

A bird-eye view of R

Yufeng Huang

Assistant Professor of Marketing, Simon Business School

July 25/26, 2022

A bird-eye view of R

- ▶ We need to learn R from the basics, but before that let me give a quick, but general introduction
- ▶ It's like learning a language
 - ▶ you don't start with the alphabet
 - ▶ you start by hearing others talk
 - ▶ but at some point you should formally know that there are 26 letters, basic grammar, how to write a sentence, how to give a speech, how to write a report, etc.
- ▶ You might find some of today's examples difficult
 - ▶ not design for us to understand all details, but hopefully we can build some general ideas from these examples
 - ▶ it's like first time you hear people speak English, you want to get a feeling of it

Things we want to learn in this lecture note

- ▶ Calculato**R**
- ▶ Variables and expressions
- ▶ Functions and packages
- ▶ Help and error code
- ▶ Throughout: readability and style

CalcuatoR

CalcuatoR

You can type numeric expressions in R and it will evaluate it for you

```
2 + 3
```

```
## [1] 5
```

```
3 * 4
```

```
## [1] 12
```

```
3^3
```

```
## [1] 27
```

```
sqrt(9) # square root
```

```
## [1] 3
```

Functions, and expressions involving more than one function

```
sqrt(9)  
log(12)  
exp(-1)
```

```
# pi is a built-in constant  
(1.2 - exp(2))^2 + (log(3) / pi)
```

Comments

The hash symbol `#` indicates a comment. Anything to the right of a `#` is ignored in execution.

```
# this is a comment
```

```
but this is not a comment
```

```
## Error:  <text>:3:5:  unexpected symbol
```

```
## 2:
```

```
## 3:  but this
```

```
##      ^
```

Use comments to talk to human readers

```
sqrt(9)          # simple comment here

# but sometimes it's good to write multiple
#   lines of comments for readability

# -----
# and you can use it to section your code
# -----
```


Use comments to talk to human readers

```
# write a header section if needed

# illustration.R
# -----
# This function illustrates the purpose of a header
#   and unfortunately has no other purposes
# Author: Yufeng Huang
# Version: 0.0.0.2
# Date: 07-25-2022
# Bug report to: yufeng.huang@simon.rochester.edu
# -----
```

Remark

- ▶ Also note that all the code on these lecture notes are actual R code
 - ▶ they run and produce results when I compile the slides
 - ▶ so you do not have to verify the results
- ▶ That means, we can focus on understanding the code and results (and take extra notes if needed) rather than typing all the code yourselves
- ▶ Also, feel *completely free* to ask any questions if you do not understand anything

Variables

Use assignment to create variables

```
# assignment with 'arrow' (standard in R)
a <- 2 + 3

# assignment with 'equal' (standard in many languages)
b = 2 - 3

# double equal '==' is for comparison
a == b

## [1] FALSE
```

Let's spend a second to think about this

- ▶ Human language “a is equal to b” might mean:
 - ▶ create a and make it equal to b
 - ▶ I assert that a is equal to b and tell me if I'm right or wrong
- ▶ For the computer, these two are completely different!
 - ▶ differentiated by “=” (or “<-”) and “==”
- ▶ Machine only understands literal statements and cannot tolerate any ambiguities

One word on single equal vs arrow

```
# totally fine for assignment
```

```
a = 2 + 3
```

```
# another use of single equal is to call an argument in the function
```

```
rm(list = ls()) # "list" is the name of an argument
```

```
# try not to mix assignment expressions
```

```
b <- 2
```

```
b = b + 2
```

R is case sensitive

```
# Z is different from z
Z <- 2
z <- 1
Z + z

## [1] 3

# but note that human eyes are not good at distinguishing cases
```

Your turn: what are a and b?

```
a <- 2  
a <- 1  
a      # what is a?  
  
b <- 1  
b <- b + b  
b <- b + b  
b      # what is b?
```

- Point: top-to-bottom order in executing simple expressions

Objects

- ▶ Everything in R is an object
 - ▶ variables, functions, ...
- ▶ To list all objects in the current session, you can either use `objects()` or `ls()`
 - ▶ to be specific it lists data, variables and user-defined functions
 - ▶ also available in the environment tab in RStudio

```
# current objects
objects()

## [1] "a" "b" "z" "Z"

# alternatively
ls()

## [1] "a" "b" "z" "Z"
```

Reserved variables

certain variable/function names are reserved: c, F, T, sum, pi, ...

```
# reserved variables are the ones that are pre-defined
```

```
F
```

```
## [1] FALSE
```

```
pi
```

```
## [1] 3.141593
```

```
# you CAN use those but you SHOULD NOT
```

```
pi <- 1 + 1
```

```
pi
```

```
## [1] 2
```

Expressions

Expressions

```
# expressions as assignment
a <- log(1)    # log() in R is natural log

# or to print
print(a)      # equivalent to just 'a'

## [1] 0

# semi-colon ends an expression, not required...
a;           # still prints a

## [1] 0

# ...unless multiple expressions in the same line
a <- 1 + 1; a <- a + 1; a    # hard to read, not advised

## [1] 3
```

Your turn: what are the outputs in the R console?

```
# expression 1
```

```
1 + 1
```

```
# expression 2
```

```
a = 1 + 1
```

```
# expression 3
```

```
a == 1 + 1
```

Your turn: why different output?

```
# expressions can be multi-line
```

```
a <- log(132) + exp(4)/log(2) +  
sqrt(log(2 + 5)^3)
```

```
a
```

```
## [1] 86.36575
```

```
# but easy to make mistake this way
```

```
a <- log(132) + exp(4)/log(2)  
+ sqrt(log(2 + 5)^3)
```

```
## [1] 2.714465
```

```
a
```

```
## [1] 83.65128
```

Indentation¹

```
# consider indenting the statement
```

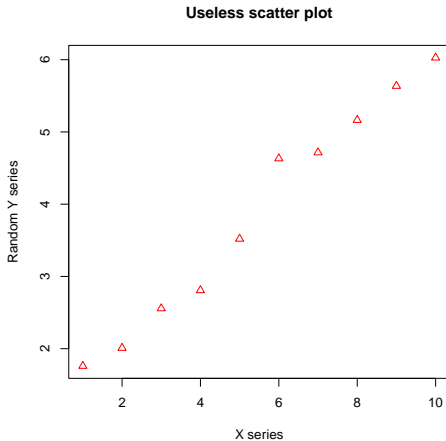
```
a <- log(132) + exp(4)/log(2) +  
  sqrt(log(2 + 5)^3)
```

```
# indentation makes things readable
```

```
x.series <- 1:10  
y.series <- NA  
for (i in 1:length(x.series)) {  
  y.series[i] <- 1 + 0.5*x.series[i] + rnorm(1)/5  
}
```

¹The second part is the “for-loop” that we saw yesterday and we’ll these in Week 3

```
# often multi-line expressions for plots
plot(x.series, y.series,
     xlab = "X series", ylab = "Random Y series",
     main = "Useless scatter plot", col = "red", pch = 24)
```



Your turn: What is y?

```
x <- 1  
y <- x^2  
x <- 2  
print(y)           # what is y?
```

Expressions are evaluated at the spot

```
x <- 1  
y <- x^2  
x <- 2  
print(y)  
## [1] 1
```

Q: What if we want to evaluate the general $y = x^2$ for any x ?

Taking stock

- ▶ Variables, assigning variables, and simple expressions that involve variables
- ▶ Programming languages (including R) are literal
 - ▶ cannot understand things that are ambiguous
 - ▶ practical note to beginners: the code that you write down are usually clear to the computer but they might not be what you mean
- ▶ Expressions are executed according to fixed order of precedence

```
# conventional order of precedence in math
x + y * z

# functions: always evaluate the argument (or: input) before the function itself
f(g(x + 1))

# top to bottom when it comes to multiple expressions; exceptions in Week 3
a <- b + 1
b <- a + 1
```

**R is free both as in “free beer”
and as in “free speech”**

Functions (= free speech)

User-defined functions

```
# define a function squared(x)
squared <- function(x) {
  y <- x^2          # body of the function
  return(y)         # returns y as output
}
```

User-defined functions (con'd)

```
# squared(x) is defined for any x
squared(1)

## [1] 1

squared(2)

## [1] 4
```

User-defined functions (con'd)

```
# squared(x) is defined for any x
```

```
squared(1)
```

```
## [1] 1
```

```
squared(2)
```

```
## [1] 4
```

```
# even if the operation "^" is illegal
```

```
squared("today is sooo sunny")
```

```
## Error in x^2: non-numeric argument to binary operator
```


Built-in functions

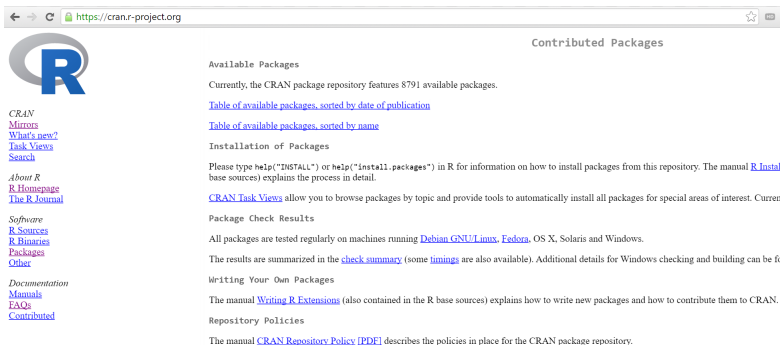
```
# sqrt() comes from the 'base' package  
sqrt(4)  
## [1] 2  
  
sqrt(1)  
## [1] 1
```

Packages (= free beer)²

²For the users; free speech for the authors.

Packages

- ▶ A package is an organized collection of functions (and datasets/other stuff)
- ▶ Written by developers and users (free speech for them)
 - ▶ distributed on the Comprehensive R Archive Network (CRAN)
 - ▶ downloadable via `install.packages()` (free beer for you)



The screenshot shows the CRAN website in a web browser. The address bar displays <https://cran.r-project.org>. The page features the CRAN logo on the left, which includes the text "CRAN", "Mirrors", "What's new?", "Task Views", "Search", "About R", "R Homepage", "The R Journal", "Software", "R Sources", "R Binaries", "Packages", "Other", "Documentation", "Manuals", "FAQs", and "Contributed". The main content area is titled "Contributed Packages" and includes sections for "Available Packages" (stating 8791 packages are available), "Installation of Packages" (providing help text and links to task views), "Package Check Results" (listing operating systems), "Writing Your Own Packages" (linking to the manual), and "Repository Policies" (linking to the policy PDF).

Available Packages

Currently, the CRAN package repository features 8791 available packages.

[Table of available packages, sorted by date of publication](#)

[Table of available packages, sorted by name](#)

Installation of Packages

Please type `help("INSTALL")` or `help("install.packages")` in R for information on how to install packages from this repository. The manual [R Install](#) base sources) explains the process in detail.

[CRAN Task Views](#) allow you to browse packages by topic and provide tools to automatically install all packages for special areas of interest. Current

Package Check Results

All packages are tested regularly on machines running [Debian GNU/Linux](#), [Fedora](#), OS X, Solaris and Windows.

The results are summarized in the [check summary](#) (some [timings](#) are also available). Additional details for Windows checking and building can be found in the [Writing Your Own Packages](#) manual.

Writing Your Own Packages

The manual [Writing R Extensions](#) (also contained in the R base sources) explains how to write new packages and how to contribute them to CRAN.

Repository Policies

The manual [CRAN Repository Policy \[PDF\]](#) describes the policies in place for the CRAN package repository.

Example: to fit a line³

```
# I want to run a linear regression
linear.fit <- lm(y.series ~ x.series)

## Error in lm(y.series ~ x.series):  could not find function "lm"
```

³Note: `lm()` comes from 'stats' package which usually comes preloaded; for illustration purposes I unloaded 'stats' in the background

Example: to fit a line³

```
# I want to run a linear regression
linear.fit <- lm(y.series ~ x.series)

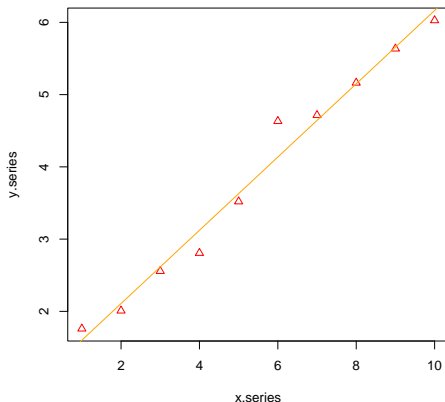
## Error in lm(y.series ~ x.series): could not find function "lm"

# but I need to load 'stats' first
library('stats')
```

³Note: `lm()` comes from 'stats' package which usually comes preloaded; for illustration purposes I unloaded 'stats' in the background

Example: to fit a line (con'd)

```
# we can run the regression after 'stats' is loaded
linear.fit <- lm(y.series ~ x.series)
plot(x.series, y.series, col = "red", pch = 24)
abline(linear.fit, col = "orange")      # add line
```



Help

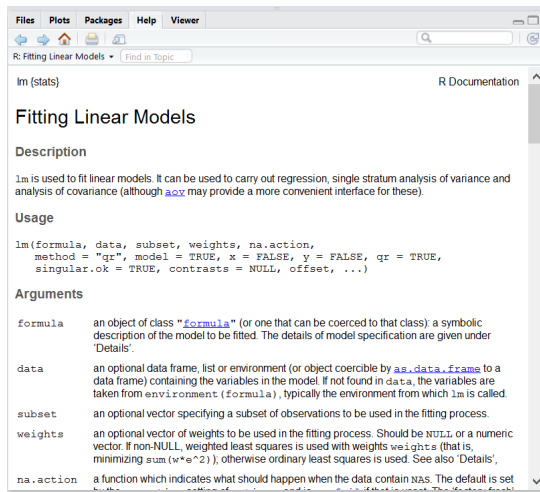
Help

```
# '?function_name' seeks help
?lm

# equivalent
help(lm)

# double '??' searches for the generic term 'lm'
??lm
```


Reading help



The screenshot shows the R help window for the `lm` function. The window has a menu bar with 'Files', 'Plots', 'Packages', 'Help', and 'Viewer'. Below the menu bar is a search bar with the text 'Find in Topic'. The main content area is titled 'lm (stats)' and 'R Documentation'. The title 'Fitting Linear Models' is prominently displayed. Below the title is the 'Description' section, which states that `lm` is used to fit linear models and can be used for regression, single stratum analysis of variance, and analysis of covariance. The 'Usage' section shows the function signature: `lm(formula, data, subset, weights, na.action, method = "qr", model = TRUE, x = FALSE, y = FALSE, qr = TRUE, singular.ok = TRUE, contrasts = NULL, offset, ...)`. The 'Arguments' section lists the parameters: `formula` (a symbolic description of the model), `data` (an optional data frame or environment), `subset` (an optional vector specifying a subset of observations), `weights` (an optional vector of weights), and `na.action` (a function which indicates what should happen when the data contain NAs).

Files Plots Packages Help Viewer

R: Fitting Linear Models Find in Topic

lm (stats) R Documentation

Fitting Linear Models

Description

`lm` is used to fit linear models. It can be used to carry out regression, single stratum analysis of variance and analysis of covariance (although [aov](#) may provide a more convenient interface for these).

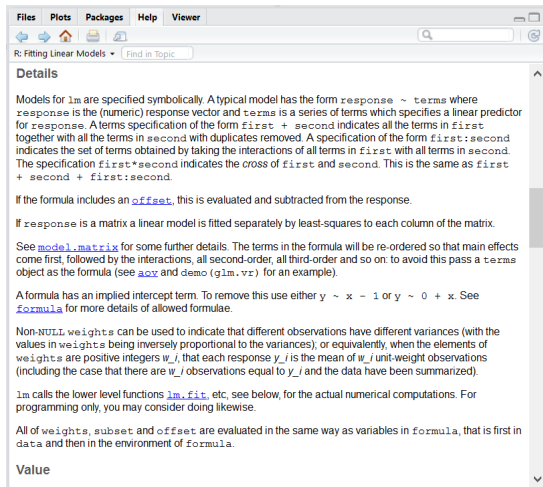
Usage

```
lm(formula, data, subset, weights, na.action,
    method = "qr", model = TRUE, x = FALSE, y = FALSE, qr = TRUE,
    singular.ok = TRUE, contrasts = NULL, offset, ...)
```

Arguments

<code>formula</code>	an object of class " formula " (or one that can be coerced to that class): a symbolic description of the model to be fitted. The details of model specification are given under 'Details'.
<code>data</code>	an optional data frame, list or environment (or object coercible by as.data.frame to a data frame) containing the variables in the model. If not found in <code>data</code> , the variables are taken from <code>environment(formula)</code> , typically the environment from which <code>lm</code> is called.
<code>subset</code>	an optional vector specifying a subset of observations to be used in the fitting process.
<code>weights</code>	an optional vector of weights to be used in the fitting process. Should be <code>NULL</code> or a numeric vector. If non- <code>NULL</code> , weighted least squares is used with weights <code>weights</code> (that is, minimizing $\sum(w \cdot e^2)$); otherwise ordinary least squares is used. See also 'Details'.
<code>na.action</code>	a function which indicates what should happen when the data contain NAs. The default is set by the option <code>na.action</code> of the package <code>stats</code> .

Reading help (con'd)



The screenshot shows the R help window for the topic "lm: Fitting Linear Models". The window has a menu bar with "Files", "Plots", "Packages", "Help", and "Viewer". Below the menu bar is a toolbar with icons for back, forward, home, search, and other navigation functions. The main content area is titled "Details" and contains the following text:

Models for `lm` are specified symbolically. A typical model has the form `response ~ terms` where `response` is the (numeric) response vector and `terms` is a series of terms which specifies a linear predictor for `response`. A terms specification of the form `first + second` indicates all the terms in `first` together with all the terms in `second` with duplicates removed. A specification of the form `first:second` indicates the set of terms obtained by taking the interactions of all terms in `first` with all terms in `second`. The specification `first*second` indicates the cross of `first` and `second`. This is the same as `first + second + first:second`.

If the formula includes an `offset`, this is evaluated and subtracted from the response.

If `response` is a matrix a linear model is fitted separately by least-squares to each column of the matrix.

See [model.matrix](#) for some further details. The terms in the formula will be re-ordered so that main effects come first, followed by the interactions, all second-order, all third-order and so on: to avoid this pass a `terms` object as the formula (see [aov](#) and `demo(glm.vr)` for an example).

A formula has an implied intercept term. To remove this use either `y ~ x - 1` or `y ~ 0 + x`. See [formula](#) for more details of allowed formulae.

Non-NULL `weights` can be used to indicate that different observations have different variances (with the values in `weights` being inversely proportional to the variances); or equivalently, when the elements of `weights` are positive integers w_i that each response y_i is the mean of w_i unit-weight observations (including the case that there are w_i observations equal to y_i and the data have been summarized).

`lm` calls the lower level functions `lm.fit`, etc, see below, for the actual numerical computations. For programming only, you may consider doing likewise.

All of `weights`, `subset` and `offset` are evaluated in the same way as variables in `formula`, that is first in data and then in the environment of `formula`.

Value

Reading help (con'd)

[lm.influence](#) for regression diagnostics, and [glm](#) for **generalized** linear models.

The underlying low level functions, [lm.fit](#) for plain, and [lm.wfit](#) for weighted regression fitting.

More `lm()` examples are available e.g., in [anscombe](#), [attitude](#), [freeny](#), [LifeCycleSavings](#), [longley](#), [stackloss](#), [swiss](#).

`biglm` in package [biglm](#) for an alternative way to fit linear models to large datasets (especially those with many cases).

Examples

```
require(graphics)

## Annette Dobson (1990) "An Introduction to Generalized Linear Models".
## Page 9: Plant Weight Data.
ctl <- c(4.17, 5.58, 5.18, 6.11, 4.50, 4.61, 5.17, 4.53, 5.33, 5.14)
trt <- c(4.81, 4.17, 4.41, 3.59, 5.87, 3.83, 6.03, 4.89, 4.32, 4.69)
group <- gl(2, 10, 20, labels = c("Ctl", "Trt"))
weight <- c(ctl, trt)
lm.D9 <- lm(weight ~ group)
lm.D90 <- lm(weight ~ group - 1) # omitting intercept

anova(lm.D9)
summary(lm.D90)

opar <- par(mfrow = c(2,2), oma = c(0, 0, 1.1, 0))
plot(lm.D9, las = 1)      # Residuals, Fitted, ...
par(opar)

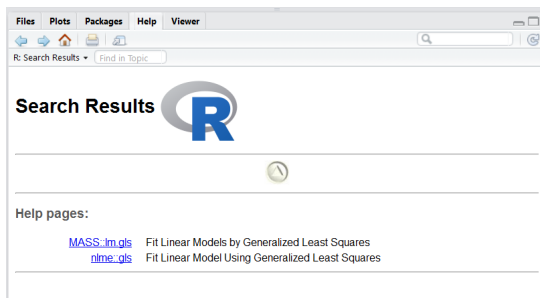
### less simple examples in "See Also" above
```

[Package *stats* version 3.3.1 [index](#)]

Help: general search

```
# '??' to search for a specific purpose  
??'fit a linear model'
```

```
# can also Google
```



Error code (and how to use them as diagnostics)

```
# no such function (a message rather than an error code)
?cool

## No documentation for 'cool' in specified packages and libraries:
## you could try '??cool'

# wrong data type on the power function (within function squared())
squared("today is sunny")

      ## Error in x^2: non-numeric argument to binary operator

# within the object (data) 'cars', no variable 'dits'
lm(formula = speed ~ dits, data = cars)

      ## Error in eval(predvars, data, env): object 'dits' not found

# no object 'car', should be 'cars'; note that error is not on 'dits'
lm(formula = speed ~ dits, data = car)

      ## Error in is.data.frame(data): object 'car' not found
```


To

Installing R on Mac - Warning messages: Setting LC_CTYPE failed ...

Setting LC_CTYPE failed, using "C" 2: Setting ...

You visited this page on 7/31/17.

Taking stock

- ▶ R is free both as in “free beer” and as in “free speech”
 - ▶ you can define virtually any function on your own
 - ▶ you can use functions that others have defined and built
- ▶ There are so many functions out there and you will have to read a lot of help files
 - ▶ help files are in fixed format so it is reasonably easy to get used to understanding them
- ▶ Code running into error is a daily thing
 - ▶ many times you can figure out these errors reasonably quickly
 - ▶ but also many times it is much easier to seek others (including Google) for help
 - ▶ and remember to ask questions in (and out of) class!

House-keeping

Good house-keeping



Bad house-keeping



House-keeping

- ▶ We talked about coding style, indentation, read-ability, spacing throughout this class
- ▶ The point is that the machine can read your code if it's correct
- ▶ But humans cannot
 - ▶ easier to make mistakes
 - ▶ easier to be mis-interpreted by others
 - ▶ ignored by others
- ▶ Just like writing a bad article
- ▶ This is important and I will dedicate more time on this later on

Bad example earlier

```
# for the computer these are 3 separate expressions
a <- log(132) + exp(4)/log(2)    # assignment, no output
+ sqrt(log(2 + 5)^3)           # evaluate and print
a                                # then prints a

## [1] 2.714465
## [1] 83.65128
```

Bad example earlier

```
# for the computer these are 3 separate expressions
a <- log(132) + exp(4)/log(2)    # assignment, no output
+ sqrt(log(2 + 5)^3)           # evaluate and print
a                                # then prints a

## [1] 2.714465
## [1] 83.65128
```

```
# for a human, easily confused with this
a <- log(132) + exp(4)/log(2) + # assignment; code continues
sqrt(log(2 + 5)^3)
a                                # then prints a

## [1] 86.36575
```

Another example

- ▶ Suppose I want to evaluate the root x of quadratic equations with the general form

$$a \cdot x^2 + b \cdot x + c = 0$$

where a , b and c are parameters

- ▶ Mathematically there are two solutions of x , jointly expressed as

$$x = \frac{-b \pm \sqrt{b^2 - 4ac}}{2a}$$

- ▶ Suppose there are 3 such equations:

- ▶ $-x^2 + 2x + 1 = 0$

- ▶ $x^2 - 2x + 1 = 0$

- ▶ $x^2 - 4x + 3 = 0$

Poor house-keeping

```
# approach I: brute-force calculatoR

x11<-(-2+sqrt(2^2-4*(-1)*1))/(2*(-1))
x12<-(-2-sqrt(2^2-4*(-1)*1))/(2*(-1))
x21<-(2+sqrt((-2)^2-4*1*1))/(2*1)
x22<-(2-sqrt((-2)^2-4*1*1))/(2*1)
x31<-(4+sqrt((-4)^2-4*1*3))/(2*1)
x32<-(4-sqrt((-4)^2-4*1*3))/(2*1)

# was there a typo?
```


Wrong approach (but good intention)

```
# approach II: write down a "generic" expression
```

```
x1<-(-b+sqrt(b^2-4*a*c))/(2*a)
```

```
x2<-(-b-sqrt(b^2-4*a*c))/(2*a)
```

```
# let's only look at one case
```

```
a<--1;b<-2;c<-1
```

```
print(x1);print(x2)
```

```
# your turn: what's wrong?
```

Good approach⁴

```
# approach III: write a function

quadratic <- function(a, b, c) {
  root <- sqrt(b^2 - 4*a*c)      # used more than once
  x1 <- (-b + root) / (2*a)
  x2 <- (-b - root) / (2*a)
  list(x1, x2)
}

# evaluate the function at the same case
x <- quadratic(-1, 2, 1)
x

## [[1]]
## [1] -0.4142136
##
## [[2]]
## [1] 2.414214
```

⁴Also note indentation and spacing

Summary: important points

- ▶ Simple expressions
- ▶ Comment
- ▶ Single vs double equal
- ▶ Order of precedence and order of executing multi-line code (top to bottom)
- ▶ Variables, assign value to variables, replace value (by re-assigning)
- ▶ General understanding of the R environment
 - ▶ indentation
 - ▶ multi-line expressions
 - ▶ functions (preview) and packages
 - ▶ help
 - ▶ error code, warning messages