R workshop

An introduction of R and RStudio

Peiyi Zhou

Octerber 15, 2025

University College London

Table of contents

This tutorial is based on the online R tutorial provided by W3Schools, which can be found at *`https://www.w3schools.com/r/`*.

What is R and RStudio?

What is R?

R is a statistical programming language widely used for data analysis, simulation and evaluation.

Normally, R is preferred in academia rather than industries (where programming languages like Python are more commonly used). However, in recent days it acts more importantly, as it gives a user-friendly interface and behave powerful in statistical learning.

Figure 1: The logo for R.

There is no perfect programming language, only suitable programming language for specific task!

Some advantages of R:

1. Similarity coding principle with Python and other programming languages (eg. selection, iteration, etc.).
2. Free software, and users can create their own R packages which can be made available to R community for academic purposes. For example, the GNAR package in R [Leeming et al., 2020]: *https://cran.r-project.org/web/packages/GNAR/index.html*
3. R provides comprehensive help with all its packages, functions and datasets.

What is RStudio?

RStudio is an editor that can operate R programming - it is somewhat like the Jupyter notebook or the Visual Studio for coding Python.

1. We can run R codes in the console of RStudio.
2. We can generate LaTeX markdown file by using R markdown.

Figure 2: The logo for RStudio.

Download R and RStudio

You must install R on your personal machine first! For doing this, visit: *https://cran.r-project.org* where you will find instructions on how to download and install R on Windows, MAC and Linux devices.

(Tip: install all your programming languages in C Drive!)

After that we will be able to install the RStudio from *https://posit.co/download/rstudio-desktop/*.

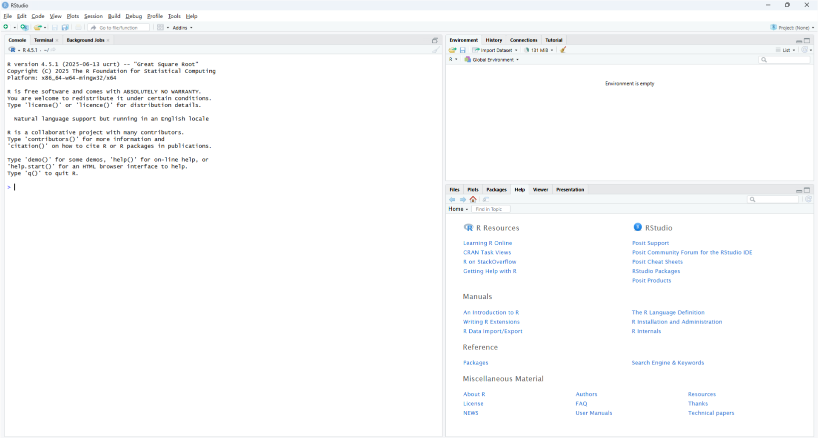
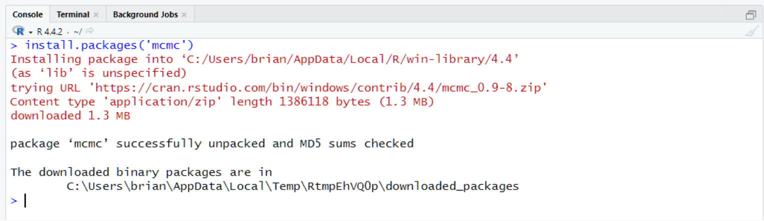


Figure 3: RStudio Interface

Start examples: using R console

- Installation of necessary packages.



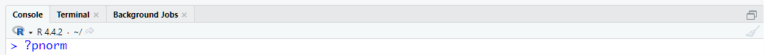
```
Console  Terminal  Background Jobs
R 4.4.2 · ~/
> install.packages('mcmc')
Installing package into 'C:/Users/brian/AppData/Local/R/win-library/4.4'
(as 'lib' is unspecified)
trying URL 'https://cran.rstudio.com/bin/windows/contrib/4.4/mcmc_0.9-8.zip'
Content type 'application/zip' length 1386118 bytes (1.3 MB)
downloaded 1.3 MB

package 'mcmc' successfully unpacked and MD5 sums checked

The downloaded binary packages are in
        C:\Users\brian\AppData\Local\Temp\RtmpEhVQ0p\downloaded_packages
>
```

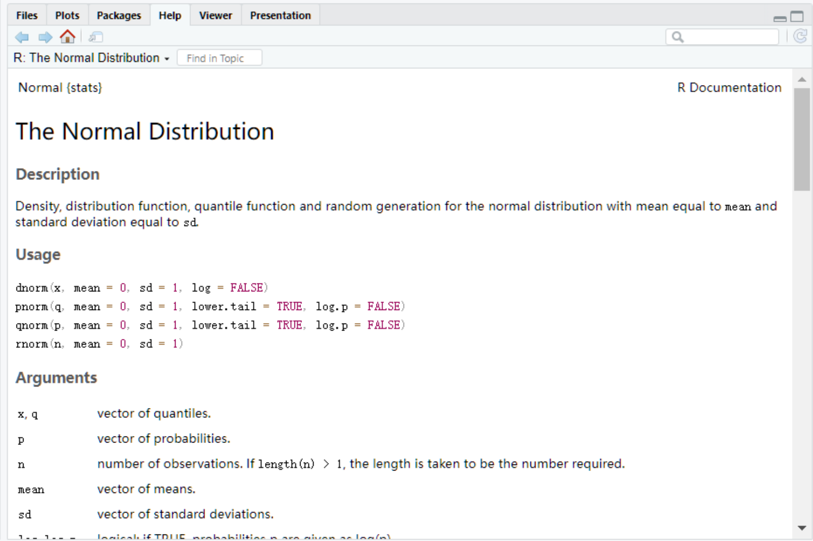
Figure 4: Installation of package MCMC.

- Help for certain R function.



```
Console  Terminal  Background Jobs
R 4.4.2 · ~/
> ?pnorm
```

Figure 5: Information check for pnorm function.


Files Plots Packages Help Viewer Presentation

R: The Normal Distribution

Find in Topic

Normal {stats} R Documentation

The Normal Distribution

Description

Density, distribution function, quantile function and random generation for the normal distribution with mean equal to `mean` and standard deviation equal to `sd`.

Usage

```
dnorm(x, mean = 0, sd = 1, log = FALSE)
pnorm(q, mean = 0, sd = 1, lower.tail = TRUE, log.p = FALSE)
qnorm(p, mean = 0, sd = 1, lower.tail = TRUE, log.p = FALSE)
rnorm(n, mean = 0, sd = 1)
```

Arguments

`x, q`	vector of quantiles.
`p`	vector of probabilities.
`n`	number of observations. If `length(n) > 1`, the length is taken to be the number required.
`mean`	vector of means.
`sd`	vector of standard deviations.

Figure 6: Complete information for pnorm and also some related functions.

R basics

Math in R: numerical operations

R can be used as a calculator that can perform simple arithmetic operations.

We will run the following examples in R console to see the expected results.

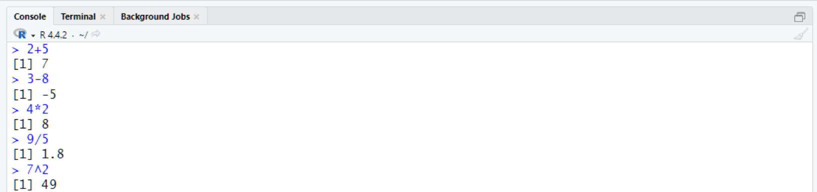


Figure 7: Numerical operations in R console.

Print the results

R console can also print the results in different data types.

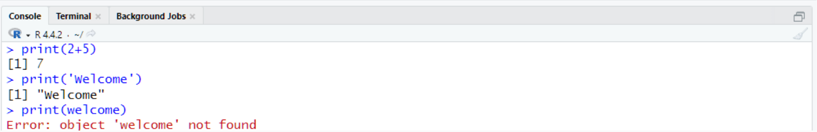


Figure 8: Printing.

Comments

Like Python and other programming languages, R can also insert comments by using #.

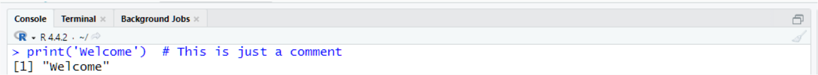


Figure 9: Comment.

R variables and objects

Creating variables and objects

A variable/object in R is created by first assigning a value to it. Such assignation is either done by using '=' or '<-'.

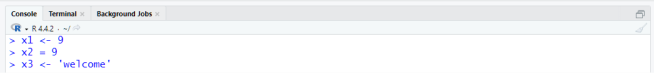


Figure 10: Assign variables in different ways.

You can see the values for the stored variables as well.

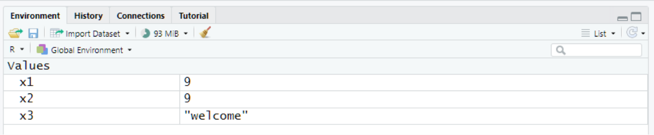


Figure 11: Stored variables in R.

You can remove the variable (which are usually temporary variables) to free your memory by using *rm* (short for ‘remove’).

To output the stored variables, we just need to type the name of the variable.

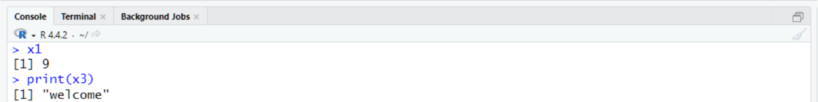


Figure 12: Output of stored variables.

Variable data types

Variables can store data of different types, and different types can do different things.

Common data types for variables in R include:

- `numeric`: like decimal in Python, eg. $45.3, -8.4, 6.0, 9$.
- `integer`: note that 6 is a `numeric`, to result the data type be `integer`, whether we change the form as 6L (where L stands for the integer), or apply `as.integer`.
- `complex`: complex numbers, eg. $1 + i$, where i is the imaginary part.
- `character`: same as 'string' in Python.
- `logical`: same as 'boolean' in Python.

The data type for variables can be examined by using `class`.

```
Console   Terminal ×   Background Jobs ×
R 4.4.2 · ~/
> class(x1)
[1] "numeric"
> class(as.integer(x1))
[1] "integer"
> class(x3)
[1] "character"
> class(1+i)
Error: object 'i' not found
> class(1+1i)
[1] "complex"
> class(is.numeric(x3))
[1] "logical"
> class(9 > 8)
[1] "logical"
```

Figure 13: Data type examples, examined by `class`.

R operators

R operators

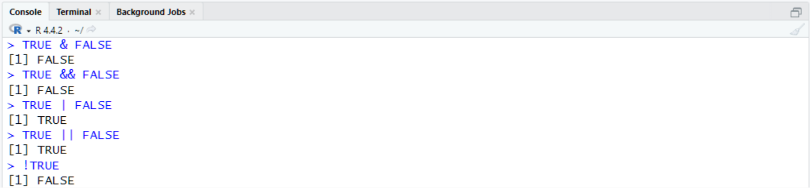
We conclude commonly used operators in R for different data types.

Arithmetic operators	
Operator	Meaning
$+$	addition
$-$	subtraction
$*$	multiplication
$/$	division
$\wedge$	exponentiation
sqrt	square root
%%	modulus
%/%	integer division

Note that in R we denote constants like π as *pi*, and e as *exp(1)*. The exponent of e is written as *exp(x)* instead.

Comparison operators	
Operator	Meaning
$==$	equal
$! =$	not equal
$>$	greater than
$<$	less than
$>=$	greater or equal to
$<=$	less or equal to

<table>
<tr><th colspan="2">Logical operators</th></tr>
<tr><th>Operator</th><th>Meaning</th></tr>
<tr><td>&</td><td>Element-wise Logical AND operator.</td></tr>
<tr><td>&&</td><td>Logical AND operator.</td></tr>
<tr><td>|</td><td>Element-wise Logical OR operator.</td></tr>
<tr><td>||</td><td>Logical OR operator.</td></tr>
<tr><td>!</td><td>Logical NOT operator.</td></tr>
</table>



```
Console   Terminal ×   Background Jobs ×
R 4.4.2 · ~/
> TRUE & FALSE
[1] FALSE
> TRUE && FALSE
[1] FALSE
> TRUE | FALSE
[1] TRUE
> TRUE || FALSE
[1] TRUE
> !TRUE
[1] FALSE
```

Figure 14: Logical operator examples.

A quick example for explaining the difference of element-wise logical operator and the (vectorised) logical operator.

```
Console  Terminal ×  Background Jobs ×
R 4.4.2 · ~/
> c(1,3) == c(1,4)  # analogous with &
[1]  TRUE FALSE
> all(c(1,3) == c(1,4))  # analogous with &&
[1] FALSE
```

Figure 15: A toy example for explaining the essence of element-wise and vectorised logical operator.

In selection where we only need single TRUE/FALSE statement, we prefer using && and ||.

Selection and Iteration in R

The syntax for selection in R (i.e. using *`if...else`*) is rather similar with Python and other programming languages. We here use a toy example that combine with the logical operators introduced before.

```
> # Recall that we define x1=9
> if ((is.numeric(x1)) && (x1 > 4)){
+     print('This is a numeric number that is greater than 4.')
+ } else if ((is.integer(x1)) && (x1 > 6)){
+     print('This is an integer that is greater than 6.')
+ } else{
+     print('Otherwise.')
+ }
[1] "This is a numeric number that is greater than 4."
```

Figure 16: A example of R selection.

R iteration: *for* loop

Like Python, there are two ways in iteration in R: ***while*** loop and *for* loop.

The *for* loop restricts the number of iterations. The syntax of *for* loop can be seen in the below example.

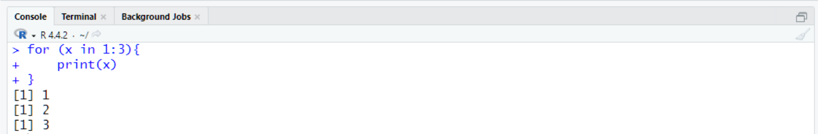


Figure 17: A simple *for* loop example.

In R the indexing starts from 1 (instead of 0 in Python and other programming languages).

We can apply `for` loop for an updating of elements in the vector/matrix using proper indexing.

Exercise: How to do the following matrix update?

$$\begin{pmatrix} 1 & 2 & 3 \\ 4 & 5 & 6 \\ 7 & 8 & 9 \end{pmatrix} \longrightarrow \begin{pmatrix} 9 & 8 & 7 \\ 6 & 5 & 4 \\ 3 & 2 & 1 \end{pmatrix}$$

```
Console   Terminal ×   Background Jobs ×
R 4.4.2 · ~/
> mat <- matrix(c(1:9), nrow = 3, ncol = 3, byrow = TRUE)  # create a matrix by rows
> mat
     [,1] [,2] [,3]
[1,]    1    2    3
[2,]    4    5    6
[3,]    7    8    9
> # Examine the number of rows and columns for the matrix
> nrow(mat);  ncol(mat)
[1] 3
[1] 3
> # Now we apply an update on mat by using double for loop
> for (i in 1:nrow(mat)){
+     for (j in 1:ncol(mat)){
+         mat[i,j] <- 9 - mat[i,j] + 1
+     }
+ }
> mat  # see the updated mat
     [,1] [,2] [,3]
[1,]    9    8    7
[2,]    6    5    4
[3,]    3    2    1
```

Figure 18: Update each entry in the matrix by using double `for` loop.

`for` loop in R can also be stopped earlier once we set certain stopping criteria, which can be done by using `break`.

Exercise: Suppose we now work on the updated matrix, how do we determine indices that have values greater than or equal to 5?

```
Console   Terminal ×   Background Jobs ×
R 4.4.2 · ~/
> for (i in 1:nrow(mat)){
+     for (j in 1:ncol(mat)){
+         if (mat[i,j] < 5){
+             break
+         }
+         # print more than 1 things separated by spaces
+         cat('The required index is:', i, j, '\n')
+     }
+ }
The required index is: 1 1
The required index is: 1 2
The required index is: 1 3
The required index is: 2 1
The required index is: 2 2
```

Figure 19: Print out the indices with entries ≥ 5 for the updated matrix using *`for`* loop.

We might also skip unnecessary iteration without terminating the `for` loop by using `next`.

Exercise: Similar with the previous example, but how do we output indices for the updated matrix that having entries being odd?

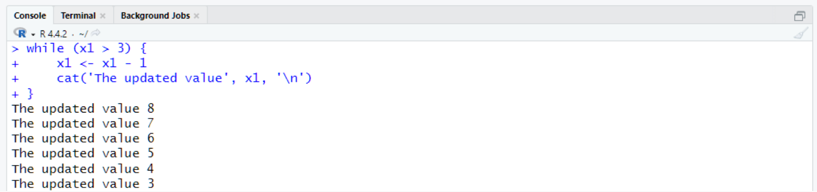
```
> for (i in 1:nrow(mat)){
+     for (j in 1:ncol(mat)){
+         if (mat[i,j] %% 2 == 0){
+             next
+         }
+         # print more than 1 things separated by spaces
+         cat('The required index is:', i, j, '\n')
+     }
+ }
The required index is: 1 1
The required index is: 1 3
The required index is: 2 2
The required index is: 3 1
The required index is: 3 3
```

Figure 20: Print out the indices with entries being odd for the updated matrix using *for* loop.

R iteration: `while` loop

We also introduce the ***while*** loop. Unlike ***for*** loop, we must need a stopping criteria otherwise the loop will never terminate.

We can even stop earlier by using the ***break***, or skip an iteration without stopping it by using ***next***. The use of these two are similar with ***for*** loop.




```
> while (x1 > 3) {
+     x1 <- x1 - 1
+     cat('The updated value', x1, '\n')
+ }
The updated value 8
The updated value 7
The updated value 6
The updated value 5
The updated value 4
The updated value 3
```

Figure 21: Simple ***while*** loop example.

Function

R functions

Finally we introduce the way for defining functions in R. This can be done by using *function*.

Normally we first assign the function with a certain name, and then use *function* to define how to use it for certain purposes.

Console Terminal Background Jobs

R 4.4.2 · ~/

```
> hello_func <- function(language){
+     if (language == 'English'){
+         print('Hello.')
+     } else if (language == 'French'){
+         print('Bonjour.')
+     } else if (language == 'German'){
+         print('Hallo.')
+     } else{
+         print('Try to use listed languages: English, French or German')
+     }
+ }
> hello_func('German')
[1] "Hallo."
```

Figure 22: A simple function for translation of ‘hello’ in three different languages: English, French and German.

Exercise: Could you write a function that takes a squared matrix as an argument and return a new matrix that having all entries reversed? i.e. We want a function that can do

$$\begin{pmatrix} 1 & 2 & 3 \\ 4 & 5 & 6 \\ 7 & 8 & 9 \end{pmatrix} \longrightarrow \begin{pmatrix} 9 & 8 & 7 \\ 6 & 5 & 4 \\ 3 & 2 & 1 \end{pmatrix}$$

```
Console   Terminal   Background Jobs
R 4.5.1 · ~/
> inv_entry <- function(mat){
+     new_mat <- matrix(0, nrow = nrow(mat), ncol = ncol(mat))  # initialise
+     for (i in 1:nrow(mat)){
+         for (j in 1:ncol(mat)){
+             new_mat[i,j] <- mat[nrow(mat) - i + 1, ncol(mat) - j + 1]
+         }
+     }
+     new_mat  # return the new matrix
+ }
```

Figure 23: Coding for such function achieving 'inverse entries' purpose.

```
> test_mat1 <- matrix(c(1:4), nrow = 2, ncol = 2, byrow = TRUE)
> inv_entry(test_mat1)
     [,1] [,2]
[1,]    4    3
[2,]    2    1
> test_mat2 <- matrix(c(1:16), nrow = 4, ncol = 4, byrow = TRUE)
> inv_entry(test_mat2)
     [,1] [,2] [,3] [,4]
[1,]   16   15   14   13
[2,]   12   11   10    9
[3,]    8    7    6    5
[4,]    4    3    2    1
```

Thank you for your listening!

References

Leeming, K., Nason, G. P., and Nunes, M. A. (2020).
GNAR: Methods for Fitting Network Time Series Models.
R package version 1.1.1, available at
https://CRAN.R-project.org/package=GNAR.