DATA7001 - Introduction to Data Science
Basic R programming

Xin GUO

Email: xin.guo@uq.edu.au
Office: **Priestley Building (67), Room 447**
Office Hours: **Tuesday 10–11 AM**

R

R is a programming language. R also refers to a free software environment for statistical computing and visualization. It compiles and runs on a wide variety of UNIX platforms, Windows and MacOS. R was originally designed for statistical computing, and now people also use it to do numerical analysis. It is stable, powerful, and widely used.

Figure: the R logo.

- An interpreter-based programming, graphics and statistics package.
- Free, stable, can be extended.
- Can easily perform standard statistical and numerical analysis.
- Can be programmed to handle non-standard cases.
- For complex tasks, it is often used as a first step to interface with `C` or `FORTRAN`.
- Almost all new statistical methodologies are published with ready-to-use packages built with **R**.

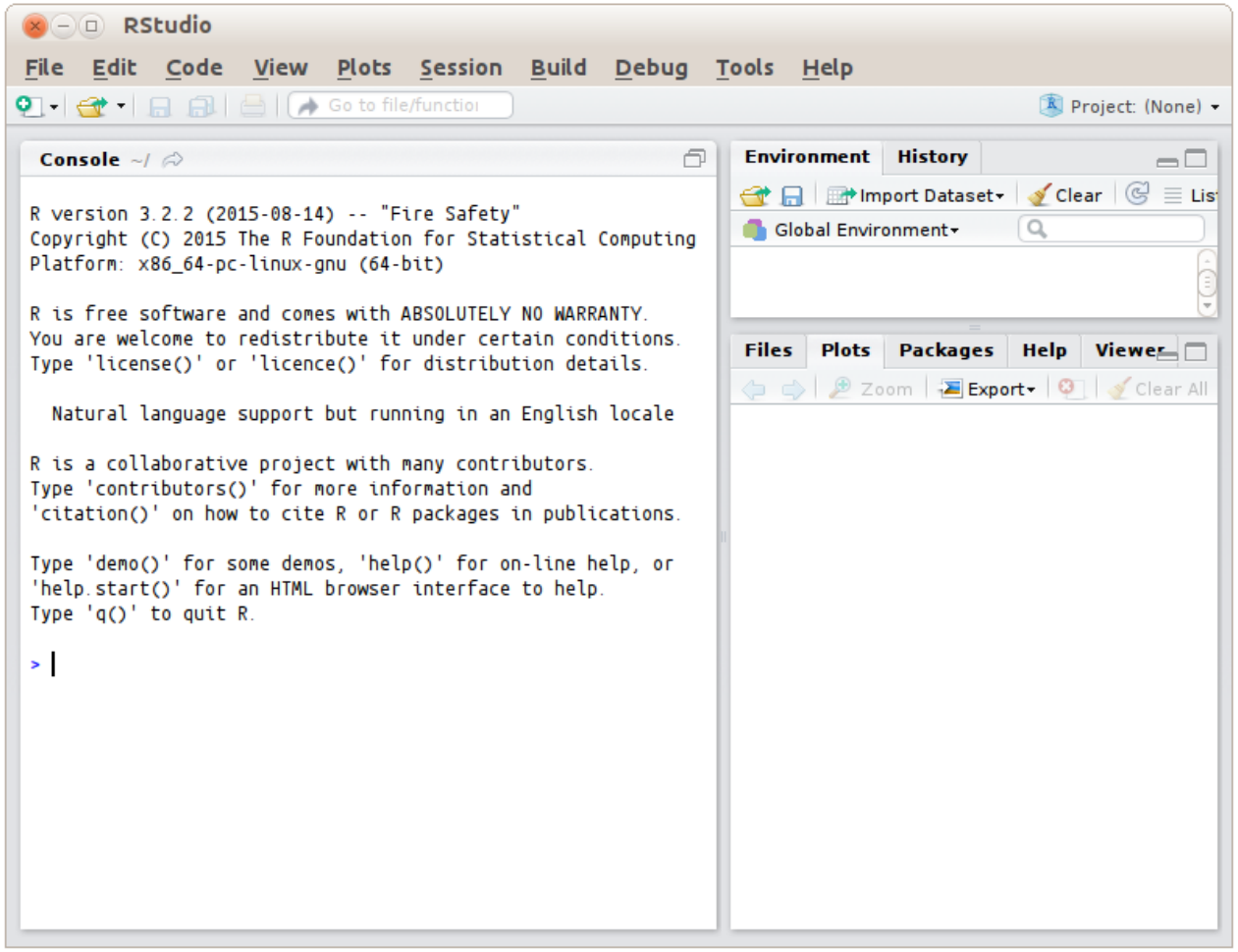
R is free in the sense of "free beer", so that you can download it, run it, and re-distribute it with no cost. R is free also in the sense of "free speech", so that you can modify it, and release your modified version (called "fork") of R (for minimizing confusion, R requires that if you modify and release your version of R, you must use some different name). R is usually used in the command-line mode, which already offers a lot of features and freedom. However, there is a graphical user interface (GUI) shell available for R, called RStudio, available for download at `https://www.rstudio.com`.

Launching R

```
xin@xin-tc-M91p: ~
xin@xin-tc-M91p:~$ R

R version 3.2.2 (2015-08-14) -- "Fire Safety"
Copyright (C) 2015 The R Foundation for Statistical Computing
Platform: x86_64-pc-linux-gnu (64-bit)

R is free software and comes with ABSOLUTELY NO WARRANTY.
You are welcome to redistribute it under certain conditions.
Type 'license()' or 'licence()' for distribution details.

  Natural language support but running in an English locale

R is a collaborative project with many contributors.
Type 'contributors()' for more information and
'citation()' on how to cite R or R packages in publications.

Type 'demo()' for some demos, 'help()' for on-line help, or
'help.start()' for an HTML browser interface to help.
Type 'q()' to quit R.

>
```

The welcome message of R environment on Linux system.

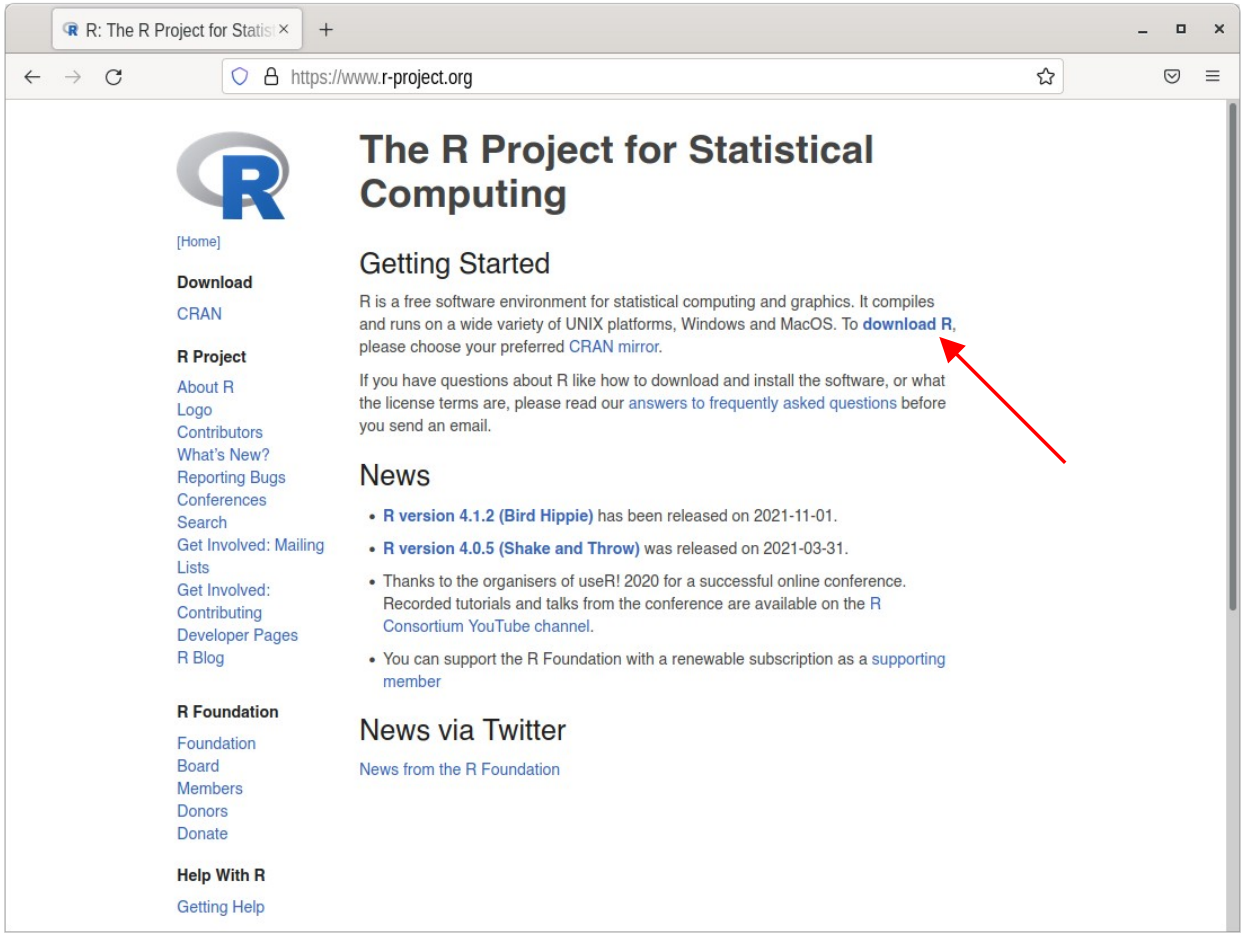
If you launch RStudio, you get the following welcome message. RStudio did nothing but just call R.



The welcome message of RStudio.

Downloading and installing R

visit `https://www.r-project.org/`



A server closer to you may provide faster downloading.

CRAN - Mirrors

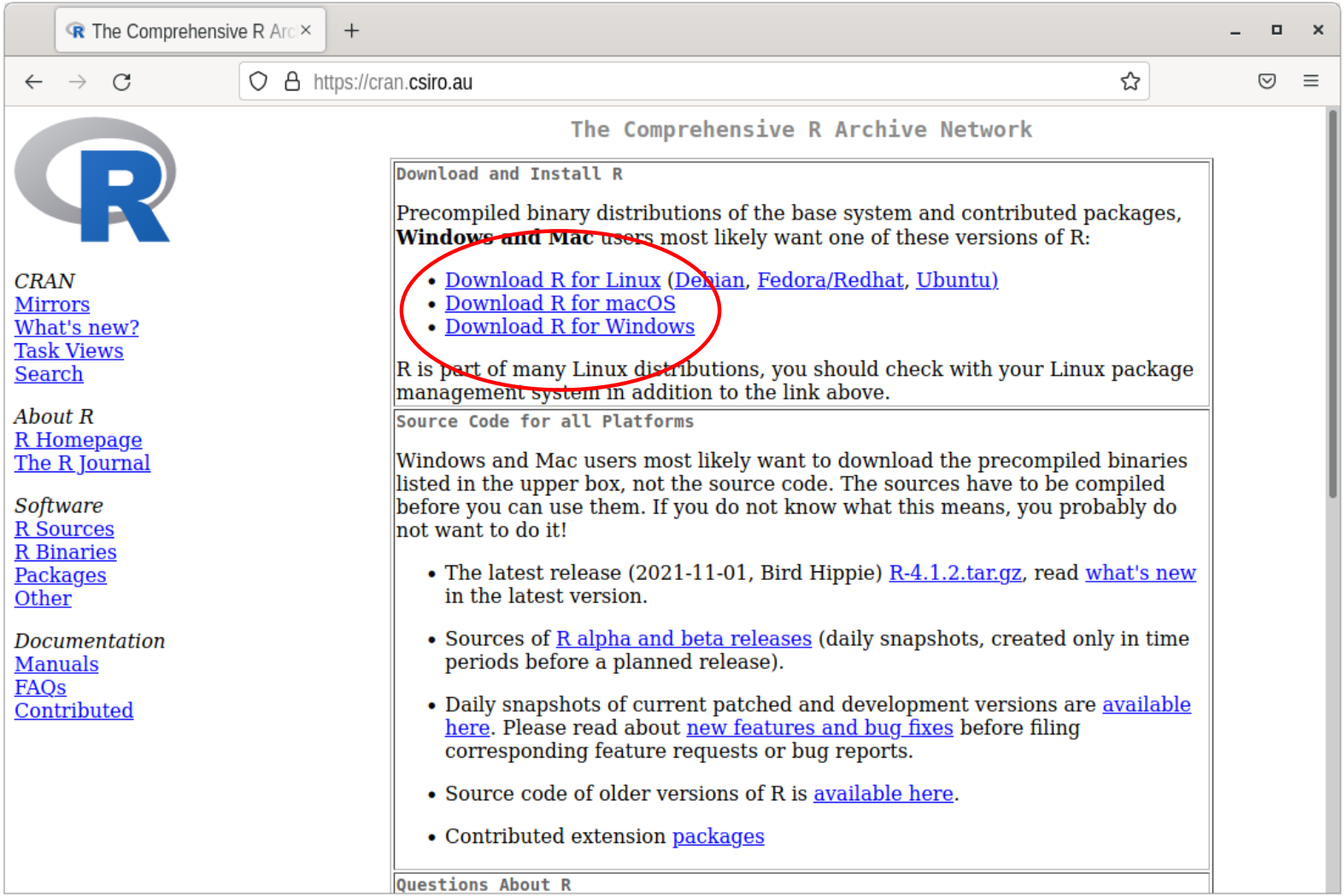
https://cran.r-project.org/mirrors.html

CRAN Mirrors

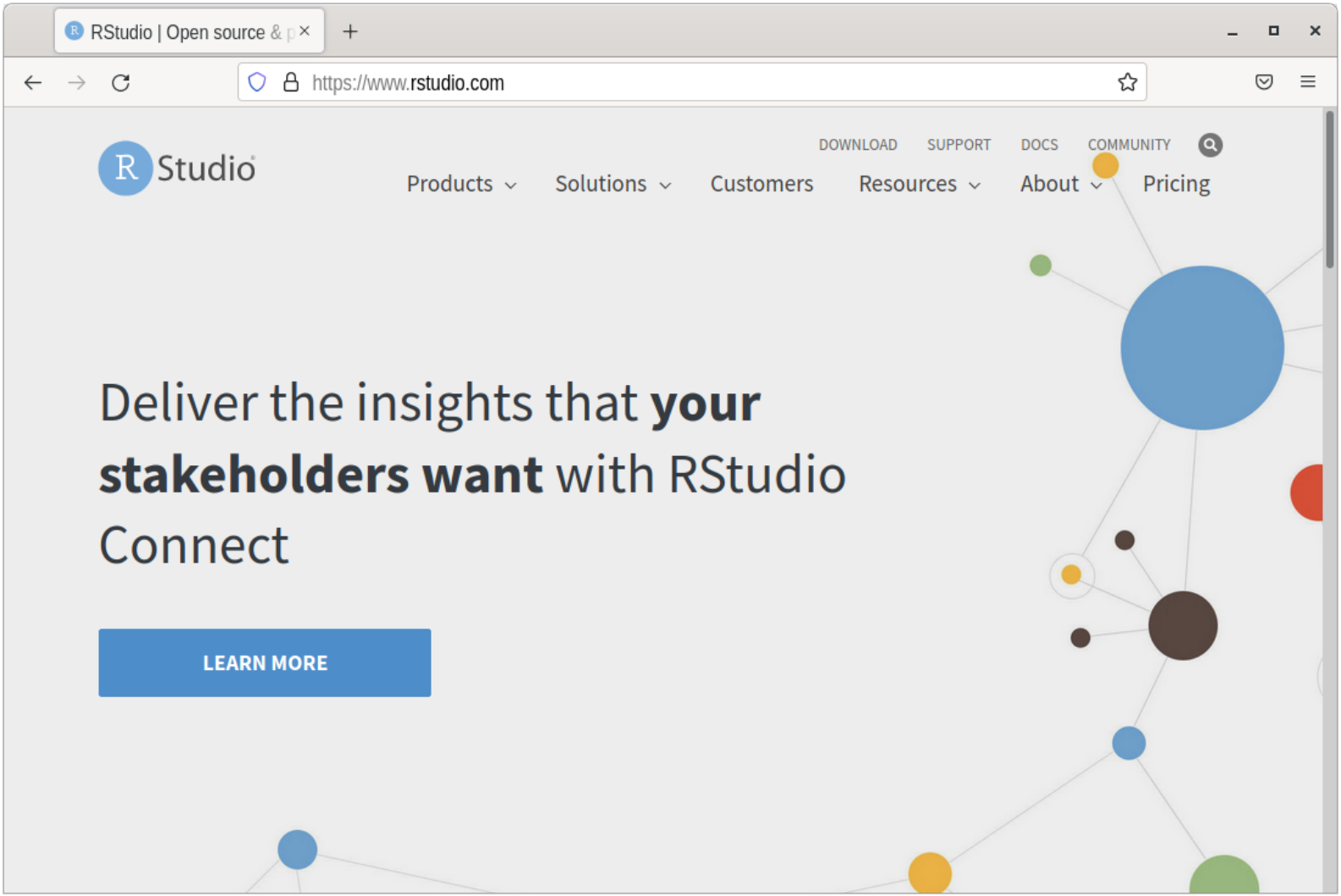
The Comprehensive R Archive Network is available at the following URLs, please choose a location close to you. Some statistics on the status of the mirrors can be found here: main page, windows release, windows old release.

If you want to host a new mirror at your institution, please have a look at the CRAN Mirror HOWTO.

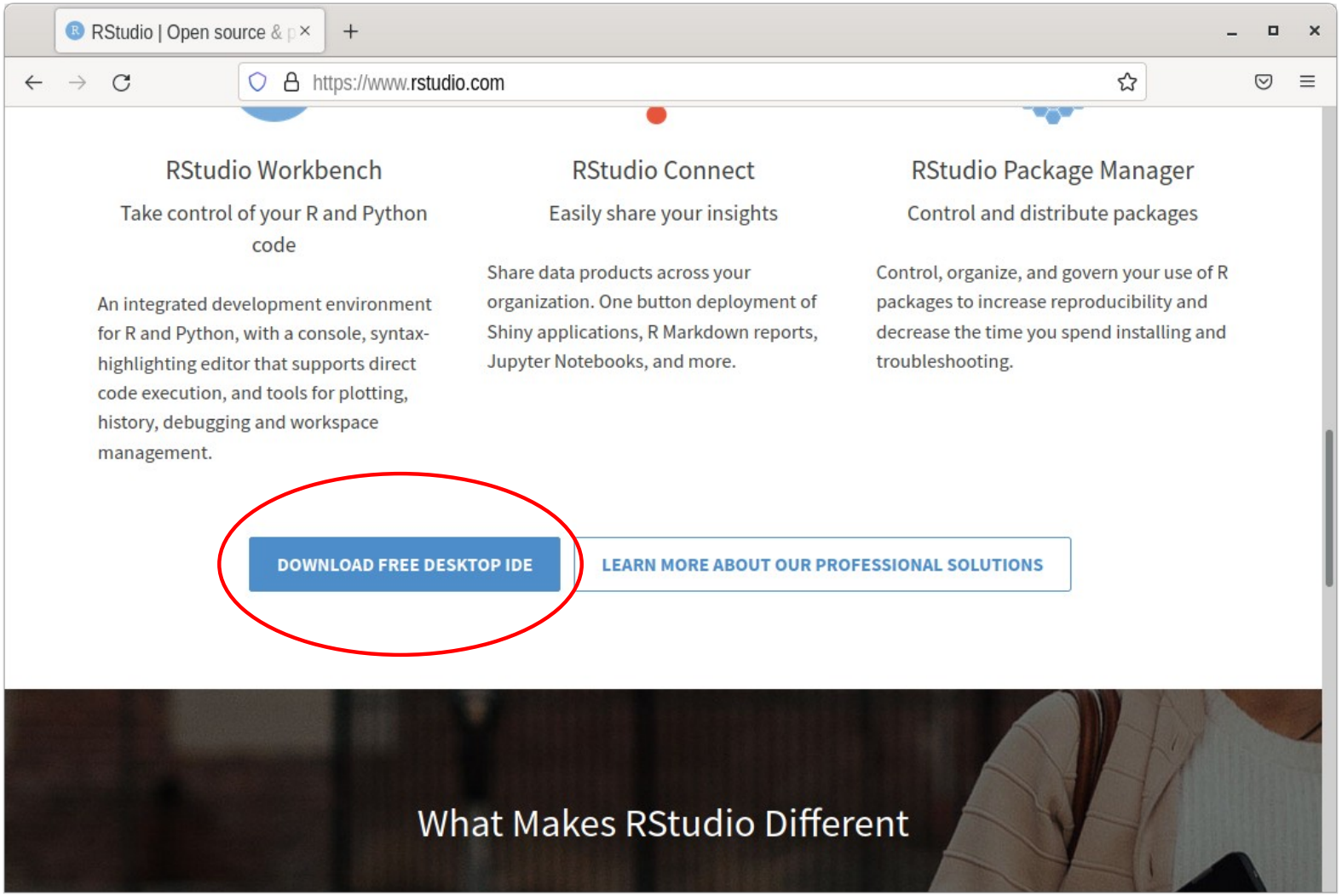
0-Cloud	
https://cloud.r-project.org/	Automatic redirection to servers worldwide, currently sponsored by Rstudio
Algeria	
https://cran.usthb.dz/	University of Science and Technology Houari Boumediene
Argentina	
http://mirror.fcaglp.unlp.edu.ar/CRAN/	Universidad Nacional de La Plata
Australia	
https://cran.csiro.au/	CSIRO
https://mirror.aarnet.edu.au/pub/CRAN/	AARNET
https://cran.ms.unimelb.edu.au/	School of Mathematics and Statistics, University of Melbourne
https://cran.curtin.edu.au/	Curtin University
Austria	
https://cran.wu.ac.at/	Wirtschaftsuniversität Wien
Belgium	
https://www.freestatistics.org/cran/	Patrick Wessa
https://ftp.belnet.be/mirror/CRAN/	Belnet, the Belgian research and education network
Brazil	
https://cran-r.c3sl.ufpr.br/	Universidade Federal do Parana
https://cran.fiocruz.br/	Oswaldo Cruz Foundation, Rio de Janeiro
https://vps.fmvz.usp.br/CRAN/	University of Sao Paulo, Sao Paulo
https://brieger.esalq.usp.br/CRAN/	University of Sao Paulo, Piracicaba


The Comprehensive R Arc
https://cran.csiro.au
The Comprehensive R Archive Network
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
Contributed
Download and Install R
Precompiled binary distributions of the base system and contributed packages, Windows and Mac users most likely want one of these versions of R:
Download R for Linux (Debian, Fedora/Redhat, Ubuntu)
Download R for macOS
Download R for Windows
R is part of many Linux distributions, you should check with your Linux package management system in addition to the link above.
Source Code for all Platforms
Windows and Mac users most likely want to download the precompiled binaries listed in the upper box, not the source code. The sources have to be compiled before you can use them. If you do not know what this means, you probably do not want to do it!
The latest release (2021-11-01, Bird Hippie) R-4.1.2.tar.gz, read what's new in the latest version.
Sources of R alpha and beta releases (daily snapshots, created only in time periods before a planned release).
Daily snapshots of current patched and development versions are available here. Please read about new features and bug fixes before filing corresponding feature requests or bug reports.
Source code of older versions of R is available here.
Contributed extension packages
Questions About R

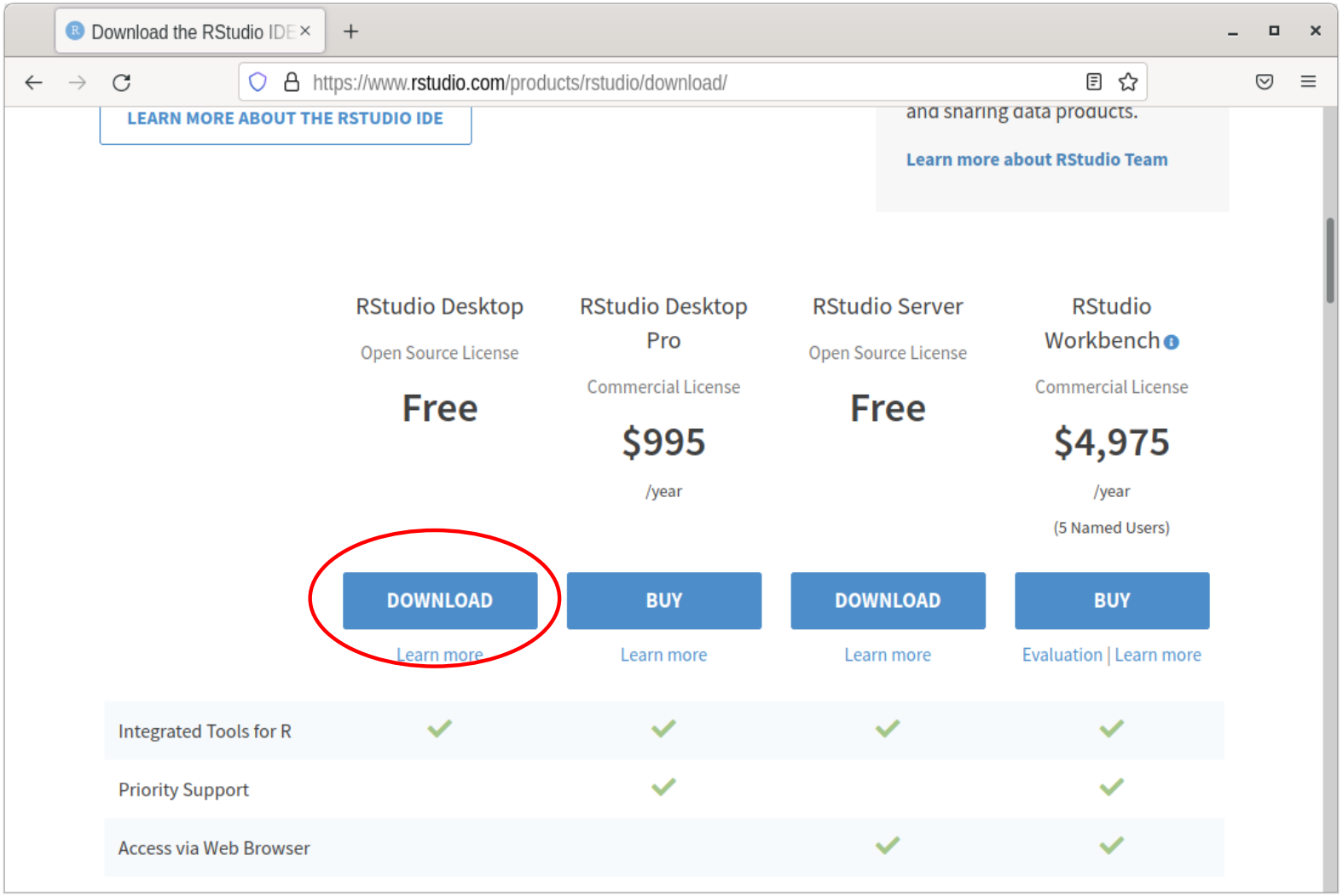
RStudio is an integrated development environment (IDE) for R, which is optional and may provide some help. RStudio is available at `https://www.rstudio.com/`



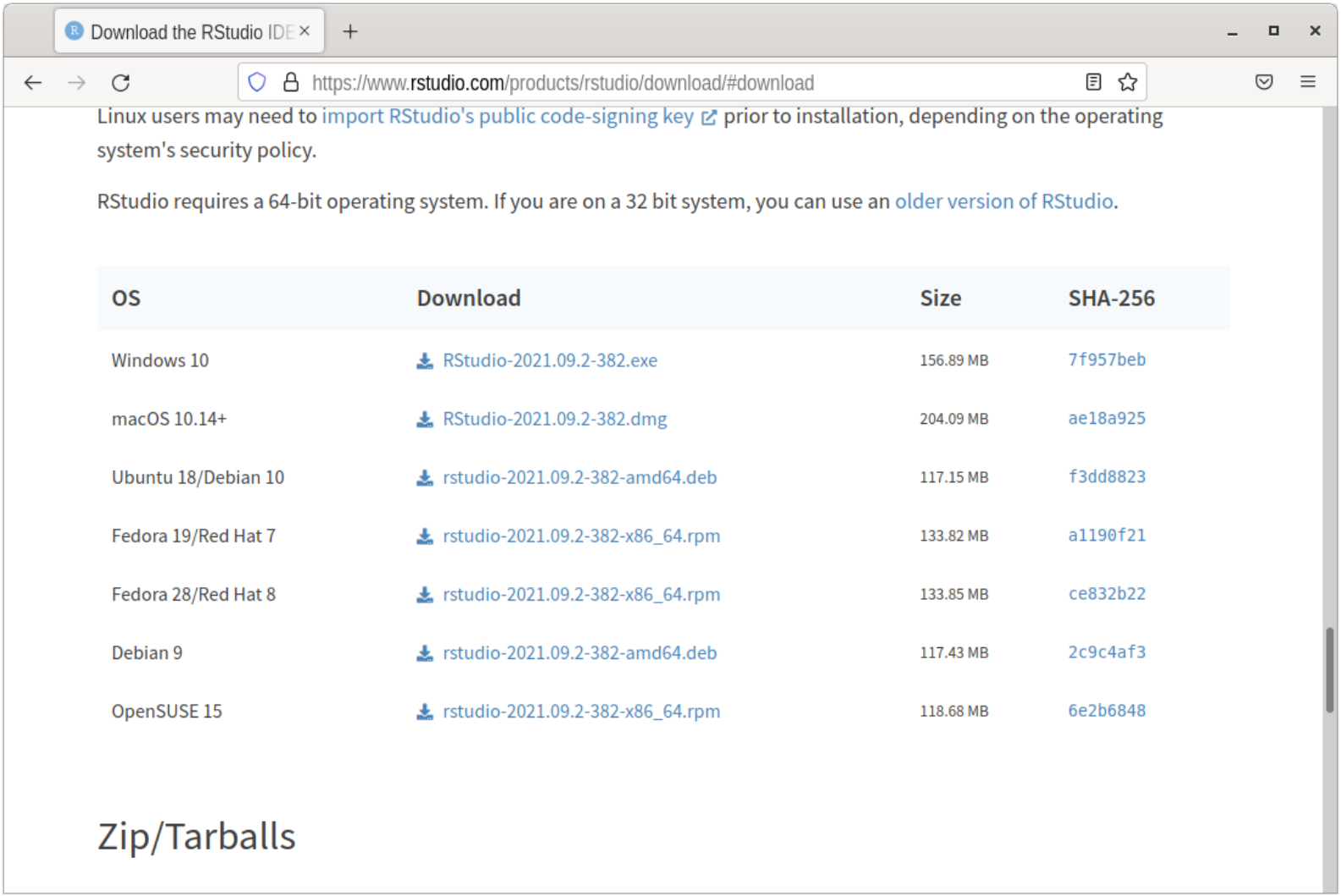
On the homepage of RStudio, scroll down to find the download link.



You may explore the “RStudio Server” but the basic “RStudio Desktop” is good enough.



Select the package that matches your operating system.



Some simple commands

- You can directly type commands at the prompt:

```
> 2 + 3
[1] 5
> 5 * pi
[1] 15.70796
> rnorm(20)
 [1] -0.57442492  1.68545014 -0.01613824 -0.14110880  0.59271169
 [6]  0.08165454  0.16220551 -0.53236657 -0.11977163  0.56726110
[11] -0.19887324 -0.77690512 -1.10581382 -0.70444775 -0.03470233
[16] -0.92088930 -2.95547369  0.22873348 -0.30088267 -0.27960355
```

- If you hit enter before completely entering a command, you will get a + prompt. You must complete the command or type CTR+C (Esc in MSW) to continue.

```
> 3 +
+ 5
[1] 8
```

- All arithmetic operations are represented via standard symbols (+ - * /) and have the usual order of precedence.

```
> 3 + 4 * 2
[1] 11
> sin(pi / 6)
[1] 0.5
> exp(log(2) + log(3))
[1] 6
> atan(Inf)/pi
[1] 0.5
```

Vectors

- R has six common types of atomic vectors: integer, double, logical, character, complex, and raw[1].

```
> a <- c(1, 3, 5, 7, 9)
> a
[1] 1 3 5 7 9
> typeof(a)
[1] "double"
> length(a)
[1] 5
> b <- 1:5
> typeof(b)
[1] "integer"
> x <- (a >= 5)
> x
[1] FALSE FALSE  TRUE  TRUE  TRUE
> typeof(x)
[1] "logical"
> x[2]
[1] FALSE
```

```
> y <- c("hello", "world", "!")
> typeof(y)
[1] "character"
> length(y)
[1] 3
> nchar(y)
[1] 5 5 1
> z <- 3 + 5i
> typeof(z)
[1] "complex"
> z^2
[1] -16+30i
> sin(z)
[1] 10.47251-73.46062i
> exp(z)
[1] 5.69751-19.26051i
```

- In each vector, all the elements must have the same atomic data type.

[1] In this course we will not expand complex and raw.

List

List is a data structure like vector, but can have components of mixed data types.

```
> a <- list(1:4, sin, "hello", TRUE, 1+3i)
> a[[1]]
[1] 1 2 3 4
> a[[2]](pi/6)
[1] 0.5
> a[[3]]
[1] "hello"
> a[[4]]
[1] TRUE
> Arg(a[[5]])
[1] 1.249046
> Arg(a[[5]]) -> e
> Mod(a[[5]]) -> r
> r * (cos(e) + 1i * sin(e))
[1] 1+3i
> names(a) <- letters[1:5]
> a$a
[1] 1 2 3 4
> a$b
function (x)  .Primitive("sin")
> a$c
[1] "hello"
```

Reading documentation

Documentation is important for learning **R**. Type `?log`, or `?"log"`, for example, to see the help page of the function `log`. Why quoting? Because sometimes we need the help page of operators. Try `?"+"` and `?+` separately. Press `q` to quit the help page.

```
xin@xin-tc-M91p: ~
log                     package:base                    R Documentation

Logarithms and Exponentials

Description:

     '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| << 1.

     'exp' computes the exponential function.

     'expm1(x)' computes exp(x) - 1 accurately also for |x| << 1.

Usage:

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

     log1p(x)

     exp(x)
     expm1(x)

Arguments:

       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)'.
:
```

Help page of the logarithm functions.

Save your commands in a file

- Each R session works under a directory ("folder"). To see the current working directory, type the following command

```
> getwd()
[1] "/home/xin/Downloads"
```

- To change the current working directory, use the function "`setwd`".

```
> setwd("/home/xin/Documents")
> getwd()
[1] "/home/xin/Documents"
```

- For example, one can make a file "`a.R`" (different filename extensions are OK, for example you can use the text file "`a.txt`") under the working directory. Save the following code to the file `a.R`.

```
print("hello, world")
```

- Then in the R session, "`source`" it.

```
> source("a.R")
[1] "hello, world"
```

Loops

- The following `for` loop is used to find $e = 2.71828\cdots = \sum_{i=0}^{\infty} \frac{1}{i!}$. There are two key words: `for` and `in`.

```
a <- 0
for(i in 0:18){
  a <- a + 1/factorial(i)
}
### below are the results:
> a
[1] 2.718282
> 0:18
 [1]  0  1  2  3  4  5  6  7  8  9 10 11 12 13 14 15 16 17 18
> typeof(0:18)
[1] "integer"
```

- One more example:

```
for(i in c("Brisbane", "New York")){
  print(paste("I love", i))
}
### below are the results:
[1] "I love Brisbane"
[1] "I love New York"
```

- R also provides the `while` loop

```
a <- 0
while(a < 4){
  a <- a + 1
  print(a)
}
### below are the results:
[1] 1
[1] 2
[1] 3
[1] 4
```

- The keyword "`break`" is used to break from a loop. Save the following code to "`a.R`".

```
a <- 0
repeat{
  a <- a + 1
  if(a > 4) break
  print(a)
}
```

  In the R session,

```
> source("a.R")
[1] 1
[1] 2
[1] 3
[1] 4
```

Note that, for practicing programming, the best way is to program, especially on some problems well motivated.

Example

Use the `for` loop to find

$$\sum_{k=1}^{100} k.$$

If you get the answer 5050, your code is probably correct.

Example

Recall the definition of the Fibonacci numbers: $a_1 = a_2 = 1$ and for any $i \geq 3$,

$$a_i = a_{i-1} + a_{i-2}.$$

Find the first 100 Fibonacci numbers.

Vectors (mathematical concept)

An array $\boldsymbol{x}$ of n real numbers $x_1, \ldots, x_n$ is called a *vector*, and it is written as

$$\boldsymbol{x} = \begin{bmatrix} x_1 \\ x_2 \\ \vdots \\ x_n \end{bmatrix}, \quad \text{or} \quad \boldsymbol{x}' = [x_1, x_2, \cdots, x_n].$$

- Here the prime denotes the operation of transposing a column to a row.
- The number n is referred to as the *dimension* of the vector $\boldsymbol{x}$.
- The *coordinates* $x_1, \ldots, x_n$ can also be complex numbers, in which case $\boldsymbol{x}$ is a complex vector. However in this course we consider ONLY real numbers and real vectors.

- The set of all the real vectors of dimension n, is denoted by $\mathbb{R}^n$.
- One can scale a vector $\boldsymbol{x}$ by multiplying it by a constant c.

$$\boldsymbol{x} = \begin{bmatrix} x_1 \\ x_2 \\ \vdots \\ x_n \end{bmatrix} \quad \Rightarrow \quad c\boldsymbol{x} = \begin{bmatrix} cx_1 \\ cx_2 \\ \vdots \\ cx_n \end{bmatrix}.$$

- Vectors can be added.

$$\boldsymbol{x} + \boldsymbol{y} = \begin{bmatrix} x_1 \\ x_2 \\ \vdots \\ x_n \end{bmatrix} + \begin{bmatrix} y_1 \\ y_2 \\ \vdots \\ y_n \end{bmatrix} = \begin{bmatrix} x_1 + y_1 \\ x_2 + y_2 \\ \vdots \\ x_n + y_n \end{bmatrix}.$$

- **inner product**: if vectors $\boldsymbol{x}$ and $\boldsymbol{y}$ have the same dimension n,

$$\langle \boldsymbol{x}, \boldsymbol{y} \rangle := \sum_{i=1}^{n} x_i y_i = \langle \boldsymbol{y}, \boldsymbol{x} \rangle .$$

- **length**, or **norm**:

$$\|\boldsymbol{x}\| := \sqrt{\langle \boldsymbol{x}, \boldsymbol{x} \rangle} = \sqrt{x_1^2 + x_2^2 + \cdots + x_n^2}.$$

- For any constant c,

$$\langle c\boldsymbol{x}, \boldsymbol{y} \rangle = c \langle \boldsymbol{x}, \boldsymbol{y} \rangle ;$$
$$\|c\boldsymbol{x}\| = \sqrt{\langle c\boldsymbol{x}, c\boldsymbol{x} \rangle} = \sqrt{c^2 \langle \boldsymbol{x}, \boldsymbol{x} \rangle} = |c| \sqrt{\langle \boldsymbol{x}, \boldsymbol{x} \rangle} = |c| \|\boldsymbol{x}\|.$$

- **triangle inequality**: (proof left as a warming up exercise)

$$\|\boldsymbol{x} + \boldsymbol{y}\| \leq \|\boldsymbol{x}\| + \|\boldsymbol{y}\|.$$

Basic linear algebra with R

There are several ways to generate a vector in R.

```
> a <- 1:5
> a
[1] 1 2 3 4 5
> b <- 5:1
> b
[1] 5 4 3 2 1
> x <- seq(1, 5, length = 10)
> x
 [1] 1.000000 1.444444 1.888889 2.333333 2.777778 3.222222 3.666667
 [8] 4.111111 4.555556 5.000000
> runif(5)
[1] 0.1664853 0.6454725 0.3244731 0.4217759 0.3419001
> rep(1, times = 5)
[1] 1 1 1 1 1
> rep(1:5, each = 3)
 [1] 1 1 1 2 2 2 3 3 3 4 4 4 5 5 5
> rep(1:5, times = 3)
 [1] 1 2 3 4 5 1 2 3 4 5 1 2 3 4 5
> rep(1:5, length = 14)
 [1] 1 2 3 4 5 1 2 3 4 5 1 2 3 4
```

We may take a look at the properties of a vector

```
> x <- 1:5
> length(x)
[1] 5
> typeof(x)
[1] "integer"
> str(x)
 int [1:5] 1 2 3 4 5
> object.size(x)
80 bytes
> object.size(1:10000)
40048 bytes
> object.size(1:100000)
400048 bytes
> object.size(1:1000000)
4000048 bytes
> object.size(as.double(1:10000))
80048 bytes
>
> y <- seq(1, 5, len = 10000)
> head(y)
[1] 1.0000 1.0004 1.0008 1.0012 1.0016 1.0020
> tail(y)
[1] 4.9980 4.9984 4.9988 4.9992 4.9996 5.0000
> length(y)
[1] 10000
> typeof(y)
[1] "double"
> object.size(y)
80048 bytes
```

By default, R takes vectors as column matrices, so a transpose of a vector is a row matrix.

```
> a
[1] 1 2 3 4 5
> t(a)
     [,1] [,2] [,3] [,4] [,5]
[1,]    1    2    3    4    5
> a <- 1:5
> t(a) # THE TRANSPOSE OF a
     [,1] [,2] [,3] [,4] [,5]
[1,]    1    2    3    4    5
> t(t(a))
     [,1]
[1,]    1
[2,]    2
[3,]    3
[4,]    4
[5,]    5
```

However, vectors do not have the "dimension" attribute, and only matrices have.

```
> dim(a)
NULL
> dim(t(a))
[1] 1 5
> dim(t(t(a)))
[1] 5 1
> attributes(t(a))
$dim
[1] 1 5

> attributes(a)
NULL
```

R does not have handy functions for inner product or vector norms. These operations are usually implemented through functions `sum` and `sqrt`.

```
> a <- 1:5
> a * 2
[1]  2  4  6  8 10
> a ^ 2
[1]  1  4  9 16 25
> 2 ^ a
[1]  2  4  8 16 32

> a + rep(10, length = 5)
[1] 11 12 13 14 15
> b <- rep(1, length = 5)
> b
[1] 1 1 1 1 1
> sum(a * b) # INNER PRODUCT
[1] 15
> sqrt(sum(b ^ 2)) # VECTOR NORM
[1] 2.236068

> sqrt(sum((-2 * b) ^ 2))
[1] 4.472136
> 2 * sqrt(sum(b ^ 2))
[1] 4.472136
```

Matrix

- **Matrix** A, of dimension $m \times n$, is just a table of (real) numbers.

$$A = \begin{bmatrix} A_{1,1} & A_{1,2} & \cdots & A_{1,n} \\ A_{2,1} & A_{2,2} & \cdots & A_{2,n} \\ \vdots & \vdots & \ddots & \vdots \\ A_{m,1} & A_{m,2} & \cdots & A_{m,n} \end{bmatrix}.$$

- $A_{i,j}$ is called the (i,j)-entry of the matrix A.
- **Transpose**: A' is a $n \times m$ matrix, defined by

$$A' = \begin{bmatrix} A_{1,1} & A_{2,1} & \cdots & A_{m,1} \\ A_{1,2} & A_{2,2} & \cdots & A_{m,2} \\ \vdots & \vdots & \ddots & \vdots \\ A_{1,n} & A_{2,n} & \cdots & A_{m,n} \end{bmatrix}.$$

Example: $\begin{bmatrix} 1 & 2 \\ 3 & 4 \\ 5 & 6 \end{bmatrix} \overset{\text{transpose}}{\Longrightarrow} \begin{bmatrix} 1 & 3 & 5 \\ 2 & 4 & 6 \end{bmatrix}$

- Write $\mathbb{R}^{m\times n}$ the space of all the $n \times m$ matrices.
- Matrix addition: $A = A_{m\times n}$, $B = B_{m\times n}$, then $A + B$ is an $m \times n$ matrix with $(A + B)_{i,j} = A_{i,j} + B_{i,j}$. Matrix $A - B$ is similarly defined.
- Matrix multiplication: $A = A_{m\times n}$, $B = B_{n\times p}$, then AB is a $m \times p$ matrix with

$$(AB)_{i,j} = \sum_{k=1}^{n} A_{i,k} B_{k,j}$$

- Hadamard product: $A, B \in \mathbb{R}^{m\times n}$, $(A \circ B)_{i,j} = A_{i,j} B_{i,j}$.
- Let c be a constant, then cA is an $m \times n$ matrix with $(cA)_{i,j} = cA_{i,j}$.
- Let $A = A_{m\times n}$ be a matrix and $x = x_{n\times 1}$ be a vector, then Ax is an $m \times 1$ vector with $(Ax)_i := A_{i,1}x_1 + A_{i,2}x_2 + \cdots + A_{i,n}x_n$.

Examples

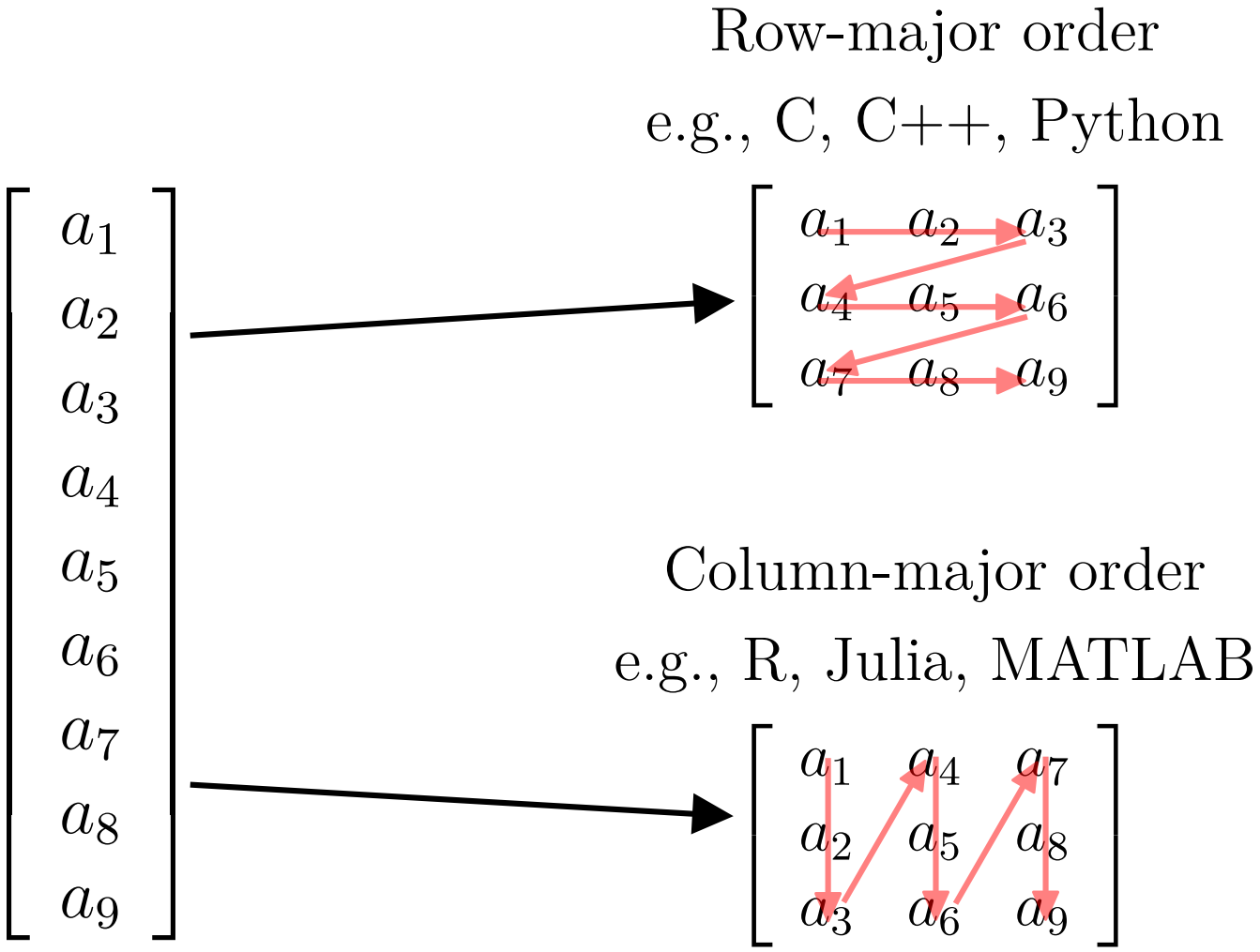
$$\begin{bmatrix} a & b \\ c & d \end{bmatrix} + \begin{bmatrix} \alpha & \beta \\ \gamma & \delta \end{bmatrix} = \begin{bmatrix} a+\alpha & b+\beta \\ c+\gamma & d+\delta \end{bmatrix} \in \mathbb{R}^{2\times 2},$$

$$\begin{bmatrix} a & b \\ c & d \end{bmatrix} \circ \begin{bmatrix} \alpha & \beta \\ \gamma & \delta \end{bmatrix} = \begin{bmatrix} a\alpha & b\beta \\ c\gamma & d\delta \end{bmatrix} \in \mathbb{R}^{2\times 2},$$

$$\begin{bmatrix} a & b & c \\ d & e & f \end{bmatrix} \begin{bmatrix} \alpha \\ \beta \\ \gamma \end{bmatrix} = \begin{bmatrix} a\alpha + b\beta + c\gamma \\ d\alpha + e\beta + f\gamma \end{bmatrix} \in \mathbb{R}^{2},$$

$$\begin{bmatrix} a & b \\ c & d \\ e & f \end{bmatrix} \begin{bmatrix} \alpha & \beta \\ \gamma & \delta \end{bmatrix} = \begin{bmatrix} a\alpha + b\gamma & a\beta + b\delta \\ c\alpha + d\gamma & c\beta + d\delta \\ e\alpha + f\gamma & e\beta + f\delta \end{bmatrix} \in \mathbb{R}^{3\times 2}.$$

In R, generating a matrix is easy. Note that R is "column-major", i.e., when we feed a vector into a matrix, elements of the vector are fed into the matrix one column after another.



```
> a <- matrix(1:6, nrow = 3)
> b <- matrix((1:6) * 100, ncol = 2)
> a
     [,1] [,2]
[1,]    1    4
[2,]    2    5
[3,]    3    6
> b
     [,1] [,2]
[1,]  100  400
[2,]  200  500
[3,]  300  600
> a + b # matrix addition
     [,1] [,2]
[1,]  101  404
[2,]  202  505
[3,]  303  606
> a * b # Hadamard product
     [,1] [,2]
[1,]  100 1600
[2,]  400 2500
[3,]  900 3600
> x <- c(7, 8)
> x
[1] 7 8
> a %*% x # matrix-vector product
     [,1]
[1,]   39
[2,]   54
[3,]   69
```

```
> b[1:2,] # a submatrix
     [,1] [,2]
[1,]  100  400
[2,]  200  500
> a %*% b[1:2,] # matrix-matrix product
     [,1] [,2]
[1,]  900 2400
[2,] 1200 3300
[3,] 1500 4200
> a
     [,1] [,2]
[1,]    1    4
[2,]    2    5
[3,]    3    6
> a + 10 # adding a scalar to a matrix
     [,1] [,2]
[1,]   11   14
[2,]   12   15
[3,]   13   16
> a * 10 # multiplying a matrix by a scalar
     [,1] [,2]
[1,]   10   40
[2,]   20   50
[3,]   30   60
```

For linear equation systems where the coefficient matrix is a square matrix of full rank,

$$\underbrace{\begin{bmatrix} A_{1,1} & A_{1,2} & \cdots & A_{1,n} \\ A_{2,1} & A_{2,2} & \cdots & A_{2,n} \\ \vdots & \vdots & \ddots & \vdots \\ A_{n,1} & A_{n,2} & \cdots & A_{n,n} \end{bmatrix}} \begin{bmatrix} x_1 \\ x_2 \\ \vdots \\ x_n \end{bmatrix} = \begin{bmatrix} y_1 \\ y_2 \\ \vdots \\ y_n \end{bmatrix}$$

$$A \quad \boldsymbol{x} = \boldsymbol{y}$$

the solution is $\boldsymbol{x} = A^{-1}\boldsymbol{y}$, where A^{-1} denotes the matrix inverse of A. Then $A^{-1}A = AA^{-1} = I$, which is the identity matrix.

Finding matrix inverse, and solving linear equations are easy in R.

```
> A <- matrix(c(1,2,3,6,5,4,8,7,9), nrow = 3)
> A
     [,1] [,2] [,3]
[1,]    1    6    8
[2,]    2    5    7
[3,]    3    4    9
> x <- c(2,3,5)
> x
[1] 2 3 5
> y <- A %*% x
> y
     [,1]
[1,]   60
[2,]   54
[3,]   63
> solve(A, y) # method 1
     [,1]
[1,]    2
[2,]    3
[3,]    5
> solve(A, y) - x # numerical errors
              [,1]
[1,]  1.776357e-15
[2,]  4.440892e-16
[3,] -8.881784e-16
> 10^(-15)
[1] 1e-15
```

```
> solve(A)
           [,1]       [,2]       [,3]
[1,] -0.8095238  1.0476190 -0.0952381
[2,] -0.1428571  0.7142857 -0.4285714
[3,]  0.3333333 -0.6666667  0.3333333
> A %*% solve(A)
              [,1]         [,2]         [,3]
[1,]  1.000000e+00 0.000000e+00 4.440892e-16
[2,]  1.665335e-16 1.000000e+00 1.665335e-16
[3,] -1.665335e-16 3.330669e-16 1.000000e+00
> solve(A) %*% y # method 2, slower
     [,1]
[1,]    2
[2,]    3
[3,]    5
> solve(A) %*% y - x
              [,1]
[1,] -1.332268e-15
[2,]  1.332268e-15
[3,]  8.881784e-16
```

The following example (demonstrated via the Julia language) shows the limit on precision of `double`. This is also an example that in general, floating-point addition and multiplication are not associative (i.e. one may have $a + b + c \neq a + (b + c)$ and $a \times b \times c \neq a \times (b \times c)$)

```
julia> eps(1.0)
2.220446049250313e-16

julia> eps(1.0) / 2 + 1.0 - 1.0
0.0

julia> eps(1.0) / 2 + (1.0 - 1.0)
1.1102230246251565e-16

julia> bitstring(1.0 + eps(1.0))
"0011111111110000000000000000000000000000000000000000000000000001"

julia> bitstring(1.0 + eps(1.0)/2)
"0011111111110000000000000000000000000000000000000000000000000000"

julia> bitstring(1.0)
"0011111111110000000000000000000000000000000000000000000000000000"
```

There are some other matrix operations.

```
> a <- matrix(1:9, nrow = 3)
> a
     [,1] [,2] [,3]
[1,]    1    4    7
[2,]    2    5    8
[3,]    3    6    9
> cbind(1, a, t(a))
     [,1] [,2] [,3] [,4] [,5] [,6] [,7]
[1,]    1    1    4    7    1    2    3
[2,]    1    2    5    8    4    5    6
[3,]    1    3    6    9    7    8    9
> rbind(sqrt(1:3), a)
     [,1]     [,2]     [,3]
[1,]    1 1.414214 1.732051
[2,]    1 4.000000 7.000000
[3,]    2 5.000000 8.000000
[4,]    3 6.000000 9.000000
```

```
> b <- matrix(1:6, nrow = 2)
> b
     [,1] [,2] [,3]
[1,]    1    3    5
[2,]    2    4    6
> apply(b, 1, sum)
[1]  9 12
> apply(b, 2, prod)
[1]  2 12 30
> sum(1:100)
[1] 5050
> prod(1:4)
[1] 24
```

Example (exercise, Fibonacci sequence, revisited)

Recall the Fibonacci sequence $\{a_n\}_{n=1}^{\infty}$ defined by $a_1 = a_2 = 1$ and then iteratively by $a_{n+2} = a_{n+1} + a_n$ for any integer $n \geq 1$. The iterative relation could be written in the vector form as

$$\begin{bmatrix} a_{n+2} \\ a_{n+1} \end{bmatrix} = \begin{bmatrix} 1 & 1 \\ 1 & 0 \end{bmatrix} \begin{bmatrix} a_{n+1} \\ a_n \end{bmatrix}, \quad \text{for } n \geq 1.$$

Use the above form to find a_{30}. Do you have a faster way to compute a_{30}?

Vectors and Indexing

A vector is an 1-dim array of data of the same *basic data types*. These basic data types include: integer, double, logical, character, complex (complex numbers), etc.

```
> a <- seq(0, 1, by = 0.1)
> a
 [1] 0.0 0.1 0.2 0.3 0.4 0.5 0.6 0.7 0.8 0.9 1.0
> seq(0, 1, len = 5)
[1] 0.00 0.25 0.50 0.75 1.00
> names(a) <- paste0("the", 1:11)
> a
 the1  the2  the3  the4  the5  the6  the7  the8  the9 the10 the11
  0.0   0.1   0.2   0.3   0.4   0.5   0.6   0.7   0.8   0.9   1.0
> typeof(a)
[1] "double"
> str(a)
 Named num [1:11] 0 0.1 0.2 0.3 0.4 0.5 0.6 0.7 0.8 0.9 ...
 - attr(*, "names")= chr [1:11] "the1" "the2" "the3" "the4" ...
> a["the3"]
the3
 0.2
> a[3]
the3
 0.2
> a[-3]
 the1  the2  the4  the5  the6  the7  the8  the9 the10 the11
  0.0   0.1   0.3   0.4   0.5   0.6   0.7   0.8   0.9   1.0
```

The components of a vector could be retrieved by number indexes, logical indexes, and names. We can even change the vector values in this way.

```
> a <- seq(0, 1, by = 0.1)
> names(a) <- paste0("the",1:11)
> a[7]
the7
 0.6
> a[1] <- 8
> a>0.8
 the1  the2  the3  the4  the5  the6  the7  the8  the9 the10 the11
 TRUE FALSE FALSE FALSE FALSE FALSE FALSE FALSE FALSE  TRUE  TRUE
> a[a>0.8] <- 12
> a
 the1  the2  the3  the4  the5  the6  the7  the8  the9 the10 the11
 12.0   0.1   0.2   0.3   0.4   0.5   0.6   0.7   0.8  12.0  12.0
> a[a>0.8] <- 1:2
Warning message:
In a[a > 0.8] <- 1:2 :
  number of items to replace is not a multiple of replacement length
> a
 the1  the2  the3  the4  the5  the6  the7  the8  the9 the10 the11
  1.0   0.1   0.2   0.3   0.4   0.5   0.6   0.7   0.8   2.0   1.0
> a[a>0.8] <- 1:5
Warning message:
In a[a > 0.8] <- 1:5 :
  number of items to replace is not a multiple of replacement length
> a
 the1  the2  the3  the4  the5  the6  the7  the8  the9 the10 the11
  1.0   0.1   0.2   0.3   0.4   0.5   0.6   0.7   0.8   2.0   3.0
```

We now try number indexes

```
> a <- 4:9
> names(a) <- paste0("ID", 1:6)
> a
ID1 ID2 ID3 ID4 ID5 ID6
  4   5   6   7   8   9
> a[-1]
ID2 ID3 ID4 ID5 ID6
  5   6   7   8   9
> a[1]
ID1
  4
> a[-(3:5)]
ID1 ID2 ID6
  4   5   9
```

We now try logical indexes

```
> a <- c("Tom", "John", "Xin", "Kosaku", "Rukawa")
> b <- c("Kosaku", "Xin", "Rukawa", "Barack", "Vladimir")
> a[a %in% b]
[1] "Xin"    "Kosaku" "Rukawa"
> a %in% b
[1] FALSE FALSE  TRUE  TRUE  TRUE
> b[b %in% a]
[1] "Kosaku" "Xin"    "Rukawa"
> b[!(b %in% a)]
[1] "Barack" "Vladimir"
> ?setdiff
> union(a,b)
[1] "Tom"    "John"   "Xin"    "Kosaku" "Rukawa" "Barack" "Vladimir"
> intersect(a,b)
[1] "Xin"    "Kosaku" "Rukawa"
```

We now try name indexes

```
> scores <- runif(5)
> names(scores) <- c("Tom", "John", "Xin", "Kosaku", "Rukawa")
> scores
       Tom       John        Xin     Kosaku     Rukawa
0.28629591 0.31875371 0.03418924 0.51401365 0.49811182
> names(scores)[scores > 0.5]
[1] "Kosaku"
> names(sort(scores, decreasing=T))
[1] "Kosaku" "Rukawa" "John"   "Tom"    "Xin"
```

Example

Use the following code to generate a score table for 100 students. Now the president plans to locate the best two students. Write code to find their names.

```
> set.seed(123)
> scores <- runif(100)
> names(scores) <- paste0("student", 1:100)
> head(scores)
 student1  student2  student3  student4  student5  student6
0.2875775 0.7883051 0.4089769 0.8830174 0.9404673 0.0455565
```

Example

Use the following code to generate a sequence $\{x_n\}_{n=1}^{100}$. Define

$$y_n = \sum_{k=1, k\neq n}^{100} x_k.$$

Write your code to find the vector $(y_n)_{n=1}^{100}$.

```
> set.seed(123)
> x <- runif(100)
> head(x)
[1] 0.2875775 0.7883051 0.4089769 0.8830174 0.9404673 0.0455565
```

Matrix indexing is similar

```
> a <- 1:5 %*% t(2:6)
> a
     [,1] [,2] [,3] [,4] [,5]
[1,]    2    3    4    5    6
[2,]    4    6    8   10   12
[3,]    6    9   12   15   18
[4,]    8   12   16   20   24
[5,]   10   15   20   25   30
> rownames(a) <- paste0("Rw", 1:5)
> colnames(a) <- paste0("Cl", 1:5)
> a
    Cl1 Cl2 Cl3 Cl4 Cl5
Rw1   2   3   4   5   6
Rw2   4   6   8  10  12
Rw3   6   9  12  15  18
Rw4   8  12  16  20  24
Rw5  10  15  20  25  30
> a[Rw3, Cl4]
Error: object 'Rw3' not found
> a["Rw3", "Cl4"]
[1] 15
> a["Rw3", 2:4]
Cl2 Cl3 Cl4
  9  12  15
```

Example

Use the following code to simulate the score table of a class. Find the average score of math, with top 2 and bottom 2 scores removed. Find the math score of the student with highest chem score.

```
> set.seed(123)
> sc <- matrix(runif(75), nrow = 25)
> rownames(sc) <- paste0("Student", 1:25)
> colnames(sc) <- c("bio", "math", "chem")
```

Reading data from files, and writing data to files

We explain the usage of the functions "`read.table`" and "`write.table`".

```
> set.seed(123)
> a <- matrix(runif(20), nrow = 5)
> a
          [,1]      [,2]      [,3]       [,4]
[1,] 0.2875775 0.0455565 0.9568333 0.89982497
[2,] 0.7883051 0.5281055 0.4533342 0.24608773
[3,] 0.4089769 0.8924190 0.6775706 0.04205953
[4,] 0.8830174 0.5514350 0.5726334 0.32792072
[5,] 0.9404673 0.4566147 0.1029247 0.95450365
> write.table(a, file = "a.txt", col.names = FALSE, row.names = FALSE, sep = ",")
> b <- read.table("a.txt", sep = ",")
> b - a
             V1            V2            V3            V4
1 -2.220446e-16  4.163336e-17 -1.110223e-16  4.440892e-16
2  3.330669e-16  2.220446e-16  4.996004e-16  8.326673e-17
3  1.110223e-16 -1.110223e-16  4.440892e-16 -3.469447e-17
4 -4.440892e-16  2.220446e-16 -2.220446e-16  3.885781e-16
5 -3.330669e-16  3.330669e-16 -2.775558e-16 -4.440892e-16
```

a.txt (~) - GVIM3

File Edit Tools Syntax Buffers Window Help

```
0.287577520124614,0.0455564993899316,0.956833345349878,0.899824970401824
0.788305135443807,0.528105488047004,0.453334156190977,0.24608773435466
0.4089769218117,0.892419044394046,0.677570635452867,0.0420595335308462
0.883017404004931,0.551435014465824,0.572633401956409,0.327920719282702
0.940467284293845,0.456614735303447,0.102924682665616,0.954503649147227
~
                                                        1,1           All
```

One may directly store R objects to files.

```
> set.seed(123)
> a <- matrix(runif(20), nrow = 5)
> save(a, file = "a.RData")
> b <- a
> rm(a)
> a
Error: object 'a' not found
> load("a.RData")
> a
          [,1]      [,2]      [,3]       [,4]
[1,] 0.2875775 0.0455565 0.9568333 0.89982497
[2,] 0.7883051 0.5281055 0.4533342 0.24608773
[3,] 0.4089769 0.8924190 0.6775706 0.04205953
[4,] 0.8830174 0.5514350 0.5726334 0.32792072
[5,] 0.9404673 0.4566147 0.1029247 0.95450365
> b - a
     [,1] [,2] [,3] [,4]
[1,]    0    0    0    0
[2,]    0    0    0    0
[3,]    0    0    0    0
[4,]    0    0    0    0
[5,]    0    0    0    0
```

Sometimes, to store a data spreadsheet that contains strings, we use the data type "`data frame`" in R. It is stored as a list equipped with the data frame structure. We will come back to the dataset "`iris`" later.

```
> 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
> is.data.frame(iris)
[1] TRUE
> as.matrix(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"
> as.matrix(head(iris)[, 1:4])
  Sepal.Length Sepal.Width Petal.Length Petal.Width
1          5.1         3.5          1.4         0.2
2          4.9         3.0          1.4         0.2
3          4.7         3.2          1.3         0.2
4          4.6         3.1          1.5         0.2
5          5.0         3.6          1.4         0.2
6          5.4         3.9          1.7         0.4
> typeof(iris)
[1] "list"
```

Functions

- A function is a fixed sequence of code that takes arguments, and returns output. The following function adds two numbers up.

```
add <- function(x, y){
  return(x + y)
}
### we call the function:
> add(2,3)
[1] 5
```

- The keyword `return` terminates the program flow and returns immediately the value of the formula. If a function has no `return` command, the value of the last formula will be returned.

```
add1 <- function(x, y){
  x + y
}
### we call the function:
> add1(2,3)
[1] 5
```

Example (Fibonacci sequence, revisited)

The Fibonacci sequence $f_1, f_2, \ldots$ is defined by $f_1 = 1$, $f_2 = 1$, and for any $k \geq 3$, $f_k = f_{k-1} + f_{k-2}$.

$$1, 1, 2, 3, 5, 8, 13, 21, 34, 55, 89, 144, 233, 377, 610, 987, 1597, \cdots$$

Make a function which takes argument k and returns f_k. Note that in the code below, we also used the `if-else` statement.

```
fibo <- function(k){
  if(k <= 2){
    return(1)
  } else {
    return(fibo(k-1) + fibo(k-2))
}}
```

```
### below are the results:
> fibo(10)
[1] 55
```

Note that the above function `fibo` calls itself! This is called recursion. On the one hand, each call of the function would generate a lot of memory overhead, the efficiency of the program is reduced; on the other hand however, this speeds up programming and saves the programmer's time.

Discrete probability distributions

We use binomial distributions as an example. Recall that for any positive integer n and real number $0 < p < 1$, a random variable X has the binomial distribution Binomial(n, p) if

$$\mathrm{Prob}(X = k) = \binom{n}{k} p^k (1-p)^{n-k}, \quad \text{for any } X = 0, 1, \ldots, n.$$

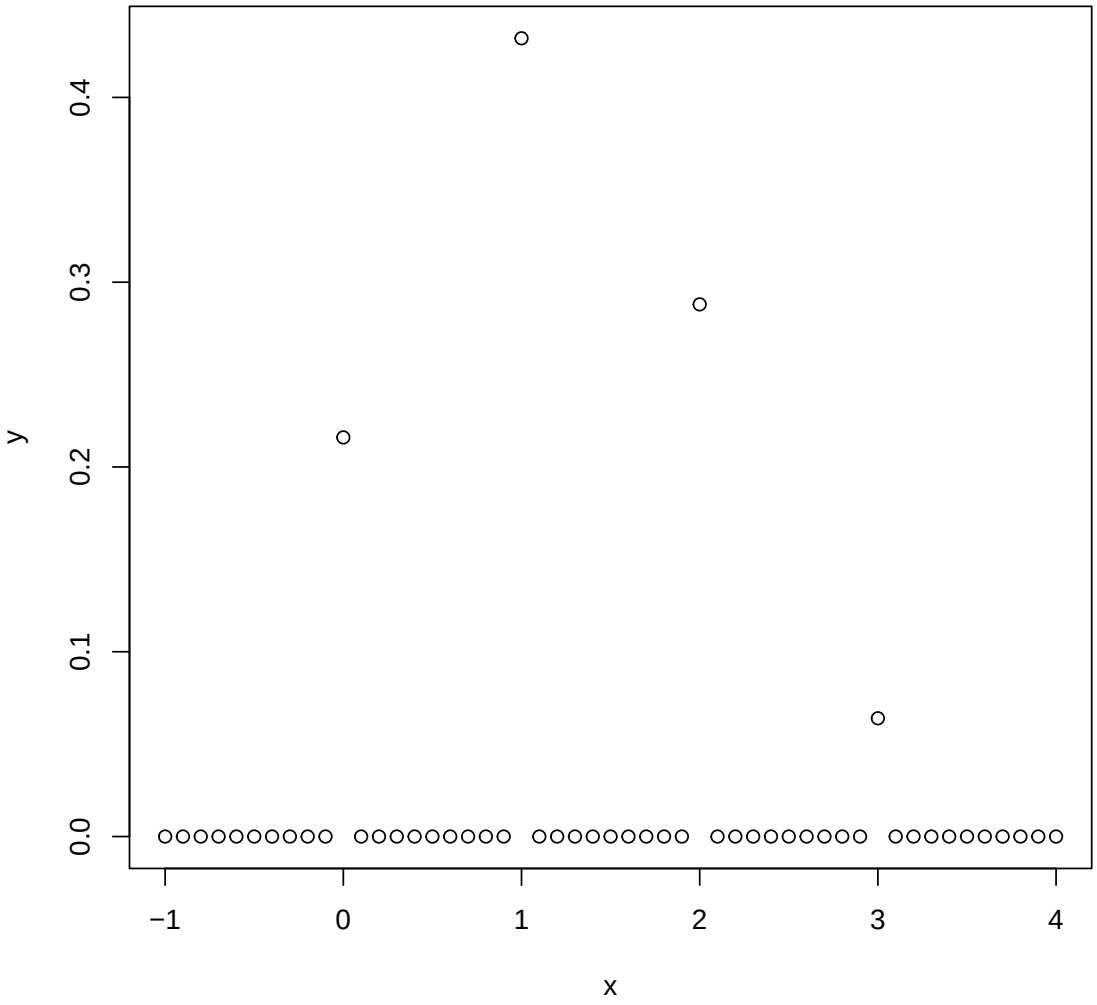
Here $\binom{n}{k} := \frac{n!}{k!(n-k)!}$. The function $f_{n,p}(k) = \binom{n}{k} p^k (1-p)^{n-k}$ of k, is called the probability function (or probability mass function, PMF). In R, $f_{n,p}$ is implemented as

$$f_{n,p}(k) = \texttt{dbinom}(\texttt{x} = k, \texttt{size} = n, \texttt{prob} = p)$$

Here, the placeholders `x`, `size`, and `prob` are called formal parameters. When $k \notin \{0, 1, \ldots, n\}$, one simply takes $f_{n,p}(k) = 0$.

```
Binomial                  package:stats                 R Documentation

The Binomial Distribution

Description:

     Density, distribution function, quantile function and random
     generation for the binomial distribution with parameters ‘size’
     and ‘prob’.

     This is conventionally interpreted as the number of ‘successes’ in
     ‘size’ trials.

Usage:

     dbinom(x, size, prob, log = FALSE)
     pbinom(q, size, prob, lower.tail = TRUE, log.p = FALSE)
     qbinom(p, size, prob, lower.tail = TRUE, log.p = FALSE)
     rbinom(n, size, prob)

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.

    size: number of trials (zero or more).

    prob: probability of success on each trial.
...
```

```r
x <- seq(-1, 4, by = 0.1)
y <- dbinom(x = x, size = 3, prob = 0.4)
plot(x, y)
```



The distribution function (also called the cumulative distribution function, CDF) is defined for all $x \in \mathbb{R}$,

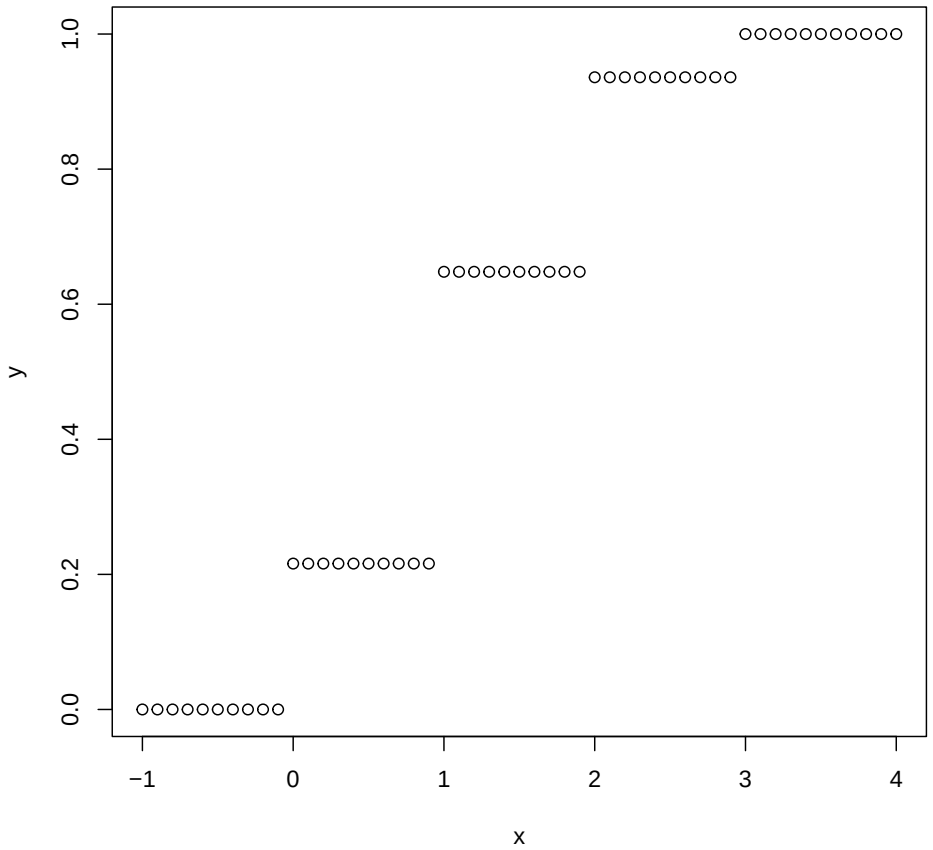
$$F_{n,p}(x) = \sum_{k \leq x} f_{n,p}(k).$$

Therefore CDF F for any probability distribution is non-decreasing, and

$$\lim_{x \to \infty} F(x) = 1$$
$$\lim_{x \to -\infty} F(x) = 0$$

The quantile $Q_{n,p}(\xi)$ for $0 < \xi < 1$ is defined as the smallest value $Q = Q_{n,p}(\xi)$ such that $F_{n,p}(Q) \geq \xi$. Here we skip the discussion.

```r
x <- seq(-1, 4, by = 0.1)
y <- pbinom(q = x, size = 3, prob = 0.4)
plot(x, y)
```



The function `rbinom` generates random numbers from a specified binomial distribution.

```
> set.seed(123)
> rbinom(n = 10, size = 3, prob = 0.4)
 [1] 1 2 1 2 3 0 1 2 1 1
> x <- rbinom(n = 100000, size = 3, prob = 0.4)
> table(x)
x
    0     1     2     3
21659 43256 28703  6382
> table(x) / length(x)
x
      0       1       2       3
0.21659 0.43256 0.28703 0.06382
> dbinom(x = 0:3, size = 3, prob = 0.4)
[1] 0.216 0.432 0.288 0.064
```

Continuous probability distributions

We use normal distributions as an example. Recall that for any real number $\mu \in \mathbb{R}$, and any positive variance $\sigma^2 > 0$, a random variable X has the normal distribution $N(\mu, \sigma^2)$ if

$$\mathrm{Prob}(X \leq x) = \int_{-\infty}^{x} \frac{1}{\sqrt{2\pi\sigma^2}} \exp\left\{-\frac{(t-\mu)^2}{2\sigma^2}\right\} dt.$$

The density function for $-\infty \leq x \leq \infty$

$$f_{\mu,\sigma^2}(x) = \frac{1}{\sqrt{2\pi\sigma^2}} \exp\left\{-\frac{(x-\mu)^2}{2\sigma^2}\right\}$$

is implemented in R as

$$f_{\mu,\sigma^2}(x) = \texttt{dnorm}(\texttt{x} = x, \texttt{mean} = \mu, \texttt{sd} = \sigma)$$

The CDF for $-\infty \leq x \leq \infty$

$$F_{\mu,\sigma^2}(x) = \int_{-\infty}^{x} f_{\mu,\sigma^2}(t)dt = \int_{-\infty}^{x} \frac{1}{\sqrt{2\pi\sigma^2}} \exp\left\{-\frac{(t-\mu)^2}{2\sigma^2}\right\} dt$$

is implemented in R as

$$F_{\mu,\sigma^2}(x) = \texttt{pnorm}(\texttt{q} = x, \texttt{mean} = \mu, \texttt{sd} = \sigma)$$

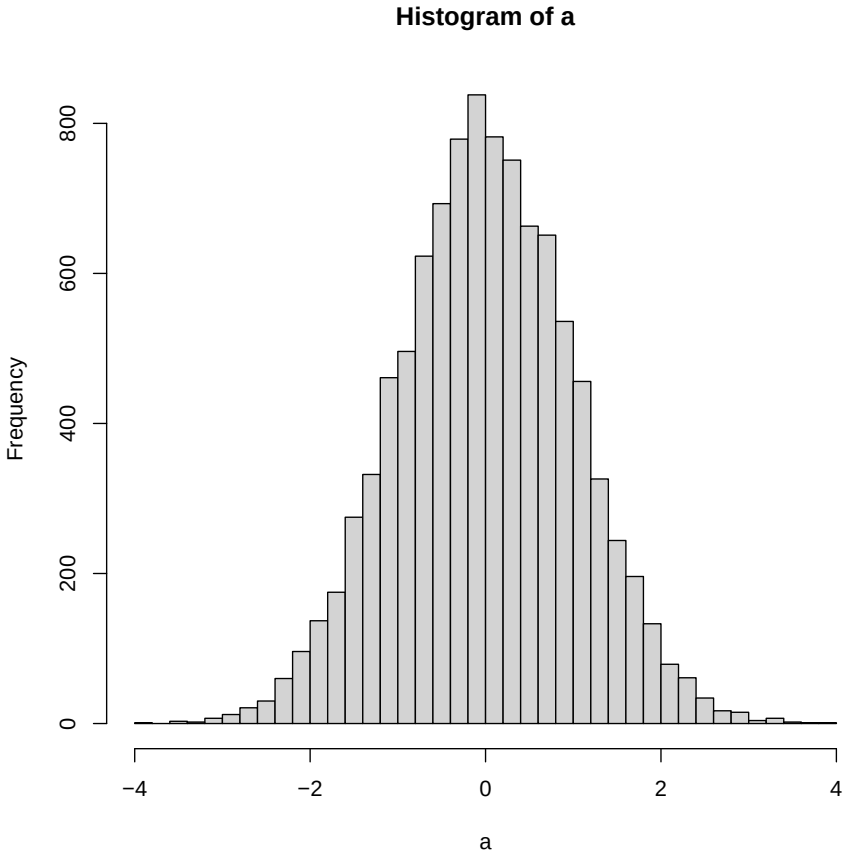
The quantile $Q_{\mu,\sigma^2}(x) = F^{-1}_{\mu,\sigma^2}(x)$ for $0 \leq x \leq 1$ is the inverse function of F_{μ,σ^2}, and is implemented in R as

$$Q_{\mu,\sigma^2}(x) = \texttt{qnorm}(\texttt{p} = x, \texttt{mean} = \mu, \texttt{sd} = \sigma)$$

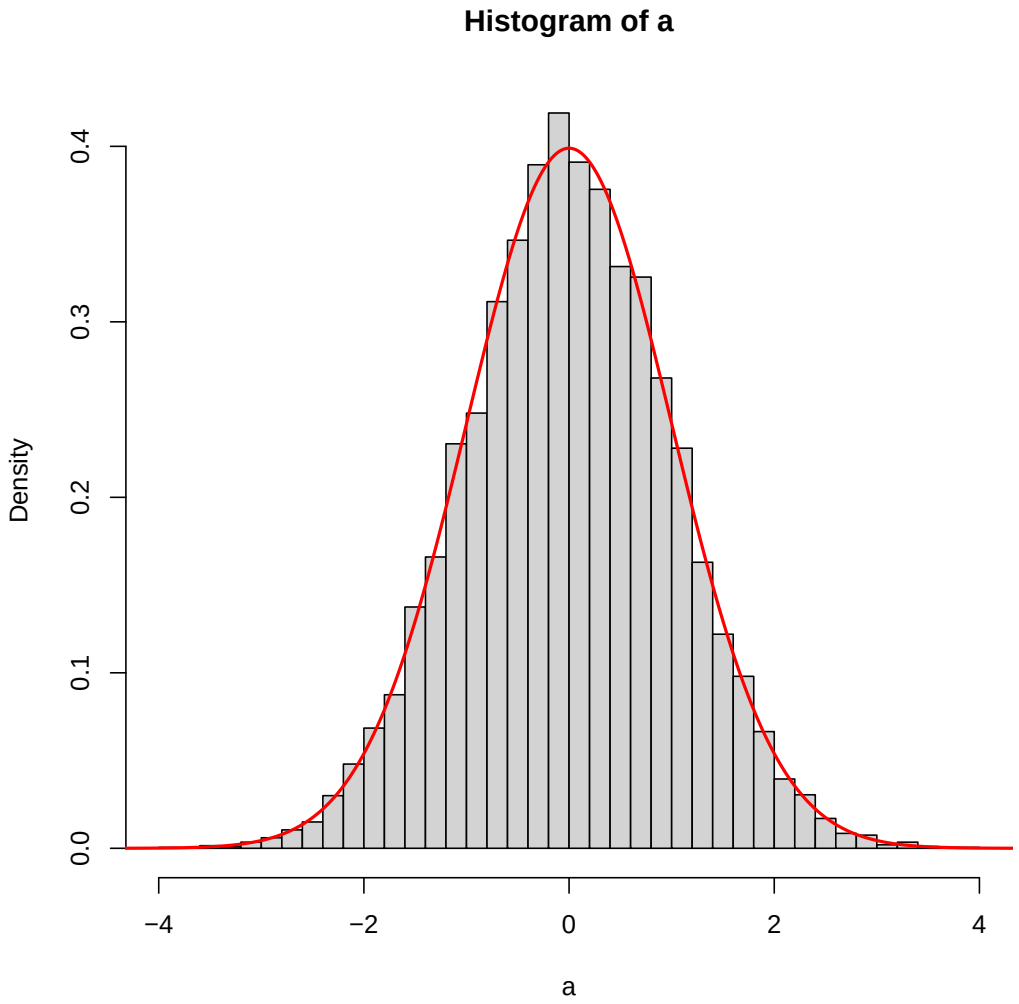
For a comprehensive list of probability distributions (discrete and continuous) implemented in R, see `https://CRAN.R-project.org/view=Distributions`.

The function `rnorm` generates random numbers from a specified normal distribution.

```
> set.seed(123)
> a <- rnorm(n = 10000) # N(0,1) by default
> head(a) # get the first several (default=6) elements
[1] -0.56047565 -0.23017749  1.55870831  0.07050839  0.12928774  1.71506499
> tail(a) # get the last several (default=6) elements
[1] -0.5896389 -1.0247840 -0.3671164 -0.7574729  1.0793289 -0.6449242
> hist(a, breaks = 30)
```



```r
set.seed(123)
a <- rnorm(n = 10000) # N(0,1) by default
hist(a, breaks = 30, freq = FALSE)
x <- seq(-5, 5, length = 500)
lines(x, dnorm(x), lwd = 2, col = "red")
```



Example (exercise)

In theory, $Q_{\mu,\sigma^2}(F_{\mu,\sigma^2}(x)) \equiv x$ for any real number x. However, the output of the R command

```
qnorm(p = pnorm(q = 10))
```

is `Inf` (infinity, ∞). Please try the command yourself, explain the output, and find a fix.

Plotting

`plot` is a generic function to plot variables. The default use is `plot(x, y)` which plots a vector `y` against another vector `x` provided they have the same length.

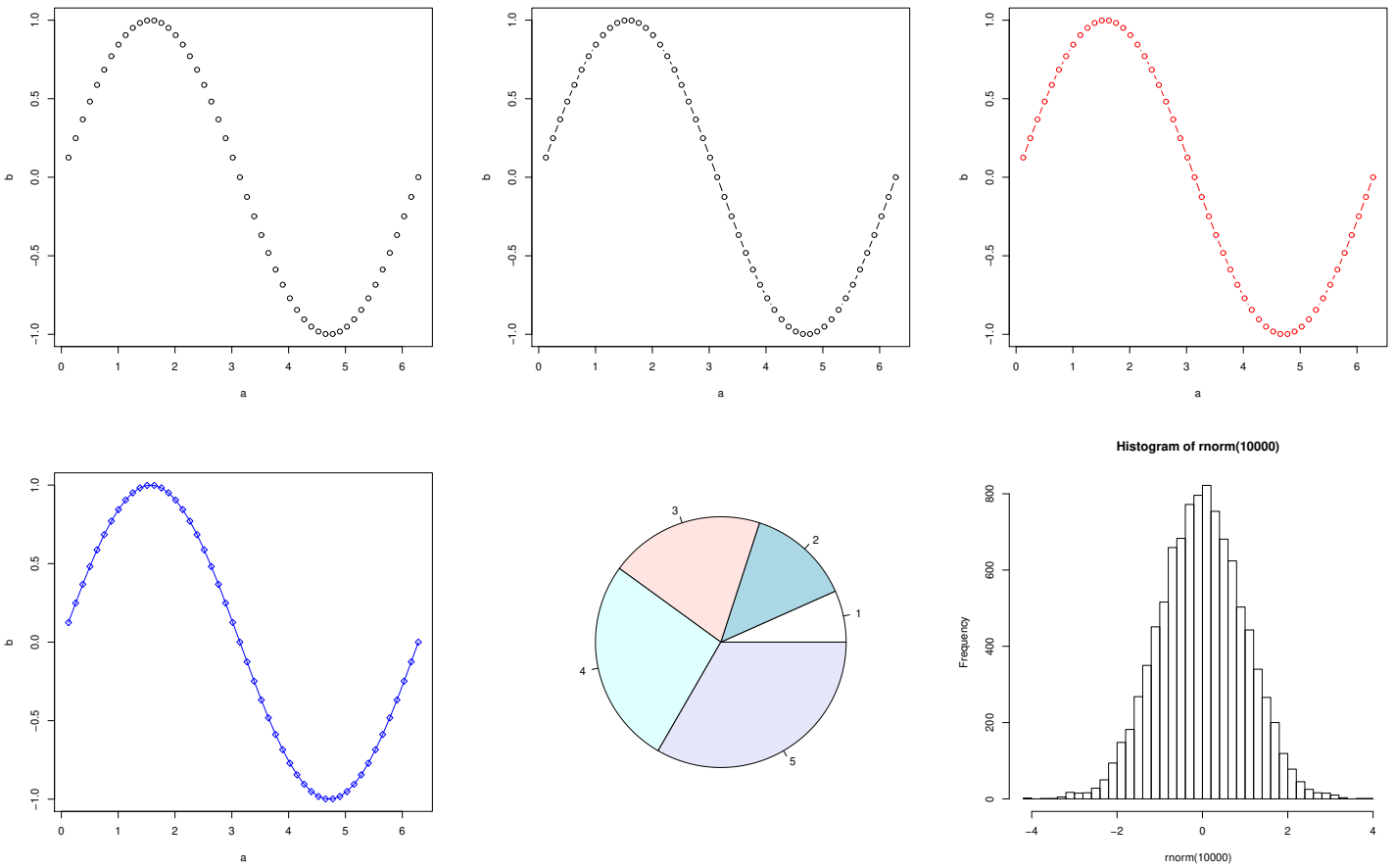


Figure: Some sample plots.

```
> a <- (1:50) / 50 * 2 * pi
> b <- sin(a)
> plot(a,b)
> plot(a,b,type="b")
> plot(a,b,type="b",col="red")
> plot(a,b,type="o",col="blue",pch=23)
> pie(1:5)
> hist(rnorm(10000),breaks=30)
```

For more options, see `?plot` and `?plot.default`. One may save the plot into files.

```
> setEPS()
> postscript("a.eps")
> pie(1:5)
> dev.off()
null device
          1
> bmp("a.bmp")
> pie(1:6)
> dev.off()
null device
          1
```

Simple Optimization

R has a simple implementation of optimization function.

Example

Using R to find the minimizer of the following function

$$f(x_1, x_2, x_3) = x_1^2 + (x_2 - 2)^2 + (x_3 + 2)^2$$

```
f <- function(x) x[1]^2 + (x[2] - 2)^2 + (x[3] + 2)^2
```

Note

- If the body of a function has only one sentence, the curly bracket {} can be ignored.
- Obviously the minimizer is $(0, 2, -2)$. Knowing the true result helps us to verify the program output.

```
optim(c(1,1,1), f, control = list(reltol = 1e-8))
### below are the results:
$par
[1]  4.366903e-07  2.000140e+00 -2.000049e+00

$value
[1] 2.19478e-08

$counts
function gradient
     186       NA

$convergence
[1] 0

$message
NULL
```

Note

- The output item `par` is the minimizer obtained of the function.
- The output item `value` is the minimum value of f.
- The output item `counts` gives the time that f is called. Roughly speaking, the more times the function is called, the longer the program takes to run. Sine we have never provided the gradient information, we have not called the gradient function.
- The output item `convergence` has value 0 if the function has confidence that a local minimizer is achieved. The value is not 0 if some potential problems are detected.

Recall

$$f(x_1, x_2, x_3) = x_1^2 + (x_2 - 2)^2 + (x_3 + 2)^2.$$

We feed the `optim` function below, also with the gradient

$$\nabla f(x_1, x_2, x_3) = \begin{pmatrix} \partial f / \partial x_1 \\ \partial f / \partial x_2 \\ \partial f / \partial x_3 \end{pmatrix} = \begin{pmatrix} 2x_1 \\ 2(x_2 - 2) \\ 2(x_3 + 2) \end{pmatrix}.$$

We define

```
gra.f <- function(x){
  2 * c(x[1], x[2] - 2, x[3] + 2)
}
```

The updated optimization code is

```
optim(c(1,1,1), f, gr = gra.f, method = "CG", control = list(reltol = 1e-8))
### below are the results:
$par
[1]  1.275786e-07  2.000000e+00 -2.000000e+00

$value
[1] 1.790392e-13

$counts
function gradient
      15        7
$convergence
[1] 0
$message
NULL
```

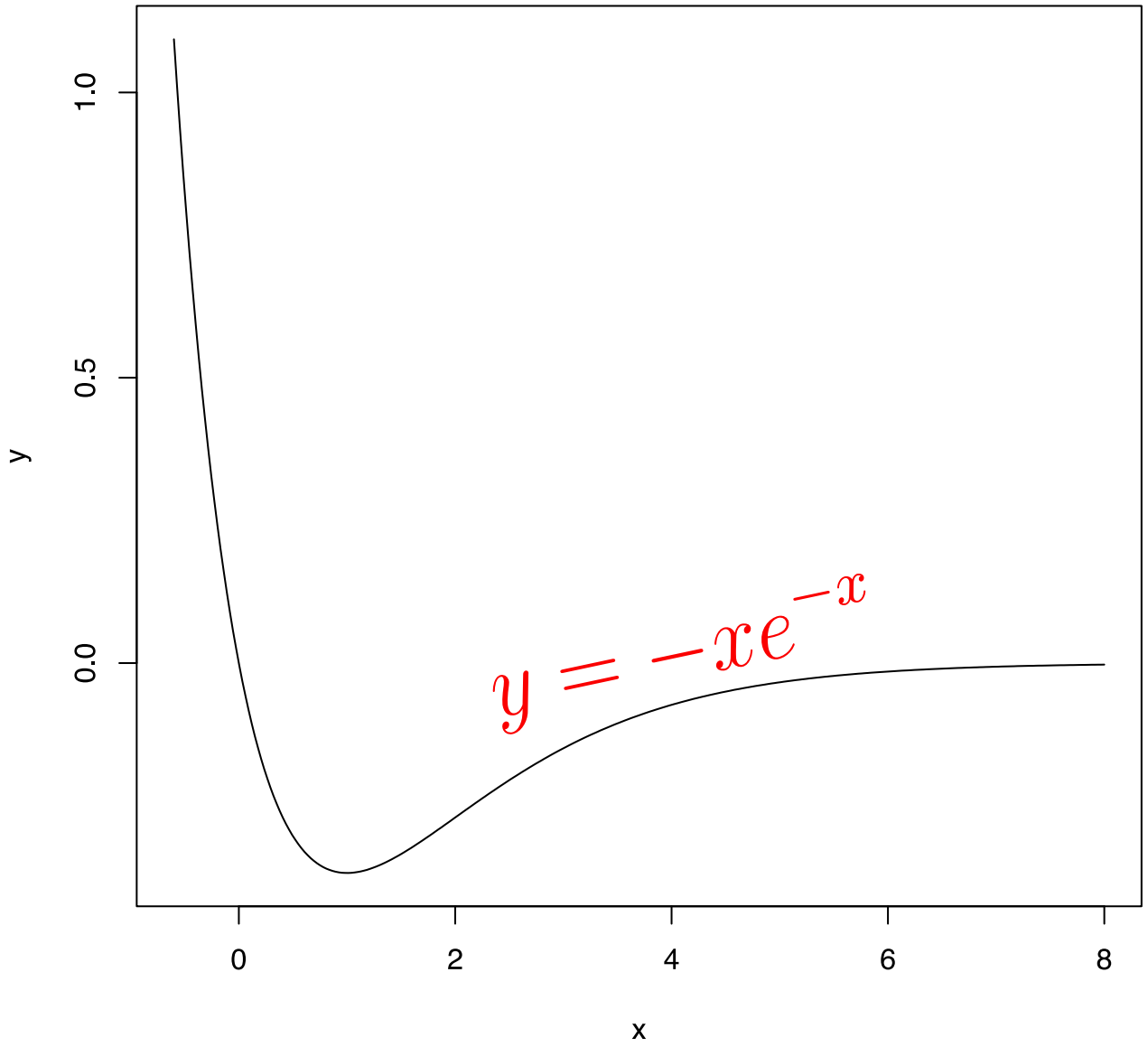
We see that although we call the gradient function 7 times, the times we call the function f is reduced, from 186, to 15. More importantly, the precision is much better than the previous run.

Example

Use R to find the minimizer of the function

$$f(x) = -xe^{-x}.$$

Note that you may use calculus to verify that the minimizer is $x = 1$ where $f(1) = -e^{-1} = -0.3679$.


1.0
0.5
0.0
y
0
2
4
6
8
x
$y = -xe^{-x}$

Example (exercise, Maximum likelihood estimation, MLE)

Let $x_1, \ldots, x_n$ be drawn independently from Poisson(λ). If λ is unknown, one may estimate λ by maximizing the log-likelihood function

$$\ell(\lambda) = \log \prod_{i=1}^{n} e^{-\lambda} \frac{\lambda^{x_i}}{x_i!} = \sum_{i=1}^{n} \{-\lambda + x_i \log \lambda - \log(x_i!)\}$$

Set λ and n, and generate data yourself. Then write a function `mle` to minimize ℓ by the above R function `optim`. Compare your solution with the analytical minimizer $\hat{\lambda} = \frac{1}{n} \sum x_i$.

Integration, 1-dim

Note that for integration with more than 1 dimension, methods like Monte-Carlo may be better.

Example

Use R to find the value of the integral

$$\int_{-\infty}^{\infty} \exp(-x^2)dx.$$

Note that the true value of the integral is

$$\int_{-\infty}^{\infty} \exp(-x^2)dx = 2\int_{0}^{\infty} \exp(-x^2)dx$$
$$\overset{y=x^2}{=} \int_{0}^{\infty} y^{\frac{1}{2}-1}e^{-y}dy = \Gamma\left(\frac{1}{2}\right) = \sqrt{\pi} = 1.77245385.$$

```
integrate(function(x) exp(-x^2), lower = -Inf, upper = Inf)
### below are the results:
1.772454 with absolute error < 4.3e-06
```

Note

- We see that the numerical integration is already very precise. We skip the algorithm behind.
- In R, one uses `Inf` to denote infinity ∞.
- We see that in the above command we have used a function

  ```
  function(x) exp(-x^2)
  ```

  which is passed to the function `integrate` immediately after it is generated. We did not even give it a name! This kind of function is called anonymous function.

Example

Use R to evaluate the following function

$$\Phi(x) := \frac{1}{\sqrt{2\pi}} \int_{-\infty}^{x} \exp\left\{-\frac{t^2}{2}\right\} dt.$$

Note that this function is the cumulative distribution function of the standard normal distribution $N(0,1)$. R has a high-performance implementation of this function: `pnorm`. After writing your function, you may compare it with `pnorm`.

Solving Nonlinear Equations

Numerically, R implements some algorithms to solve nonlinear equations, e.g. Newton's method. It is not therefore guaranteed that all the roots can be found.

Example

Find the solution of the equation $f(x) = 0$ with

$$f(x) = e^x + x.$$

```
install.packages("nleqslv") # install the package, just for the first time
library("nleqslv")
f <- function(x) exp(x) + x
nleqslv(1, f) # 1: initial guess
```

```
### below are the results:
$x
[1] -0.5671433

$fvec
[1] 8.346891e-10

$termcd
[1] 1

$message
[1] "Function criterion near zero"

$scalex
[1] 1

$nfcnt
[1] 6

$njcnt
[1] 1

$iter
[1] 6
```

So the solution is $x = -0.5671433$.

Example (exercise)

Find the solution to the equation $f(x) = 0$ with

$$f(x) = x^3 - x^2 + x - 1.$$

In R, there is a function `polyroot` specially designed to find the roots of polynomials. Let $a = (a_1, a_2, \ldots, a_{n+1}) \in \mathbb{R}^{n+1}$ be the coefficient vector for the polynomial

$$p(x) = a_1 + a_2 x + a_3 x^2 + \cdots + a_{n+1} x^n,$$

then we may use the command `polyroot(a)` to find ALL the roots of p. For the above example,

```
> polyroot(c(-1,1,-1,1))
[1] 0+1i 0-1i 1+0i
```

We see that some complex roots are also found.