Introduction to R

ECON 515 Fall 2024

Zhan Gao

Introduction

What is R?

To quote the R project website:

> R is a free software environment for statistical computing and graphics. It compiles and runs on a wide variety of UNIX platforms, Windows and MacOS.

What does that mean?

- R was created for the statistical and graphical work required by econometrics.
- R has a vibrant, thriving online community. (stack overflow)
- Plus it's **free** and **open source**.

Why are we using R?

- Many alternatives:
  - STATA v.s.R v.s. Python v.s MATLAB v.s. Julia v.s. C/C++
  - No clear answer...
- Advantages of R
  - R is **free** and **open source** (v.s. STATA and MATLAB)
  - Designed for data work. Favored by the academia, particular the statistics and econometrics community, which means most of the times you can apply most up-to-date methods with packages developed by the authors. (v.s. all others)
    - Example: C-Lasso, Bubble test
  - R is very **flexible and powerful** — adaptable to nearly any task, *e.g.*, 'metrics, spatial data analysis, machine learning, web scraping, data cleaning, website building, teaching.
    - Well-developed environment. Rich sources of packages. (v.s. Julia.)
  - R imposes **no limitations** on your amount of observations, variables, memory, or processing power. (v.s. Stata)
  - If you put in the work, you will come away with a **valuable and marketable** tool.
  - https://github.com/matloff/R-vs.-Python-for-Data-Science

Installation

1. Download and install R.

- It comes with a GUI, which opens when you run the R application (e.g. from the applications folder on Mac). You can also run it from the command line.

2. Download and install RStudio.

- A powerful integrated development environment (IDE) can be very helpful.

3. *Download Git. (Optional but recommended.)*

- Version control is important in practice, particularly when many people work on the same project.

4. *Create an account on GitHub and register for a student/educator discount. (Optional but recommended.)*

- For installation guidance and troubleshooting: Jenny Bryan's http://happygitwithr.com.

Some OS-specific extras

To help smooth some software installation issues further down the road, please also do the following (depending on your OS):

- **Windows:** Install Rtools.
- **Mac:** Install Homebrew. I also recommend that you configure/open your C++ toolchain (see here.)
- **Linux:** None (you should be good to go).

Checklist

[check] Do you have the most recent version of R?

```
version$version.string
```

```
## [1] "R version 4.3.3 (2024-02-29)"
```

[check] Do you have the most recent version of RStudio? (The preview version is fine.)

```
RStudio.Version()$version
## Requires an interactive session but should return something like "[1] '1.2.5001'"
```

[check] Have you updated all of your R packages?

```
update.packages(ask = FALSE, checkBuilt = TRUE)
```

Additional R Resources

- Official Manual
  - R-Introduction

- Course materials
  - EC 510 and EC 607 taught by Prof. Grant McDermott at the University of Oregon.
  - ECON 5170 taught by Prof. Zhentao Shi at the Chinese University of Hong Kong.
  - Tibshirani, R. Statistics 36-350 taught by Prof. Ryan Tibshirani
  - Coursera/Udacity/Udemy/YouTube/Bilibili...
- Books
  - Grolemund G. Hands-On Programming with R
  - Wickham, H. and Grolemund, G. R for data science
  - Wickham, H Advanced R
  - Adams. C. P. Learning Microeconometrics with R
- Stack Overflow/Google oriented programming

Basic arithmetic

R is a powerful calculator and recognizes all of the standard arithmetic operators:

```
1+2 ## Addition
```

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

```
6-7 ## Subtraction
```

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

```
5/2 ## Division
```

```
## [1] 2.5
```

```
2^3 ## Exponentiation
```

```
## [1] 8
```

We can also invoke modulo operators (integer division & remainder). - Very useful when dealing with time, for example.

```
100 %/% 60 ## How many whole hours in 100 minutes?
```

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

```
100 %% 60 ## How many minutes are left over?
```

```
## [1] 40
```

Logical operators

R also comes equipped with a full set of logical operators (and Boolean functions), which follow standard programming protocol. For example:

```
1 > 2
```

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

```
1 == 2
```

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

```
1 > 2 | 0.5 ## The "|" stands for "or" (not a pipe a la the shell)
```

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

```
1 > 2 & 0.5 ## The "&" stands for "and"
```

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

```
isTRUE (1 < 2)
```

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

```
4 %in% 1:10
```

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

```
is.na(1:10)
```

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

```
# etc..
```

You can read more about these logical operators here and here.

Assignment

In R, we can use either `=` or `<-` to handle assignment. The `<-` is really a `<` followed by a `-`.

Assignment with <-

`<-` is normally read aloud as "gets". You can think of it as a (left-facing) arrow saying *assign in this direction*.

```
a <- 10 + 5
a
```

```
## [1] 15
```

f course, an arrow can point in the other direction too (i.e. `->`). So, the following code chunk is equivalent to the previous one, although used much less frequently.

```
10 + 5 -> a
```

Assignment with =

You can also use `=` for assignment.

```r
b = 10 + 10 ## Note that the assigned object *must* be on the left with "=".
b
```

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

More discussion about <- vs =: https://github.com/Robinlovelace/geocompr/issues/319#issuecomment-427376764 and https://www.separatinghyperplanes.com/2018/02/why-you-should-use-and-never.html.

Objects

1. Everything is an object.
2. Everything has a name.

Different *types* (or *classes*) of objects.

Vectors

The c() function creates vectors. This is one of the objects we'll use to store data.

```r
myvec <- c(1, 2, 3)
print(myvec)
```

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

Shortcut for consecutive numbers:

```r
myvec <- 1:3
print(myvec)
```

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

Basic algebraic operations all work entrywise on vectors.

```r
myvec <- c(1, 3, 7)
myvec2 <- c(5, 14, 3)
myvec3 <- c(9, 4, 8)

myvec + myvec2
```

```
## [1]  6 17 10
```

```r
myvec / myvec2
```

```
## [1] 0.2000000 0.2142857 2.3333333
```

```r
myvec * (myvec2^2 + sqrt(myvec3))
```

```
## [1]  28.00000 594.00000  82.79899
```

So are the binary logical operations `& | !=`.

```
# logical vectors
logi_1 = c(T,T,F)
logi_2 = c(F,T,T)

logi_12 = logi_1 & logi_2
print(logi_12)
```

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

You can "slice" a vector (grab subsets of it) in several different ways: Vector selection is specified in square bracket `a[ ]` by either positive integer or logical vector.

```
myvec <- 7:20
myvec[8]
```

```
## [1] 14
```

```
myvec[2:5]
```

```
## [1]  8  9 10 11
```

```
# vector of booleans for whether each entry of myvec is greater than 13
indvec <- myvec > 13
indvec
```

```
##  [1] FALSE FALSE FALSE FALSE FALSE FALSE FALSE  TRUE  TRUE  TRUE  TRUE  TRUE
## [13]  TRUE  TRUE
```

```
indvec2 <- myvec == 8
indvec2
```

```
##  [1] FALSE  TRUE FALSE FALSE FALSE FALSE FALSE FALSE FALSE FALSE FALSE FALSE
## [13] FALSE FALSE
```

```
# only outputs entries of myvec for which indvec is true
myvec[indvec]
```

```
## [1] 14 15 16 17 18 19 20
```

```
# same thing but all in one line, without having to define indvec
myvec[myvec>13]
```

```
## [1] 14 15 16 17 18 19 20
```

Some useful vector functions:

```
length(myvec)
```

```
## [1] 14
```

```r
mean(myvec)
```

```
## [1] 13.5
```

```r
var(myvec)
```

```
## [1] 17.5
```

Matrices

Matrices are just collections of several vectors of the same length.

```r
myvec <- c(1, 3, 7)
myvec2 <- c(5, 14, 3)
myvec3 <- c(9, 4, 8)
# creates matrix whose columns are the inputs of myvec
mat_1 <- cbind(myvec, myvec2, myvec3)
print(mat_1)
```

```
##      myvec myvec2 myvec3
## [1,]     1      5      9
## [2,]     3     14      4
## [3,]     7      3      8
```

```r
# now they're rows instead
mat_2 <- rbind(myvec, myvec2, myvec3)
print(mat_2)
```

```
##        [,1] [,2] [,3]
## myvec     1    3    7
## myvec2    5   14    3
## myvec3    9    4    8
```

```r
# Define a matrix by its element
mat_3 <- matrix(1:8, 2, 4)
print(mat_3)
```

```
##      [,1] [,2] [,3] [,4]
## [1,]    1    3    5    7
## [2,]    2    4    6    8
```

```r
mat_4 <- matrix(1:8, 2, 4, byrow = TRUE)
print(mat_4)
```

```
##      [,1] [,2] [,3] [,4]
## [1,]    1    2    3    4
## [2,]    5    6    7    8
```

Matrix algebra works the same way as vector algebra - it's all done entrywise with the * and + operators. If you want to do matrix multiplication, use %*%

```
dim(mat_1)
```

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

```
mat_1 * mat_2
```

```
##      myvec myvec2 myvec3
## [1,]     1     15     63
## [2,]    15    196     12
## [3,]    63     12     64
```

```
mat_1 %*% mat_2 #Note that this differs from the elementwise product
```

```
##      [,1] [,2] [,3]
## [1,]  107  109   94
## [2,]  109  221   95
## [3,]   94   95  122
```

Other often used matrix operations:

```
t(mat_1) # Transpose
```

```
##        [,1] [,2] [,3]
## myvec     1    3    7
## myvec2    5   14    3
## myvec3    9    4    8
```

```
solve(mat_1) # Inverse
```

```
##                [,1]        [,2]         [,3]
## myvec  -0.146842878  0.01908957  0.155653451
## myvec2 -0.005873715  0.08076358 -0.033773862
## myvec3  0.130690162 -0.04698972  0.001468429
```

```
eigen(mat_1) # eigenvalues and eigenvectors
```

```
## eigen() decomposition
## $values
## [1] 18.699429  8.556685 -4.256114
##
## $vectors
##            [,1]       [,2]        [,3]
## [1,] -0.4631214  0.3400869  0.87156297
## [2,] -0.7270618 -0.6713411 -0.03609066
## [3,] -0.5068528  0.6585150 -0.48895343
```

For more operations, check out https://www.statmethods.net/advstats/matrix.html.

"Slicing" matrices:

```
mat_1[1, 1]
```

```
## myvec
##     1
```

```
mat_1[2, ]
```

```
##  myvec myvec2 myvec3
##      3     14      4
```

Data frames

data.frame is a two-dimensional table that stores the data, similar to a spreadsheet in Excel. A matrix is also a two-dimensional table, but it only accommodates one type of elements. Real world data can be a collection of integers, real numbers, characters, categorical numbers and so on. Data frame is the best way to organize data of mixed type in R.

```
df_1 <- data.frame(a = 1:2, b = 3:4)
print(df_1)
```

```
##   a b
## 1 1 3
## 2 2 4
```

```
df_2 <- data.frame(name = c("Jack", "Rose"), score = c(100, 90))
print(df_2)
```

```
##   name score
## 1 Jack   100
## 2 Rose    90
```

```
print(df_2[, 1])
```

```
## [1] "Jack" "Rose"
```

```
print(df_2$name)
```

```
## [1] "Jack" "Rose"
```

Lists

A vector only contains one type of elements. *list* is a basket for objects of various types. It can serve as a container when a procedure returns more than one useful object. For example, recall that when we invoke `eigen`, we are interested in both eigenvalues and eigenvectors, which are stored into `$value` and `$vector`, respectively.

```
x_list <- list(a = 1:2, b = "hello world")
print(x_list)
```

```
## $a
## [1] 1 2
##
## $b
## [1] "hello world"
```

```
print(x_list[[1]]) # Different from vectors and matrices
```

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

```
print(x_list$a)
```

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

Functions

You do things using functions. Functions come pre-written in packages (i.e. "libraries"), although you can — and should — write your own functions too. - In the developing stage, it allows us to focus on a small chunk of code. It cuts an overwhelmingly big project into manageable pieces. - A long script can have hundreds or thousands of variables. Variables defined inside a function are local. They will not be mixed up with those outside of a function. Only the input and the output of a function have interaction with the outside world. - If a revision is necessary, We just need to change one place. We don't have to repeat the work in every place where it is invoked.

```
# Built-in function
sum(c(3, 4))
```

```
## [1] 7
```

```
# User-defined function
add_func <- function(x, y){
    return(x + y)
}
add_func(3, 4)
```

```
## [1] 7
```

Package

A pure clean installation of R is small, but R has an extensive ecosystem of add-on packages. This is the unique treasure for R users. Most packages are hosted on CRAN. A common practice today is that statisticians upload a package to CRAN after they write or publish a paper with a new statistical method. They promote their work via CRAN, and users have easy access to the state-of-the-art methods.

A package can be installed by `install.packages("package_name")` and invoked by `library(package_name)`.

```r
# Function from a package
stats::sd(1:10)
```

```
## [1] 3.02765
```

```r
# Package isntall
# install.packages("glmnet")
library(glmnet)
```

It is also common for authors to make packages available on GitHub or on their websites. You can often use the devtools package or the remotes packages to install these, following instructions on the project website.

```r
if (!requireNamespace("remotes")) {
  install.packages("remotes")
}
```

```
## Loading required namespace: remotes
```

```r
remotes::install_github("kolesarm/RDHonest")
```

```
## Using GitHub PAT from the git credential store.
```

```
## Skipping install of 'RDHonest' from a github remote, the SHA1 (7391e914) has not chan
##   Use `force = TRUE` to force installation
```

Help System

The help system is the first thing we must learn for a new language. In R, if we know the exact name of a function and want to check its usage, we can either call `help(function_name)` or a single question mark `?function_name`. If we do not know the exact function name, we can instead use the double question mark `??key_words`. It will provide a list of related function names from a fuzzy search.

Example: `?seq`, `??sequence`

For many packages, you can also try the `vignette()` function, which will provide an introduction to a package and it's purpose through a series of helpful examples.

Example: `vignette("dplyr")`

Example: OLS estimation

OLS estimation with one x regressor and a constant. Graduate textbook expresses the OLS in matrix form

$$\hat{\beta} = (X'X)^{-1}X'y.$$

To conduct OLS estimation in R, we literally translate the mathematical expression into code.

Step 1: We need data Y and X to run OLS. We simulate an artificial dataset.

```r
# simulate data
set.seed(111) # can be removed to allow the result to change

# set the parameters
n <- 100
b0 <- matrix(1, nrow = 2 )

# generate the data
e <- rnorm(n)
X <- cbind( 1, rnorm(n) )
Y <- X %*% b0 + e
```

Step 2: translate the formula to code

```r
# OLS estimation
(bhat <- solve( t(X) %*% X, t(X) %*% Y ))
```

```
##           [,1]
## [1,] 0.9861773
## [2,] 0.9404956
```

```r
class(bhat)
```

```
## [1] "matrix" "array"
```

```r
# User-defined function
ols_est <- function(X, Y) {
    bhat <- solve( t(X) %*% X, t(X) %*% Y )
    return(bhat)
}
(bhat_2 <- ols_est(X, Y))
```

```
##           [,1]
## [1,] 0.9861773
## [2,] 0.9404956
```

```r
class(bhat_2)
```

```
## [1] "matrix" "array"
```

```r
# Use built-in functions
(bhat_3 <- lsfit(X, Y, intercept = FALSE)$coefficients)
```

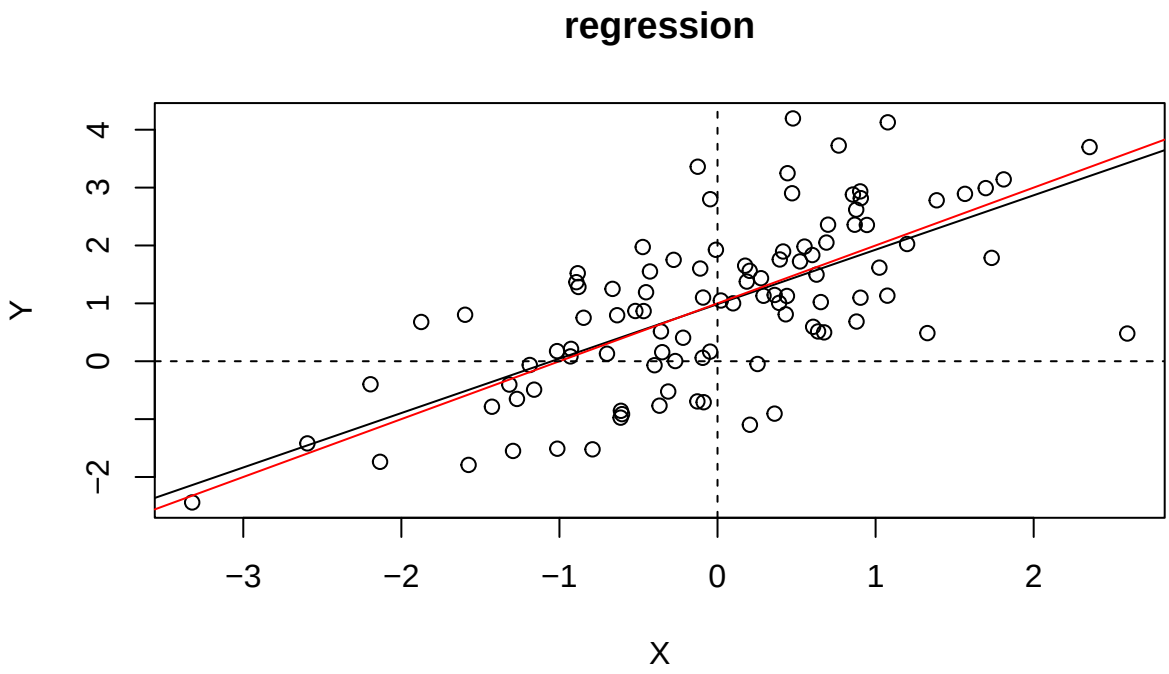
```
##        X1        X2
## 0.9861773 0.9404956
```

```r
class(bhat_3)
```

```
## [1] "numeric"
```

Step 3 (additional): plot the regression graph with the scatter points and the regression line. Further compare the regression line (black) with the true coefficient line (red).

```
# plot
plot(y = Y, x = X[,2], xlab = "X", ylab = "Y", main = "regression")
abline(a = bhat[1], b = bhat[2])
abline(a = b0[1], b = b0[2], col = "red")
abline(h = 0, lty = 2)
abline(v = 0, lty = 2)
```



Step 4: In econometrics we are often interested in hypothesis testing. The t-statistic is widely used. To test the null $H_0 : \beta_2 = 1$, we compute the associated t-statistic. Again, this is a translation.

$$t = \frac{\hat{\beta}_2 - \beta_{02}}{\hat{\sigma}_{\hat{\beta}_2}} = \frac{\hat{\beta}_2 - \beta_{02}}{\sqrt{[(X'X)^{-1}\hat{\sigma}^2]_{22}}}.$$

where $[\cdot]_{22}$ is the (2,2)-element of a matrix.

```
# calculate the t-value
bhat2 = bhat[2] # the parameter we want to test
e_hat = Y - X %*% bhat
sigma_hat_square = sum(e_hat^2)/ (n-2)
Sigma_B = solve( t(X) %*% X  ) * sigma_hat_square
t_value_2 = ( bhat2 - b0[2]) / sqrt( Sigma_B[2,2] )
print(t_value_2)
```

```
## [1] -0.5615293
```

Exercise: Can you write a function with both $\hat{\beta}$ and t-value as outputs?

Input and Output

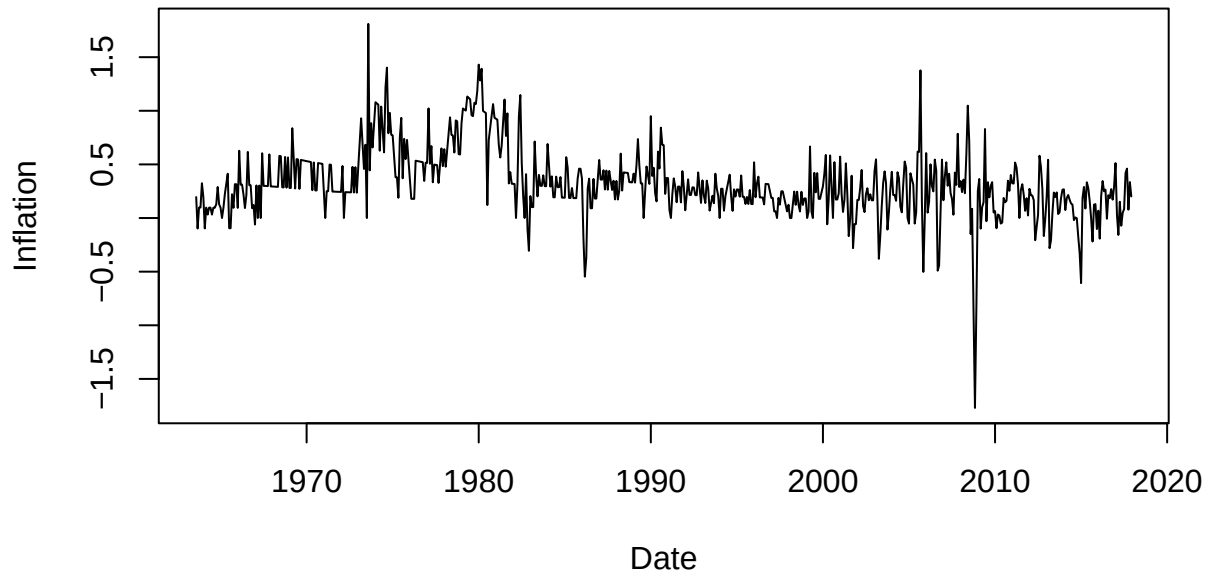
Read local data

Recommend the use of `csv` format, a plain ASCII file format.

- `read.table()` or `read.csv()` imports data from an ASCII file into an R session.
- `write.table()` or `write.csv()` exports the data in an R session to an ASCII file.

Example: FRED-MD Data

```
# Read data
fred_md <- read.csv("./data/fred_md_data.csv")
fred_md <- fred_md[-1, ] # remove the first row (transform code)
# Read in CPI and calculate the inflation
cpi <- fred_md$CPIAUCSL
infl_cpi = 100 * (cpi[-1] - cpi[-length(cpi)]) / cpi[-length(cpi)]
# Transform the format of date
date <- zoo::as.yearmon(zoo::as.Date( fred_md[, 1], format = "%m/%d/%Y" ))
date <- date[-1]
plot(date, infl_cpi, type = 'l', xlab = "Date", ylab = "Inflation")
```



Data included in packages

```
library(AER)
data("CreditCard")
head(CreditCard)
```

```
##   card reports      age income       share expenditure owner selfemp dependents
## 1  yes       0 37.66667 4.5200 0.033269910  124.983300   yes      no          3
## 2  yes       0 33.25000 2.4200 0.005216942    9.854167    no      no          3
```

```
## 3  yes       0 33.66667 4.5000 0.004155556   15.000000   yes      no          4
## 4  yes       0 30.50000 2.5400 0.065213780  137.869200    no      no          0
## 5  yes       0 32.16667 9.7867 0.067050590  546.503300   yes      no          2
## 6  yes       0 23.25000 2.5000 0.044438400   91.996670    no      no          0
##   months majorcards active
## 1     54          1     12
## 2     34          1     13
## 3     58          1      5
## 4     25          1      7
## 5     64          1      5
## 6     54          1      1
```

Read data from internet

Besides loading a data file on the local hard disk, We can directly download data from internet. Here we show how to retrieve the stock daily data of *Apple Inc.* from *Yahoo Finance*, and save the dataset locally. A package called `quantmod` is used.

```
quantmod::getSymbols("AAPL", src = "yahoo")
```

```
## Registered S3 method overwritten by 'quantmod':
##   method            from
##   as.zoo.data.frame zoo
```

```
## [1] "AAPL"
```

```
tail(AAPL)
```

```
##            AAPL.Open AAPL.High AAPL.Low AAPL.Close AAPL.Volume AAPL.Adjusted
## 2024-09-19    224.99    229.82   224.63     228.87    66781300        228.87
## 2024-09-20    229.97    233.09   227.62     228.20   318679900        228.20
## 2024-09-23    227.34    229.45   225.81     226.47    54146000        226.47
## 2024-09-24    228.65    229.35   225.73     227.37    43556100        227.37
## 2024-09-25    224.93    227.29   224.02     226.37    42308700        226.37
## 2024-09-26    227.30    228.50   225.41     227.52    36636700        227.52
```

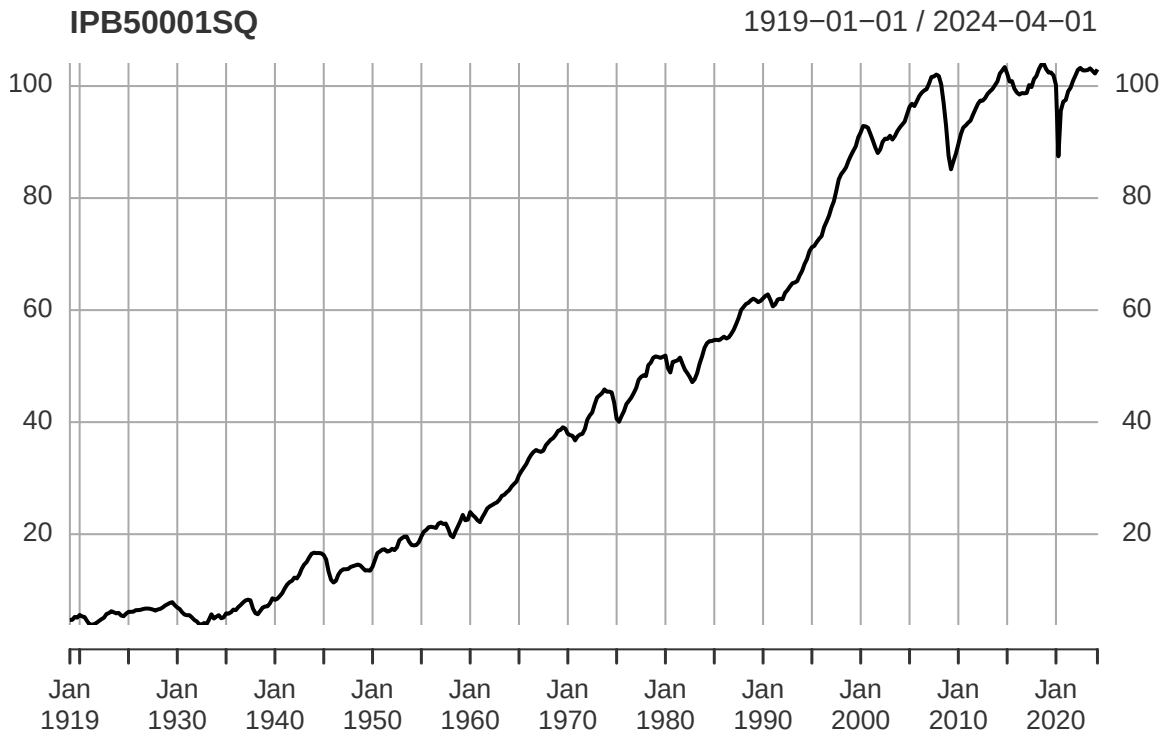
```
plot(AAPL$AAPL.Close)
```

Another example: Industrial Production: Total Index in FRED data

```
quantmod::getSymbols.FRED(Symbols = c("IPB50001SQ"), env = .GlobalEnv)
```

```
## [1] "IPB50001SQ"
```

```
plot(IPB50001SQ)
```



Flow Control

Flow control is common in all programming languages. `if` is used for choice, and `for` or `while` is used for loops.

`if ... else ...`

```
x <- 1
if (x == 1) {
    print("Yes!")
} else if (x == 0) {
    print("No!")
} else {
    print("I don't understand.")
}
```

```
## [1] "Yes!"
```

`for` loop

```
for (i in 1:3) {
    x <- rnorm(i)
    print(x)
}
```

```
## [1] -0.2986997
## [1] -0.5935356  0.8267038
## [1] -2.1614064 -0.9210425  0.5171631
```

When we use loops, we should minimize the tasks within the loop. One principle is that always pre-specify the dimension and allocate memory for variables outside the loop.

Example Empirical coverage of confidence intervals.

```
CI <- function(x) {
  # construct confidence interval
  # x is a vector of random variables
  n <- length(x)
  mu <- mean(x)
  sig <- sd(x)
  upper <- mu + 1.96 / sqrt(n) * sig
  lower <- mu - 1.96 / sqrt(n) * sig
  return(list(lower = lower, upper = upper))
}

Rep <- 100000
```

```r
sample_size <- 1000
mu <- 2

# Append a new outcome after each loop
pts0 <- Sys.time() # check time
for (i in 1:Rep) {

    x <- rpois(sample_size, mu)
    bounds <- CI(x)
    out_i <- ((bounds$lower <= mu) & (mu <= bounds$upper))
    if (i == 1) {
        out <- out_i
    } else {
        out <- c(out, out_i)
    }

}
mean(out)
```

```
## [1] 0.95023
```

```r
cat("Takes", Sys.time() - pts0, "seconds\n")
```

```
## Takes 9.672461 seconds
```

```r
# Initialize the result vector
out <- rep(0, Rep)
pts0 <- Sys.time() # check time
for (i in 1:Rep) {
    x <- rpois(sample_size, mu)
    bounds <- CI(x)
    out[i] <- ((bounds$lower <= mu) & (mu <= bounds$upper))
}
mean(out)
```

```
## [1] 0.94893
```

```r
cat("Takes", Sys.time() - pts0, "seconds\n")
```

```
## Takes 3.531373 seconds
```

Vectorization

In R and other high-level languages, for loops are SLOW. If you have to loop through many many elements and do expensive operations for each element, your code will take forever. There are several ways to speed up loops, one way relies on vectorization.

Many operations naively done in for-loops can instead be done using matrix operations.

Example: 2-D Random Walk

Consider an example of generating 2-D random walk (example borrowd from Ross Ihaka's online note)

- A 2-D discrete random walk:
  - Start at the point $(0, 0)$
  - For $t = 1, 2, \cdots$, take a unit step in a randomly chosen direction
    - N, S, E, W
- Naive Implementation with `for` loop

```
rw2d_loop <- function(n) {
    xpos <- rep(0, n)
    ypos <- rep(0, n)
    xdir <- c(TRUE, FALSE)
    step_size <- c(1, -1)
    for (i in 2:n) {
        if (sample(xdir, 1)) {
            xpos[i] <- xpos[i - 1] + sample(step_size, 1)
            ypos[i] <- ypos[i - 1]
        } else {
            xpos[i] <- xpos[i - 1]
            ypos[i] <- ypos[i - 1] + sample(step_size, 1)
        }
    }
    return(data.frame(x = xpos, y = ypos))
}
```

- Vectorization without loops

```
rw2d_vec <- function(n) {
    xsteps <- c(-1, 1, 0, 0)
    ysteps <- c(0, 0, -1, 1)
    dir <- sample(1:4, n - 1, replace = TRUE)
    xpos <- c(0, cumsum(xsteps[dir]))
    ypos <- c(0, cumsum(ysteps[dir]))

    return(data.frame(x = xpos, y = ypos))
}
```

- Comparison

```
n <- 100000
t0 <- Sys.time()
df <- rw2d_loop(n)
```

```r
cat("Naive implementation: ", difftime(Sys.time(), t0, units = "secs"))
```

```
## Naive implementation:  0.5797138
```

```r
t0 <- Sys.time()
df2 <- rw2d_vec(n)
cat("Vectorized implementation: ", difftime(Sys.time(), t0, units = "secs"))
```

```
## Vectorized implementation:  0.002183914
```

Example: Standard Error under Heteroskedasticity

In linear regression model with heteroskedasticity, the asymptotic distribution of the OLS estimator is

$$\sqrt{n}\left(\widehat{\beta}-\beta_0\right) \xrightarrow{d} N\left(0, E\left[x_i x_i'\right]^{-1} \operatorname{var}\left(x_i e_i\right) E\left[x_i x_i'\right]^{-1}\right)$$

where $\operatorname{var}\left(x_i e_i\right)$ is estimated by

$$\frac{1}{n}\sum_{i=1}^{n} x_i x_i' \widehat{e}_i^2 = \frac{1}{n} X'DX = \frac{1}{n}\left(X'D^{1/2}\right)\left(D^{1/2}X\right)$$

where D is a diagonal matrix of $\left(\widehat{e}_1^2, \widehat{e}_{2,}^2, \ldots, \widehat{e}_n^2\right)$.

1. Literally sum $\hat{e}_i^2 x_i x_i'$
2. Compute $X'DX$ with dense central matrix
3. Compute $X'DX$ with sparse central matrix
4. Do cross product to $X\hat{e}$. It takes advantage of the element-by-element operation in R

- Consider a linear probability model in which the outcome variable takes binary value $\{0,1\}$,

  $$y_i = x_i'\beta + v_i,$$

  and we assume $\mathbb{E}\left(v_i|x_i\right) = 0$. This implies:

  $$\mathbb{E}\left(y_i|x_i\right) = \Pr\left(y_i = 1\right) = x_i'\beta.$$

  Note that $v_i = \begin{cases} 1 - x_i'\beta & y_i = 1 \\ -x_i'\beta & y_i = 0 \end{cases}$, then

  $$\operatorname{var}\left(v_i|x_i\right) = x_i'\beta\left(1 - x_i'\beta\right)$$

- The error term is heteroskedastic.
- DGP and OLS estimation

```r
lpm <- function(n) {

  # set the parameters
```

```r
  b0 <- matrix(c(-1, 1), nrow = 2)

  # generate the data from a Probit model
  e <- rnorm(n)
  X <- cbind(1, rnorm(n))
  Y <- as.numeric(X %*% b0 + e >= 0)
  # note that in this regression bhat does not converge to b0
  # because the model is mis-specified

  # OLS estimation
  bhat <- solve(t(X) %*% X, t(X) %*% Y)
  e_hat <- Y - X %*% bhat
  return(list(X = X, e_hat = as.vector(e_hat)))
}
```

```r
# Set up
n <- 50
Rep <- 1000
data.Xe <- lpm(n)
X <- data.Xe$X
e_hat <- data.Xe$e_hat

# Estimation
est_func <- function(X, e_hat, opt) {
    if (opt == 1) {
        for (i in 1:n) {
            XXe2 <- matrix(0, nrow = 2, ncol = 2)
            XXe2 <- XXe2 + e_hat[i]^2 * X[i, ] %*% t(X[i, ])
        }
    } else if (opt == 2) {
        e_hat2_M <- matrix(0, nrow = n, ncol = n)
        diag(e_hat2_M) <- e_hat^2
        XXe2 <- t(X) %*% e_hat2_M %*% X
    } else if (opt == 3) {
        e_hat2_M <- Matrix::Matrix(0, ncol = n, nrow = n)
        diag(e_hat2_M) <- e_hat^2
        XXe2 <- t(X) %*% e_hat2_M %*% X
    } else if (opt == 4) {
        Xe <- X * e_hat
        XXe2 <- t(Xe) %*% Xe
    }

    XX_inv <- solve(t(X) %*% X)
    sig_B <- XX_inv %*% XXe2 %*% XX_inv
```

```
    return(sig_B)
}

# Compare the speed
for (opt in 1:4) {
    pts0 <- Sys.time()
    for (iter in 1:Rep) {
        sig_B <- est_func(X, e_hat, opt)
    }
    cat("n =", n, ", Rep =", Rep, ", opt =", opt, ", time =", Sys.time() - pts0, "\n")
}
```

```
## n = 50 , Rep = 1000 , opt = 1 , time = 0.1547518
## n = 50 , Rep = 1000 , opt = 2 , time = 0.120152
## n = 50 , Rep = 1000 , opt = 3 , time = 0.1816239
## n = 50 , Rep = 1000 , opt = 4 , time = 0.01119304
```

- Increase the sample size

```
n <- 2000
data.Xe <- lpm(n)
X <- data.Xe$X
e_hat <- data.Xe$e_hat
```

```
for (opt in 1:4) {
    pts0 <- Sys.time()
    for (iter in 1:Rep) {
        sig_B <- est_func(X, e_hat, opt)
    }
    cat("n =", n, ", Rep =", Rep, ", opt =", opt, ", time =", Sys.time() - pts0, "\n")
}
```

```
## n = 2000 , Rep = 1000 , opt = 1 , time = 5.354386
## n = 2000 , Rep = 1000 , opt = 2 , time = 22.85418
## n = 2000 , Rep = 1000 , opt = 3 , time = 0.2245958
## n = 2000 , Rep = 1000 , opt = 4 , time = 0.04970598
```

Parallel computing

We can exploit the power of multicore machines to speed up loops by parallel computing. The packages `foreach` and `doParallel` are useful for parallel computing.

```
capture <- function() {
    x <- rpois(sample_size, mu)
    bounds <- CI(x)
    return(((bounds$lower <= mu) & (mu <= bounds$upper)))
```

```
}

# Workhorse packages
library(foreach)
library(doParallel)

Rep <- 100000
sample_size <- 1000

registerDoParallel(8) # open 8 CPUs to accept incoming jobs.
pts0 <- Sys.time() # check time
out <- foreach(icount(Rep), .combine = c) %dopar% {
    x <- rpois(sample_size, mu)
    bounds <- CI(x)
    ((bounds$lower <= mu) & (mu <= bounds$upper))
}
cat("parallel loop takes", Sys.time() - pts0, "seconds\n")
```

```
## parallel loop takes 4.328318 seconds
```

```
pts0 <- Sys.time() # check time
out <- foreach(icount(Rep), .combine = c) %do% {
    x <- rpois(sample_size, mu)
    bounds <- CI(x)
    ((bounds$lower <= mu) & (mu <= bounds$upper))
}
cat("sequential loop takes", Sys.time() - pts0, "seconds\n")
```

```
## sequential loop takes 8.439488 seconds
```

We change the nature of the task a bit.

```
Rep <- 200
sample_size <- 200000

pts0 <- Sys.time() # check time
out <- foreach(icount(Rep), .combine = c) %dopar% {
    x <- rpois(sample_size, mu)
    bounds <- CI(x)
    ((bounds$lower <= mu) & (mu <= bounds$upper))
}
cat("parallel loop takes", Sys.time() - pts0, "seconds\n")
```

```
## parallel loop takes 0.2144861 seconds
```

```
pts0 <- Sys.time() # check time
out <- foreach(icount(Rep), .combine = c) %do% {
```

```
    x <- rpois(sample_size, mu)
    bounds <- CI(x)
    ((bounds$lower <= mu) & (mu <= bounds$upper))
}
cat("sequential loop takes", Sys.time() - pts0, "seconds\n")
```

```
## sequential loop takes 1.012217 seconds
```

Statistics

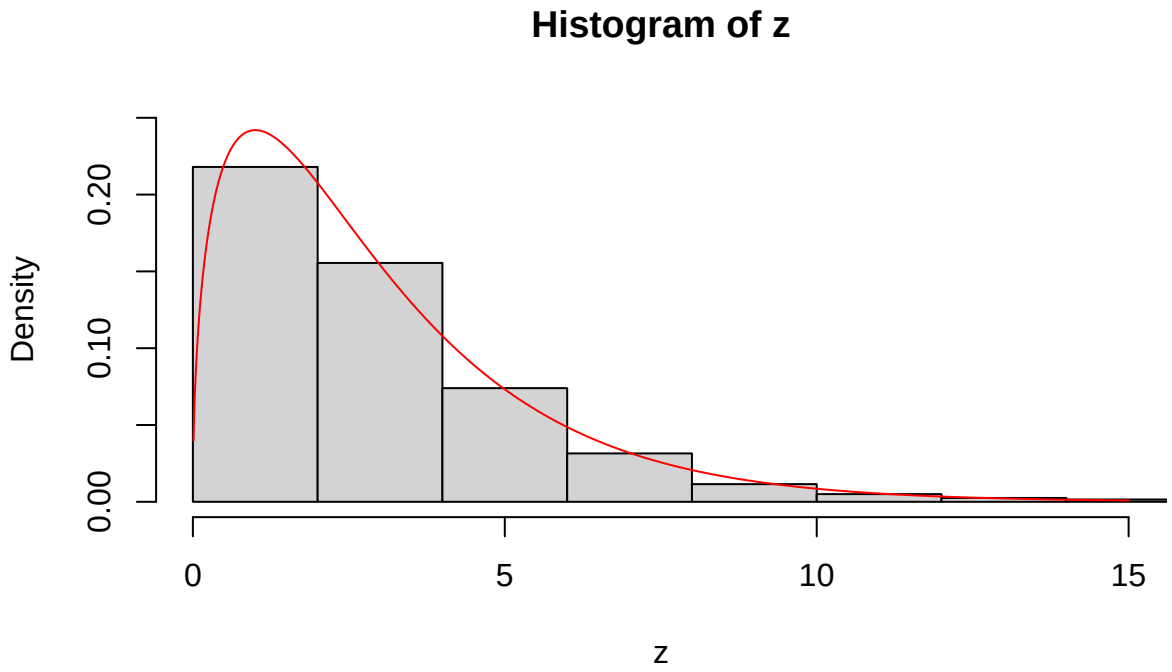
R is a language created by statisticians. It has elegant built-in statistical functions. `p` (probability), `d` (density for a continuous random variable, or mass for a discrete random variable), `q` (quantile), `r` (random variable generator) are used ahead of the name of a probability distribution, such as `norm` (normal), `chisq` (χ^2), `t` (t), `weibull` (Weibull), `cauchy` (Cauchy), `binomial` (binomial), `pois` (Poisson), to name a few.

Example

This example illustrates the sampling error.

1. Plot the density of $\chi^2(3)$ over an equally spaced grid system `x_axis = seq(0.01, 15, by = 0.01)` (black line).
2. Generate 1000 observations from $\chi^2(3)$ distribution. Plot the kernel density, a nonparametric estimation of the density (red line).

```
set.seed(100)
x_axis <- seq(0.01, 15, by = 0.01)
y <- dchisq(x_axis, df = 3)
z <- rchisq(1000, df = 3)
hist(z, freq = FALSE, xlim = range(0.01, 15), ylim = range(0, 0.25))
lines(y = y, x = x_axis, type = "l", xlab = "x", ylab = "density", col = "red")
```



Statistical models are formulated as `y~x`, where `y` on the left-hand side is the dependent variable, and `x` on the right-hand side is the explanatory variable. The built-in OLS function is `lm`. It is called by `lm(y~x, data = data_frame)`.

All built-in regression functions in R share the same structure. Once one type of regression is understood, it is easy to extend to other regressions.

```
# Linear models
n <- 100
x <- rnorm(n)
y <- 0.5 + 1 * x + rnorm(n)
result <- lm(y ~ x)
summary(result)
```

```
##
## Call:
## lm(formula = y ~ x)
##
## Residuals:
##      Min       1Q   Median       3Q      Max
## -2.49919 -0.53555 -0.08413  0.46672  2.74626
##
## Coefficients:
##             Estimate Std. Error t value Pr(>|t|)
## (Intercept)  0.43953    0.09161   4.798 5.74e-06 ***
## x            1.03031    0.10158  10.143  < 2e-16 ***
## ---
## Signif. codes:  0 '***' 0.001 '**' 0.01 '*' 0.05 '.' 0.1 ' ' 1
```

```
##
## Residual standard error: 0.9091 on 98 degrees of freedom
## Multiple R-squared:  0.5122, Adjusted R-squared:  0.5072
## F-statistic: 102.9 on 1 and 98 DF,  p-value: < 2.2e-16
```

```
class(result)
```

```
## [1] "lm"
```

```
typeof(result)
```

```
## [1] "list"
```

```
result$coefficients
```

```
## (Intercept)           x
##   0.4395307   1.0303095
```

An simple example of logit regression using the Swiss Labor Market Participation Data (Cross-section data originating from the health survey SOMIPOPS for Switzerland in 1981).

```
library("AER")
data("SwissLabor", package="AER")
head(SwissLabor)
```

```
##   participation   income age education youngkids oldkids foreign
## 1            no 10.78750 3.0         8         1       1      no
## 2           yes 10.52425 4.5         8         0       1      no
## 3            no 10.96858 4.6         9         0       0      no
## 4            no 11.10500 3.1        11         2       0      no
## 5            no 11.10847 4.4        12         0       2      no
## 6           yes 11.02825 4.2        12         0       1      no
```

```
SwissLabor <- SwissLabor %>%
    mutate(participation = ifelse(participation == "yes", 1, 0))
glm_fit <- glm(
    participation ~ age + education + youngkids + oldkids + foreign,
    data = SwissLabor,
    family = "binomial")
summary(glm_fit)
```

```
##
## Call:
## glm(formula = participation ~ age + education + youngkids + oldkids +
##     foreign, family = "binomial", data = SwissLabor)
##
## Coefficients:
##             Estimate Std. Error z value Pr(>|z|)
## (Intercept)  2.085961   0.540535   3.859 0.000114 ***
```

```
## age         -0.527066   0.089670  -5.878 4.16e-09 ***
## education   -0.001298   0.027502  -0.047 0.962371
## youngkids   -1.326957   0.177893  -7.459 8.70e-14 ***
## oldkids     -0.072517   0.071878  -1.009 0.313024
## foreignyes   1.353614   0.198598   6.816 9.37e-12 ***
## ---
## Signif. codes:  0 '***' 0.001 '**' 0.01 '*' 0.05 '.' 0.1 ' ' 1
##
## (Dispersion parameter for binomial family taken to be 1)
##
##     Null deviance: 1203.2  on 871  degrees of freedom
## Residual deviance: 1069.9  on 866  degrees of freedom
## AIC: 1081.9
##
## Number of Fisher Scoring iterations: 4
```

Monte Carlo Simulation

We demonstrate the skills from this notes by a Monte Carlo simulation practice. In such experiments, we sample data from a specified statistical model and examine the finite sample performance of estimation/inference procedures.

For different data generating processes and different sample sizes n, we simulate data and estimate the model for many times and summarized the results (in most cases) by measures (empirical) Bias, RMSE, size and empirical power functions, or empirical density plots.

Generically, bias, RMSE and size are calculated by

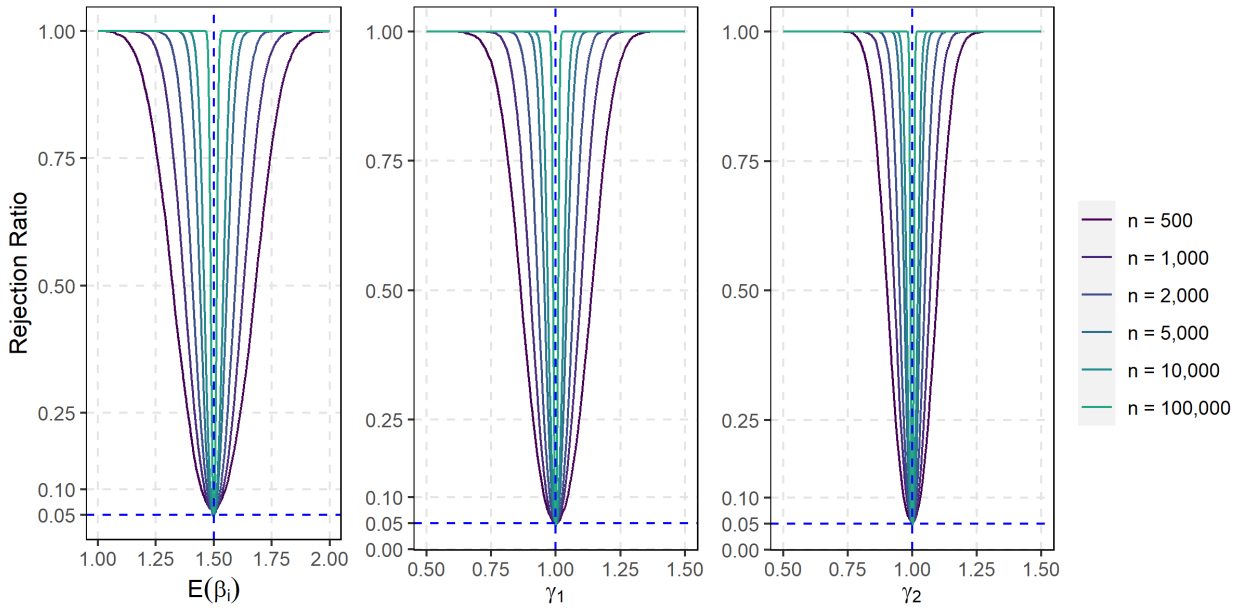
$$bias = R^{-1}\sum_{r=1}^{R}\left(\hat{\theta}^{(r)} - \theta_0\right),$$

$$RMSE = \left(R^{-1}\sum_{r=1}^{R}\left(\hat{\theta}^{(r)} - \theta_0\right)^2\right)^{1/2},$$

$$Size = R^{-1}\sum_{r=1}^{R}\mathbf{1}\left[\left|\hat{\theta}^{(r)} - \theta_0\right|/\hat{\sigma}_{\hat{\theta}}^{(r)} > \text{cv}_{0.05}\right].$$

for true parameter θ_0, its estimate $\hat{\theta}^{(r)}$, the estimated standard error of $\hat{\theta}^{(r)}$, $\hat{\sigma}_{\hat{\theta}}^{(r)}$.

$$Power(\theta_\delta) = R^{-1}\sum_{r=1}^{R}\mathbf{1}\left[\left|\hat{\theta}^{(r)} - \theta_\delta\right|/\hat{\sigma}_{\hat{\theta}}^{(r)} > \text{cv}_{0.05}\right]$$



Major components: 1. Data generating functions. 2. Estimation Functions. 3. Master files. Repeatedly generate data and estimate the model. 4. Results generating functions.

```
Initialize result_container
for (i in [different setups]) {
    for (r in 1:number_of_replications) {
        data = generate_data_function(sample size, parameters, ...)
        result_container[i, r] = statistical_operation(data, ...)
    }
}
list(bias, rmse, size, power, ...) = result_summary_function(result_container, ...)
```

Example Omitted variable bias.

Consider a linear regression model $y_i = \alpha + x_{i1}\beta_1 + x_{i2}\beta_2 + u_i$. (y_i, x_i) are i.i.d. with

$$u_i \sim i.i.d.N(0,1), \quad (x_{i1}, x_{i2})' \sim i.i.d.N\left(\begin{pmatrix}0\\1\end{pmatrix}, \begin{pmatrix}\sigma_1^2 & \rho\sigma_1\sigma_2\\ \rho\sigma_1\sigma_2 & \sigma_2^2\end{pmatrix}\right).$$

True parameters are $a = 0.11$, $\beta = (0.22, 0.33)'$, $\rho = 0.5$, $\sigma_1 = 1$, and $\sigma_2 = 4$.

We investigate the finite sample performance of the OLS estimator in two scenarios: 1. Regress y on x_1 and x_2. 2. Regress y on x_1 only.

Run 1000 replications and report the bias and RMSE.

```
# The data generating process (DGP)
dgp <- function(n, a = 0.11, b = c(0.22, 0.33), mu = c(0,1),
                sigma1 = 1, sigma2 = 4, rho = 0.1){

    Sigma <-  matrix(c(sigma1^2, rho*sigma1*sigma2,
```

```
                      rho*sigma1*sigma2, sigma2^2), 2, 2)
    X <-  MASS::mvrnorm(n = n, mu = mu, Sigma = Sigma)
    y <-  a + X %*% b + rnorm(n)

    return( list(y = y, X = X) )
}


# Fix the seed for random number generator
# so that the results are replicable
set.seed(200)
# Setup
n <- 1000
p <- 3
a0 <- 0.11
b0 <- c(0.22, 0.33)
Rep <- 1000
# Container
Coef_hat <- matrix(0, p, Rep)
Coef_hat_unspec <- matrix(0, p-1, Rep)
# Estimation
for(i in 1:Rep){
    data <- dgp(n)
    Coef_hat[, i] <- lsfit(data$X, data$y)$coefficients
    Coef_hat_unspec[, i] <- lsfit(data$X[, 1], data$y)$coefficients
}

# A function that summarizes results
result_sum <- function(coef_hat, para_0) {
    Rep <- ncol(coef_hat)
    data.frame(mean = rowMeans(coef_hat),
               bias = rowMeans(coef_hat) - para_0,
               RMSE = sqrt(rowMeans((coef_hat - matrix(para_0, length(para_0), Rep))^2)))
}

cat("Correct-specified Model: \n")
```

```
## Correct-specified Model:
```

```
result_sum(Coef_hat, c(a0, b0))
```

```
##        mean          bias        RMSE
## 1 0.1110109  0.0010109166 0.032403825
## 2 0.2198700 -0.0001299870 0.031024382
## 3 0.3295965 -0.0004034982 0.007871069
```

```r
cat("Under-specified Model: \n")
```

```
## Under-specified Model:
```

```r
result_sum(Coef_hat_unspec, c(a0, b0[1]))
```

```
##        mean      bias      RMSE
## 1 0.4413665 0.3313665 0.3355070
## 2 0.3501253 0.1301253 0.1397925
```

Example: Bias of OLS estimator for the AR(1)

Since the regressors in AR(1) regression,

$$y_t = \rho y_{t-1} + u_t$$

are not strictly exogenous, the ordinary least square estimator

$$\hat{\rho}_{OLS} = \left(\sum_{t=2}^{T} y_{t-1}^2 \right)^{-1} \left(\sum_{t=2}^{T} y_{t-1} y_t \right)$$

is biased in finite sample. Kendall (1954) derives the bias formula for the case $|\rho| < 1$ and innovations u_t are normally distributed:

$$\mathrm{E}\left(\hat{\rho}_{OLS}\right) - \rho = -\frac{1+3\rho}{T} + O\left(\frac{1}{T^2}\right)$$

The bias matters when T is small.

- Bias of OLS estimator for the AR(1)

```r
set.seed(100)
n_vec <- c(10, 20, 30, 50, 100)
num_n <- length(n_vec)
rho <- 0.5
R <- 2000
rho_hat_mat <- matrix(NA, R, num_n) # Initialize result container
for(i in 1:num_n) {
    n <- n_vec[i]
    for(r in 1:R){
        y <- arima.sim(model = list(order = c(1, 0, 0), ar = rho), n = n)
        rho_hat_mat[r, i] <- as.numeric(
            ar(y, order.max = 1, aic = FALSE, method = "ols")$ar
        )
    }
}

data.frame(
    n = n_vec,
```

```
    bias = colMeans(rho_hat_mat) - rho,
    formula = -(1 + 3 * rho) / n_vec
)
```

```
##     n        bias     formula
## 1  10 -0.25481602 -0.25000000
## 2  20 -0.12655888 -0.12500000
## 3  30 -0.08788927 -0.08333333
## 4  50 -0.04736005 -0.05000000
## 5 100 -0.02640868 -0.02500000
```

- You may try other values of ρ and T to complete the picture.

Data example: Welch and Goyal (2008)

- Welch, I. and A. Goyal (2008). A comprehensive look at the empirical performance of equity premium prediction. The Review of Financial Studies 21(4), 1455–1508.
- Both monthly and quarterly data of S&P 500 index and potential predictors including macroeconomic variables and financial variables
- Widely used to study stock return predictablity
  - Koo, B., H. M. Anderson, M. H. Seo, & W. Yao (2020). High-dimensional predictive regression in the presence of cointegration. Journal of Econometrics 219(2), 456–477.
  - Lee, J. H., Shi, Z., & Gao, Z. (2022). On LASSO for predictive regression. Journal of Econometrics, 229(2), 322-349.
- Data source: http://www.hec.unil.ch/agoyal/

```
# Load the raw data
data_raw <- read_excel("./data/PredictorData2018.xlsx",
                       sheet = "Monthly",
                       col_names = TRUE,
                       na = "NaN")
head(data_raw)
```

```
## # A tibble: 6 x 18
##   yyyymm Index   D12   E12 `b/m`    tbl    AAA    BAA    lty   ntis   Rfree
##    <dbl> <dbl> <dbl> <dbl> <dbl>  <dbl>  <dbl>  <dbl>  <dbl>  <dbl>   <dbl>
## 1 192612  13.5 0.69   1.24 0.441 0.0307 0.0468 0.0568 0.0354 0.0509 0.00256
## 2 192701  13.2 0.697  1.23 0.444 0.0323 0.0466 0.0561 0.0351 0.0508 0.00269
## 3 192702  13.8 0.703  1.22 0.429 0.0329 0.0467 0.0559 0.0347 0.0517 0.00274
## 4 192703  13.9 0.71   1.21 0.470 0.032  0.0462 0.0554 0.0331 0.0464 0.00267
## 5 192704  14.2 0.717  1.20 0.457 0.0339 0.0458 0.0548 0.0333 0.0505 0.00282
## 6 192705  14.9 0.723  1.19 0.435 0.0333 0.0457 0.055  0.0327 0.0553 0.00278
## # i 7 more variables: infl <dbl>, ltr <dbl>, corpr <dbl>, svar <dbl>,
## #   csp <dbl>, CRSP_SPvw <dbl>, CRSP_SPvwx <dbl>
```

- Construct variables

```
# Construct the data frame, start from the dates
data_wg <- data.frame(
    date = zoo::as.yearmon(lubridate::ymd(data_raw$yyyymm, truncated = 2))
)

# Construct predictors according to Welch and Goyal (2008)
attach(data_raw)

data_wg$ExReturn <- log(Index) - log(c(1,Index[-length(Index)]))
data_wg$dp <- log(D12) - log(Index)
data_wg$dy <- log(D12) - log( c(1, Index[-length(Index)]) )
data_wg$ep <- log(E12) - log(Index)
data_wg$tms <- lty - tbl
data_wg$dfy <- BAA - AAA
data_wg$dfr <- corpr - ltr

detach(data_raw)
```

- Put variables together

```
data_wg <- cbind(data_wg,
                 data_raw[c("b/m", "tbl", "ltr",
                            "ntis", "svar", "infl")])
data_wg <- data_wg[-1, ]
head(data_wg)
```

```
##       date     ExReturn        dp        dy        ep     tms    dfy     dfr
## 2 Jan 1927 -0.020974552 -2.942374 -2.963349 -2.374773  0.0028 0.0095 -0.0019
## 3 Feb 1927  0.046588832 -2.979535 -2.932946 -2.430353  0.0018 0.0092 -0.0019
## 4 Mar 1927  0.006481838 -2.976535 -2.970053 -2.445079  0.0011 0.0092 -0.0170
## 5 Apr 1927  0.017082266 -2.984225 -2.967143 -2.471309 -0.0006 0.0090  0.0060
## 6 May 1927  0.050905075 -3.025963 -2.975058 -2.531446 -0.0006 0.0093 -0.0120
## 7 Jun 1927 -0.009434032 -3.007309 -3.016743 -2.531330  0.0027 0.0097  0.0112
##         b/m    tbl     ltr       ntis         svar         infl
## 2 0.4437056 0.0323  0.0075 0.05083276 0.0004698947 -0.011299435
## 3 0.4285009 0.0329  0.0088 0.05168097 0.0002873343 -0.005714286
## 4 0.4697651 0.0320  0.0253 0.04636992 0.0009241928 -0.005747126
## 5 0.4567541 0.0339 -0.0005 0.05051790 0.0006025886  0.000000000
## 6 0.4347826 0.0333  0.0109 0.05527903 0.0003917338  0.005780347
## 7 0.4523852 0.0307 -0.0069 0.05882564 0.0008245770  0.011494253
```

- A `dplyr` way to doing this (Refer to `dplyr` and Chapter 3, 7 - 13 of R for data science)

```
data_wg <- data_raw %>%
    transmute(
```

```
        date = zoo::as.yearmon(lubridate::ymd(yyyymm, truncated = 2)),
        ExReturn = log(Index) - log(c(1,Index[-length(Index)])),
        dp = log(D12) - log(Index),
        dy = log(D12) - log( c(1, Index[-length(Index)]) ),
        ep = log(E12) - log(Index),
        tms = lty - tbl,
        dfy = BAA - AAA,
        dfr = corpr - ltr,
        bm = `b/m`,
        tbl = tbl,
        ltr = ltr,
        ntis = ntis,
        svar = svar,
        infl = infl
    ) %>%
    filter(
        row_number() > 1
    )
print(data_wg, n = 10)
```

```
## # A tibble: 1,104 x 14
##    date      ExReturn    dp    dy    ep       tms     dfy     dfr    bm    tbl
##    <yearmon>    <dbl> <dbl> <dbl> <dbl>     <dbl>   <dbl>   <dbl> <dbl>  <dbl>
##  1 Jan 1927  -0.0210  -2.94 -2.96 -2.37  0.00280  0.0095  -0.0019 0.444 0.0323
##  2 Feb 1927   0.0466  -2.98 -2.93 -2.43  0.00180  0.0092  -0.0019 0.429 0.0329
##  3 Mar 1927   0.00648 -2.98 -2.97 -2.45  0.00110  0.0092  -0.017  0.470 0.032
##  4 Apr 1927   0.0171  -2.98 -2.97 -2.47 -0.000600 0.009    0.006  0.457 0.0339
##  5 May 1927   0.0509  -3.03 -2.98 -2.53 -0.000600 0.0093  -0.012  0.435 0.0333
##  6 Jun 1927  -0.00943 -3.01 -3.02 -2.53  0.0027   0.0097   0.0112 0.452 0.0307
##  7 Jul 1927   0.0630  -3.06 -3.00 -2.60  0.00370  0.0095  -0.0047 0.415 0.0296
##  8 Aug 1927   0.0435  -3.10 -3.05 -2.66  0.00590  0.00920  0.0007 0.396 0.027
##  9 Sep 1927   0.0423  -3.13 -3.09 -2.71  0.0062   0.00880  0.0131 0.381 0.0268
## 10 Oct 1927  -0.0546  -3.07 -3.12 -2.66  0.0017   0.0087  -0.0044 0.414 0.0308
## # i 1,094 more rows
## # i 4 more variables: ltr <dbl>, ntis <dbl>, svar <dbl>, infl <dbl>
```

```
x <- data_wg %>%
    mutate(date = zoo::as.Date.yearmon(date))
print(x, n = 6)
```

```
## # A tibble: 1,104 x 14
##   date       ExReturn    dp    dy    ep       tms    dfy     dfr    bm    tbl
##   <date>        <dbl> <dbl> <dbl> <dbl>     <dbl>  <dbl>   <dbl> <dbl>  <dbl>
## 1 1927-01-01 -0.0210  -2.94 -2.96 -2.37  0.00280  0.0095 -0.0019 0.444 0.0323
## 2 1927-02-01  0.0466  -2.98 -2.93 -2.43  0.00180  0.0092 -0.0019 0.429 0.0329
```

```
## 3 1927-03-01  0.00648 -2.98 -2.97 -2.45  0.00110  0.0092 -0.017  0.470 0.032
## 4 1927-04-01  0.0171  -2.98 -2.97 -2.47 -0.000600 0.009   0.006  0.457 0.0339
## 5 1927-05-01  0.0509  -3.03 -2.98 -2.53 -0.000600 0.0093 -0.012  0.435 0.0333
## 6 1927-06-01 -0.00943 -3.01 -3.02 -2.53  0.0027   0.0097  0.0112 0.452 0.0307
## # i 1,098 more rows
## # i 4 more variables: ltr <dbl>, ntis <dbl>, svar <dbl>, infl <dbl>
```

Visualization

- "One picture is worth ten thousand words".
- `plot` is a generic command for graphs in r-base.
  - For preliminary statistical graphs.
- `matplot` for multiple objects
- `ggplot2`: Advanced system for high-quality statistical graphs.
  - Layered grammar of graphics
- Plot single time series

```
ggplot(data = x, aes(x = date, y = dp)) +
    geom_line() +
    labs(x = "data", y = "DP ratio") +
    theme(
        panel.background = element_rect(fill = "transparent",
                                        colour = NA_character_),
        plot.background = element_rect(fill = "transparent",
                                       colour = NA_character_),
        panel.border = element_rect(
            linetype = 1,
            colour = "black",
            fill = NA
        ),
        panel.grid.major = element_line(linetype = 2, color = "grey90")
    )
```

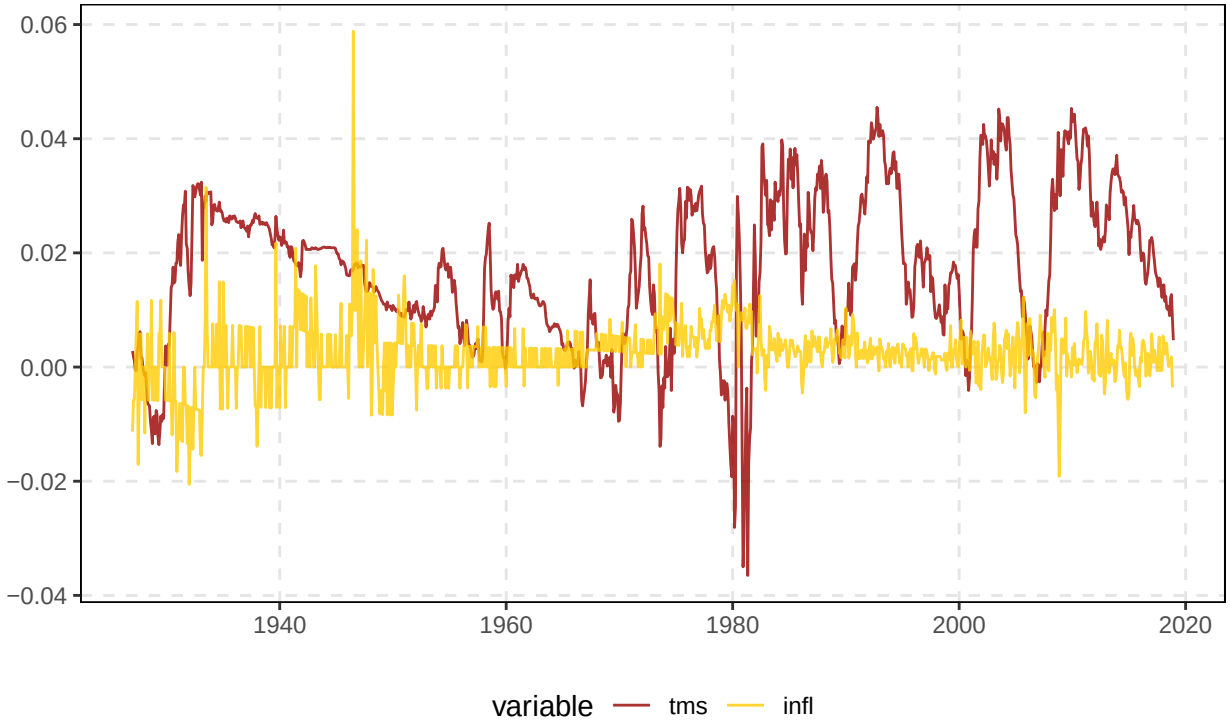
- Plot multiple time series in one plot

```
x_plot <- reshape2::melt(
    x %>% select(date, tms, infl), id = "date"
)
head(x_plot)
```

```
##         date variable   value
## 1 1927-01-01      tms  0.0028
## 2 1927-02-01      tms  0.0018
## 3 1927-03-01      tms  0.0011
## 4 1927-04-01      tms -0.0006
## 5 1927-05-01      tms -0.0006
## 6 1927-06-01      tms  0.0027
```

```
ggplot(data = x_plot) +
    geom_line(mapping = aes(x = date, y = value, color = variable), alpha = 0.8) +
    scale_color_manual(values = c("#990000", "#FFCC00")) +
    labs(x = NULL, y = NULL) +
    theme(
        panel.background = element_rect(fill = "transparent",
                                        colour = NA_character_),
        plot.background = element_rect(fill = "transparent",
                                       colour = NA_character_),
        panel.border = element_rect(
            linetype = 1,
```

```
            colour = "black",
            fill = NA
        ),
        legend.position = "bottom",
        panel.grid.major = element_line(linetype = 2, color = "grey90")
    )
```



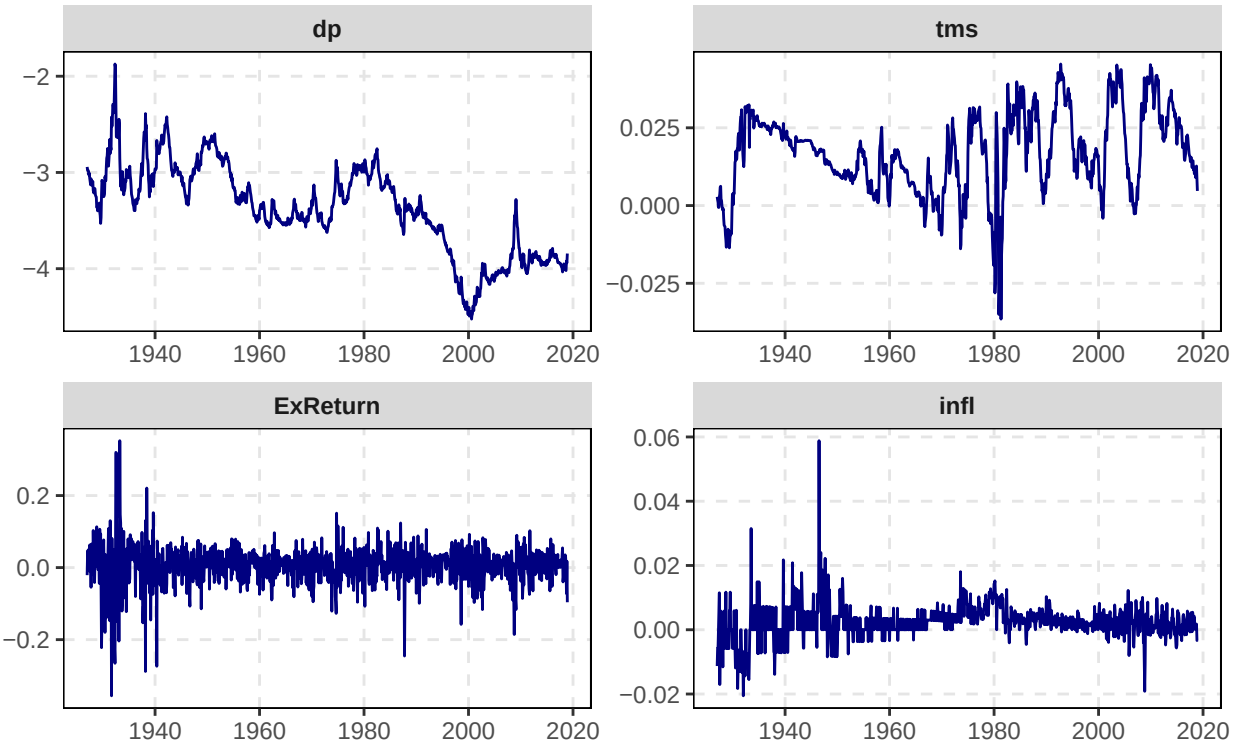
- Plot multiple time series in different plots

```
x_plot <- reshape2::melt(
    x %>% select(date, dp, tms, ExReturn, infl), id = "date"
)
head(x_plot)
```

```
##         date variable     value
## 1 1927-01-01       dp -2.942374
## 2 1927-02-01       dp -2.979535
## 3 1927-03-01       dp -2.976535
## 4 1927-04-01       dp -2.984225
## 5 1927-05-01       dp -3.025963
## 6 1927-06-01       dp -3.007309
```

```
p <- ggplot(data = x_plot) +
    geom_line(mapping = aes(x = date, y = value), color = "navyblue") +
    facet_wrap( ~ variable, ncol = 2, scales = "free")
p + labs(x = NULL, y = NULL) +
    theme(
```

```
        panel.background = element_rect(fill = "transparent",
                                        colour = NA_character_),
        plot.background = element_rect(fill = "transparent",
                                       colour = NA_character_),
        panel.border = element_rect(
            linetype = 1,
            colour = "black",
            fill = NA
        ),
        panel.grid.major = element_line(linetype = 2, color = "grey90"),
        strip.text =  element_text(face = "bold")
    )
```



Rmarkdown

R Markdown is document format in which we can execute R code and write documents using Markdown grammar.

> R Markdown provides an authoring framework for data science. You can use a single R Markdown file to both
>
> - save and execute code
> - generate high quality reports that can be shared with an audience

A friendly introduction to R Markdown by RStudio: https://rmarkdown.rstudio.com/lesson-1.html

Tips

- Learn R the hard way
- Never use absolute path
- Always write comments
- Style, see The tidyverse style guide
- Version control and collaboration
  - Git and Github
  - A quick demonstration in class
  - details in the note and references therein
  - McDermott's EC 607 course, Lecture 2 is superb
  - Michael Stepner's introduction
- Data work
  - Patrick Ball: Principled Data Processing
  - Patrick Ball: The Task Is A Quantum Of Workflow
  - Gentzkow and Shapiro Code and Data for the Social Sciences: A Practitioner's Guide
- A summary of useful links: Jonathan Dingel's webpage

Acknowledgement

Acknowledgements: This set of slides is heavily dependent on

- Grant McDermott's lecture notes at University of Oregon (EC 607)
- Zhentao Shi's lecture notes at the Chinese University of Hong Kong (ECON 5170).