MiCM Workshop Series - R Programming Beyond the Basics

Efficient Coding and Computing

- Yi Lian

- August 13, 2019

Link to workshop material https://github.com/ly129/MiCM (https://github.com/ly129/MiCM)

Outline

Morning

1. **An overview of efficiency**

    - General rules
    - R-specific rules
    - Time your program in R
        - Illustrations of the rules
2. **Efficient coding**

    - Powerful functions in R

      ```
      - aggregate(), by(), apply() family
      - ifelse(), cut() and split()
      ```

    - Write our own functions in R

      ```
      - function()
      ```

    - Examples and exercises
        - Categorization, conditional operations, etc..

Afternoon

1. **Efficient computing**

    - Parallel computing

      ```
      - Package 'parallel'
      ```

    - Integration with C++

      ```
      - Package 'Rcpp'
      ```

    - Integration with Fortran
2. **Exercises**

    - Examples and exercises
        - Implement our own functions written in R, Rcpp or Fortran!

Important note! There are MANY advanced and powerful packages that do different things. There are too many and they are too diverse to be covered in this workshop.

Here is a list of some awesome packages. https://awesome-r.com/ (https://awesome-r.com/)

1. An overview of efficiency

Why?

- Clean and tidy codes make everything easier - edit, debug, reproduce, etc..
- Era of big data/machine learning/AI
- Large sample size and/or high dimension

1.1 General rules

- All operations take time (CPU)
- Reading/writing data takes time
  - Memory allocation and re-allocation
  - A not really appropriate illustration

- Objects and operations take memory
  - http://adv-r.had.co.nz/memory.html (http://adv-r.had.co.nz/memory.html)
  - e.g. R will do "garbage collection" automatically when it needs more memory, which takes time
- Setups take time (overhead)
- Programming languages are different and are fast/slow at different things
- **Efficient coding $\neq$ efficient computing**
  - Shorter codes do not necessarily lead to shorter run time.
- **Avoid duplicated operations, especially expensive operations**
  - Matrix mulplications, inversion, etc..
  - Store the results that will be used later as objects.
- **Test your program**

1.2 R-specific rules

- R emphasizes flexibility but not speed
  - Very good for research
- R is designed to be better with **vectorized** operations than loops
- Without specific setups, R only uses 1 **CPU** core
  - Setting up parallel (multicore) computing takes time (overhead)
- Use well-developped R functions and packages
  - Some of them have core computations written in other languages, e.g. C, C++, Fortran
  - These functions usually make coding and computing more efficient at the same time.

Detailed illustration in 1.3 Time your program in R.

1.2.1 To understand vectorized operations and to facilitate integration with other programs, we need to know R data types and structures

R data types

```
- numeric
    - integer
    - double (default)
- logical
- character
- factor
- ...
```

In [1]:

```
# double
class(5); is.double(5)
```

'numeric'

TRUE

In [2]:

```
# integer
class(5L); is.double(5L)
```

'integer'

FALSE

In [3]:

```
object.size(rep(5, 1000))
object.size(rep(5L, 1000))
```

```
8048 bytes
```

```
4048 bytes
```

In [4]:

```
# How precise is double precision?
options(digits = 22) # show more digits in output
print(1/3)
options(digits = 7) # default
```

```
[1] 0.3333333333333333148296
```

In [5]:

```
# logical
class(TRUE); class(F)
```

'logical'

'logical'

In [6]:

```
# character
class("TRUE")
```

'character'

In [7]:

```
# Not important for this workshop
fac <- as.factor(c(1, 5, 11, 3))
fac
```

1 5 11 3

▶ **Levels**:

In [8]:

```
class(fac)
```

'factor'

In [9]:

```
fac.ch <- as.factor(c("B", "a", "1", "ab", "b", "A"))
fac.ch
```

B a 1 ab b A

▶ **Levels**:

R data structures

```
- Scalar *
- Vector
- Matrix
- Array
- List
- Data frame
- ...
```

In [10]:

```r
# Scalar - a vector of length 1
myscalar <- 5
myscalar
```

5

In [11]:

```r
class(myscalar)
```

'numeric'

In [12]:

```r
# Vector
myvector <- c(1, 1, 2, 3, 5, 8)
myvector
```

1 1 2 3 5 8

In [13]:

```r
class(myvector)
```

'numeric'

In [14]:

```r
# Matrix - a 2d array
mymatrix <- matrix(c(1, 1, 2, 3, 5, 8), nrow = 2, byrow = FALSE)
mymatrix
```

A matrix:
2 × 3 of
type dbl

1	2	5
1	3	8

In [15]:

```r
class(mymatrix)
```

'matrix'

In [16]:

```
# Array - not important for this workshop
myarray <- array(c(1, 1, 2, 3, 5, 8, 13, 21, 34, 55, 89, 144), dim = c(2, 2, 3))
print(myarray) # print() is not needed if run in R or Rstudio.
```

```
, , 1

     [,1] [,2]
[1,]    1    2
[2,]    1    3

, , 2

     [,1] [,2]
[1,]    5   13
[2,]    8   21

, , 3

     [,1] [,2]
[1,]   34   89
[2,]   55  144
```

In [17]:

```
class(myarray)
```

'array'

In [18]:

```
# List - very important for the workshop
mylist <- list(Title = "Efficient Coding and Computing",
               Duration = c(3, 3),
               sections = as.factor(c(1, 2, 3, 4)),
               Date = as.Date("2019-08-13"),
               Lunch_provided = FALSE,
               Feedbacks = c("Amazing!", "Great workshop!", "Yi is the best!", "
Wow!")
)
print(mylist) # No need for print if running in R or Rstudio
```

```
$Title
[1] "Efficient Coding and Computing"

$Duration
[1] 3 3

$sections
[1] 1 2 3 4
Levels: 1 2 3 4

$Date
[1] "2019-08-13"

$Lunch_provided
[1] FALSE

$Feedbacks
[1] "Amazing!"        "Great workshop!" "Yi is the best!" "Wow!"
```

In [19]:

```
class(mylist)
```

'list'

In [20]:

```
# Access data stored in lists
mylist$Title
```

'Efficient Coding and Computing'

In [21]:

```
# or
mylist[[6]]
```

'Amazing!' 'Great workshop!' 'Yi is the best!' 'Wow!'

In [22]:

```
# Further
mylist$Duration[1]
mylist[[6]][2]
```

3

'Great workshop!'

In [23]:

```
# Elements in lists can have different data types
lapply(mylist, class) # We will talk about lapply() later
```

$Title
'character'
$Duration
'numeric'
$sections
'factor'
$Date
'Date'
$Lunch_provided
'logical'
$Feedbacks
'character'

In [24]:

```
# Elements in list can have different lengths
lapply(mylist, length)
```

$Title

1

$Duration

2

$sections

4

$Date

1

$Lunch_provided

1

$Feedbacks

4

In [25]:

```
# Data frames - most commonly used for analyses
head(mtcars)
```

A data.frame: 6 × 11

	mpg	cyl	disp	hp	drat	wt	qsec	vs	am	gear	carb
	<dbl>	<dbl>	<dbl>	<dbl>	<dbl>	<dbl>	<dbl>	<dbl>	<dbl>	<dbl>	<dbl>
Mazda RX4	21.0	6	160	110	3.90	2.620	16.46	0	1	4	4
Mazda RX4 Wag	21.0	6	160	110	3.90	2.875	17.02	0	1	4	4
Datsun 710	22.8	4	108	93	3.85	2.320	18.61	1	1	4	1
Hornet 4 Drive	21.4	6	258	110	3.08	3.215	19.44	1	0	3	1
Hornet Sportabout	18.7	8	360	175	3.15	3.440	17.02	0	0	3	2
Valiant	18.1	6	225	105	2.76	3.460	20.22	1	0	3	1

In [26]:

```
# Access a column (variable) in data frames
mtcars$mpg
```

21 21 22.8 21.4 18.7 18.1 14.3 24.4 22.8 19.2 17.8 16.4 17.3 15.2 10.4 10.4 14.7 32.4 30.4 33.9 21.5 15.5 15.2 13.3 19.2 27.3 26 30.4 15.8 19.7 15 21.4

1.2.2 To show CPU usage

In [27]:

```
# Let's try to invert a large matrix.
A <- diag(4000)
# A.inv <- solve(A)
```

1.2.3 To show integration with other languages

In [28]:

```
# optim() in R calls C programs, run optim to see source code.
# optim
```

1.3 Time your program in R

```
    - proc.time(), system.time()
    - microbenchmark()
```

Illustrations of R rules for efficiency.

Example Calculate the square root of 1 to 1,000,000 using three different operations:

1. Vectorized

In [29]:

```
# Vectorized operation
t <- system.time( x1 <- sqrt(1:1000000) )
head(x1)
```

1 1.4142135623731 1.73205080756888 2 2.23606797749979 2.44948974278318

2. For loop with memory pre-allocation

In [30]:

```
# We can do worse
# For loop with memory pre-allocation
x2 <- rep(NA, 1000000)
t0 <- proc.time()
for (i in 1:1000000) {
    x2[i] <- sqrt(i)
}
t1 <- proc.time()

identical(x1, x2) # Check whether results are the same
```

TRUE

3. For loop without memory pre-allocation

In [31]:

```
# Even worse
# For loop without memory pre-allocation
x3 <- NULL
t2 <- proc.time()
for (i in 1:1000000) {
    x3[i] <- sqrt(i)
}
t3 <- proc.time()

identical(x2, x3) # Check whether results are the same
```

TRUE

In [32]:

```
# As we can see, R is not very good with loops.
t; t1 - t0; t3 - t2
# ?proc.time
```

```
   user  system elapsed 
  0.006   0.005   0.011 

   user  system elapsed 
  0.071   0.004   0.076 

   user  system elapsed 
  0.294   0.072   0.369 
```

How did R execute these three sets of codes?

The better we know how programming languages work, how computers work in general, the better codes we can write.

1. Vectorized

```
x1 <- sqrt(1:1000000)

- sqrt 1, sqrt 2, ..., sqrt 1e6
- Save everything in x1 and put it in memory.
```

2. For loop with memory pre-allocation

```
x2 <- rep(NA, 1000000)
for (i in 1:1000000) { x2[i] <- sqrt(i) }

- Make a vector x2 of length 1e6 and set all elements to NA.
- Put it in memory.
- Setup for loop.
- 1st step
    - Find x2 in memory
    - Change the 1st element to sqrt 1
    - Put new x2 back in memory, delete old x2
- 2nd step
    - Find x2 in memory
    - Change the 2nd element to sqrt 2
    - Put new x2 back in memory, delete old x2
- ...
- - 1e6th step
    - Find x2 in memory
    - Change the 1e6th element to sqrt 1e6
    - Put new x2 back in memory, delete old x2
```

3. For loop without memory pre-allocation

```
x3 <- NULL
for (i in 1:1000000) { x3[i] <- sqrt(i) }

- Make an empty object x3 (NULL has length 0)
- Put it in memory
- Setup for loop.
- 1st step
    - Find x3 in memory
    - Change the 1st element to .., wait x3 has length 0
    - Make a new x3 that has length 1
    - Change the 1st element to sqrt 1
    - Put new x3 back in memory.., wait
      The memory allocated for old x3 is not enough for new x3
    - Find some new space in memory for new x3
    - Put new x3 back in memory, delete old x3
- 2nd step
    - Find x3 in memory
    - Change the 2nd element to .., wait x3 has length 1
    - Make a new x3 that has length 2
    - Copy the old x3 and paste as the first 1 element of new x3
    - Change the 2nd element to sqrt 2
    - Put new x3 back in memory.., wait
      The memory allocated for old x3 is not enough for new x3
    - Find some new space in memory for new x3
    - Put new x3 back in memory, delete old x3
- ...
- 1e6th step
    - Find x3 in memory
    - Change the 1e6th element to .., wait x3 has length 999999
    - Make a new x3 that has length 1e6
    - Copy the old x3 and paste as the first 999999 elements of new x
3
    - Change the 1e6th element to sqrt 1e6
    - Put new x3 back in memory.., wait
      The memory allocated for old x3 is not enough for new x3
    - Find some new space in memory for new x3..
    - Put new x3 back in memory, delete old x3
```

As a result, there will not be a lot of loops in this workshop.

However, I still use the third one sometimes.

- Speed is not always my major concern. Especially if I am only executing the code once. Or I am working on reasonably sized data and/or fairly inexpensive computations.
- Typing takes time too. Compare NULL vs. rep(NA, 1000000)

```
- capslock, n, u, l, l
- r, e, p, shift + 9, shift + n, shift + a, , , 1000000, shift + 0
```

- Thinking takes time as well. Loop is more intuitive. Sometimes I have to think to get the size of the result object because it can be matrices, arrays, etc.

```
- matrix(rep(NA, n * p), nrow = n)
```

Take-home message

- Use vectorized operations rather than loops for speed.
- Balance between speed, your need for speed, your own laziness, etc.., based on what you are doing.

In [33]:

```
# microbenchmark runs the code multiple times and take a summary
library(microbenchmark)
result <- microbenchmark(sqrt(1:1000000),
                         for (i in 1:1000000) {x2[i] <- sqrt(i)},
                         unit = "s", times = 20
                        )
summary(result)
# Result in seconds
```

A data.frame: 2 × 8

expr	min	lq	mean	median	uq	max	ne
<fct>	<dbl>	<dbl>	<dbl>	<dbl>	<dbl>	<dbl>	<dl
sqrt(1:1e+06)	0.003910112	0.004903612	0.007846455	0.005464531	0.01071618	0.01427238	
for (i in 1:1e+06) { x2[i] <- sqrt(i) }	0.059107341	0.059319024	0.069171916	0.059795169	0.07439276	0.11318731	

Example Calculate the square root using sqrt( ) vs. our own implementation.

In [34]:

```
# Use well-developped R functions
result <- microbenchmark(sqrt(500),
                         500^0.5,
                         unit = "ns", times = 1000
                        )
summary(result)
# Result in nanoseconds
```

A data.frame: 2 × 8

expr	min	lq	mean	median	uq	max	neval
<fct>	<dbl>	<dbl>	<dbl>	<dbl>	<dbl>	<dbl>	<dbl>
sqrt(500)	118	152	176.687	168	185	1583	1000
500^0.5	209	263	309.043	291	319	11331	1000

In summary, keep the rules in mind, know what you want to do, test your program, time your program.

2. Efficient coding

R has many powerful and useful functions that we can use to achieve efficient coding and computing.

2.1 Powerful functions in R

Let's play with some data.

In [35]:

```
data <- read.csv("https://raw.githubusercontent.com/ly129/MiCM/master/sample.csv
", header = TRUE)
head(data, 10)
```

A data.frame: 10 × 8

X	Sex	Wr.Hnd	NW.Hnd	Pulse	Smoke	Height	Age
<int>	<fct>	<dbl>	<dbl>	<int>	<fct>	<dbl>	<dbl>
1	Male	21.4	21.0	63	Never	180.00	19.000
2	Male	19.5	19.4	79	Never	165.00	18.083
3	Female	16.3	16.2	44	Regul	152.40	23.500
4	Female	15.9	16.5	99	Never	167.64	17.333
5	Male	19.3	19.4	55	Never	180.34	19.833
6	Male	18.5	18.5	48	Never	167.00	22.333
7	Female	17.5	17.0	85	Heavy	163.00	17.667
8	Male	19.8	20.0	NA	Never	180.00	17.417
9	Female	13.0	12.5	77	Never	165.00	18.167
10	Female	18.5	18.0	75	Never	173.00	18.250

In [36]:

```
summary(data)
```

```
       X                Sex          Wr.Hnd          NW.Hnd            Pu
lse
 Min.   :  1.00   Female:47   Min.   :13.00   Min.   :12.50   Min.
: 40.00
 1st Qu.: 25.75   Male  :53   1st Qu.:17.50   1st Qu.:17.45   1st Qu
.: 50.25
 Median : 50.50               Median :18.50   Median :18.50   Median
: 71.50
 Mean   : 50.50               Mean   :18.43   Mean   :18.39   Mean
: 69.90
 3rd Qu.: 75.25               3rd Qu.:19.50   3rd Qu.:19.52   3rd Qu
.: 84.75
 Max.   :100.00               Max.   :23.20   Max.   :23.30   Max.
:104.00
                                                              NA's
:6
   Smoke        Height          Age
 Heavy: 6   Min.   :152.0   Min.   :16.92
 Never:79   1st Qu.:166.4   1st Qu.:17.58
 Occas: 5   Median :170.2   Median :18.46
 Regul:10   Mean   :171.8   Mean   :20.97
            3rd Qu.:179.1   3rd Qu.:20.21
            Max.   :200.0   Max.   :73.00
            NA's   :13
```

a1. Calculate the mean writing hand span of all individuals

```
mean(x, trim = 0, na.rm = FALSE, ...)
```

In [ ]:

```
```

a2. Calculate the mean height of all individuals, exclude the missing values

In [ ]:

```
```

In [ ]:

```
```

a3. Calculate the mean of all continuous variables

```
apply(X, MARGIN, FUN, ...)
```

In [37]:

```
cts.var <- sapply(X = data, FUN = is.double) # We'll talk about sapply later.
cts <- data[ , cts.var]
head(cts)
```

A data.frame: 6 × 4

Wr.Hnd	NW.Hnd	Height	Age
<dbl>	<dbl>	<dbl>	<dbl>
21.4	21.0	180.00	19.000
19.5	19.4	165.00	18.083
16.3	16.2	152.40	23.500
15.9	16.5	167.64	17.333
19.3	19.4	180.34	19.833
18.5	18.5	167.00	22.333

In [ ]:

```
```

b1. Calculate the count/proportion of females and males

```
table(...,
  exclude = if (useNA == "no") c(NA, NaN),
  useNA = c("no", "ifany", "always"),
  dnn = list.names(...), deparse.level = 1)

prop.table()
```

In [ ]:

```
```

In [ ]:

```
```

b2. Calculate the count in each Smoke group

In [ ]:

```

```

b3. Calculate the count of males and females in each Smoke group

In [ ]:

```

```

In [ ]:

```

```

c1. Calculate the standare deviation of writing hand span of females

```
aggregate()
tapply()
by()
```

In [ ]:

```

```

In [ ]:

```

```

In [ ]:

```

```

In [ ]:

```

```

In [38]:

```
# Return a list using tapply()
```

aggregate( ), by( ) and tapply( ) are all connected. They give different types of output.

c2. Calculate the standard deviation of writing hand span of all different Sex-Smoke groups

In [ ]:

In [ ]:

c3. Calculate the standard deviation of writing hand and non-writing hand span of all Sex-Smoke groups

In [ ]:

Let's try to figure out what aggregate( ) is doing

```
print()
```

In [ ]:

Exercise.

1. Repeat b1-b3 using aggregate( )

In [ ]:

1. Make histograms of writing hand span for all eight Sex-Smoke groups using aggregate( )

In [ ]:

d1. Categorize 'Age' - make a new binary variable 'Adult'

```
ifelse(test, yes, no)
```

In [39]:

```
adult <- 18

head(data)
```

A data.frame: 6 × 8

X	Sex	Wr.Hnd	NW.Hnd	Pulse	Smoke	Height	Age
<int>	<fct>	<dbl>	<dbl>	<int>	<fct>	<dbl>	<dbl>
1	Male	21.4	21.0	63	Never	180.00	19.000
2	Male	19.5	19.4	79	Never	165.00	18.083
3	Female	16.3	16.2	44	Regul	152.40	23.500
4	Female	15.9	16.5	99	Never	167.64	17.333
5	Male	19.3	19.4	55	Never	180.34	19.833
6	Male	18.5	18.5	48	Never	167.00	22.333

R has if (test) {opt1} else {opt2}, what is the advantage of ifelse( )?

In [40]:

```
if (data$Age >= 18) {
    data$Adult2 = "Yes"
} else {
    data$Adult2 = "No"
}
head(data)
```

```
Warning message in if (data$Age >= 18) {:
“the condition has length > 1 and only the first element will be use
d”
```

A data.frame: 6 × 9

X	Sex	Wr.Hnd	NW.Hnd	Pulse	Smoke	Height	Age	Adult2
<int>	<fct>	<dbl>	<dbl>	<int>	<fct>	<dbl>	<dbl>	<chr>
1	Male	21.4	21.0	63	Never	180.00	19.000	Yes
2	Male	19.5	19.4	79	Never	165.00	18.083	Yes
3	Female	16.3	16.2	44	Regul	152.40	23.500	Yes
4	Female	15.9	16.5	99	Never	167.64	17.333	Yes
5	Male	19.3	19.4	55	Never	180.34	19.833	Yes
6	Male	18.5	18.5	48	Never	167.00	22.333	Yes

In [41]:

```
# Delete Adult2
data <- subset(data, select=-c(Adult2))
```

ifelse( ) is vectorized!!!

d2. Categorize 'Wr.Hnd' into 5 groups - make a new categorical variable with 5 levels

```
1. =< 16: Stephen Curry
2. 16~18: Drake
3. 18~20: Fred VanVleet
4. 20~22: Jeremy Lin
5. >  22: Kawhi Leonard
```

Can we still use ifelse( )?

```
cut(x, breaks, labels = NULL, right = TRUE, ...)
```

In [42]:

```
cut.points <- c(0, 16, 18, 20, 22, Inf)
# labels as default
```

In [43]:

```
# Set labels to false
```

In [44]:

```
# Customized labels
label <- c("Curry", "Drake", "VanVleet", "Lin", "Leonard")
```

e1. Calculate the mean Wr.Hnd span of each Hnd.group

In [ ]:

```

```

e2. Calcuate the mean Wr.Hnd span of each Hnd.group without using aggregate, by, tapply

```
split(x, f, ...)
lapply(X, FUN, ...)
sapply(X, FUN, ..., simplify = TRUE)
```

In [45]:

```
cut.points <- c(0, 16, 18, 20, 22, Inf)
```

In [46]:

```
# lapply
```

In [47]:

```
# sapply
# See what simplify does
```

In [48]:

```
# vapply *
# Safer than sapply(), and a little bit faster
# because FUN.VALUE has to be specified that length and type should match
# Any idea why it can be a little bit faster? Recall...
# va <- vapply(Wr.Hnd.grp, summary, FUN.VALUE = c("Min." = numeric(1),
#                                                  "1st Qu." = numeric(1),
#                                                  "Median" = numeric(1),
#                                                  "lalalalala" = numeric(1),
#                                                  "3rd Qu." = numeric(1),
#                                                  "Max." = numeric(1)))
# va
```

f. Calculate the 95% sample confidence intervals of Wr.Hnd in each Smoke group.

One variable for lower bound and one variable for upper bound.

$$CI = \bar{x} \pm t_{n-1,0.025} \times \sqrt{\frac{s^2}{n}}$$

where $\bar{x}$ is the sample mean and s^2 is the sample variance.

In [49]:

```
# aggregate(Wr.Hnd~Smoke, data = data, FUN = ...)
# tapply(X = data$Wr.Hnd, INDEX = list(data$Smoke), FUN = ...)
```

Unfortunately, I do not know any function in R that does this calculation.

But I know how to do it step by step.

In [50]:

```
sample.mean <- NULL
sample.sd <- NULL
n <- 10
t <- qt(p = 0.025, df = n - 1, lower.tail = FALSE)
lb <- sample.mean - t * sample.sd / sqrt(n)
ub <- sample.mean + t * sample.sd / sqrt(n)

# How many times did we aggregate according to the group? Can on aggregate only
once?
```

Or, we can make our own function and integrate it into aggregate( ), by( ), or tapply( ) !!!

2.2 Write our own functions in R

A function takes in some inputs and gives outputs

In [51]:

```
# The structure
func_name <- function(argument){
    statement
}
```

Example 1. Make a function for $f(x) = 2x$

In [52]:

```
# Build the function
times2 <- function(x) {
    fx = 2 * x
    return(fx)
}
# Use the function
times2(x = 5)
# or
times2(5)
```

10

10

Example 2. Make a function to calculate the integer division of a by b, return the integer part and the modulus.

In [53]:

```
# R has operators that do this
9 %/% 2
9 %% 2
```

4

1

```
floor( ) takes a single numeric argument x and returns a numeric vector c
ontaining the largest integers not greater than the corresponding elements
of x.
```

In [54]:

```
int.div <- function(){

}
```

In [55]:

```
# class(result)
# Recall: how do we access the modulus?
```

Example 3. Make the simplest canadian AI chatbot

In [56]:

```
# No need to worry about the details here.
# Just want to show that functions do not always have to return() something.
AIcanadian <- function(who, reply_to) {
    system(paste("say -v", who, "Sorry!"))
}
# AIcanadian("Alex", "Sorry I stepped on your foot.")
```

In [57]:

```
# Train my chatbot - AlphaGo style.
# I'll let Alex and Victoria talk to each other.
# MacOS has their voices recorded.
chat_log <- rep(NA, 8)
# for (i in 1:8) {
#     if (i == 1) {
#         chat_log[1] <- "Sorry I stepped on your foot."
#         system("say -v Victoria Sorry, I stepped on your foot.")
#     } else {
#         if (i %% 2 == 0)
#             chat_log[i] <- AIcanadian("Alex", chat_log[i - 1])
#         else
#             chat_log[i] <- AIcanadian("Victoria", chat_log[i - 1])
#     }
# }
# chat_log
```

Example 4. Check one summary statistic by Smoke group of our 'data' data.

Function arguments can be basically anything, say another function.

In [58]:

```
data_summary <- function(func) {
    data <- read.csv("https://raw.githubusercontent.com/ly129/MiCM/master/sample
.csv", header = TRUE)
    by(data = data$Wr.Hnd, INDICES = list(data$Smoke), FUN = func)
}
data_summary(quantile)
```

```
: Heavy
   0%   25%   50%   75%  100% 
14.00 17.20 17.50 20.35 23.20 
------------------------------------------------------------ 
: Never
   0%   25%   50%   75%  100% 
13.00 17.50 18.50 19.35 22.00 
------------------------------------------------------------ 
: Occas
  0%  25%  50%  75% 100% 
15.4 16.5 19.0 19.1 22.2 
------------------------------------------------------------ 
: Regul
    0%    25%    50%    75%   100% 
16.300 18.125 19.600 20.375 22.500 
```

Exercise: make a function to calculate sample confidence intervals (2.1 f)

In [59]:

```
# sample.mean <- NULL
# sample.sd <- NULL
# n <- NULL
# t <- qt(p = 0.025, df = n - 1, lower.tail = FALSE)
# lb <- sample.mean - t * sample.sd / sqrt(n)
# ub <- sample.mean + t * sample.sd / sqrt(n)

sample_CI <- function(x) {
}

aggregate(Wr.Hnd~Smoke, data = data, FUN = sample_CI)
```

A data.frame: 4 × 2

Smoke	Wr.Hnd
<fct>	<list>
Heavy	NULL
Never	NULL
Occas	NULL
Regul	NULL

3. Efficient computing

We often want to minimize the resources used to do certain computation.

Time is usually the most important resource.

Other resources are relatively less important.

3.1 Parallel computing

When multiple tasks are independent of each other. We can use (up to) all the CPU cores at the same time to do the tasks simultaneously. Check CPU usage if interested.

In [60]:

```
library(parallel)
detectCores()
```

4

In [61]:

```
mat.list <- sapply(c(1, 5, 200, 250, 1800, 2000), diag)
print(head(mat.list, 2)) # print() makes the output here look the same as in R/R
studio
```

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

[[2]]
     [,1] [,2] [,3] [,4] [,5]
[1,]    1    0    0    0    0
[2,]    0    1    0    0    0
[3,]    0    0    1    0    0
[4,]    0    0    0    1    0
[5,]    0    0    0    0    1
```

Here we compare lapply( ) and its multi-core version mclapply( ) and parLapply( ) in the 'parallel' package.

```
- lapply()
- mclapply(..., mc.preschedule = TRUE)   # without load balancing
- mclapply(..., mc.preschedule = FALSE)  # with load balancing
```

mcapply( ) sets up a pool of mc.cores workers just for this computation

mclapply( ) is not available on Windows. For those of you using Windows computers

https://www.apple.com/ca/mac/ (https://www.apple.com/ca/mac/) or https://ubuntu.com (https://ubuntu.com) or

```
- parLapply                              # without load balancing
- parLapplyLB                            # with load balancing
```

To use parLapply( ), we need to set up a cluster, and we need to close the cluster after we are done. The good part is that we can put several parLapply( ) calls within the cluster.

In [62]:

```
system.time(
    sc <- lapply(mat.list, solve)
)
```

```
   user  system elapsed 
  2.919   0.054   2.978 
```

In [63]:

```
system.time(
    mc <- mclapply(mat.list, solve, mc.preschedule = TRUE, mc.cores = 3)
)
```

```
   user  system elapsed 
  1.292   0.167   1.861 
```

In [64]:

```
system.time(
    mc <- mclapply(mat.list, solve, mc.preschedule = FALSE, mc.cores = 3)
)
```

```
   user  system elapsed 
  1.291   0.195   1.878 
```

In [65]:

```
t <- proc.time()
cl <- makeCluster(3) # Use 3 cores
pl <- parLapply(cl = cl, X = mat.list, fun = solve)
stopCluster(cl)
proc.time() - t
```

```
   user  system elapsed 
  0.386   0.090   4.197 
```

In [66]:

```
t <- proc.time()
cl <- makeCluster(3)
pl <- parLapplyLB(cl = cl, X = mat.list, fun = solve)
stopCluster(cl)
proc.time() - t
```

```
   user  system elapsed 
  0.499   0.121   3.490 
```

In [67]:

```
# Two parallel calls within one cluster.
t <- proc.time()
cl <- makeCluster(3)
pl_nb <- parLapply(cl = cl, X = mat.list, fun = solve)
pl_lb <- parLapplyLB(cl = cl, X = mat.list, fun = solve)
stopCluster(cl)
proc.time() - t
# This takes shorter than the sum of the previous two. Why?
```

```
   user  system elapsed 
  0.868   0.154   7.513 
```

For both mclapply( ) and parLapply( ), setting up parallel computing takes time (overhead).

In [68]:

```
t <- proc.time()
cl <- makeCluster(3)
stopCluster(cl)
proc.time() - t
```

```
   user  system elapsed 
  0.010   0.006   0.720 
```

Load-balancing is tricky.

"Load balancing is potentially advantageous when the tasks take quite dissimilar amounts of computation time, or where the nodes are of disparate capabilities."

If 1000 tasks need to be allocated to 10 nodes (CPUs, cores, etc.)

- without load-balancing, 100 tasks are sent to each of the nodes.
- with load-balancing, tasks are sent to a node one at a time. Overhead is high.

Take-home message

- Parallel computing exists in R
- They are not faster than non-parallel computing by a factor of number of cores used.
- R is still slow.

The solution is lower-level programming languages.

- C
- C++
- Fortran

3.2 Integration with C++

The 'Rcpp' package

```
- "Seamless R and C++ Integration"
```

1. Install 'Rcpp' package in R
2. Install compiler

   - Windows: Rstudio should ask you to install Rtools when you source your cpp code.
     - If not, you can download and install Rtools on the exact same webpage where you downloaded R.
   - MacOS:
     - Install XCode Command Line Tools. Open Terminal, paste and run

       ```
       xcode-select --install
       ```

     - Install gfortran-6.1 binary and clang compiler (also on the exact same webpage as the R download)

       [https://cran.r-project.org/bin/macosx/tools/ (https://cran.r-project.org/bin/macosx/tools/)](https://cran.r-project.org/bin/macosx/tools/)

   "Setup is extra work on macOS, but it is above our pay grade to change that." - Dirk Eddelbuettel [https://github.com/RcppCore/RcppArmadillo/issues/249 (https://github.com/RcppCore/RcppArmadillo/issues/249)](https://github.com/RcppCore/RcppArmadillo/issues/249)
3. File -> New File -> C++ File
4. Code C++
   - Try not to forget ';' at the end of lines;
   - Every object that has ever appeared has to be defined;
   - Have to use loops to do a lot of things such as matrix calculations (not slow though);

Example 1. Create an R function that calculates the square root of vectors in C++.

In [69]:

```
library(Rcpp)
sourceCpp("sqrt_cpp.cpp")
square_root(1:4)
# We return a NumericVector in the .cpp file. So we get an R vector.
```

1 1.4142135623731 1.73205080756888 2

The addition of the 'RcppArmadillo' package

```
- "Armadillo is a C++ linear algebra library aiming towards a good balan
ce between speed and ease of use."
```

http://arma.sourceforge.net (http://arma.sourceforge.net)

Linear algebra in Armadillo http://arma.sourceforge.net/armadillo_joss_2016.pdf (http://arma.sourceforge.net/armadillo_joss_2016.pdf)

In base C++, operations like matrix multiplication requires loops.

Example 2. Create an R function that calculates matrix multiplication in C++.

In [70]:

```
sourceCpp("mm_cpp.cpp")
```

In [71]:

```
# Now we can call the function using the name defined in the .cpp file
set.seed(20190813)
a <- matrix(rnorm(100000), ncol = 50000)  # 2 x 50000 matrix
b <- matrix(rnorm(200000), nrow = 50000)  # 50000 x 4 matrix

mat_mul(a, b)
# We return an Rcpp::List in the .cpp file. So we get an R list here.
# mat_mul(b, a)
```

$MatrixMultiplication

A matrix: 2 × 4 of type dbl

-345.3068	359.6366	-54.33261	-182.1485
-190.4709	85.7216	-330.53902	121.1807

$rows

2

$cols

4

In [72]:

```
bchmk <- microbenchmark(a %*% b,
                        mat_mul(a, b),
                        unit = "us", times = 100
                       )
summary(bchmk)
```

A data.frame: 2 × 8

expr	min	lq	mean	median	uq	max	neval
<fct>	<dbl>	<dbl>	<dbl>	<dbl>	<dbl>	<dbl>	<dbl>
a %*% b	738.83	743.7275	775.0459	754.637	786.785	973.247	100
mat_mul(a, b)	586.69	588.0115	614.3580	590.760	614.089	1637.623	100

In [73]:

```
# Here we make an R function that calls the C++ function
mmc <- function(a, b) {
    result <- mat_mul(a, b)$MatrixMultiplication
    return(result)
}
mmc(a, b)
```

A matrix: 2 × 4 of type dbl

-345.3068	359.6366	-54.33261	-182.1485
-190.4709	85.7216	-330.53902	121.1807

In [74]:

```
# Another way to do this. Here you do not need to have a separate .cpp file.
# A naive .cpp function is made here.
library(RcppArmadillo)
cppFunction(depends = "RcppArmadillo",
            code = 'arma::mat mm(arma::mat& A, arma::mat& B){
                        return A * B;
                    }'
)
```

In [75]:

```r
mm(a, b)
# mm(b, a)
```

A matrix: 2 × 4 of type dbl

-345.3068	359.6366	-54.33261	-182.1485
-190.4709	85.7216	-330.53902	121.1807

In [76]:

```r
# We can wrap this naive function in an R function to manipulate input and outpu
t in R
mmc2 <- function(A, B) {
    if (ncol(A) == nrow(B)) {
        return(mm(A, B))
    } else {
        stop("non-conformable arguments")
    }
}
mmc2(a, b)
# mmc2(b, a)
```

A matrix: 2 × 4 of type dbl

-345.3068	359.6366	-54.33261	-182.1485
-190.4709	85.7216	-330.53902	121.1807

3.3 Integration with Fortran

- Old but even faster
  - We will see that Fortran sacrifices a LOT for speed.
- Require a compiler too
  - MacOS
    - You might have noticed that we have installed a compiler called gfortran
  - Windows
    - A lot of work. See "Fortran_Setup_Win.txt". [https://github.com/ly129/MiCM/blob/master/Fortran_Setup_Win.txt (https://github.com/ly129/MiCM/blob/master/Fortran_Setup_Win.txt)](https://github.com/ly129/MiCM/blob/master/Fortran_Setup_Win.txt)
- R can call fortran subroutines
  - Basically a kind of function
  - Through a .so file (Shared Object, MacOS) and a .dll file (Dynamic Link Library, Windows)
    - MacOS Terminal

      ```
      cd file_path
      R CMD SHLIB -o file_name.so file_name.f90
      ```

    - Windows Command Prompt

      ```
      cd file_path
      gfortran -shared -o file_name.dll file_name.f90
      ```

- Requires pointers for communication between programs
  - Pointers point to a locations on the computer's memory
    - Two different programs cannot share data directly.
  - The same applies to C++. 'Rcpp' package handles it for us.
  - Test your program in Fortran. The communication between programs make it hard to debug.
    - Make a Fortran program. Then in terminal/command prompt, type

      ```
      cd file_path
      gfortran file_name.f90
      ./a.out (MacOS) | a.exe (Windows)
      ```

In [77]:

```
set.seed(20190813)

ra <- 2
ca <- 4
rb <- 4
cb <- 3

A <- matrix(rnorm(ra*ca), nrow = ra)
B <- matrix(rnorm(rb*cb), nrow = rb)

A; B
```

A matrix: 2 × 4 of type dbl

-0.7265744	0.3488403409	-1.56818199	0.47863529
-1.0882637	-0.0002502522	0.03957236	0.01071728

A matrix: 4 × 3 of type dbl

-3.1124977	1.1331331	-0.1618750
-0.4561366	-0.6835550	0.6946241
-1.5031855	0.1793388	-0.5950162
0.5016549	0.3443689	-0.1799074

In [78]:

```
# Load the executable .so file (MacOS) or .dll file (Windows)
dyn.load("mm_for.so")
```

In [79]:

```
# Check whether the "mat_mul_for" function is loaded into R
is.loaded("mat_mul_for")
```

TRUE

In [80]:

```
result <- .Fortran("mat_mul_for",
                   A = as.double(A),
                   B = as.double(B),
                   AB = double(ra * cb),  # note the difference here
                   RowA = as.integer(ra),
                   ColA = as.integer(ca),
                   RowB = as.integer(rb),
                   ColB = as.integer(cb),
                   RowAB = as.integer(ra),
                   ColAB = as.integer(cb)
)
result
class(result)
```

$A

-0.72657435955062   -1.08826373174186   0.348840340876   -0.000250252212517976
-1.56818199047583   0.0395723591446798   0.478635292694497   0.0107172814771946

$B

-3.11249770245218   -0.456136568911746   -1.5031855452531   0.501654927877872
1.13313311393103   -0.68355497968992   0.179338752934091   0.344368887483415
-0.161874950719064   0.694624084455543   -0.595016170289594   -0.179907360239921

$AB

4.69972044222275   3.33322429270281   -1.17816571795269   -1.222189054142
1.20691072131186   0.150514495323954

$RowA

2

$ColA

4

$RowB

4

$ColB

3

$RowAB

2

$ColAB

3

'list'

We can wrap it in an R function as well.

In [81]:

```
mmf <- function(A, B) {
    ra <- nrow(A)
    ca <- ncol(A)
    rb <- nrow(B)
    cb <- ncol(B)

    if (ca == rb) {
        result <- .Fortran("mat_mul_for",
                           A = as.double(A),
                           B = as.double(B),
                           AB = double(ra * cb),
                           RowA = as.integer(ra),
                           ColA = as.integer(ca),
                           RowB = as.integer(rb),
                           ColB = as.integer(cb),
                           RowAB = as.integer(ra),
                           ColAB = as.integer(cb)
                           )
        mm <- matrix(result$AB, nrow = result$RowAB, byrow = F)
    } else {
        stop('non-conformable arguments')
    }
    return(list(Result = mm,
                Dimension = c(result$RowAB, result$ColAB)
               )
          )
}
```

In [82]:

```
set.seed(20190813)

ra <- 2
ca <- 50000
rb <- 50000
cb <- 3

A <- matrix(rnorm(ra*ca), nrow = ra)
B <- matrix(rnorm(rb*cb), nrow = rb)

mmf(A, B)
```

$Result

A matrix: 2 × 3 of type dbl

-345.3068	359.6366	-54.33261
-190.4709	85.7216	-330.53902

$Dimension

2 3

In [83]:

```
A %*% B
```

A matrix: 2 × 3 of type dbl

-345.3068	359.6366	-54.33261
-190.4709	85.7216	-330.53902

4. Exercises

Make a function in R using R and/or Rcpp and/or R call Fortran that does

Level 1 Integer division of two integers using a loop

```
    I have 9 dollars to buy donuts for my colleagues. The donuts are 2 dol
lars each.
```

$$\begin{array}{cc} 9 > 2 \rightarrow 9 - 2 = 7 & 1 \text{ donut} \\ 7 > 2 \rightarrow 7 - 2 = 5 & 2 \text{ donuts} \\ 5 > 2 \rightarrow 5 - 2 = 3 & 3 \text{ donuts} \\ 3 > 2 \rightarrow 3 - 2 = 1 & 4 \text{ donuts} \\ \multicolumn{2}{c}{1 < 2 \rightarrow \text{stop}} \end{array}$$

In [84]:

```
# Something like this.
9 %/% 2; 9%%2
```

4

1

Level 2 Element-wise integer division for two integer vectors

In [85]:

```
# Something like this.
c(15, 14, 13, 12) %/% c(6, 5, 4, 3)
c(15, 14, 13, 12) %% c(6, 5, 4, 3)
```

2 2 3 4

3 4 1 0

If you like loops, loops in C++ and Fortran are fast.

Level 3 Linear regression

The formula for the point estimates is

$$\boldsymbol{\beta} = (\boldsymbol{X}^\top \boldsymbol{X})^{-1} \boldsymbol{X}^\top \boldsymbol{Y}$$

- matrix transpose in R: t(X)
- matrix inverse in R: solve(X)
- matrix-matrix and matrix-vector mulplication in R: X %*% Y

In [86]:

```
# If you enter the right X and Y in your function, you should get the following
result
lm(Wr.Hnd~NW.Hnd+Age, data = data)
```

```
Call:
lm(formula = Wr.Hnd ~ NW.Hnd + Age, data = data)

Coefficients:
(Intercept)       NW.Hnd          Age
     1.1109       0.9535      -0.0103
```

Level 4 Gradient descent to calculate the minimum value of a given function, with user-supplied gradient function.

- Gradient descent is an iterative algorithm therefore we have to use loops. Here we can really see the speed advantage of C++ and Fortran over R.

In [87]:

```
# Something like this, both inputs are R functions.
GD <- function(objective_function, gradient_function, initial_value) {
    statements
}
```

Level 5 Specific task in your own research.

In [ ]:

```

```