List and Functional Programming

Song Liu (song.liu@bristol.ac.uk)

GA 18, Fry Building,

Microsoft Teams (search "song liu").

So far, we have talked about

1. R is an interpreted language.
    - Interpreted vs. Compiled Language
    - Pros and Cons
2. Scalar compute in R (Mostly similar to C)
    - Scalar types: numeric, logical, character
    - Arithmetic operators: `+,-,*,/`
    - Logical operators: `&&, ||`
    - Flow control: `if-else` and `if-else if` ladder.
    - Loops: `for`, `while`
    - Function definition.
    - Scalar compute in R is slow.

So far, we have talked about

3. Vector compute in R
   - Vector compute commands are translated into SIMD instructions, thus is highly efficient.
   - Vector construction
   - Vectors are always passed by value.
   - Elementwise vector operators (`+,-,*,/,|,&`)
     - vector in, vector out.
   - Indexing vectors, `:` Symbol.
4. Matrix compute in R
   - Matrix construction
   - Elementwise matrix operators (`+,-,*,/,|,&`)
     - matrix in, matrix out.
   - Matrix multiplication/transposition/inversion.

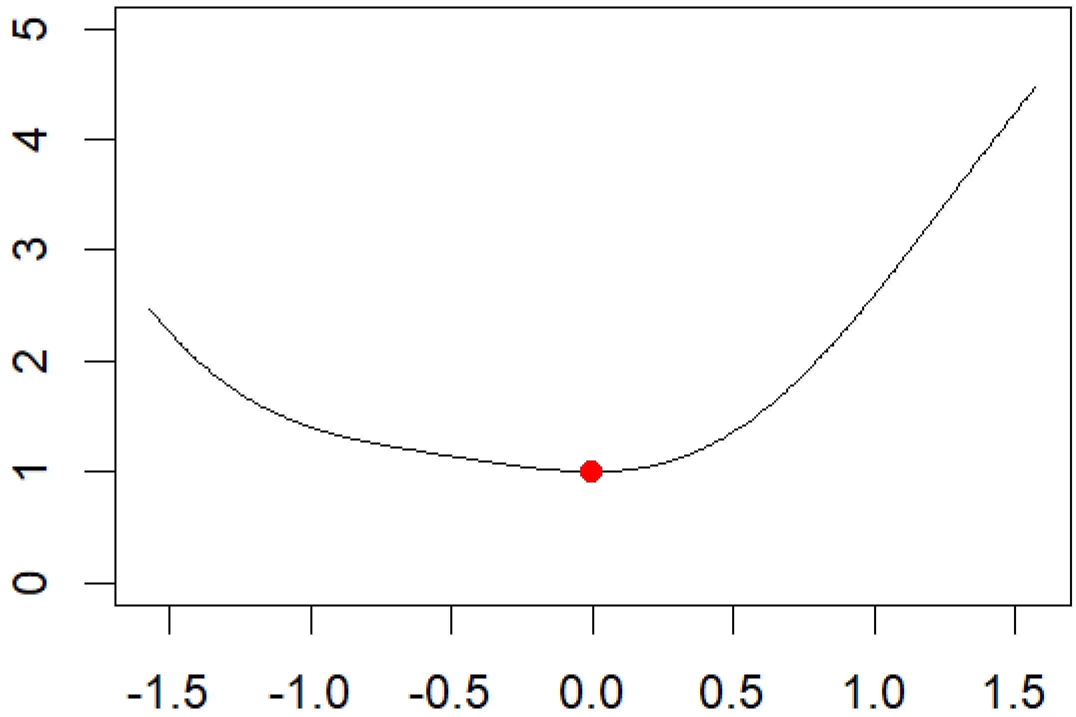
So far, we have talked about

5. Levels of Vectorization
    - Vectorization can happen at different levels.
    - Vectorization at different levels will have different performance impact.
    - **Matrix multiplication** with three different levels of vectorization and compare speed.
    - `pdist` **function** with three different levels of vectorization and compare speed.

So far, we have talked about

6. Other Vector/Matrix operators
    - Vector/Matrix concatenation
    - Insertion and deletion of vector element
        - Insertion and deletion by reassignments
    - Recycling
    - `apply`
7. Graphics
    - `plot`
    - `points`
    - `lines`

Demo: Gradient Descent Visualized.



List

- List is another important data structure in R.
- It combines objects **with different types**.
  - Matrix/vector only supports a single type of data.
  - Similar to the `struct` in C programming.
- See ART: Section 4.

Creating List

```
song_liu <- list(name = "song", male = T, salary = 10)
```

- This creates a list contains three elements:
  - Character data: `name`
  - Logic data: `male`
  - Numeric data: `salary`
- `name`, `male` and `salary` are called "tags" for values `"song", "T", 10` respectively.

Indexing List

You can list all tags and their corresponding values by simply typing the name of the list at the console.

```
> song_liu
$name
[1] "song"

$male
[1] TRUE

$sal
[1] 10
```

Indexing List

- To obtain the value bound to a specific tag, we can use the `$` sign.

  ```
  song_liu$male
  [1] TRUE
  ```

- Or you can index a list without using tags:

  ```
  song_liu[[2]]
  [1] TRUE
  ```

  where `[[2]]` is the index of the element.

Creating List without Tags

- In fact, you can create a list without using any tag:

```
song_liu <- list("song", T, 10)
> song_liu
[[1]]
[1] "song"
[[2]]
[1] TRUE
[[3]]
[1] 10
```

- Then you will have to access all elements in the list using their indices.

```
> song_liu[[2]]
[1] TRUE
```

  - You **cannot** use vector to index list:

```
> song_liu[[2:3]]
Error in song_liu[[2:3]] : subscript out of bounds
```

Add Element to List

```
song_liu <- list(name = "song", male = T, sal = 10)
song_liu$department <- "math"
song_liu[[5]] <- 1987
> song_liu
$name
[1] "song"

$male
[1] TRUE

$sal
[1] 10

$department
[1] "math"

[[5]]
[1] 1987
```

Delete Element from List

```
song_liu$department <- NULL
> song_liu
$name
[1] "song"
$male
[1] TRUE
$sal
[1] 10
[[4]]
[1] 1987
```

- Notice that after deleting `department`, the value 1987 moved up by one position, with a new tag `4`.
- In R, all modifications to an existing vector/list involves creating a modified copy of the old vector/list, and reassign it to the original variable.
  - Extra memory allocation!

Nested List

List itself can contain lists.

```
song_liu <- list(name = "song", male = T, sal = 10,
                 grades = list(CPP=90, MAT = 100, CALC = 70))
> song_liu$grades$CPP
[1] 90
> song_liu[[4]][[1]]
[1] 90
```

```
jack <- list(name = "jack", male = T, sal = 10,
                 grades = list(CPP=90, MAT = 100, CALC = 70))
students <- list(song_liu, jack)  # Now the list students
# contains two elements, which are both lists.
```

Functional Programming

- So far, we have introduced two programming paradigms
  - **Procedural Programming (PP)**: Your program is divided into several subtasks and you write **functions** for each subtask.
  - **Object Oreinted Programming (OOP)**: Your program is divided into several pieces called "objects" and objects contain **data as well as procedures**.
- PP and OOP divide the program **by features** thus is suitable for developing APPs with complicated logics and components.

Functional Programming

- However, most data science program has a simple programming pipeline:
    - Apply(Op1, Data1) -> Apply(Op1, Data1) -> ... -> Final Result
- Functional Programming (FP) view our program as a pipeline, focusing on writing data-operating functions and applying such functions to our data.
- R supports functional programming natively.
    - C/C++ also supports functional programming via some advanced language features.

A Simple FP Example

- Write a simple data operating function

```
# add the input by 1.
add <- function(x) {return(x+1)}
```

- Applying this function on some dummy data.

```
l <- list(1,2)
lapply(l, add)
[[1]]
[1] 2

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

- Here, `lapply` applies the `add` function to each element of the list `l`, producing a new list.

A Simple FP Example

- We can also convert the list output to a vector by using `unlist`:

```
l <- list(1,2)
unlist(lapply(l, add))
[1] 2 3
```

- or using `sapply`

```
l <- list(1,2)
sapply(l, add)
[1] 2 3
```

Functions are Variables

- In FP, functions are variables too, thus they can be passed to other functions as input arguments.
- In the previous example, `add` is a function that was passed to the `apply` function as an input argument.
- This property allows us to write clean and easily readable code.

Gradient Descent Revisited

```
f <- function (x){
  return(sin(x)^3+x^2+1)
}

df <- function(x){
  return(3*sin(x)^2*cos(x) + 2*x)
}

x <- -1.5
while( abs(df(x)) > .01){
  x <- x - .1*df(x)
}

x
f(x)
```

- This code is clean enough. However, our gradient descent algorithm should not depend on `f` since it is a generic algorithm applies to all functions.

Gradient Descent Revisited

```
f <- function (x){
  return(sin(x)^3+x^2+1)
}

df <- function(x){
  return(3*sin(x)^2*cos(x) + 2*x)
}

# gradient descent, takes two functions as inputs
# f, function to be minimized, df, derivative of f,
# x, initial search point.
grad_desc <- function(f, df, x){
  while( abs(df(x)) > .01){
    x <- x - .1*df(x)
  }
  return(list(x, f(x)))
}

print(grad_desc(f, df, -1.5))
```

Gradient Descent Revisited

- In this example, the initial search point `x` is data and `f` and `df` are data operating functions.
- `grad_desc` tells the program how `f` and `df` are applied to the data and produces the final outcome.

Gradient Descent Revisited

- Since functions are variables, they can be elements of a list too. This leads to a further simplification of the input arguments of `grad_desc`.

```
grad_desc <- function(problem, x){
f <- problem$func
df <- problem$deri

while( abs(df(x)) > .01){
    x <- x - .1*df(x)
}
return(list(x, f(x)))
}

# creating a list with two functions as elements.
problem <- list(func = f, deri = df)
# more readable
print(grad_desc(problem, -1.5))
```

Conclusion

1. List in R can contain data with different types.
2. Lists can be nested.
3. FP focuses on data operating functions and how these functions are applied on data.
4. In FP, functions are variables.