Stat243 PS5

Meng Wang, SID: 21706745

October 18, 2015

1 Problem 1

1.1 (a)

In R, variable $.Machine$ include the information of the numerical characteristics of the machine R is running on. Particularly, the $.Machine\$double.base$ will return us the base and $.Machine\$double.ulp.digits$ return the largest negative integer i such that $1 + double.base^i! = 1$. Let us check out those two numbers in my machine

```
.Machine$double.base

## [1] 2

.Machine$double.ulp.digits

## [1] -52
```

As we can see, the smallest floating number it can reach is $2^{-52} = 2.220446e - 16$. So my machine is 16 digits precision, by 16 I mean, the 15 digit is precise but the 16 digit may not. Since 1.000000000001 is much bigger than $2.220446e - 16$, we could test our results,

```
 num <- 1.000000000001
 options(digits = 16)
 num

## [1] 1.000000000001
```

1.2 (b)

Let see the result first,

```
x <- c(1, rep(1e-16, times = 10000))
options(digits = 20)
sum(x)

## [1] 1.0000000000009996448
```

As we can see from the result, it does give the up to 16 digits accuracy.

1.3 (c)

Write the same code in python, here is the code

```
import numpy as np
####### Problem 1(c) ############
vec = np.array([1e-16]*(10001))
vec[0] = 1.0
print '%.15f' % np.sum(vec)
```

Run the code in terminal. Here is the result

```
 1.000000000000099853459
```

Python returns up to 15 digits precision result.

1.4 (d)

Let us do the sum manually with for loop. First, we put 1 at the first beginning.

```
d1_sum <- 0.0
for(i in 1:length(x)){
  d1_sum = d1_sum + x[i]
}
options(digits = 20)
d1_sum

## [1] 1
```

The result is wrong. It is very easy to see since we know from 1(a) that the smallest number we could add to 1 without being neglected is $2^{-52} = 2.220446e - 16$. If we add 1 as the last,

```
d2_sum <- 0.0
for(i in 1:10000){
  d2_sum = d2_sum + 1e-16
}
d2_sum = d2_sum + 1.0
options(digits = 20)
d2_sum

## [1] 1.0000000000010000889
```

Now the result is correct. Play the same game in Python, here is the code

```
####### Problem 1(d) ############
# use for loop, but add 1 at the beginning
d1_sum = 0.0
d1_sum = d1_sum + 1.0
for i in range(10000):
        d1_sum = d1_sum + 1e-16
print "========== 1d =============="
print "====== Add 1 first ====="
print '%.20f' % d1_sum

# use for loop, add 1 at the end
d2_sum = 0.0
for i in range(10000):
        d2_sum = d2_sum + 1e-16
d2_sum = d2_sum + 1.0
```

```
print "====== Add 1 last ====="
print '%.20f' % d2_sum
```

Here is the result

```
========== 1d =============
====== Add 1 first =====
1.00000000000000000000
====== Add 1 last =====
1.00000000000100008890
```

Python returns the same result. As we discussed, the precision for R and Python are both 16 digits, the reason why it loses digits when we add them one by one, is because the smallest number to add to number 1 is $2^{-52} = 2.220446e-16$, any number smaller than this number will be neglected.

1.5 (e)(f)

Apparently, function *sum*() in R is not just add the number from left to right. It has been optimized to handle the accuracy problem. I checked the source code *summary.c* (https://github.com/wch/r-source/blob/trunk/src/main/summa for function sum in R. It seems when R does the sum operation, it actually uses long double variable to do the sum operation and then truncate the result to double precision (Line 115 - 131, function rsum).

2 Problem 2

Let us see an example of matrix transform. Define two matrix with the same value, one matrix has integer element while the other has double element. Apply the matrix transform on both matrix and see how long each will take. The transform operator can tell us how fast R can move two different kinds of data in the memory.

```
############## Problem 2 ####################
int_A <- matrix(as.integer(13), nrow = 1e4, ncol = 1e4)
double_A <- matrix(as.numeric(13.0), nrow = 1e4, ncol = 1e4)

system.time( t(int_A))
system.time(t(double_A))
```

Here is the result

```
> system.time( t(int_A))
   user  system elapsed
  0.236   0.040   0.277
> system.time(t(double_A))
   user  system elapsed
  0.401   0.068   0.471
```

Another simple test is to find out the max value of the matrix. This can test the speed of numeric comparison of integer and double.

```
> system.time( max(int_A ))
   user  system elapsed
  0.100   0.000   0.099
> system.time( max(double_A) )
   user  system elapsed
  0.406   0.000   0.404
```

It is clear that for basic data operation(copying and moving in the memory, comparison), integer type is much faster than double type.
** R code saved in HW5.R; Python code saved in HW5.py; all codes have been run and tested*