STAT243 Problem set 5

Tianyi Zhang

October 18, 2015

1 Problem 1

1.1 1A

At most 16 digits accuracy stored.

```
1+1e-12

## [1] 1.000000000001000088901

1+(1e-16)*10000

## [1] 1.000000000001000088901
```

1.2 1B

The sum() function gives a different anwser than add 1+1e-12. It has 15 digits accuracy stored.

```
vec_R<-c(1,rep(1e-16,10000))
sum(vec_R)

## [1] 1.000000000000999644811
```

1.3 1C

Python gives a different answer than R, it has 14 digtis accuracy when using the sum function.

```
import numpy as np
import decimal
vec_python=np.array([1e-16]*(10001))
vec_python[0]=1
print decimal.Decimal(np.sum(vec_python))


## 1
```

1.4 1D

In R, after I wrote a for loop to do the summation, I found that putting 1 at the **last** position of the vector will give an accurate result (The maximum digits accuracy that a computer can store, 16 digits). However, putting 1 in the first position will not give a right answer, it will simply return a 1 for the result of the summation.

```
function_R<-function(vec_R){
  a=vec_R[1]
  for (i in 2:length(vec_R)){
    a=a+vec_R[i]
  }
  return(a)
}
## Vec_first has one at its first position, and vec_last has one at the last.
vec_first<-c(1,rep(1e-16,10000))
vec_last<-c(rep(1e-16,10000),1)
function_R(vec_first)

## [1] 1

function_R(vec_last)

## [1] 1.000000000001000088901
```

In Python, when I wrote the loop with the same functionality. It almost yields the same result as R. When I put 1 in the **last** position of the array, the for loop gives the most accurate result (16 digits accuracy). When 1 is in the first position, the answer is wrong,(just 1.0 is returned).

```
import numpy as np
import decimal
def function_python(vec_python):
  a=0
  for i in range(len(vec_python)):
    a=a+vec_python[i]
  return(a)

# This array has one as its first element.
vec_first=np.array([1e-16]*(10001))
vec_first[0]=1
# This array has one as its last element.
vec_last=np.array([1e-16]*(10001))
vec_last[10000]=1

# Result
print decimal.Decimal(function_python(vec_first))
print decimal.Decimal(function_python(vec_last))


## 1
## 1.0000000000010000889005823410116136074066162109375
```

1.5 1E

It is obvious that the sum() function **does not simply add numbers from left to right.** The following two give the same result, while adding left to right will give a different result from the for loop in 1D. Basically, sum() function is independent with the order of elements within a vector.

```
vec_first<-c(1,rep(1e-16,10000))
vec_last<-c(rep(1e-16,10000),1)
sum(vec_first)

## [1] 1.000000000000999644811

sum(vec_last)

## [1] 1.000000000001000088901
```

1.6 1F

My guess is that the sum() function does not care about the order in a vector, since I have tried inserting 1 in mulitple positions in a vector, and all of them will return the same result. I have searched a number of documentations stating that simply summing a sequance of n (finite) number of floating numbers has the **worst** case precision, because the error grows with n. We know the sum() function in R basically called the sum function in C, which involves so-called compensated summation method, which sometimes carry arbitrary re-ordering of the sequence.
Reference: `http://www.drdobbs.com/floating-point-summation/184403224`

2 Problem 2

First I compared the difference in the calculation speed in the vector level, including the addition, multiplication and subsetting. Generally, if we were using double to carry the calculation, it would be faster than do it with integer, which slightly contradicts to what have been talked in class. My guess is that R tends to prevent integer overflow, and thus convert all integers to double before the calculation and then convert them back after the calculation. One should notice that subsetting will yield nearly the same amount of time for integers and doubles, which makes sense because the operations of subsetting do not involve the conversion between integers and double.

```
options(digits=7)
float_set<-as.double(1:10000)
is.double(float_set)

## [1] TRUE

int_set<-as.integer(1:10000)
is.integer(int_set)

## [1] TRUE

subset_sample<-sample(10000,1000)
object_size(int_set)

## 40 kB

object_size(float_set)

## 80 kB

## Vector multiplication (elementwise), the final result's type remains the same
## as before arithmatic operations
microbenchmark(a<-int_set*int_set,b<-float_set*float_set)
```

```
## Unit: microseconds
##                          expr    min     lq      mean  median      uq
##      a <- int_set * int_set 64.542 80.565 103.24771 85.8695 115.290
##  b <- float_set * float_set 10.966 22.177  76.92982 46.3125  65.212
##       max neval
##   534.167   100
##  2089.776   100
```

```
typeof(a)
```

```
## [1] "integer"
```

```
typeof(b)
```

```
## [1] "double"
```

```
microbenchmark(int_set+int_set,float_set+float_set)
```

```
## Unit: microseconds
##                   expr    min      lq      mean  median     uq      max
##      int_set + int_set 50.053 57.0325 113.13362 74.4365 92.572 2129.187
##  float_set + float_set  9.229 13.0200  27.55945 18.4290 30.643  135.216
##  neval
##    100
##    100
```

```
##Vector subsetting, there is no difference in the time of vector subsetting
microbenchmark(int_set[subset_sample],float_set[subset_sample])
```

```
## Unit: microseconds
##                       expr   min     lq     mean  median     uq    max
##    int_set[subset_sample] 8.158 8.5975 11.76148  8.9665 13.450 40.461
##  float_set[subset_sample] 8.693 9.4725 12.57420 12.7120 14.339 36.226
##  neval
##    100
##    100
```

Now let us assess the Time spent for matrix operations (in linear algebra and elementwise). It takes roughly the same time to do matrix mulitiplication for floating point and integers, and we found that their result type were both double. Floating will be a little faster because the calculation will first transform the integer matrix to double in case of **integer overflow**, but that time spent is trivial compared to the time spent for matrix multipilication. To illustrate this, floating point calculation is faster than integer calculation in elementwise multiplication, the time difference is more significant because elementwise multiplication itself is lighter. Thus converting integer to double spends a larger proportion of time. Notice that the subsetting operations for matrices with integers and doubles will have the same speed for the same reason in the vector subsetting, there is no need for conversion. One interesting try is to compute the inverse of a matrix with integer and floatings, and it almost yield the same time probably because the function solve is smart, and it does not take too long to perform the conversion.

```
options(digits=7)
## Create a matrix with 100 rows and 100 columns
int_mat<-matrix(int_set,nrow=100,byrow=TRUE)
float_mat<-matrix(float_set,nrow=100,byrow=TRUE)
typeof(int_mat)
```

```
## [1] "integer"
```

```
typeof(float_mat)

## [1] "double"

sample2<-sample(100,50)
## Matrix multiplication
microbenchmark(x<-int_mat%*%int_mat,y<-float_mat%*%float_mat)

## Unit: microseconds
##                          expr     min       lq     mean  median       uq
##      x <- int_mat %*% int_mat 925.185 926.9515 1204.386 933.211 1137.622
##  y <- float_mat %*% float_mat 901.933 903.8825 1086.465 937.047 1108.483
##       max neval
##  8136.563   100
##  2601.126   100

typeof(x)

## [1] "double"

typeof(y)

## [1] "double"

## Elementwise multiplication
microbenchmark(x<-int_mat*int_mat,y<-float_mat*float_mat)

## Unit: microseconds
##                        expr    min     lq      mean  median       uq
##      x <- int_mat * int_mat 71.426 90.008 115.21680 94.9535 117.4625
##  y <- float_mat * float_mat 12.349 15.029  45.93681 50.6140  65.2825
##       max neval
##  1336.798   100
##   113.138   100

typeof(x)

## [1] "integer"

typeof(y)

## [1] "double"

## Matrix subsetting
microbenchmark(int_mat[sample2,],float_mat[sample2,])

## Unit: microseconds
##                  expr    min     lq     mean  median      uq    max neval
##    int_mat[sample2, ] 22.677 23.191 27.47208 23.7380 31.7600 48.930   100
##  float_mat[sample2, ] 22.952 23.586 31.07101 24.0535 40.2775 94.955   100

## Taking inverse of a matrix
sample_mat<-sample(20000,100)
random_mat_int<-matrix(as.integer(sample_mat),nrow=10)
random_mat_float<-matrix(as.double(sample_mat),nrow=10)
typeof(random_mat_float)
```

```
## [1] "double"

typeof(random_mat_int)

## [1] "integer"

microbenchmark(solve(random_mat_int),solve(random_mat_float))

## Unit: microseconds
##                      expr    min     lq     mean median      uq     max
##     solve(random_mat_int) 32.353 33.577 43.91502 35.516 36.7140 801.418
##   solve(random_mat_float) 31.678 33.492 36.04812 35.213 36.0755  75.632
##   neval
##     100
##     100
```

Note: I have consulted the problem in this problem set with Yueqi Feng, Boying Gong and Jianglong Huang.