Stat243: Problem Set 5

Sicun Huang

October 19, 2015

1.

(a) When storing 1.000000000001 on a computer, we have 16 digits of accuracy.

```
options( digits = 22 )
1.000000000001

## [1] 1.000000000001000088901
```

(b) Using sum function did not result in the same level of accuracy as we got from (a); instead of 16, answer has 12 digits of accuracy.

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

## [1] 1.000000000000999644811
```

(c) In python, the result from the sum function was of even lower accuracy; it was an integer.

```
import numpy as np
from decimal import Decimal

x = np.array([1]+[1e-16]*(10000))
print( Decimal( sum(x) ) )

## 1
```

(d) Having 1 as the first value in the vector in R returned an integer 1 i.e. the result has 0 digits of accuracy. If 1 is the last value instead of the first, we get an answer with 16 digits of accuracy, which is what we expected.

```
y <- 1
for( i in 1:10000 ){
   y <- y + 1e-16
}
y

## [1] 1

z <- 0
for( i in 1:10000 ){
  z <- z + 1e-16
}
z <- z+1
z

## [1] 1.000000000001000088901
```

In python, however, we get 1.0 from either summing order.

```
y = 1
for i in range(10000):
    y = y+1e-16
print(y)

## 1.0
```

```
z = 0
for i in range(10000):
    z = z+1e-16
z = z+1
print(z)

## 1.0
```

(e) Observe that since its result wasn't the same as the first for loop, R's sum function isn't simply summing numbers from left to right. Although it's didn't achieve the level of accuracy we were expecting either.

2. As we can see from the results below, overall, integer calculations in R are faster than floating point calculations.

```
library(microbenchmark)

a <- rep(1, 1000000)
b <- rep(as.numeric(1.00000000), 1000000)

microbenchmark(sum(a), sum(b))

## Unit: microseconds
##    expr                    min                     lq
##  sum(a) 823.8440000000000509317 846.2670000000000527507
##  sum(b) 823.9120000000000345608 845.2345000000000254659
##                      mean                  median                      uq
##  918.9971500000000332875 872.9650000000000318323 913.0454999999999472493
##  938.2676800000000412183 878.0000000000000000000 933.2490000000000236469
##                       max neval
##  2056.2899999999999963620   100
##  1956.8109999999999921783   100

microbenchmark(a-10000, b-10000)

## Unit: microseconds
##       expr                    min                     lq
##  a - 10000 771.8089999999999690772 867.8109999999999217835
##  b - 10000 763.1670000000000300133 849.0584999999999809006
##                      mean                  median                      uq
##  2222.0625199999999949505 1755.1289999999999905413 2759.5245000000000443833
##  1859.1128699999999929860 1432.4510000000000021828 2457.4745000000000261934
##                       max neval
##  34995.5639999999999848660   100
##  28114.1840000000000110595   100

microbenchmark(a^10000, b^10000)
```

```
## Unit: milliseconds
##     expr                     min                       lq
##  a^10000 2.45220899999999861075 2.61818250000000079154
##  b^10000 2.45343499999999810427 2.56547850000000217085
##                     mean                  median                       uq
##  3.62674538000000018583 3.24221300000000011624 4.47510199999999691227
##  3.65462260000000165466 3.34094049999999868663 4.54117899999999632564
##                      max neval
##  7.37091399999999965951   100
##  6.03184100000000008185   100
```

```r
microbenchmark(c <- a[1:9999], d <- b[1:9999])
```

```
## Unit: microseconds
##            expr                    min                       lq
##  c <- a[1:9999] 41.18599999999994316 106.26699999999959073
##  d <- b[1:9999] 40.72099999999653255 105.64300000000006821
##                     mean                  median                       uq
##  122.88295000000081172 115.27600000000104592 133.58150000000054570
##  154.11234999999921783 114.58500000000079581 124.54449999999993179
##                       max neval
##   234.27000000000102318   100
##  3570.57099999999126885   100
```