STAT243: PS 5

Xinyue Zhou

October 19, 2015

1.

a)

I expect a 15 digits accuracy after the decimal point, Since the machine epsilon is around 10^{-16}.

```
.Machine$double.eps

## [1] 2.220446e-16

options(digits = 22)
1+1e-12

## [1] 1.0000000000001000088901
```

b)

It do give the answer around $1 + 10^{-12}$, but is as less accurate as directly calculation in part (a). It's also 16-digit accuracy (15 after decimal points).

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

## [1] 1.0000000000000999644811
```

c)

In this part, before I do the summation, I substitude the first element of the vector *vec* to be 1. It gave me almost the same answer in part b).

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

The result: 1.0000000000009985

d)

If 1 is the first element of vector during the summation, then I found the result is extremely inaccurate:

```
x = c(1,rep(1e-16,10000))
ret = 0
for (i in 1: length(x)){
  ret=ret+ x[i]
}
ret

## [1] 1
```

Put 1 as the last element. Redo b) all of this in a loop. This gave the exact the right answer as the computer store of $1 + 10^{-12}$.

```
ret = 0
for (i in 2: length(x)){
  ret=ret+ x[i]
}
ret +1

## [1] 1.000000000001000088901
```

Redo it in python:
When 1 is the first element:

```
import numpy as np
import decimal as dl
ret = 0
vec = np.array([1e-16]*(10001))
vec[0] = 1
for xx in vec:
  ret = ret + xx
dl.Decimal(ret)
```

The result: Out[65]: Decimal('1').
The number has been round to 1, which is not accurate.

```
import numpy as np
import decimal as dl
ret = 0
vec = np.array([1e-16]*(10001))
vec[len(vec)-1] = 1
for xx in vec:
  ret = ret + xx
dl.Decimal(ret)
```

Result: Decimal('1.000000000001000088900582341011613607406616210937 5')
This gave the right answer as shown in part a).

e) and f)

No, the function *sum()* is not just to sum the vector from its left to right, or apply sum on vector will give the exactly same result in for loop in part d). However, it doesn't. When the element 1 is on the left, instead of 1, the sum gives a more accurate number 1.00000000000999644811.

I have find out the C source code online at https://github.com/wch/r-source/blob/trunk/src/main/summary.c about sum() function. Here is part of it (I just truncate the part for real number):

```
case REALSXP:
       if(ans_type == INTSXP) {
ans_type = REALSXP;
if(!empty) zcum.r = Int2Real(icum);
    }
    updated = rsum(REAL(a), XLENGTH(a), &tmp, narm);
    if(updated) {
zcum.r += tmp;
    }
    break;
```

Take a look at rsum()

```
static Rboolean rsum(double *x, R_xlen_t n, double *value, Rboolean narm)
{
    LDOUBLE s = 0.0;
    Rboolean updated = FALSE;

    for (R_xlen_t i = 0; i < n; i++) {
  if (!narm || !ISNAN(x[i])) {
    if(!updated) updated = TRUE;
    s += x[i];
}
    }
    if(s > DBL_MAX) *value = R_PosInf;
    else if (s < -DBL_MAX) *value = R_NegInf;
    else *value = (double) s;

    return updated;
}
```

I didn't really understand the meaning of the code. But in rsum(), each value of the vector has been compared with maximum positive value in R and minimus negetive value, the value of summation has be adjusted to another number. I think that is why using sum() can have a more accurate result.

2.

First, I create two objects storing number from 1 to 10000000 in two different types, integer and double. It is shown as below.

```
library(pryr)
int_vec=1:10000000
float_vec = seq(1.0,10000000.0, by=1.0)
c(typeof(int_vec),typeof(float_vec))

## [1] "integer" "double"

object_size(int_vec)

## 40 MB

object_size(float_vec)

## 80 MB
```

From the above, I found that the floating type is as twice size as that of integer vector. Next, I will evaluate the speed of calculation of these two types.

```
library(microbenchmark)
options(digits=4)
#basic calculation
#plus on each element
microbenchmark(int_plus = int_vec+5
               ,float_plus=float_vec+5)

## Unit: milliseconds
##        expr   min    lq  mean median    uq   max neval
##    int_plus 25.97 33.53 35.23  34.18 35.51 75.70   100
##  float_plus 19.50 26.49 29.05  27.49 28.81 69.55   100
```

```r
microbenchmark(int_mean = mean(int_vec)
               ,float_mean =mean(float_vec))
```

```
## Unit: milliseconds
##        expr    min     lq   mean median    uq   max neval
##    int_mean  9.187  9.264  9.931  9.407 10.22 13.41   100
##  float_mean 18.387 18.627 19.630 19.173 20.10 24.74   100
```

```r
microbenchmark(int_max = max(int_vec)
               ,float_max =max(float_vec))
```

```
## Unit: milliseconds
##       expr    min     lq   mean median    uq   max neval
##    int_max  9.145  9.202  9.867  9.411 10.22 12.39   100
##  float_max 12.229 12.306 13.038 12.623 13.49 16.60   100
```

```r
#subseting
mask = seq.int(1,10000000,2)
microbenchmark(int_subsetting = int_vec[mask],
               float_subsetting=float_vec[mask])
```

```
## Unit: milliseconds
##              expr   min    lq  mean median    uq   max neval
##    int_subsetting 51.23 59.27 65.40  63.96 68.54 111.8   100
##  float_subsetting 55.35 63.51 69.58  67.67 72.04 111.7   100
```

```r
#reverse a vector
microbenchmark(int_reversing = rev(int_vec)
               ,float_reversing=rev(float_vec))
```

```
## Unit: milliseconds
##             expr   min    lq  mean median    uq   max neval
##    int_reversing 56.76 67.30 80.04  73.74 84.39 140.2   100
##  float_reversing 68.06 76.32 88.40  82.07 93.03 134.2   100
```

```r
#tranpose a vector
microbenchmark(int_transposing = t(int_vec)
               ,float_transposing=t(float_vec))
```

```
## Unit: milliseconds
##               expr   min    lq mean median    uq   max neval
##    int_transposing 15.36 16.56 22.6  17.46 24.70 73.22   100
##  float_transposing 20.90 23.03 33.0  25.93 41.09 80.08   100
```

From the above operations on vector, I found that almost all the operations on integer is faster than that of floating points type, beside plusing number on each element of the vector. The reason why this special case happened, I think, is because in R the number (5 in my example) is the type of "double" by default. Therefore, the operation between floating points number and integer is slower than that between floating number and floating number.