Problem Set 5

Alexander Brandt
SID: 24092167

October 19 2015

Friendly Collaborators: Milos Atz

1

1.1

The number of precision is 1.000000000001 is 13 (the number of digits after the decimal + the significant first digit).

1.2

This method gives the right answer up the accuracy expected (i.e. the numbers that start at the 14th digit round correctly the the 13th digit). So it does give the correct accuracy from part a).

```
options(digits=22)
# Create the list, init an empty sum variable, then sum
numbers <- c(1, rep(1 * 10^(-16), 10000))
sum(numbers)

## [1] 1.0000000000000999644811
```

1.3

We perform the same calcuation, just using python instead of R. It gives the right answer ONLY if the floating point is cast first to Digit. If the code chunk is run "as is" from the problem statement, it sums to 1.

```
from decimal import *
import numpy as np
# Create the array, casting each element as a Decimal object
vec = np.array([Decimal(1e-16)]*(10001))
# Make 1.0 the FIRST element
vec[0] = 1
print Decimal(np.sum(vec))

## 1.0000000000010000000000000000
```

1.4

Trying it in a for loop with the 1 as the leading value gives the incorrect answer. It only gives the correct value to one significant digit.

```
options(digits=22)
# Create the list, init an empty sum variable, then loop
# using the previous numbers list/array
my_sum <- 0
for (i in 1:length(numbers))
{
    my_sum <- my_sum + numbers[i]
}
my_sum

## [1] 1
```

Trying it with 1 at the end of the calcuation, and using the for loop, gives the correct answer.

```
options(digits=22)
# Create the list, init an empty sum variable, then loop
revised_numbers <- c(rep(1 * 10^(-16), 10000), 1)
my_revised_sum <- 0.0
for (i in 1:length(revised_numbers))
{
  my_revised_sum <- my_revised_sum + revised_numbers[i]
}
my_revised_sum

## [1] 1.000000000001000088901
```

Now trying it with a for loop in python gives the correct answer (again, if the numbers are first cast to Decimal).

```
from decimal import *
import numpy as np
# Create the array, casting each element as a Decimal object
vec = np.array([Decimal(1e-16)]*(10001))
# Make 1.0 the FIRST element
vec[0] = 1

for_loop_sum = vec[0] + vec[1]

for i in range(2,len(vec)):
    for_loop_sum += vec[i]
print Decimal(for_loop_sum)

## 1.0000000000010000000000000000
```

Trying it with 1 at the end of the calcuation, and using the for loop in python gives the correct answer (I think one works regardless of the cast, but it is always good to be safe!).

```
from decimal import *
import numpy as np
# Create the array, casting each element as a Decimal object
vec = np.array([Decimal(1e-16)]*(10001))
# Make 1.0 the LAST element
vec[len(vec)-1] = Decimal(1)

for_loop_sum = vec[0] + vec[1]
```

```
# initialize the sum, then loop through
for i in range(2,len(vec)):
    for_loop_sum += vec[i]
print Decimal(for_loop_sum)

## 1.00000000000010000000000000000
```

1.5

The results suggest that sum is a fairly savvy method for this calcuation. It keeps track of the lowest value and works off that for the operating precision. I found this to be true when I corroborated with the C code (details below).

1.6

Calling sum shows us that the sum function is a .Primitive method, and hence is not written in R code, but is instead part of the compiled C backend that R operates on.

```
sum

## function (..., na.rm = FALSE)  .Primitive("sum")
```

So now we need to source dive into summary.c from the R installation. I'm reading off the following source `https://github.com/wch/r-source/blob/trunk/src/main/summary.c` with a little help from `https://cran.r-project.org/doc/Rnews/Rnews_2006-4.pdf`.

We see that .Internal call is being made to sum() via the name.c file. This in turn allows us to know that do_summary is embedded in summary.c. sum checks to see "if all of the arguments are integer or logical in advance, as we might overflow before we find out." It then loops over the array with a LDOUBLE (long double, high precision) variable as the returned sum object, and checks to insure that the value does not overflow (and if so, sets it to an internally defined R_PosInf or R_NegInf object). Then it sets the value pointer to the sum, and returns a boolean confirming the success/lack of success of the operation. I inspected this for the real sum operation rsum, but it is probably very similar for the complex sum csum.

2

Let's test the speed of basic arithmatic with a vector of integers vs. a vector of decimals. It seems the integer method is marginally faster, but in running this a few times, they can switch places for fastest. I'd call it a "tie."

```
options(digits=6)
v <- rep(as.integer(1), 200000000)
system.time(sum(v))

##    user  system elapsed
##   0.213   0.000   0.214

v <- rep(as.double(1), 200000000)
system.time(sum(v))

##    user  system elapsed
##   0.219   0.001   0.219
```

Let's test the speed of array subsetting with a vector of integers vs. a vector of decimals. The integer vector subsets marginally slower. I find this to be a bit confusing.

```
options(digits=6)
v <- rep(as.integer(1:5), 20000000)
system.time(v[(v %% 5) == 0])

##    user  system elapsed
##   3.241   0.588   3.848

v <- rep(as.double(1:5),  20000000)
system.time(v[(v %% 5) == 0 ])

##    user  system elapsed
##   2.049   0.603   2.669
```

Finally, let's test the speed of boolean operations with a vecotr of integers vs. a vector of decimals. The decimal method is marginally faster! This was unexpected.

```
options(digits=6)
v <- rep(as.integer(1), 2000000)
system.time(sapply(v, is.integer))

##    user  system elapsed
##   1.632   0.053   1.687

v <- rep(as.double(1), 2000000)
system.time(sapply(v, is.integer))

##    user  system elapsed
##   1.394   0.022   1.417
```