PS5 - STAT 243

Alexander Fred Ojala
Student ID: 26958060
Collaborators: Guillame Baquiast and Milos Atz

October 19th 2015

1 Problem 1

1.1 1. a)

The numbers in R are typically stored as doubles. This is a binary format occupying 8 bytes per number and its precision is 53 bits (about 16 decimal digits). R has in general, has approximately 16 digits of accuracy and uses finite precision arithmetic. Therefore any number cannot be trusted in its precision after the 16 digit representation (counted from the first digit $\neq 0$). Since the number 1.000000000001 consists of 13 digits we can expect that we would have 16 digits of accuracy if we store this number in R. (The addition of numbers needs to be done so that the smaller numbers of size 10^{-16} are added first and after that we can add the one in order not to perform operations on a number consisting of 17 digits (e.g. $1 + 10 \cdot 10^{-16}$))

1.2 1. b)

The calculations below result in the correct answer, as expected. The answer should be $1 + 10^{-12}$ and this is the result that R produces carrying out the calculations. Also note that we have 16 digits of accuracy for the number stored in the variable: *xReal*.

```
options("scipen"=TRUE, "digits"=22)

#SETUP VECTOR
x=rep(1*10^-16,10001)
x[1]=1


##DEFINE REAL, FOR COMPARISON
xReal=1.000000000001
xReal #Ok 16 digits of accuracy (18th digit is an 8, making the 17th a 1 if rounded)

## [1] 1.000000000001000088901

xReal-1 #OK 1*10^12

## [1] 1.0000889005823410116l4e-12

#SUM
xSum=sum(x)
xSum # Ok 1+10^-12, (16 numbers of accuracy, 17:th digit is 6 (it should be a 9))

## [1] 1.0000000000009996448l1

xSum-1 # Ok 1*10^-12
```

```
## [1] 9.996448113724909489974e-13

#COMPARE
xReal-xSum #16 numbers of accuracy, the 17th digit is different

## [1] 4.440892098500626161695e-16
```

1.3 1. c)

The same result was not obtained in Python. Python's sum function only returns the answer 1 (the answer should be $1 + 10^{-12}$). (It has to do with the fact that Python tries to add the numbers sequentially, in adding the number 1 is added first and then the $10^{-16} \cdot 10000$, and this cannot be done since we do not have 17 digits of accuracy.)

N.B. I needed to use the print function for knitr to display results. Prints are printed chronologically at the bottom of the code block below:

```
import numpy as np
from decimal import * #import all from decimal package*

#SETUP VECTOR
x = np.repeat(1e-16,10001)
x[0] = 1

#REAL, FOR COMPARISON
xReal = 1.000000000001 # store answer number and print
print 'Correct answer, for reference:'
print Decimal(xReal) #1st PRINT


#SUM
xSum = np.sum(x)
print 'Sum over the array:'
print Decimal(xSum) #2nd PRINT. Not OK, equals 1


#REVERSE SUM (NB OPTIONAL)
x[0] = 1e-16
x[10000] = 1
xSumReverse = np.sum(x)
print 'Reverse sum over the array (1 added last):'
print Decimal(xSumReverse) #3rd PRINT. this is OK, 1+10e-12


## Correct answer, for reference:
## 1.000000000001000088900582341011613607406616210937 5
## Sum over the array:
## 1
## Reverse sum over the array (1 added last):
## 1.000000000001000088900582341011613607406616210937 5
```

1.4 1. d)

Instead implementing a for loop and summing Forward (adding the number 1.000... first) and Reverse (adding the number 1.000... last) one vector element of a time. In R we obtain the results below.

As noted earler summing the smaller numbers first only results in the answer 1(.00000...) which in this example has 12 digits of accuracy, the 13th digit should be a 1. However it should be noted that no matter what amount of 10^{-16} we add to one would result in 1 (here ten thousand, but even millions or any arbitrary number would result in 1 when we use a loop and add 1 first). When we sum the numbers in Reverse we have 16 digits of accuracy, precisely as was the result with the sum function.

```
## FORWARD LOOP, remember that x[1]=1

xSumLoopForward=0

for (i in 1:length(x)) {
  xSumLoopForward = x[i] + xSumLoopForward
}

xSumLoopForward #wrong only returns 1

## [1] 1

## REVERSE LOOP
xSumLoopReverse=0

for (i in length(x):1) {
  xSumLoopReverse = x[i] + xSumLoopReverse
}

xSumLoopReverse #OK, same result as the sum function

## [1] 1.000000000001000088901
```

Implementing the same solution in Python yields exactly the same results as in R, which can be seen below:

```
import numpy as np
from decimal import * #import all from decimal package*

#CREATE VECTOR
x = np.repeat(1e-16,10001)
x[0] = 1


#FORWARD LOOP
xSumLoopForward=0

for i in range(0,len(x)):
  xSumLoopForward=x[i]+xSumLoopForward

print 'Forward Loop sum (1 added first):'
```

```
print Decimal(xSumLoopForward) #1st PRINT. Not OK, only returns 1


#RERVERSE LOOP
xSumLoopReverse=0

for i in range(len(x)-1,-1,-1): #Reverse loop, begins at x[10000], terminates at x[0]
  xSumLoopReverse=x[i]+xSumLoopReverse

print 'Reverse Loop sum (1 added last):'
print Decimal(xSumLoopReverse) #2nd PRINT. OK, returns correct result, as expected



## Forward Loop sum (1 added first):
## 1
## Reverse Loop sum (1 added last):
## 1.000000000001000088900582341011613607406616210937 5
```

1.5 1. e)

R's sum function did not return 1 as the result (as R did when we used a for loop to sum sequentially over the array x, when $x[1] = 1$), but it retruned a result that was correct with 16 digits of accuracy (if we count the ceiling/rounding up of a digit 9 as the correct result), namely $1 + 10^{-12}$. This suggests that R's sum function does not simply add numbers from the left to the right, but adds numbers of approximately the same magnitude first (in this case the ten thousand smaller numbers, 10^{-16}) and then adds numbers of larger magnitude after that operation has been performed in order to keep the accuracy of the result. This prevents the precision of numbers and their representation to affect the results of the sum function if R is able to avoid it (and not carry out the summation in a 'dumb' way).

2 Problem 2

A function called *timing* was implemented, which uses R's benchmark function (a wrapper aroun system.time) over two operations carried out 5 times each on an integer vector and a floating point vector. The results presented are the mean time for evaluation of the functions (total time (named *elapsed*), system time (named $sys.self$) and user time (named $user.self$)) as well as the relative total time it took for the functions to be evaluated (given under the column *relative*).

In the Documentation of *proc.time* it says that: "The 'system time' is the CPU time charged for execution by the system on behalf of the calling process. [..] The 'user time' is the CPU time charged for the execution of user instructions of the calling process."

N.B. All functions below have been tested so that they perserve the type (checked with R function $typeof()$) when the operation has been carried out. That is also why I used the %/%-operator to carry out the integer division, because otherwise R would transform the integer array into a double in order to carry out the operation.

2.1 Results

- **Arithmetic operations:** The system time is almost half for all the integer arithmetic operations. However the user time is almost always more than twice as fast for the float arithmetic operations. Also the user time takes about 4 times as much time for the integer operations compared to its own system

time. Therefore the elapsed time for the floating number arithmetic operations are always quicker in R if we look at the tests.

- **Subsetting:** For subsetting vectors it is the reverse. The system time is (almost always, a bit) faster for floating point numbers, while the user time is a lot faster for integer numbers. All types of subsetting is quicker with integer vectors for both system and user time.
- **Other interesting operations:** When we try other operations like *mean*, *max* and *median* the integer vectors generally are faster than the floating point vectors (the time results are about the same for the max operation).

2.2 Conclusion

Many of the results were unexptected and inconsistent in the results. The notion was that the integer operations would perform better overall, however it did not. Especially it was unexpected that the arithmetic operation was faster (user time and total time) for the floating point numbers.

```
#SETUP AND VECTOR CHECK
vecInt<-rep(1:10,10^7)
vecFloat<-as.double(vecInt)

typeof(vecInt)

## [1] "integer"

typeof(vecFloat)

## [1] "double"

object_size(vecInt)

## 400 MB

object_size(vecFloat)

## 800 MB

options(digits=4)

#Define function for timing
timing <- function(int,float) {
  cat("Operations: \nint = ")
  print(int)
  cat("float = ")
  print(float)
  benchmark(int=eval(int),float=eval(float),replications = 5, columns=c('test','elapsed','user.self','sy
}

## ARITHMETIC TIMING

#Sum
timing( quote(sum(vecInt)) , quote(sum(vecFloat)) )

## Operations:
## int = sum(vecInt)
## float = sum(vecFloat)
##    test elapsed user.self sys.self relative
## 2 float   0.535     0.527    0.003    1.039
## 1   int   0.515     0.513    0.001    1.000
```

```
#Multiplication
timing( quote(vecInt*vecInt) , quote(vecFloat*vecFloat) )
```

```
## Operations:
## int = vecInt * vecInt
## float = vecFloat * vecFloat
##    test elapsed user.self sys.self relative
## 2 float   2.788     1.225    1.419    1.000
## 1   int   3.158     2.497    0.643    1.133
```

```
#Division, %/% forces integer division
timing( quote(vecInt%/%vecInt), quote(vecFloat/vecFloat))
```

```
## Operations:
## int = vecInt%/%vecInt
## float = vecFloat/vecFloat
##    test elapsed user.self sys.self relative
## 2 float   3.643     2.365    1.263    1.000
## 1   int   5.220     4.456    0.716    1.433
```

```
#Addition
timing( quote(vecInt+vecInt) , quote(vecFloat+vecFloat) )
```

```
## Operations:
## int = vecInt + vecInt
## float = vecFloat + vecFloat
##    test elapsed user.self sys.self relative
## 2 float   2.439     1.108    1.308    1.000
## 1   int   2.886     2.233    0.635    1.183
```

```
#Subtraction
timing( quote(vecInt-vecInt) , quote(vecFloat-vecFloat) )
```

```
## Operations:
## int = vecInt - vecInt
## float = vecFloat - vecFloat
##    test elapsed user.self sys.self relative
## 2 float   2.358     1.098    1.254     1.00
## 1   int   2.688     2.050    0.629     1.14
```

```
#Multiplication of constant
timing( quote(10L*vecInt) , quote(as.double(10)*vecFloat) )
```

```
## Operations:
## int = 10L * vecInt
## float = as.double(10) * vecFloat
##    test elapsed user.self sys.self relative
## 2 float   2.309     1.036    1.248    1.000
## 1   int   3.176     2.507    0.659    1.375
```

```
#Addition of constant
timing( quote(10L+vecInt) , quote(as.double(10)+vecFloat) )
```

```
## Operations:
## int = 10L + vecInt
## float = as.double(10) + vecFloat
##    test elapsed user.self sys.self relative
## 2 float   2.355     1.051    1.283    1.000
## 1   int   2.923     2.270    0.639    1.241
```

```
## SUBSETTING

#Extract first million values
timing( quote(vecInt[1:10^6]) , quote(vecFloat[1:10^6]) )

## Operations:
## int = vecInt[1:10^6]
## float = vecFloat[1:10^6]
##    test elapsed user.self sys.self relative
## 2 float   0.049     0.034    0.014    1.167
## 1   int   0.042     0.030    0.011    1.000

#Extract every tenth value
timing( quote(vecInt[seq(10,10^8,10)]) , quote(vecFloat[seq(10,10^8,10)] ))

## Operations:
## int = vecInt[seq(10, 10^8, 10)]
## float = vecFloat[seq(10, 10^8, 10)]
##    test elapsed user.self sys.self relative
## 2 float   2.465     2.090    0.363    1.011
## 1   int   2.439     2.024    0.372    1.000

#Extract random sample
set.seed(0)
smp<-sample(1:10^8,10^7)
timing( quote(vecInt[smp]) , quote(vecFloat[smp]) )

## Operations:
## int = vecInt[smp]
## float = vecFloat[smp]
##    test elapsed user.self sys.self relative
## 2 float   1.911     1.827    0.059    1.145
## 1   int   1.669     1.541    0.103    1.000

#Extract unique values
timing( quote(unique(vecInt)) , quote(unique(vecFloat)) )

## Operations:
## int = unique(vecInt)
## float = unique(vecFloat)
##    test elapsed user.self sys.self relative
## 2 float   11.26     8.906    2.289    1.368
## 1   int    8.23     5.914    2.281    1.000

## OTHER INTERESTING FUNCTIONS

#Max value
timing( quote(max(vecInt)) , quote(max(vecFloat)) )

## Operations:
## int = max(vecInt)
## float = max(vecFloat)
##    test elapsed user.self sys.self relative
## 2 float   0.649     0.611    0.004     1.05
## 1   int   0.618     0.592    0.005     1.00
```

```
#Mean value
timing( quote(mean(vecInt)) , quote(mean(vecFloat)) )

## Operations:
## int = mean(vecInt)
## float = mean(vecFloat)
##    test elapsed user.self sys.self relative
## 2 float   1.074     1.050    0.005     2.21
## 1   int   0.486     0.484    0.001     1.00

#Median
timing( quote(median(vecInt)) , quote(median(vecFloat)) )

## Operations:
## int = median(vecInt)
## float = median(vecFloat)
##    test elapsed user.self sys.self relative
## 2 float   11.78     9.229    2.512    1.023
## 1   int   11.51     8.941    2.346    1.000
```