Problem Set 5

Jeffrey Kwarsick

October 18, 2017

1 Problem 1

Completed the reading of the numerical linear algebra unit.

2 Problem 2

- **Demonstrate how integers are stored from 1, 2, 3, ..., 2^{53}-2, 2^{53}-1 in the (-1)^S(1.d)(2^{e-1023}) format. This problem is done with use of the library *pryr*.**

  Using the number two as an example, I demonstrate how the leading bit of the 64-bit binary representation of integers controls the sign of the integer or floating decimal number. In this case, I used positive 2 and negative 2.

```
library(pryr)
options(digits=22)
bits(2)

## [1] "01000000 00000000 00000000 00000000 00000000 00000000 00000000 00000000"

bits(-2)

## [1] "11000000 00000000 00000000 00000000 00000000 00000000 00000000 00000000"
```

  Now to show some examples of the that integers up to 2^{53} can be stored exactly using the double floating precision point format.

```
#2^10 examples
2^10-1

## [1] 1023

2^10

## [1] 1024

2^10+1

## [1] 1025

bits(2^10-1)

## [1] "01000000 10001111 11111000 00000000 00000000 00000000 00000000 00000000"
```

```r
bits(2^10)
```

```
## [1] "01000000 10010000 00000000 00000000 00000000 00000000 00000000 00000000"
```

```r
bits(2^10+1)
```

```
## [1] "01000000 10010000 00000100 00000000 00000000 00000000 00000000 00000000"
```

```r
#2^40 examples
2^40-1
```

```
## [1] 1099511627775
```

```r
2^40
```

```
## [1] 1099511627776
```

```r
2^40+1
```

```
## [1] 1099511627777
```

```r
bits(2^40-1)
```

```
## [1] "01000010 01101111 11111111 11111111 11111111 11111111 11100000 00000000"
```

```r
bits(2^40)
```

```
## [1] "01000010 01110000 00000000 00000000 00000000 00000000 00000000 00000000"
```

```r
bits(2^40+1)
```

```
## [1] "01000010 01110000 00000000 00000000 00000000 00000000 00010000 00000000"
```

```r
#2^52 examples
2^52-1
```

```
## [1] 4503599627370495
```

```r
2^52
```

```
## [1] 4503599627370496
```

```r
2^52+1
```

```
## [1] 4503599627370497
```

```r
bits(2^52-1)
```

```
## [1] "01000011 00101111 11111111 11111111 11111111 11111111 11111111 11111110"
```

```r
bits(2^52)
```

```
## [1] "01000011 00110000 00000000 00000000 00000000 00000000 00000000 00000000"

bits(2^52+1)

## [1] "01000011 00110000 00000000 00000000 00000000 00000000 00000000 00000001"

#2^53 examples
2^53-1

## [1] 9007199254740991

2^53

## [1] 9007199254740992

bits(2^53-1)

## [1] "01000011 00111111 11111111 11111111 11111111 11111111 11111111 11111111"

bits(2^53)

## [1] "01000011 01000000 00000000 00000000 00000000 00000000 00000000 00000000"
```

It is observed that all of these numbers can be stored exactly as integers (a precision of 1) because all of these numbers are smaller than the machine epsilon, 2^{-52} or 2.220446049250313080847e-16. Numbers where their base of 2^x, where is greater than 52, will not be able to be represented exactly with an integer spacing of 1. This is shown below in the subsequent parts of this problem. Overall, this demonstrates that integers up to 2^{53} can be stored exactly with a spacing of 1 because 52 bits are allocated to the mantissa, or fractional part of a common logarithim (base 10).

- **Show how 2^{53} and $2^{53}+2$ can be represented exactly but not $2^{53}+1$, so the spacing of numbers of this magnitude is 2.**

```
#Integers first
2^53

## [1] 9007199254740992

2^53+1

## [1] 9007199254740992

2^53+2

## [1] 9007199254740994

2^53+4

## [1] 9007199254740996

2^53+6
```

```
## [1] 9007199254740998

2^53+8

## [1] 9007199254741000

#now the bit formats
bits(2^53)

## [1] "01000011 01000000 00000000 00000000 00000000 00000000 00000000 00000000"

bits(2^53+1)

## [1] "01000011 01000000 00000000 00000000 00000000 00000000 00000000 00000000"

bits(2^53+2)

## [1] "01000011 01000000 00000000 00000000 00000000 00000000 00000000 00000001"

bits(2^53+4)

## [1] "01000011 01000000 00000000 00000000 00000000 00000000 00000000 00000010"

bits(2^53+6)

## [1] "01000011 01000000 00000000 00000000 00000000 00000000 00000000 00000011"

bits(2^53+8)

## [1] "01000011 01000000 00000000 00000000 00000000 00000000 00000000 00000100"
```

Above are the numbers investigated using the *bits()* function from *pryr* library. From invoking *bits()*, I observe that when adding by units of 2, it advances changes the last 8-bits, either activating or de-activating them depending on what multiple of 2 is being added. Since there are 52-bits used for the storage of the ingeter, this means that the machine epsilon is 2^{-52}. This means that for integers in the same magnitude of 2^{53}, the minimum spacing between them is $2^{-52} * 2^{53}$, which is 2.

- **Show that for numbers starting with 2^{54} that the spacing between integers that can be exactly represented is 4.**

  Going off of the same argument from the previous part of the problem. The minimum spacing between integers in the realm of 2^{54} will be **4** because the machine epsilon is 2^{-52}. This is demonstrated below –

```
#showing that minimum spacing is 4
#based off the size of the number.
2^(-52)*2^(54)

## [1] 4
```

The difference between them is 4, representing the minimum spacing that can be exactly represented between numbers in the realm of 2^{54}

- **Confirm results are consistent with the execution of 2^{53}-1, 2^{53}, and 2^{53}+1.**

  Here is the execution, confirming the results from the sub-parts of Problem 2. 2^{53}+1 cannot be represented exactly, but 2^{53}-1 and 2^{53} can be represented exactly. 2^{53}+2 was also printed to demonstrate that the spacing between numbers of the magnitude is exactly 2.

```
options(digits=22)
2^53-1

## [1] 9007199254740991

2^53

## [1] 9007199254740992

2^53+1

## [1] 9007199254740992

2^53+2

## [1] 9007199254740994
```

3 Problem 3

3.1 Part (a)

```
library(microbenchmark) #Needed for timing of the copy
#Create numeric and integer vectors of same length
numeric_vec <- rnorm(1e7)
integer_vec <- as.integer(sample(1:1e8, 1e7))

#Copy Numeric and integer vectors
copy_numeric <- numeric_vec
copy_integer <- integer_vec

#Determine time to copy each vector
system.time(copy_numeric <- numeric_vec)

##    user  system elapsed
##       0       0       0

system.time(copy_integer <- integer_vec)

##    user  system elapsed
##       0       0       0

#microbenchmark timing of the copies
microbenchmark(copy_numeric <- numeric_vec)

## Unit: nanoseconds
##                         expr min   lq                    mean median uq
```

```
##  copy_numeric <- numeric_vec  80 83.5 102.4899999999999948841   87.5 89
##   max neval
##  1572   100

microbenchmark(copy_integer <- integer_vec)

## Unit: nanoseconds
##                          expr min lq                    mean median uq  max
##  copy_integer <- integer_vec  82 85 107.3199999999999931788     89 91 1874
##  neval
##    100
```

Above, I created a vector of numerics of length 1e7 and a vector of integers of length 1e7. Using the *microbenchmark* library, I copied the both lists and found that copying the large vector of integers is slightly faster than copying a large list, of equivalent length, of numerics. The vector of integers is only copied slightly faster by roughly 3 nanonseconds, according to the results from *microbenchmark*.

3.2 Part (b)

```
library(microbenchmark)
#variable x to grab first half of each large vector
x <- length(numeric_vec)/2
#subsetting vectors
#grabbing the first half of each vector
#timing with microbenchmark
microbenchmark(subset_numeric <- numeric_vec[1:x], times = 100)

## Unit: milliseconds
##                                expr                      min
##  subset_numeric <- numeric_vec[1:x] 28.36346699999999998454
##                          lq                     mean                   median
##  39.32142650000000116961 46.07480035999999756768 46.15639500000000339242
##                          uq                      max neval
##  47.93905700000000535965 91.84356300000000317141   100

microbenchmark(subset_integer <- integer_vec[1:x], times = 100)

## Unit: milliseconds
##                                expr                      min
##  subset_integer <- integer_vec[1:x] 26.34556299999999851025
##                          lq                     mean                   median
##  28.09336599999999961597 32.28821059999999931733 29.81952249999999793317
##                          uq                      max neval
##  33.26057799999999531337 67.75653200000000708769   100

#system time also invoked to check timing
system.time(subset_numeric <- numeric_vec[1:x])

##                       user                     system
## 0.028999999999999991473487 0.000000000000000000000000
##                    elapsed
## 0.028999999999999991473487

system.time(integer_numeric <- integer_vec[1:x])
```

```
##                      user                    system
## 0.02800000000000046895821 0.00000000000000000000000
##                   elapsed
## 0.02799999999999869260137
```

Above, a subset was taken from the large numeric vector as well as the large integer vector. The variable x was defined as half the length of both large vectors and was used to grab the first half of each vector. The vector that required less time for execution, according to *microbenchmark*, was the large integer vector. The integer was faster by approximately 13% compared to the the subsetting of the numeric vector.

4 Problem 4

4.1 Part (a)

When attmpting to parallelize matrix multiplication of two n X n matrices, X and Y to produce the product XY, it would better to break matrix Y into p blocks of m=n/p columns rather individual n column-wise computations. The reason is that by breaking into different blocks, each block could be parallelized by passing each block to a different core on your machine in order to conduct computations. This way, your PC is utilizing all the computational power that a multi-core processor has to offer. Running individual column-wise computations would mean that you would likely end up with tasks to compute when compared to the number of cores at yours disposal which makes evenly dividing the computations amongst available computing resources/cores difficult. This would likely mean that all the column-wise computations would be conducted iteratively over a single core which is costly in regards to computing time. Therefore, by breaking up the number of columns into p blocks where p = number of cores, you would be maximizing the number of computations capable of being conducted by your machine, reducing computation time.

4.2 Part (b)

Approach 1 is better for communication minimization. The for Approach 1 being better for communication is that the entirety of matrix X would be sent to each core within a node along with 1 of p blocks of m columns of matrix Y. Sending the entirety of X to each core would require more memory and the product of X and the block of Y would be larger. Despite taking more memory, this method would require few communications, sending all of X and block of Y, p_i, and then receiving the product back from each core to assemble as the final product.

Approach 2 is better for minimizing memory use. The chunks that would be sent to each core would be much smaller than the entirety of each matrix, and the resulting product of these smaller chunks would also require less matrix than that of Approach 1. While this would minimize memory use during the computation, it would require more communications. The number of communications would be equal to all of the pairs of blocks of rows from X and blocks of columns of Y required to to be sent to all the cores within the nodes you have access to plus all of the communications back in order to stitch together the final product of matrices X and Y.

5 Problem 5: Extra Credit

```
options(digits = 22)
#TRUE
0.2 + 0.3 == 0.5
```

```
## [1] TRUE
```

```
0.01 + 0.49 == 0.5
```

```
## [1] TRUE
```

```r
0.1 + 0.1 == 0.2
```

```
## [1] TRUE
```

```r
0.2 + 0.2 == 0.4
```

```
## [1] TRUE
```

```r
0.4 + 0.1 == 0.5
```

```
## [1] TRUE
```

```r
0.1 + 0.5 == 0.6
```

```
## [1] TRUE
```

```r
0.2 + 0.2 + 0.3 == 0.7
```

```
## [1] TRUE
```

```r
0.2 + 0.2 + 0.2 + 0.2 == 0.8
```

```
## [1] TRUE
```

```r
0.4 + 0.4 + 0.1 == 0.9
```

```
## [1] TRUE
```

```r
0.03 + 0.03 + 0.03 == 0.09
```

```
## [1] TRUE
```

```r
0.02 + 0.03 == 0.05
```

```
## [1] TRUE
```

```r
0.001 + 0.002 == 0.003
```

```
## [1] TRUE
```

```r
#FALSE
0.2 + 0.1 == 0.3
```

```
## [1] FALSE
```

```r
0.1 + 0.1 + 0.1 == 0.3
```

```
## [1] FALSE
```

```r
0.1 + 0.1 + 0.1 + 0.1 + 0.1 + 0.1 + 0.1 + 0.1 + 0.1 == 0.9
```

```
## [1] FALSE
```

```r
0.1 + 0.2 + 0.3 == 0.6
```

```
## [1] FALSE
```

```r
0.2 + 0.2 + 0.2 == 0.6
```

```
## [1] FALSE
```

```
0.4 + 0.2 == 0.6
## [1] FALSE
0.3 + 0.3 + 0.3 == 0.9
## [1] FALSE
0.01 + 0.01 + 0.01 + 0.01 + 0.01 + 0.01 + 0.01 + 0.01 + 0.01 + 0.01 == 0.1
## [1] FALSE
#looking at the digits out to 22 places
0.1
## [1] 0.10000000000000000055511
0.2
## [1] 0.20000000000000000111022
0.3
## [1] 0.29999999999999999888978
0.4
## [1] 0.40000000000000000222045
0.5
## [1] 0.5
0.6
## [1] 0.59999999999999999777955
0.7
## [1] 0.69999999999999999555911
0.8
## [1] 0.80000000000000000444089
0.9
## [1] 0.90000000000000000222045
0.01
## [1] 0.01000000000000000020817
0.02
## [1] 0.02000000000000000041633
0.03
## [1] 0.02999999999999999888978
0.04
## [1] 0.04000000000000000083267
0.05
## [1] 0.05000000000000000277556
0.49
## [1] 0.48999999999999999911182
```

Above is my effort to attempt to discern a pattern from the manner in which some addition statements are true, but some are not. I was not able to discern any pattern from this effort, but after some research I confirmed that R is only able to represent numerics exactly "are integers and fractions whose denominator is a power of 2. All other numbers are internally rounded to (typically) 53 binary digits accuracy."[1]

1. Hornik, Kurt. *R FAQ*. 2017. https://cran.r-project.org/doc/FAQ/R-FAQ.html#Why-doesn_0027t-R-think-these-numbers-are-equal_003f