ps5

Yihuan Song

10/14/2018

question1

since A = ΓΛΓ , det(A) = det(Γ)det(Λ)det(Γ).Since Γ is an orthogonal matrix, det(Γ)det(Γ)=1. so det(A) = det(Λ). Since Λ is a diagonal matrix, det(Λ) = product of the diagonal, which is the product of the eigenvalues. Therefore, |A| is the product of the eigenvalues.

question2

The expit function doesn't work numerically on a computer for large values of z, because when z is large, exp(z) would equal to infinity due to overflow of large numbers in computer, so that the result would appear as NaN as shown below. A solution is to divide both the numerator and the denomenator by exp(z). Therefore, the expit(z) = exp(z) / (1 + exp(z)) becomes expit(z) = 1 / (1 + exp(-z)), and for large z, exp(-z) would be relatively much smaller in value compared to 1, so its value will be igorned in calculation. Then, expit(z) would approximately be 1, which is more numerically stable compared to the original form.

```
exp(10000)/(1+exp(10000))
```

```
## [1] NaN
```

```
1/(1+exp(-10000))
```

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

question 3

After adding e^12 to z, the accuracy of each element in x has already gone down to 4 digits, because the e^12 occupied the first 12 digit of precision. Therefore, while z has 16 digits of precision after the decimal point, x has four digit of precision after the decimal point. (As is shown below in R.) Therefore, as we are trying to calculate var(x), we are doing calculations using x, which has already lost precision after the fourth digit of decimals, so our result of variance of x is not the same as the variance calculated using the original z. (We have 5 digits agreed in the question where we set seed of 1, which is a coincidence. If we set another seed, 4 digits would agree, as shown below.)

```
set.seed(2)
z <- rnorm(10, 0, 1)
x <- z + 1e12
z[2]
```

```
## [1] 0.1848492
```

```
formatC(x[2],20)
```

```
## [1] "1000000000000.1848145"
```

```
formatC(var(z), 20, format = 'f')
```

```
## [1] "0.97020065227876062242"
```

```r
formatC(var(x), 20, format = 'f')
```

```
## [1] "0.97024419572618270102"
```

question 4

a) It is better to break up Y into p blocks of m = n/p columns rather than into n individual column-wise computations, because it reduces the number of times of transferring data to or from the core and the time it takes starting up a worker process. After breaking up Y, the p cores would exactly receive the p blocks of columns and process at the same time, which reduces the latency of communication, compared to sending data by each column, which will require a larger number of communication.

b) Since X and Y are n*n matrices, the memory that X or Y takes is $8n^2$ bytes. Therefore, each block of X or Y takes $8n^2$/p bytes, and storing the result takes $8n^2$ memory. For Approach A, the total memory used would be $8(p(n^2 + n^2/p)) = 8(p+1)n^2$ bytes. For Approach B, the total memory used would be $8p(n^2/p + n^2/p) = 16n^2$ bytes. Thus, Approach B is better for minimizing memory use. For Approach A, we need to pass each block of Y and the whole X matrix to each of the p cores in order to complete the computation, while for Approach B, we only need to pass one block of B and one block of A to each of the p cores to do the computaion. Therefore, Approach B would take less memory than Approach A does.

On the other hand, Approach A is better for minimizing communication. For Approach A, only p total tasks will be required to complete the computation. Therefore, p columns will correspond to exactly p cores for data communication, and it will return the results from the p cores, so the total communication is $(n^2+n^2/p)*p + n^2 = n^2(2+p)$. However, for Approach B, we need p^2 total tasks to compelete the computation. Since we are dividing both X and Y, we need to pass all p blocks of X to compute with each block of Y for p blocks of Y; therefore, we need $p^2(n^2/p+n^2/p) + n^2 = n^2(2p+1)$ number of data transfer for communication between the master and workers. Therefore, since $2p+1 > 2+p$ when $p > 1$, Approach A has a smaller amount of communication.