Numerical Analysis: homework 01

Due on Tuesday, Febrary 28, 2017

102061149 Fu-En Wang

1 Introduction

When we are given an n x n squre matrix A, then an n x n orthogonal matrix G can also be found by Gram-Schmidt process.

1.1 Gram-Schmidt process

When

$$A = \begin{bmatrix} A_1 & A_2 & ..... & A_n \end{bmatrix} \tag{1}$$

and

$$G = \begin{bmatrix} G_1 & G_2 & ..... & G_n \end{bmatrix} \tag{2}$$

Each A_i and G_i is a column of squre matrix A and G, respectively. And the algorithm of **Gram-Schmidt process** is as the following:

$$G_1 = A_1 \tag{3}$$

$$G_k = A_k - \sum_{i=1}^{k-1} \frac{(A_k^T G_i) G_i}{G_i^T G_i} \tag{4}$$

1.2 Sigma calculation

Because G is an orthogonal matrix, so $M = G^T G$ will be a diagonal matrix; in other words, each non-diagonal element

$$M[i][j], i \neq j \tag{5}$$

has to be zero. To prove this property, we calculate the error of non-diagonal elements by Sigma:

$$\sigma = \sqrt{\sum_{i=1}^{n} \sum_{j=1, i \neq j}^{n} d_{i,j}^2} \tag{6}$$

Theorectically, σ should be very closed to zeros.

In this homework, we will implement three algorithms to compare their runtime.

2 C++ Implementation

2.1 Algorithm-1

```
MAT A_t = A.transpose();
MAT G(n, n);
G[0] = A_t[0];
for(int k=1; k<n; k++){
    G[k] = 0;
    for(int i=0; i<k; i++){
        G[k] += (A_t[k] * G[i]).sum() * G[i] / (G[i] * G[i]).sum();
    }
    G[k] = A_t[k] - G[k];
}
G = G.T();
```

This is the most straightforward method for **Gram-Schmidt process**. In the innermost for-loop, we calculate $\sum_{i=1}^{k-1} \frac{(A_k^T G_i)G_i}{G_i^T G_i}$ first and subtracted by $A[k]$, and then replace $G[k]$ by the final vector.

2.2 Algorithm-2

```
MAT A_t = A.transpose();
MAT G(n, n);
G[0] = A_t[0];
for(int k=1; k<n; k++){
    G[k] = A_t[k];
    for(int i=0; i<k; i++){
        G[k] -= (G[k] * G[i]).sum() * G[i] / (G[i] * G[i]).sum();
    }
}
G = G.T();
```

Algo2 simplifies algo1 by modifying three terms and removing $G[k] = A[k]$ - $G[k]$. So there is one more subtraction in algo1 than algo2.

2.3 Algorithm-3

```
MAT A_t = A.transpose();
MAT G(n, n);
G[0] = A_t[0];
for(int k=1; k<n; k++){
    G[k] = A_t[k];
    for(int i=0; i<k; i++){
        G[k] -= (G[k] * G[i]).sum() / (G[i] * G[i]).sum() * G[i];
    }
}
G = G.T();
```

Compared to algo2, algo3 only move the multiplication of $(G_k^T G_i)$ and G_i to the last position. By such manipulation, two problems show up:

1. Will the result change? (observed by $\boldsymbol{\sigma}$)

2. Will the speed change? (observed by **RUNTIME**)

We will discuss these two questions in Discussion part.

2.4 Complexity

Because in the innermost for-loop, we have to do $\frac{n(n+1)}{2}$; but we still need to do vector operation one time, so it will become $n * \frac{n(n+1)}{2}$, which is a $\boldsymbol{O(n^3)}$ problem.

3 Discussion

3.1 Performance Evaluation

In this project, we focus on two numbers for performance:

1. $\boldsymbol{\sigma}$ (accuracy)

2. **RUNTIME** (speed)

σ can be regarded as the loss or error of our system, so we must make it small as possible as we can. As for RUNTIME, there are two way to analyze.

1. Total runtime (total time of execution)

2. Algorithm runtime (total time of algorithm part)

Total runtime consists of the time that program parses the **mX.dat** and processes other jobs, which may also take much time. But what we really care about is the time our algorithms consume, so I also track the algorithm time in each experiment. In the following part, **algo-time** is refered as algorithm runtime(seconds), while **runtime** refered as total runtime(seconds).

3.2 Algorithm-1 Perormance

Table [1] show detail results of algorithm-1.

3.2.1 Sigma

When n is large, it seems that algorithm-1 get a large σ; in other words, it is not **accurate**. By my observation, because the **magnitude** difference between $A[k]$ and $G[k]$ could be large, so the calculation error of $G[k] = A[k] - G[k]$ will be amplified, and that's why we get a large $\boldsymbol{\sigma}$

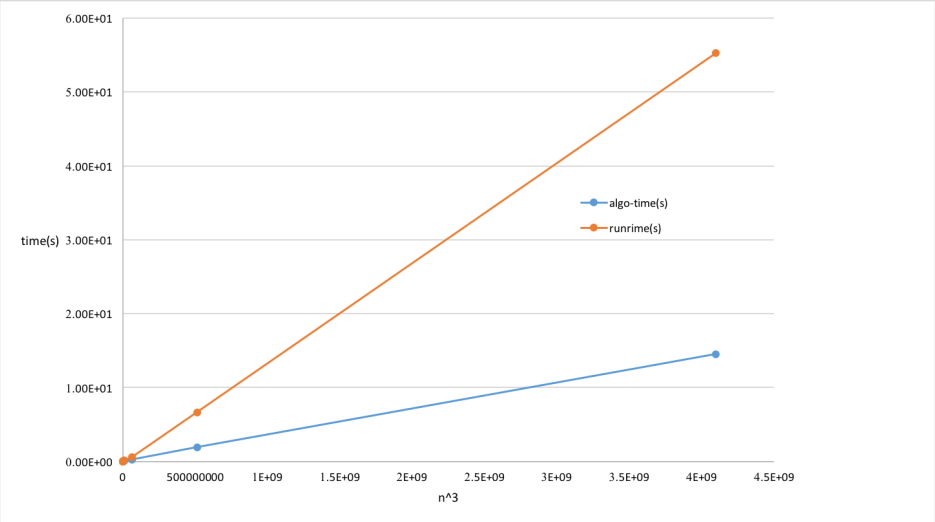
Table 1: Algorithm-1 performance

Mat	n	sigma	algo-time(s)	runrime(s)
m3	3	4.40E-15	4.05E-06	0.004
m4	10	3.06E-13	2.79E-05	0.005
m5	100	293.881	0.0118849	0.04
m6	200	477.474	0.042047	0.112
m7	400	717.106	0.242979	0.634
m8	800	1069.88	1.92496	6.629
m9	1600	1531.34	14.5427	55.245

3.2.2 n^3 and runtime

Because it is a $O(n^3)$ problem, so n^3 and runtime shoule be a straight line, shown by fig [1].

Figure 1: Algorithm-1 runtime to n^3



3.3 Algorithm-2 performance

Table [2] shows detail result of Algorithm-2

3.3.1 Sigma

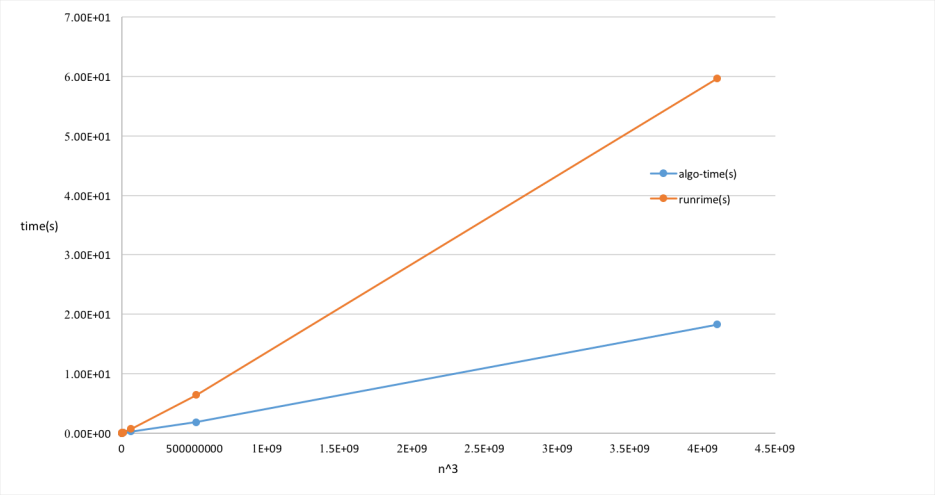
Although when n is getting larger, we will also get larger σ, but the slope σ grows is much smaller than Algorithm-1.

Table 2: Algorithm-2 performance

Mat	n	sigma	algo-time(s)	runrime(s)
m3	3	3.78E-15	4.05E-06	0.004
m4	10	1.29E-14	4.91E-05	0.009
m5	100	1.92E-13	0.00610399	0.017
m6	200	6.41E-13	0.039006	0.113
m7	400	2.29E-12	0.25009	0.663
m8	800	9.10E-12	1.84492	6.382
m9	1600	3.40E-11	18.2445	59.626

3.3.2 runtime to n^3

Figure 2: Algorithm-2 runtime to n^3



3.4 Algorithm-3 performance

Table [3] shows detail results of Algorithm-3.

Table 3: Algorithm-3 performance

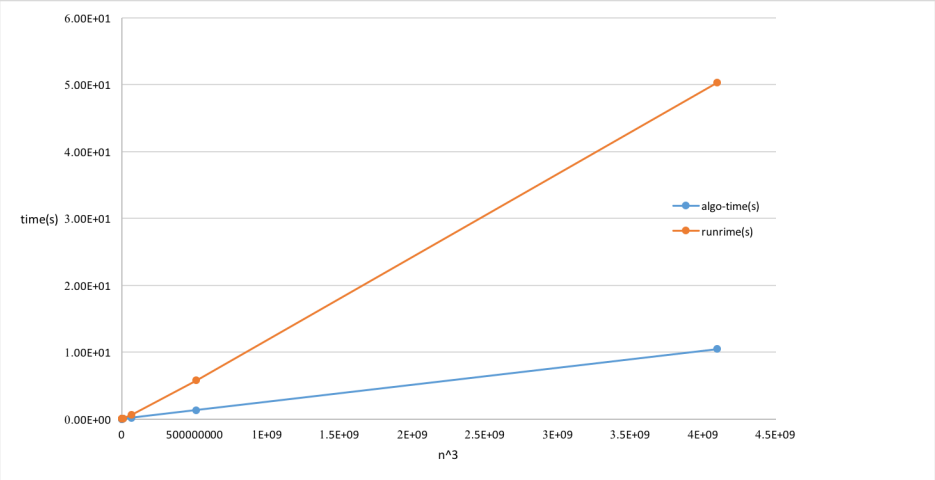
Mat	n	sigma	algo-time(s)	runrime(s)
m3	3	7.87E-15	1.91E-06	0.004
m4	10	7.83E-15	3.60E-05	0.015
m5	100	1.86E-13	0.00776792	0.02
m6	200	6.35E-13	0.022558	0.094
m7	400	2.32E-12	0.164832	0.586
m8	800	8.79E-12	1.33471	5.788
m9	1600	3.45E-11	10.458	50.275

3.4.1 Sigma

From Table [3], we can found that sigma in algorithm-2 and algoritm-3 is almost same, so they have the same accuracy.

3.4.2 runtime to n^3

Figure 3: Algorithm-3 runtime to n^3



3.5 Comparison

3.5.1 Complexity

As mentioned in Section 2.4, the complexity of three algorithms is all $\boldsymbol{O(n^3)}$, so the runtime grows up with n^3.

3.5.2 Accuracy

From Section 3.2.1, we can found that Algorithm-1 is the worst method for **Gram-Schmidt process** because it is not quite accurate. However, Algorithm-2 and Algorithm-3 almost have the same accuracy.

3.5.3 Speed

From Table [2] and Table [3], we found that both algo-time and runtime consumed by Algorithm-3 are a little smaller than Algorithm-2, so Algorithm-3 is faster than Algorithm-2.

3.5.4 Runtime Problem

From Table [1][2][3], we can see that algo-time only takes about $\frac{1}{4}$ of total execution time; After several experiments, I also found that the most time-consuming part is not our algorithm itself; Instead, it's the calculation of $G^T * G$. However, the complexity of matrix multiplication is also $O(n^3)$, which still makes sense in the result of Figure [1][2][3].