**CSE6643-HW1 Mini-Reports**

**Question 6**

**Algorithm**

1. Import numpy to use its matrix method including tril (forms lower triangular matrices), linalg.norm (gets p-norm of a matrix), and linalg.cond (gets p-norm condition number).
2. In make\_matrix, define a method that creates a lower triangular matrix with size of input m. Use numpy.random.normal to form a m-size matrix with entries of mean 0 and std sqr(m), and use numpy.tril to cut this matrix into a lower triangular matrix. Return this lower triangular matrix.
3. In main, perform the following process based on iteration i in [1:100]:
4. get desired size cur by 100 \* i;
5. call make\_matrix to make required matrix of size cur;
6. use numpy.linal.norm to compute 2-norm and inf-norm (my GT ID# is odd), and calculate the ratio;
7. use numpy.linal.cond to compute 2-norm condition number;
8. print the results;

**Result Analysis** (check sample output in the attached txt file)

In the sample output, as size increases, the ratio between 2-norm and inf-norm decreases, and the condition number increases. A possible reason behind these trends is that larger sized matrices are more likely to be ill-formed, which directly increases the condition number. Moreover, there is more likely to exist an entry that has a much higher absolute value than others, which will cause a large inf-norm that contributes to the decreasing ratio.

**Question 7**

**Algorithm**

1. Import numpy to create matrices and vectors using numpy.zeros. Import time to create stopwatches to calculate CPU times. Import matplotlib.pyplot to create plots and demonstrate end results.
2. In naive, performs naive gaussian elimination on input A and b similar to what we do by hands. Return reduced A and b.
3. In pivot, perform gaussian elimination with pivoting on input A and b. In each iteration, find the max\_index, swap rows if necessary, and perform the elimination process in naive method. Return reduced A and b.
4. In backSub, perform backward substitution based on inputs (reduced A and b). Return result x.
5. In discretize, based on given function in Q7's prompt and input (size n and lambda), produce A and b to solve. First, calculate h = 1/n^2 and call numpy.zeros to create empty matrices A and b. Populate A such that:
6. A's diagonal entries are 2.0 - lambda \* h\*\*2;
7. A's entries next to the diagonal entries are -1;

Populate b such that b's entries are (3 \* (i + 1)^2 \* h^2 - 0.5) \* h^2 for i in range [1, n]. Return populated A and b.

1. In main, store desired lambdas and n values for tests according to the prompt. Create empty maps (based on methods, keys are lambdas, values are CPU times) of lists to store computation results in the following steps and for demonstration use. For each lambda, perform the following steps on each n:
2. call discretize with current n and lambda to retrieve A and b to start with;
3. to avoid modification error, create copies of A and b for different methods;
4. start the timer, perform naive methods, end the timer, and store the elapsed time;
5. start the timer, perform pivot methods, end the timer, and store the elapsed time;

7. Demonstrate results of each lambdas by calling mathplotlib.pyplot.

**Result Analysis** (check sample output in the attached txt file)

Due to my computer’s computation limit, I did not compute n-value = 4000 for both lambda values.

In sample outputs, we can see as size grows, computational costs, in terms of CPU time, increases drastically for both naïve GE and GE with pivoting methods. Based on the data, trends of both methods seem to be cubic, which are the same to their big-O estimation.

The computational costs of the two methods are approximately the same, as the trendlines overlaps. GE with pivoting cost slightly more than naïve GE if observed closely. This result makes sense because GE with pivoting requires extra time to search and swap, which costs O(N) (each iteration) and is out-scaled by the time complexity of GE O(N^2).