COMPENG 3SK3: Computer-Aided Engineering
Assignment #2: Gauss Elimination

Instructor: Dr. Dongmei Zhao

Christabel Ehirim– L07 – EHIRIMC - 400137486

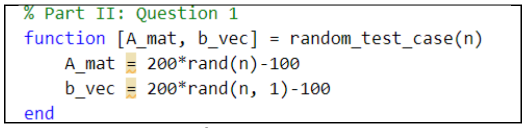
Lelna Gwet – L07 – GWETL - 400060749

<u>Part 1:</u>

See Appendix and attached Matlab files for complete code of GaussElimination() and GaussElimination_Pivoting() functions.

<u>Part 2: Implementation</u>

1. Using the built-in matlab function, rand(), random coefficient matrices and right-hand side vectors were generated for test cases.



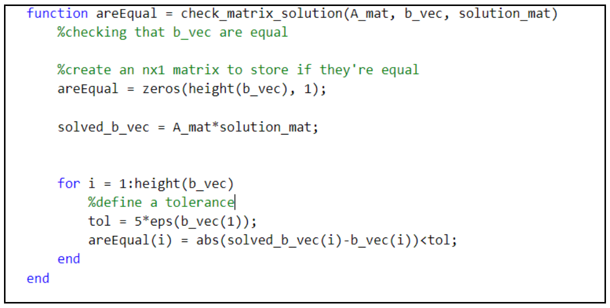
```
% Part II: Question 1
function [A_mat, b_vec] = random_test_case(n)
    A_mat = 200*rand(n)-100
    b_vec = 200*rand(n, 1)-100
end
```

Figure 1: Matlab function: random_test_case(n)

Rand(n) generates an *n x n* matrix with random floating-point digits between 0 and 1.0. By multiplying rand(n) by 200 and subtracting 100, a coefficient matrix with dimensions *n x n* is generated, with digits in the prescribed range (-100.0, 100). Similarly, Rand(n, 1) generates an nx1 column vector with random floating-point digits between 0 and 1.0. By multiplying rand(n, 1) by 200 and subtracting 100, a right-hand side vector with dimensions *n x n* is generated, with digits in the prescribed range (-100.0, 100).

2. (See Appendix A and attached Matlab files for complete implementation code).

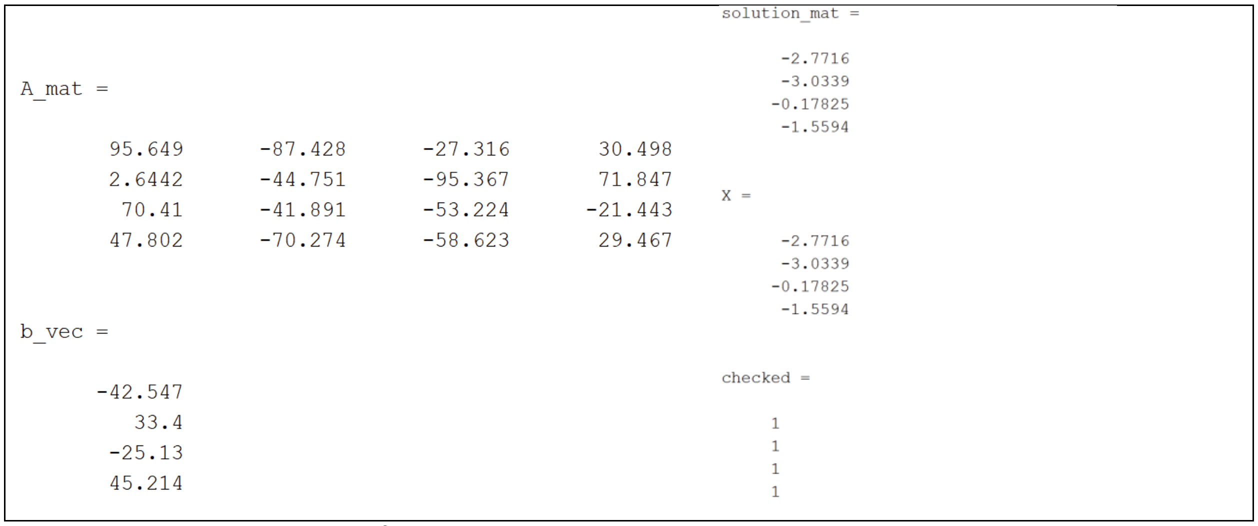
A function, *check_matrix_solution()*, takes a coefficient matrix, solution vector, and right-hand side vector and checks that the solution vector is correct by multiplying the solution vector with the coefficient matrix and comparing that resultant right-hand side vector with the right-hand side vector taken by the function.



```
function areEqual = check_matrix_solution(A_mat, b_vec, solution_mat)
    %checking that b_vec are equal

    %create an nx1 matrix to store if they're equal
    areEqual = zeros(height(b_vec), 1);

    solved_b_vec = A_mat*solution_mat;


    for i = 1:height(b_vec)
        %define a tolerance
        tol = 5*eps(b_vec(1));
        areEqual(i) = abs(solved_b_vec(i)-b_vec(i))<tol;
    end
end
```

Figure 2: Matlab function: check_matrix_solution(A_mat, b_vec, solution_mat)

To check that both right-hand side vectors are equivalent, each element in both vectors are compared individually. Since the values in both vectors are floating point digits, it can be difficult for them to be *exactly* equivalent for all decimal places in its precision. Rather than checking for equivalency with the logic equivalence, '==', the elements are first subtracted from each other. In theory, when two numbers are equivalent, the result of subtracting them would be zero. However, because these are floating point numbers that are not *exactly* equivalent, a tolerance, or very small number, is defined to quantify how "different" each number can be before they are considered different numbers (i.e. not equivalent). If the result of subtracting both values is less than the defined tolerance, then they are considered equivalent. Otherwise, they are considered not equivalent. The result of each element's equivalence is stored in a vector called, areEqual(). If the areEqual() vector is all 1's, then the right-hand side vectors are equal.



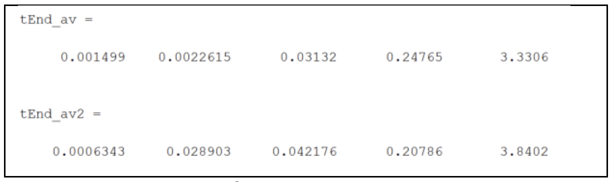
```
A_mat =

    95.649     -87.428     -27.316      30.498
    2.6442     -44.751     -95.367      71.847
     70.41     -41.891     -53.224     -21.443
    47.802     -70.274     -58.623      29.467


b_vec =

   -42.547
      33.4
    -25.13
    45.214
```

```
solution_mat =

   -2.7716
   -3.0339
  -0.17825
   -1.5594


X =

   -2.7716
   -3.0339
  -0.17825
   -1.5594


checked =

  1
  1
  1
  1
```

Figure 3: example of 4x4 matrix using check_matrix_solution(A_mat, b_vec, solution_mat)

Additionally, the Matlab built-in linear equation solver, linsolve(), was also used to verify the solution. For the 2x2, 3x3, and 4x4 systems tested, the answer from built-in solver appears to be consistently correct.

3. (See Appendix B and attached Matlab files for complete implementation code).



```
tEnd_av =

     0.001499    0.0022615     0.03132     0.24765      3.3306


tEnd_av2 =

    0.0006343     0.028903    0.042176     0.20786      3.8402
```

Figure 4a: Plotting points for Running time vs Matrix dimension, n.

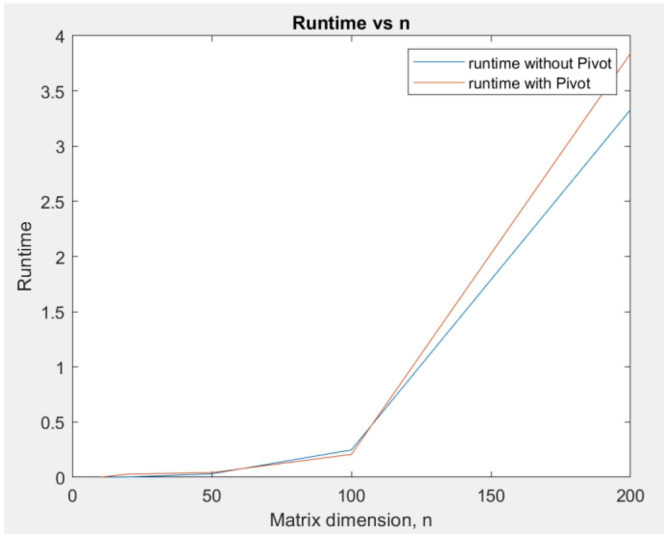


Figure 4b: Plot of Running time vs Matrix dimension, n.

As per Figure 4a, the running times for solving five randomly generated matrices of different sizes (n = 10, 20, 50, 100, 200) with the Gauss_Elimination() function were, respectively, recorded and stored in a *5 x 1* row vector, tEnd_av. Each matrix size's running time was repeated 10 times and an average of the 10 iterations was stored. The same procedure was repeated for the Gauss_Elimination_Pivoting() function and stored into the *5 x 1* row vector, tEnd_av2. The resulting vectors were plotted on the 'Runtime vs n' graph shown in Figure 4. Upon observing the resulting graph, it does appear to support the theoretical result that the running time on an *n x n* system is proportional to n^3. Though it is important to keep in mind that it is rare for a matrix to require pivoting and running times can also vary significantly within the same matrix size, a general trend was still observed between the running times required for an *n x n* matrix when using Gauss_Elimination() vs Gauss_Eliminiation_Pivoting(). In general, Gauss_Elimination_Pivoting() function tends to require slightly more running time to solve the same size matrix. At the worst case, based in the above example, it appears that adding pivoting to the Gauss Elimination algorithm adds ~15% of overhead time; which may prove to be significant at larger systems (i.e., n = 1000).

4. (See Appendix C and attached Matlab files for complete implementation code).

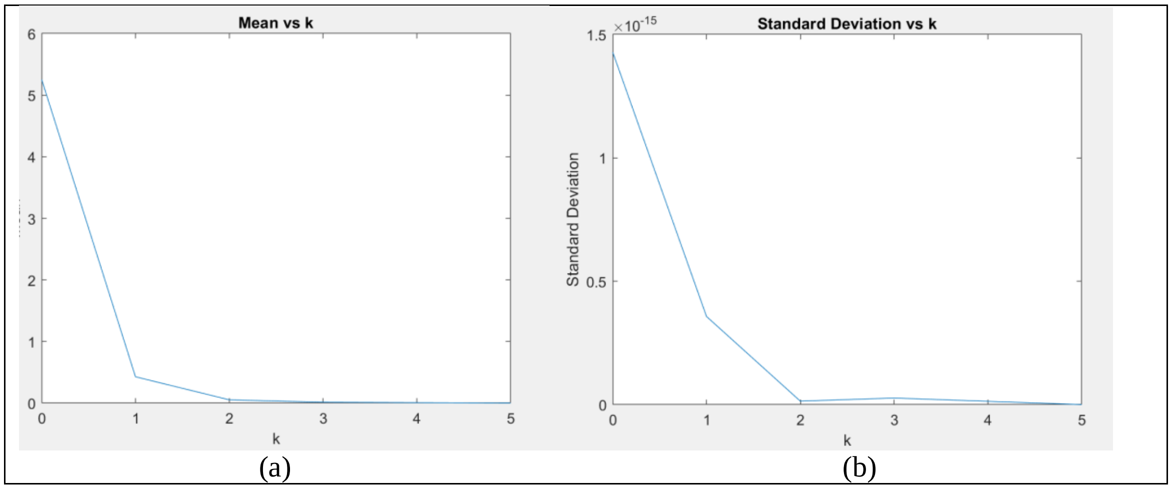


Figure 5: (a) Mean vs k plot; (b) Standard Deviation vs k plot.

An element of error, ranging from the order of ±1.0, was introduced to 100 randomly generated *10 x 10* matrices. To observe the effects of the error, for each *10 x 10* matrix, the difference between each element of the original system solution and the perturbed system solution was taken, squared, and summed. The square root of the result of this operation is then taken, resulting in the total error between the system solutions. The total error of each *10 x 10* system (out of the total 100 systems) is stored into the first column of a *100 x 5* matrix, errors_stored. Once all the errors have been computed for all 100 systems, the mean is calculated by the following definition:

$$Mean = \frac{\sum_{i=1}^{100}(\,total\ system\ errors)_i}{total\ \#\ of\ system\ errors}$$

Similarly, the standard deviation is calculated by following the following definition:

$$\sigma = \sqrt{\frac{\sum_{i=1}^{100}\left((\,total\ system\ error)_i - mean\right)^2}{(\,total\ \#\ of\ systems) - 1}}$$

This procedure was repeated with the following elements of error added: [-0.1, 0.1], [-0.01, 0.01], [-0.001, 0.001], [-0.00001, 0.00001]. Figure 5 represents a plot of the mean (5a) and standard deviation (5b) depending on the order of error introduced to a system, where k is from [-10^k, 10^{-k}]. As per the plot data, it can be observed that as the order of the added error decreases, the mean and standard deviation also decreases in a cubic manner.

Appendix A: Question 2 Implementation.

```
format short g
% testing Part II: Question 1
% random_test_case(6)

%Part II: Question 2

% WITHOUT PIVOTING
[A_mat, b_vec] = random_test_case(2);
solution_mat = Gauss_Elimination(A_mat, b_vec)
X = linsolve(A_mat, b_vec)
checked = check_matrix_solution(A_mat, b_vec, solution_mat)

[A_mat, b_vec] = random_test_case(3);
solution_mat = Gauss_Elimination(A_mat, b_vec)
checked = check_matrix_solution(A_mat, b_vec, solution_mat)
X = linsolve(A_mat, b_vec)

[A_mat, b_vec] = random_test_case(4)
solution_mat = Gauss_Elimination(A_mat, b_vec)
checked = check_matrix_solution(A_mat, b_vec, solution_mat)
X = linsolve(A_mat, b_vec)

% WITH PIVOTING [A_mat, b_vec] = random_test_case(2);
solution_mat = Gauss_Elimination_Pivoting(A_mat, b_vec)
X = linsolve(A_mat, b_vec)
checked = check_matrix_solution(A_mat, b_vec, solution_mat)

[A_mat, b_vec] = random_test_case(3);
solution_mat = Gauss_Elimination_Pivoting(A_mat, b_vec)
X = linsolve(A_mat, b_vec)
checked = check_matrix_solution(A_mat, b_vec, solution_mat)
```

```
[A_mat, b_vec] = random_test_case(4);

solution_mat = Gauss_Elimination_Pivoting(A_mat, b_vec)

X = linsolve(A_mat, b_vec)

checked = check_matrix_solution(A_mat, b_vec, solution_mat)
```

Appendix B: Question 3 Implementation.

```
format short g
n = [10, 20, 50, 100, 200];
tEnd = zeros(1, 10);
tEnd_av = zeros(1, 5);
for i = 1:5
      for j = 1:10
            [A_mat, b_vec] = random_test_case(n(1, i));
            TStart = tic;
            solution_mat = Gauss_Elimination(A_mat, b_vec);
            tEnd(j) = toc(TStart);
      end
      tEnd_av(i) = sum(tEnd)/height(tEnd);
end
tEnd_av

plot(n, tEnd_av)
hold on

n = [10, 20, 50, 100, 200];
tEnd = zeros(1, 10);
tEnd_av2 = zeros(1, 5);
for i = 1:5
      for j = 1:10
      [A_mat, b_vec] = random_test_case(n(1, i));
      TStart = tic;
      solution_mat = Gauss_Elimination_Pivoting(A_mat, b_vec);
      tEnd(j) = toc(TStart);
      end
      tEnd_av2(i) = sum(tEnd)/height(tEnd);
end
tEnd_av2
```

```
plot(n, tEnd_av2)
title('Runtime vs n')
xlabel('Matrix dimension, n')
ylabel('Runtime')
legend('runtime without Pivot', 'runtime with Pivot')
hold off
```

Appendix C: Question 4 Implementation.

```
format short g

%Part II: Question 4

%---------------------------FIRST: [-1.0, 1.0]---------------------------

% create a column vector to store the 100 error values

errors_stored = zeros(100, 5);

variance = zeros (1, 5);

std_dev = zeros (1, 5);

% generate 100 random 10 x 10 systems

n = 10;

for i = 1:100

      [A_mat, b_vec] = random_test_case(n);


      %record solution to system

      solution_mat = Gauss_Elimination(A_mat, b_vec)

      % "perturb system" --> add a number -1.0 to 1.0 to each element in

      % A_mat and b_vec


      %generate 10 by 10 matrix with numbers -1.0 to 1.0 and add it to A_mat
      A_mat_perturbed = A_mat+ (2*rand(n)-1);

      b_vec_perturbed = b_vec + (2*rand(n, 1)-1);

      perturbed_solution_mat = Gauss_Elimination(A_mat_perturbed,
      b_vec_perturbed)


      error_matrix = (solution_mat-perturbed_solution_mat).^2;

      sum_error = sum(error_matrix);

      error = sqrt(sum_error)

      errors_stored(i, 1) = error;

end

%---------------------------SECOND: [-0.1, 0.1]---------------------------

for i = 1:100

    [A_mat, b_vec] = random_test_case(n);
    %record solution to system
    solution_mat = Gauss_Elimination(A_mat, b_vec)

    % "perturb system" --> add a number -1.0 to 1.0 to each element in
    % A_mat and b_vec

    %generate 10 by 10 matrix with numbers -1.0 to 1.0 and add it to A_mat
    A_mat_perturbed = A_mat+ (0.2*rand(n)-0.1);
    b_vec_perturbed = b_vec + (0.2*rand(n, 1)-0.1);
    perturbed_solution_mat = Gauss_Elimination(A_mat_perturbed,
b_vec_perturbed)

    error_matrix = (solution_mat-perturbed_solution_mat).^2;
    sum_error = sum(error_matrix);
    error = sqrt(sum_error)
    errors_stored(i, 2) = error;
end
%---------------------------THIRD: [-0.01, 0.01]--------------------------
for i = 1:100
    [A_mat, b_vec] = random_test_case(n);

    %record solution to system
    solution_mat = Gauss_Elimination(A_mat, b_vec)

    % "perturb system" --> add a number -1.0 to 1.0 to each element in
    % A_mat and b_vec

    %generate 10 by 10 matrix with numbers -1.0 to 1.0 and add it to A_mat
    A_mat_perturbed = A_mat+ (0.02*rand(n)-0.01);
    b_vec_perturbed = b_vec + (0.02*rand(n, 1)-0.01);
    perturbed_solution_mat = Gauss_Elimination(A_mat_perturbed,
b_vec_perturbed)
```

```
    error_matrix = (solution_mat-perturbed_solution_mat).^2;
    sum_error = sum(error_matrix);
    error = sqrt(sum_error)
    errors_stored(i, 3) = error;
end
%---------------------------FOURTH: [-0.001, 0.001]-----------------------
for i = 1:100
    [A_mat, b_vec] = random_test_case(n);

    %record solution to system
    solution_mat = Gauss_Elimination(A_mat, b_vec)

    % "perturb system" --> add a number -1.0 to 1.0 to each element in
    % A_mat and b_vec

    %generate 10 by 10 matrix with numbers -1.0 to 1.0 and add it to A_mat
    A_mat_perturbed = A_mat+ (0.002*rand(n)-0.001);
    b_vec_perturbed = b_vec + (0.002*rand(n, 1)-0.001);
    perturbed_solution_mat = Gauss_Elimination(A_mat_perturbed,
b_vec_perturbed)

    error_matrix = (solution_mat-perturbed_solution_mat).^2;
    sum_error = sum(error_matrix);
    error = sqrt(sum_error)
    errors_stored(i, 4) = error;
end
%---------------------------FIFTH: [-0.00001, 0.00001]---------------------
for i = 1:100
    [A_mat, b_vec] = random_test_case(n);

    %record solution to system
    solution_mat = Gauss_Elimination(A_mat, b_vec)
    % "perturb system" --> add a number -1.0 to 1.0 to each element in
    % A_mat and b_vec

    %generate 10 by 10 matrix with numbers -1.0 to 1.0 and add it to A_mat
    A_mat_perturbed = A_mat+ (0.00002*rand(n)-0.00001);
    b_vec_perturbed = b_vec + (0.00002*rand(n, 1)-0.00001);
    perturbed_solution_mat = Gauss_Elimination(A_mat_perturbed,
b_vec_perturbed)

    error_matrix = (solution_mat-perturbed_solution_mat).^2;
    sum_error = sum(error_matrix);
    error = sqrt(sum_error)
    errors_stored(i, 5) = error;
end

mean = sum(errors_stored)./height(errors_stored)



for i=1:5
    variance(i) = (sum(errors_stored(:, i) -
mean(i)).^2)/(height(errors_stored)-1);
    std_dev(i) = sqrt(variance(i));
end
std_dev

% PLOTS
% mean vs k
% std_dev vs k
% k = 0 1 2 3 5
k = [0, 1, 2, 3, 5]
height(errors_stored)
errors_stored
```

```
figure(1)
plot(k, mean)

figure(2)
plot(k, std_dev)
```