Assignment 2

111550057 資工15 莊婷馨

1. I use Matlab to complete the process of Gaussian elimination with partial pivoting. To do partial pivoting, I select the row with the largest magnitude on or below diagonal and switch rows if needed. After pivoting, do Gaussian elimination.

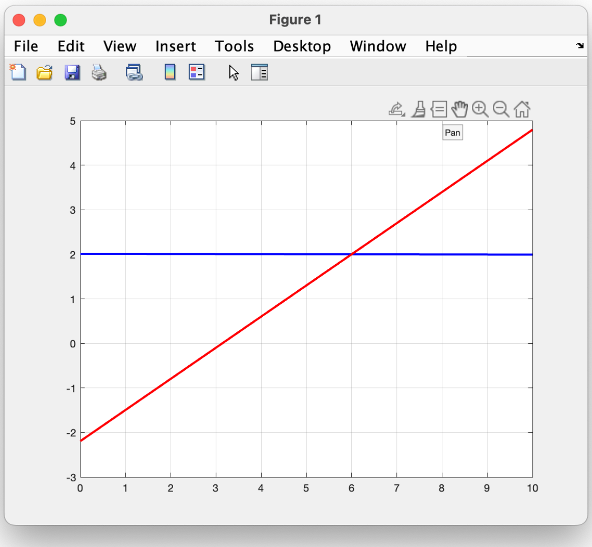
```
function x = gaussian_pivot(A,b)
    % Form augmented matrix
    aug = [A,b];

    % Forward elimination with partial pivoting
    for k = 1:n-1
        % Partial pivoting
        [~, max_row] = max(abs(aug(k:n, k)));   % [max number. max row]
        max_row = max_row + k - 1;
        if max_row ~= k
            % swap max_row and k
            aug([max_row,k], :) = aug([k, max_row], :);
        end

        % Forward elimination
        for i = k+1:n
            factor = aug(i,k) / aug(k,k);
            aug(i,k:n+1) = aug(i,k:n+1) - factor * aug(k,k:n+1);
        end
    end

    % Back substitution
    x = zeros(n,1);
    for k = n:-1:1
        x(k) = (aug(k,n+1) - aug(k,k+1:n) * x(k+1:n)) / aug(k,k);
    end
end
```

gaussian_pivot.m

The result: No row interchange needed.

```
>> problem1
x1=3.2099
x2=0.23457
x3=0.71605
```

2. I first graph the system using Matlab. It is clear that the system intersects at (6, 2).



Then I calculated the system by hand.

(a) x = [10, 1.99]

(b) x = [6.02, 2.01]

(a)

$$\left[\begin{array}{cc|c} 0.1 & 51.7 & 104 \\ 5.1 & -7.3 & 16 \end{array}\right]$$

$$\left[\begin{array}{cc|c} 0.1 & 51.7 & 104 \\ 0 & -2650 & -5280 \end{array}\right]$$

$$x_2 = \frac{-5280}{-2650} \approx 1.99$$

$$x_1 = \frac{104 - 103}{0.1} = 10$$

(b)

$$\left[\begin{array}{cc|c} 5.1 & -7.3 & 16 \\ 0.1 & 51.7 & 104 \end{array}\right] \Big) \, 0.0196$$

$$\left[\begin{array}{cc|c} 5.1 & -7.3 & 16 \\ 0 & 51.8 & 104 \end{array}\right]$$

$$x_2 = \frac{104}{51.8} \approx 2.01$$

$$x_1 = \frac{16 + 14.7}{5.1} \approx 6.02$$

(c) x = [5.99, 2]

The result of (c) match neither (a) nor (b), but is much closer to that of (b). The result after pivoting is more accurate than the result without pivoting. The result after scaled partial pivoting is more accurate than partial pivoting.

(c)

$$\begin{bmatrix} 0.0193 & 0 \\ 0 & 0.137 \end{bmatrix} \left[\begin{array}{cc|c} 0.1 & 51.7 & 104 \\ 5.1 & -7.3 & 16 \end{array}\right]$$

$$= \left[\begin{array}{cc|c} 0.00193 & 1 & 2.01 \\ 0.699 & -1 & 2.19 \end{array}\right]$$

$$\left[\begin{array}{cc|c} 0.699 & -1 & 2.19 \\ 0.00193 & 1 & 2.01 \end{array}\right] \Big) \, 0.00276$$

$$\approx \left[\begin{array}{cc|c} 0.699 & -1 & 2.19 \\ 0 & 1 & 2.00 \end{array}\right]$$

$$x_2 = \frac{2}{1} \approx 2$$

$$x_1 = \frac{4.19}{0.699} \approx 5.99$$

3. Do the LU factorization by hand. Do calculation row by row to get the result.

$$A = \begin{bmatrix} 2 & -1 & 3 & 2 \\ 2 & 2 & 0 & 4 \\ 1 & 1 & -2 & 2 \\ 1 & 3 & 4 & -1 \end{bmatrix}$$

$$= \begin{bmatrix} 1 & 0 & 0 & 0 \\ 1 & 1 & 0 & 0 \\ \frac{1}{2} & \frac{1}{2} & 1 & 0 \\ \frac{1}{2} & \frac{7}{6} & -3 & 1 \end{bmatrix} \begin{bmatrix} 2 & -1 & 3 & 2 \\ 0 & 3 & -3 & 2 \\ 0 & 0 & -2 & 0 \\ 0 & 0 & 0 & \frac{-13}{3} \end{bmatrix}$$

$$= LU$$

Times 2 to L to make its diagonal all 2's. Times $\frac{1}{2}$ to U to make the result remain the same.

Then

$$A = \begin{bmatrix} 2 & 0 & 0 & 0 \\ 2 & 2 & 0 & 0 \\ 1 & 1 & 2 & 0 \\ 1 & \frac{7}{3} & -6 & 2 \end{bmatrix} \begin{bmatrix} 1 & \frac{-1}{2} & \frac{3}{2} & 1 \\ 0 & \frac{3}{2} & \frac{-3}{2} & 1 \\ 0 & 0 & -1 & 0 \\ 0 & 0 & 0 & \frac{-13}{6} \end{bmatrix}$$

4. Use Matlab to implement a function to perform the Jacobi method. Implement the process of ensuring diagonally dominance in the function for convenience. Then iterate until accuracy to 5 significant digits.

```matlab
function [x,iter] = jacobi(A, b, x0, max_iter)
    % Initialize variables
    n = length(b);
    x = x0;
    iter = 0;
    x_prev = x0;
    for i=1:(n-1)     % Check if diagonally dominant
        [~,max_row] = max(abs(A(i:n, i)));
        max_row = max_row +i -1;
        if max_row ~= i
            % swap max_row and i
            A([max_row,i], :) = A([i, max_row], :);
            b([max_row,i], :) = b([i, max_row], :);
        end
    end

    % Iterate until convergence or max iterations
    while iter < max_iter
        for j = 1 : n
            x(j) = (b(j)/A(j,j)) - ((A(j,[1:j-1,j+1:n]) * x_prev([1:j-1,j+1:n])) / A(j,j));
        end

        % Check for convergence
        x_prev_r = round(x_prev,5,'significant');
        rounded_x = round(x,5,'significant');
        if x_prev_r == rounded_x
            break
        end
        iter = iter + 1;   % increment iteration counter
        x_prev = x;
    end
end
```

jacobi.m

The result:

```
x =
   -0.1433
   -1.3746
    0.7199

iterations used: 32
```

5. Use Matlab to implement a function to perform the Gauss-Seidel method. Implement the process of ensuring diagonally dominance in the function for convenience. Then iterate until accuracy to 5 significant digits.

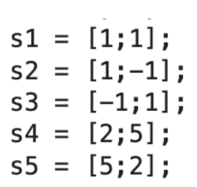
```
function [x,iter] = gauss_seidel(A, b, x0, max_iter)
    n = length(b);
    x = x0;
    iter = 0;
    x_prev = x0;
    for i=1:(n-1)
        [~,max_row] = max(abs(A(i:n, i)));
        max_row = max_row +i -1;
        if max_row ~= i
            % swap max_row and i
            A([max_row,i], :) = A([i, max_row], :);
            b([max_row,i], :) = b([i, max_row], :);
        end
    end

    % Iterate until convergence or max iterations
    while iter < max_iter
        for j = 1 : n
            x(j) = (b(j)/A(j,j)) - ((A(j,[1:j-1,j+1:n]) * x([1:j-1,j+1:n])) / A(j,j));
        end

        % Check for convergence
        x_prev_r = round(x_prev,5,'significant');
        rounded_x = round(x,5,'significant');
        if x_prev_r == rounded_x
            break
        end

        % Increment iteration counter
        iter = iter + 1;
        x_prev = x;
    end
end
```

gauss_seidel.m

The function only differs from the previous one in the timing the new value of x is used. In Gauss-Seidel, x is updated once it is calculated. In Jacobi, we wait until one iteration is completed to update the value of x.

The result: Gauss-Seidel method uses fewer iterations than Jacobi method.

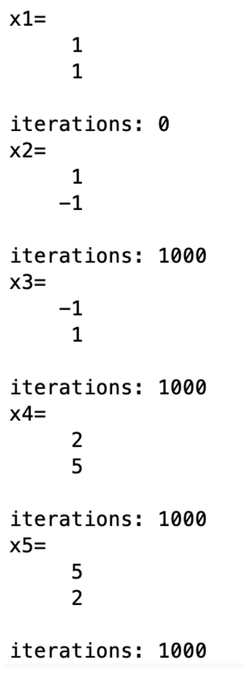
```
x =
   -0.1433
   -1.3746
    0.7199

iterations used: 13
```

6. Let 

```
s1 = [1;1];
s2 = [1;-1];
s3 = [-1;1];
s4 = [2;5];
s5 = [5;2];
```

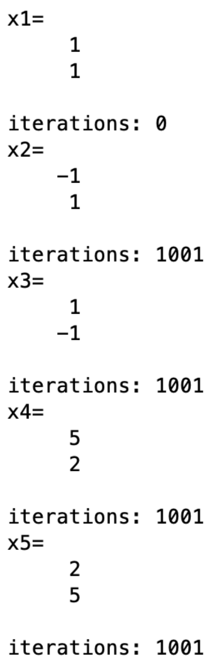
and get the results as x1,x2,x3,x4,x5 accordingly.

(a) Use the function jacobi to get the results. Compare between the results of max iteration = 1000 and 1001.



```
x1=
     1
     1

iteration: 0
x2=
     1
    -1

iterations: 1000
x3=
    -1
     1

iterations: 1000
x4=
     2
     5

iterations: 1000
x5=
     5
     2

iterations: 1000
```

max_iter=1000



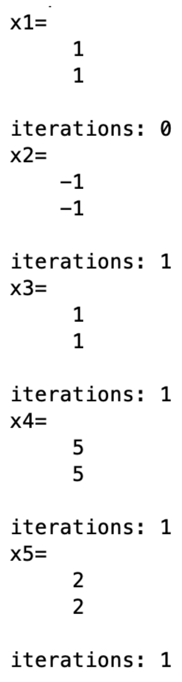
```
x1=
     1
     1

iterations: 0
x2=
    -1
     1

iterations: 1001
x3=
     1
    -1

iterations: 1001
x4=
     5
     2

iterations: 1001
x5=
     2
     5

iterations: 1001
```

max_iter=1001

Since s1 already satisfies $x = y$, it does not need any iteration.

The other starting vectors s=[a, b] iterate repeatedly between [b, a] and [a, b], which means the process will not converge.

(b) Use the function gauss_seidel to get the results.

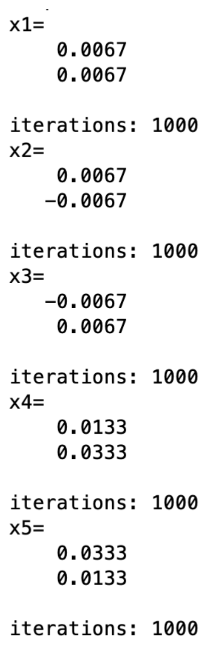


```
x1=
     1
     1

iterations: 0
x2=
    -1
    -1

iterations: 1
x3=
     1
     1

iterations: 1
x4=
     5
     5

iterations: 1
x5=
     2
     2

iterations: 1
```

Since x can be updated immediately after calculation, it only take one iteration for s2, s3, s4, s5 to get a feasible pair.

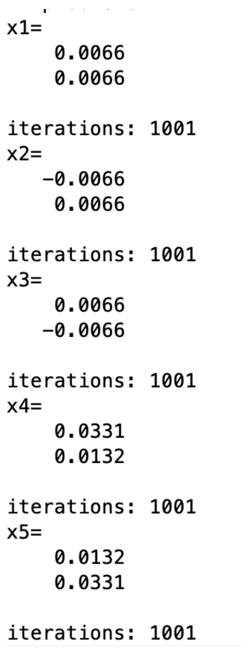
No iteration is required for s1 because it already satisfies $x = y$.

(c) Use Jacobi method to get the results:



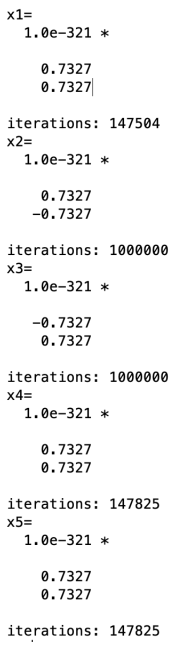
```
x1=
    0.0067
    0.0067

iterations: 1000
x2=
    0.0067
   -0.0067

iterations: 1000
x3=
   -0.0067
    0.0067

iterations: 1000
x4=
    0.0133
    0.0333

iterations: 1000
x5=
    0.0333
    0.0133

iterations: 1000
```

max_iter = 1000



```
x1=
    0.0066
    0.0066

iterations: 1001
x2=
   -0.0066
    0.0066

iterations: 1001
x3=
    0.0066
   -0.0066

iterations: 1001
x4=
    0.0331
    0.0132

iterations: 1001
x5=
    0.0132
    0.0331

iterations: 1001
```

max_iter = 1001

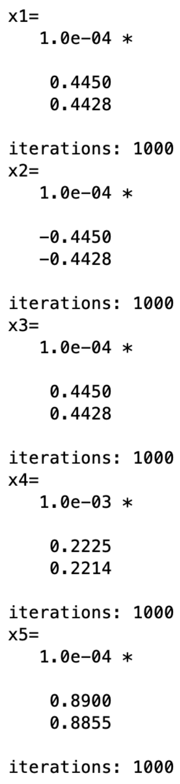


```
x1=
  1.0e-321 *

    0.7327
    0.7327

iterations: 147504
x2=
  1.0e-321 *

    0.7327
   -0.7327

iterations: 1000000
x3=
  1.0e-321 *

   -0.7327
    0.7327

iterations: 1000000
x4=
  1.0e-321 *

    0.7327
    0.7327

iterations: 147825
x5=
  1.0e-321 *

    0.7327
    0.7327

iterations: 147825
```

max_iter = 1000000

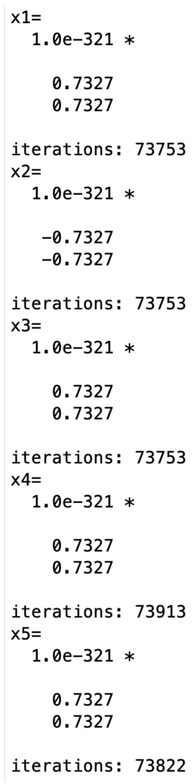
Results for s1, s4, s5 converge to a very small value at 147504 ,147825 and 147825 iterations. I think the results for s2 and s3 will not converge, since the values in the vector have opposite sign.

Use Gauss-Seidel method to get the results:



```
x1=
   1.0e-04 *

    0.4450
    0.4428

iterations: 1000
x2=
   1.0e-04 *

   -0.4450
   -0.4428

iterations: 1000
x3=
   1.0e-04 *

    0.4450
    0.4428

iterations: 1000
x4=
   1.0e-03 *

    0.2225
    0.2214

iterations: 1000
x5=
   1.0e-04 *

    0.8900
    0.8855

iterations: 1000
```

max_iter = 1000



```
x1=
  1.0e-321 *

    0.7327
    0.7327

iterations: 73753
x2=
  1.0e-321 *

   -0.7327
   -0.7327

iterations: 73753
x3=
  1.0e-321 *

    0.7327
    0.7327

iterations: 73753
x4=
  1.0e-321 *

    0.7327
    0.7327

iterations: 73913
x5=
  1.0e-321 *

    0.7327
    0.7327

iterations: 73822
```

max_iter = 1000000

All results converge at around 73700 to 74000 iterations.