Caitlyn Jones

Prof. Thomas Fai

MATH 122A: Numerical Methods and Big Data

22 September 2024

Homework 3

1. Given $A = \begin{pmatrix} 3 & 4 & 5 \\ 1 & 0 & 1 \\ -2 & 2 & 2 \end{pmatrix}$ and $b = \begin{pmatrix} 1 \\ 0 \\ -1 \end{pmatrix}$, compute $A\underline{x} = \underline{b}$.

   a. $\begin{pmatrix} 3 & 4 & 5 \\ 1 & 0 & 1 \\ -2 & 2 & 2 \end{pmatrix} \begin{pmatrix} x_1 \\ x_2 \\ x_3 \end{pmatrix} = \begin{pmatrix} 1 \\ 0 \\ -1 \end{pmatrix}$

   Pivot = 3 and multipliers are $\frac{1}{3}$ and $-\frac{2}{3}$

   First, eliminate x_1 from rows two and three using the following equations:

   $row2 = row2 - \frac{1}{3}row1$ and $row3 = row3 + \frac{2}{3}row1$.

   $row2$:
   $$1 - \frac{1}{3}(3) = 0$$
   $$0 - \frac{1}{3}(4) = -\frac{4}{3}$$
   $$1 - \frac{1}{3}(5) = -\frac{2}{3}$$
   $$0 - \frac{1}{3}(1) = -\frac{1}{3}$$

   and

   $row3$:
   $$-2 + \frac{2}{3}(3) = 0$$
   $$2 + \frac{2}{3}(4) = \frac{14}{3}$$
   $$2 + \frac{2}{3}(5) = \frac{16}{3}$$
   $$-1 + \frac{2}{3}(1) = -\frac{1}{3}$$

   After computing row2 and row3, we get the following matrices:

   $$A = \begin{pmatrix} 3 & 4 & 5 \\ 0 & -\frac{4}{3} & -\frac{2}{3} \\ 0 & \frac{14}{3} & \frac{16}{3} \end{pmatrix} \text{ and } b = \begin{pmatrix} 1 \\ \frac{-1}{3} \\ \frac{-1}{3} \end{pmatrix}$$

   Now, since $\left|\frac{14}{3}\right| > \left|\frac{-4}{3}\right|$, we are going to swap rows 2 and 3 to get the following new matrices:

$$A = \begin{pmatrix} 3 & 4 & 5 \\ 0 & \frac{14}{3} & \frac{16}{3} \\ 0 & \frac{-4}{3} & \frac{-2}{3} \end{pmatrix} \text{ and } b = \begin{pmatrix} 1 \\ \frac{-1}{3} \\ \frac{-1}{3} \end{pmatrix}$$

Now, our new pivot is $\frac{14}{3}$ and our new multiplier is $\frac{\frac{-4}{3}}{\frac{14}{3}}$ or $\frac{-2}{7}$.

Now, we must eliminate x_2 from row three using the following equation:

$$row3 = row3 + \frac{2}{7} row2$$

$$row3:$$
$$0 + \frac{2}{7}(0) = 0$$
$$-\frac{4}{3} + \frac{2}{7}\left(\frac{14}{3}\right) = 0$$
$$-\frac{2}{3} + \frac{2}{7}\left(\frac{16}{3}\right) = \frac{6}{7}$$
$$-\frac{1}{3} + \frac{2}{7}\left(\frac{-1}{3}\right) = \frac{-3}{7}$$

Now, we have completed forward elimination to receive the following matrices:

$$A = \begin{pmatrix} 3 & 4 & 5 \\ 0 & \frac{14}{3} & \frac{16}{3} \\ 0 & 0 & \frac{6}{7} \end{pmatrix} \text{ and } b = \begin{pmatrix} 1 \\ \frac{-1}{3} \\ \frac{-3}{7} \end{pmatrix}$$

Now, we can solve for x_1, x_2, and x_3 using the following system of equations:

$$\begin{aligned} 3x_1 + 4x_2 + 5x_3 &= 1 \\ \frac{14}{3}x_2 + \frac{16}{3}x_3 &= \frac{-1}{3} \\ \frac{6}{7}x_3 &= \frac{-3}{7} \end{aligned}$$

, solving for each one we get:

$$\frac{6}{7}x_3 = -\frac{3}{7}$$
$$x_3 = -\frac{1}{2}$$

$$\frac{14}{3}x_2 + \frac{16}{3}x_3 = \frac{-1}{3}$$
$$\frac{14}{3}x_2 + \frac{16}{3}\left(-\frac{1}{2}\right) = \frac{-1}{3}$$
$$\frac{14}{3}x_2 + \frac{-8}{3} = \frac{-1}{3}$$
$$\frac{14}{3}x_2 = \frac{7}{3}$$
$$x_2 = \frac{1}{2}$$

$$3x_1 + 4x_2 + 5x_3 = 1$$
$$3x_1 + 4\left(\frac{1}{2}\right) + 5\left(-\frac{1}{2}\right) = 1$$
$$3x_1 + 2 - \frac{5}{2} = 1$$
$$3x_1 - \frac{1}{2} = 1$$
$$3x_1 = \frac{3}{2}$$
$$x_1 = \frac{1}{2}$$

So, x $= \begin{pmatrix} \frac{1}{2} \\ \frac{1}{2} \\ \frac{-1}{2} \end{pmatrix}$.

b. $b = \begin{pmatrix} 2 \\ 2 \\ -1 \end{pmatrix}$, L $= \begin{pmatrix} 1 & 0 & 0 \\ \frac{-2}{3} & 1 & 0 \\ \frac{1}{3} & \frac{-2}{7} & 1 \end{pmatrix}$, U $= \begin{pmatrix} 3 & 4 & 5 \\ 0 & \frac{14}{3} & \frac{16}{3} \\ 0 & 0 & \frac{6}{7} \end{pmatrix}$, and P $= \begin{pmatrix} 1 & 0 & 0 \\ 0 & 0 & 1 \\ 0 & 1 & 0 \end{pmatrix}$.

First, solving for y:

$Ly = Pb$

$Ly = \begin{pmatrix} 2 \\ -1 \\ 2 \end{pmatrix}$

$$\begin{pmatrix} 1 & 0 & 0 \\ \frac{-2}{3} & 1 & 0 \\ \frac{1}{3} & \frac{-2}{7} & 1 \end{pmatrix}\begin{pmatrix} y_1 \\ y_2 \\ y_3 \end{pmatrix} = \begin{pmatrix} 2 \\ -1 \\ 2 \end{pmatrix}$$

Solving for y we get:

$$y_1 = 2$$
$$\frac{-2}{3}y_1 + y_2 = -1$$
$$\frac{1}{3}y_1 - \frac{2}{7}y_2 + y_3 = 2$$

$$y_1 = 2$$
$$\frac{-2}{3}y_1 + y_2 = 1$$
$$\frac{-2}{3}(2) + y_2 = 1$$
$$y_2 = \frac{1}{3}$$

$$\frac{1}{3}y_1 - \frac{2}{7}y_2 + y_3 = 2$$
$$\frac{1}{3}(2) - \frac{2}{7}\left(\frac{1}{3}\right) + y_3 = 2$$
$$\frac{2}{3} - \frac{2}{21} + y_3 = 2$$
$$\frac{4}{7} + y_3 = 2$$
$$y_3 = \frac{10}{7}$$

Therefore, $y = \begin{pmatrix} 2 \\ \frac{1}{3} \\ \frac{10}{7} \end{pmatrix}$.

Now, to solve for x:

$Ux = y$

$$Ux = \begin{pmatrix} 2 \\ \frac{1}{3} \\ \frac{10}{7} \end{pmatrix}$$

$$\begin{pmatrix} 3 & 4 & 5 \\ 0 & \frac{14}{3} & \frac{16}{3} \\ 0 & 0 & \frac{6}{7} \end{pmatrix} \begin{pmatrix} x_1 \\ x_2 \\ x_3 \end{pmatrix} = \begin{pmatrix} 2 \\ \frac{1}{3} \\ \frac{10}{7} \end{pmatrix}$$

Solving for x we get:

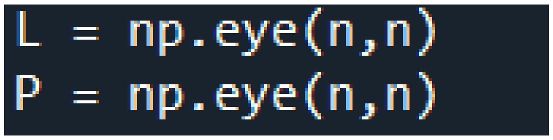
$$\begin{gathered} 3x_1 + 4x_2 + 5x_3 = 2 \\ \frac{14}{3}x_1 + \frac{16}{3}x_2 = \frac{1}{3} \\ \frac{6}{7}x_3 = \frac{10}{7} \\ x_3 = \frac{5}{3} \end{gathered}$$

$$\begin{gathered} \frac{14}{3}x_2 + \frac{16}{3}x_3 = \frac{1}{3} \\ \frac{14}{3}x_2 + \frac{16}{3}\left(\frac{5}{3}\right) = \frac{1}{3} \\ \frac{14}{3}x_2 + \frac{80}{9} = \frac{1}{3} \\ \frac{14}{3}x_2 = \frac{-77}{9} \\ x_2 = -\frac{11}{6} \end{gathered}$$

$$\begin{gathered} 3x_1 + 4x_2 + 5x_3 = 2 \\ 3x_1 + 4\left(-\frac{11}{6}\right) + 5\left(\frac{5}{3}\right) = 2 \\ 3x_1 - \frac{22}{3} + \frac{25}{3} = 2 \\ 3x_1 + 1 = 2 \\ 3x_1 = 1 \\ x_1 = \frac{1}{3} \end{gathered}$$

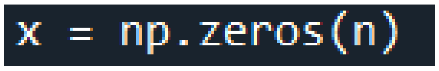
Therefore, x $= \begin{pmatrix} \frac{1}{3} \\ \frac{-11}{6} \\ \frac{5}{3} \end{pmatrix}$.

c. I first created matrices L and P, size n x n, using the numpy identity matrix function, as both of these matrices will have ones along the diagonal and zeros above and below the diagonal (L will later have multipliers below the diagonal and P will have zeros below the diagonal, but there will also be row swapping in P to retrieve a permutation matrix that is different than the identity matrix above).



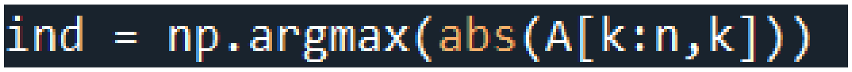
```
L = np.eye(n,n)
P = np.eye(n,n)
```

Next, I initialized the solution vector x. The values are all zero because they will be filled in with the computed values once the program is run.



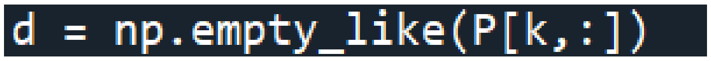
```
x = np.zeros(n)
```

Then, I began the steps for Gaussian Elimination using partial pivoting. To start, I first needed to find the maximum index in the k-th column or find the largest absolute value of the all the values in the given k-th column. This index will end up being our pivot value for this round of elimination.



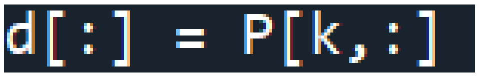
```
ind = np.argmax(abs(A[k:n,k]))
```

Next, after row swapping for matrices A, L, and b, we need to row swap for matrix P, which we initialized earlier as an identity matrix with ones along the diagonal and zeros above and below the diagonal. However, because we have row swapped the other matrices, we also create P as the permutation matrix in order to correctly compute Ly = Pb and Ux = y. To do so, we will first store the k-th row of the matrix P in an empty array that is the same size as the k-th row.



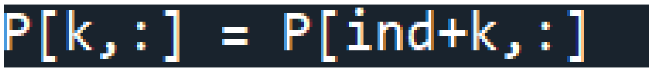
```
d = np.empty_like(P[k,:])
```

In order to store the contents of the k-th row during the row swaps, we need to now copy the contents of the k-th row into the new temporary array.



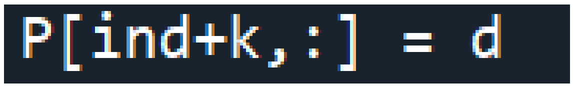
```
d[:] = P[k,:]
```

Then, we are going to swap the rows of P with the row at the index with the largest pivot value that we found earlier during the partial pivoting process.

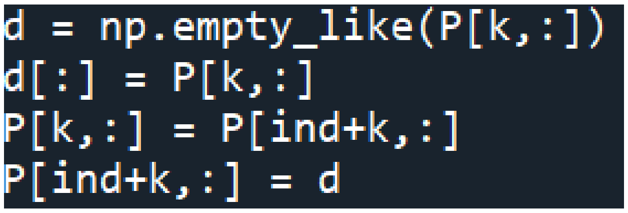


```
P[k,:] = P[ind+k,:]
```

Lastly, we need to put the contents of the original k-th that we stored in d into the row that has the largest pivot value.

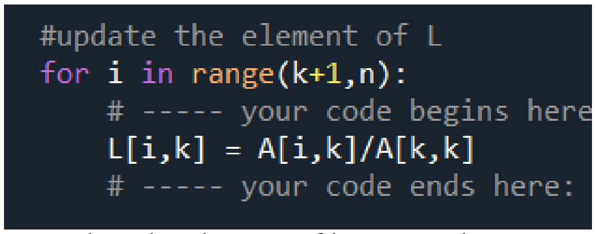


```
P[ind+k,:] = d
```



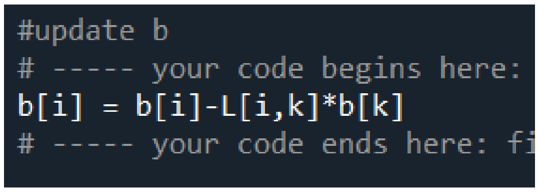
```
d = np.empty_like(P[k,:])
d[:] = P[k,:]
P[k,:] = P[ind+k,:]
P[ind+k,:] = d
```

After we have completed the row swapping, we then need to update the values of L, A, and b. To update the element of L, we must retrieve the multiplier of this iteration of partial pivoting, which will the be the value the current index of A divided by the current pivot value of A.



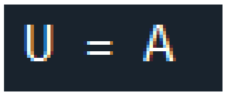
```
#update the element of L
for i in range(k+1,n):
    # ----- your code begins here
    L[i,k] = A[i,k]/A[k,k]
    # ----- your code ends here:
```

To update the element of b, we need to compute row operations with the multiplier retrieved from updating L and the current value of b.



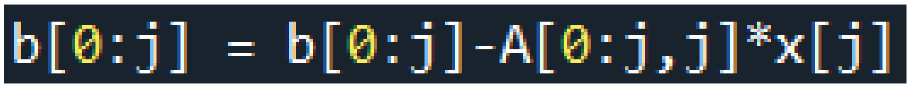
```
#update b
# ----- your code begins here:
b[i] = b[i]-L[i,k]*b[k]
# ----- your code ends here: fi
```

Next, to retrieve matrix U, we need to copy matrix A, as matrix U is equivalent to matrix A *after* all row operations have ended.



```
U = A
```

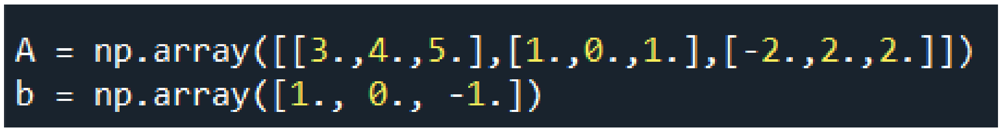
Lastly, we need to use backward substitution to solve for x, given b and A.



```
b[0:j] = b[0:j]-A[0:j,j]*x[j]
```

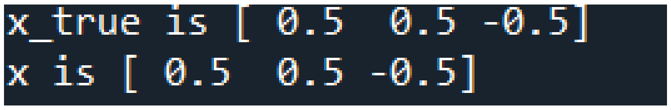
Using matrices given in part (a) and (b), let’s confirm that this program works.

For part (a):



```
A = np.array([[3.,4.,5.],[1.,0.,1.],[-2.,2.,2.]])
b = np.array([1., 0., -1.])
```

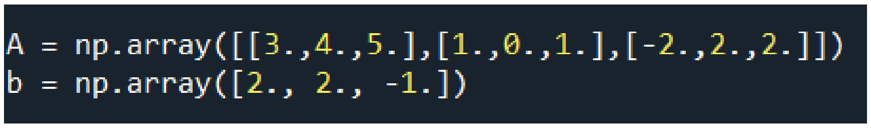
This is the result of solving the equation Ax = b:



```
x_true is [ 0.5  0.5 -0.5]
x is [ 0.5  0.5 -0.5]
```

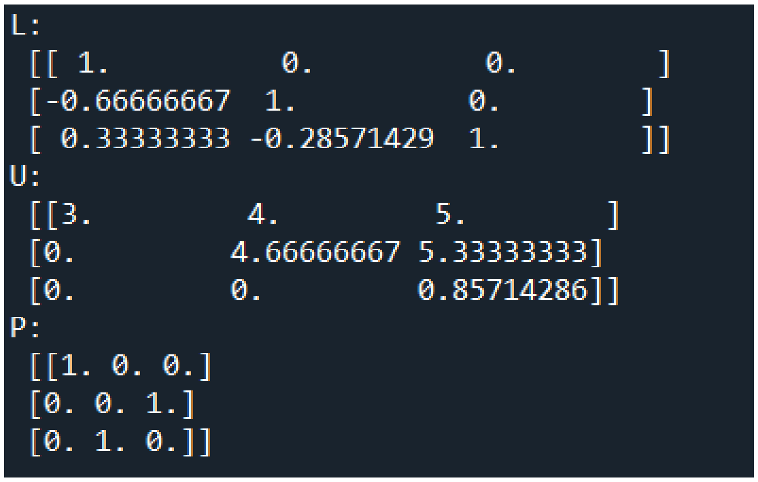
which is correct.

For part (b), we need to solve the equations Ly = Pb and Ux = y.



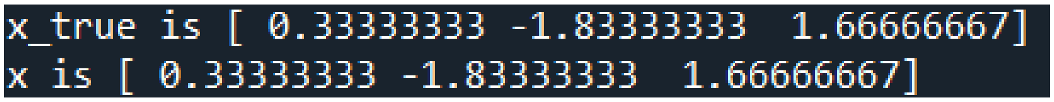
```
A = np.array([[3.,4.,5.],[1.,0.,1.],[-2.,2.,2.]])
b = np.array([2., 2., -1.])
```

Here are the following matrices L, U, and P:



```
L:
 [[ 1.          0.          0.        ]
 [-0.66666667  1.          0.        ]
 [ 0.33333333 -0.28571429  1.        ]]
U:
 [[3.         4.         5.        ]
 [0.         4.66666667 5.33333333]
 [0.         0.         0.85714286]]
P:
 [[1. 0. 0.]
 [0. 0. 1.]
 [0. 1. 0.]]
```

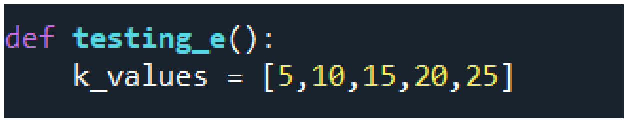
which were the matrices I got when computing by hand. Finally, here is x as the result of computing Ly = Pb and Ux = y:



```
x_true is [ 0.33333333 -1.83333333  1.66666667]
x is [ 0.33333333 -1.83333333  1.66666667]
```

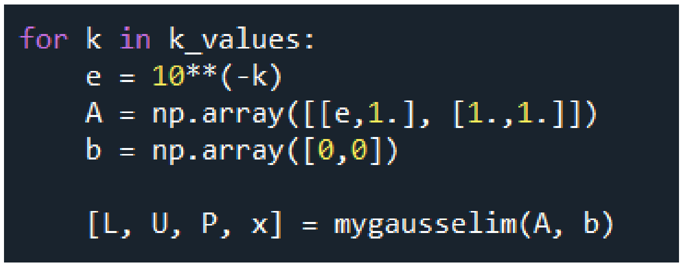
which is correct.

d. To find the LU factorizations for A, with $\epsilon = 10^{-k}$ and k = 5, 10, 15, 20, 25, I created a method `testing_e()` that will compute the values of L,U,P, L*U and P*A for each ϵ in a loop. To start, I created a variable k_values that will hold each k value.



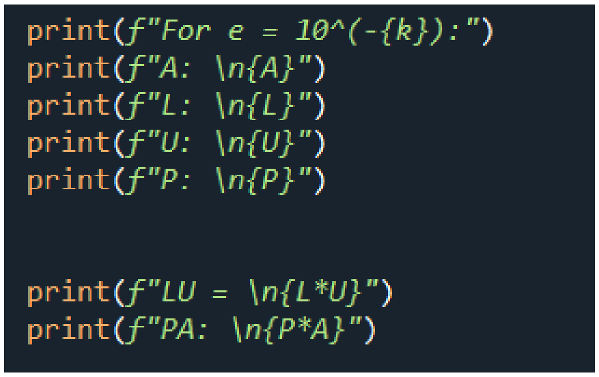
```python
def testing_e():
    k_values = [5,10,15,20,25]
```

Then, I created a loop that will create A with the given ϵ value and call `mygausselim(A,b)`, which will return L, U, P, and x.



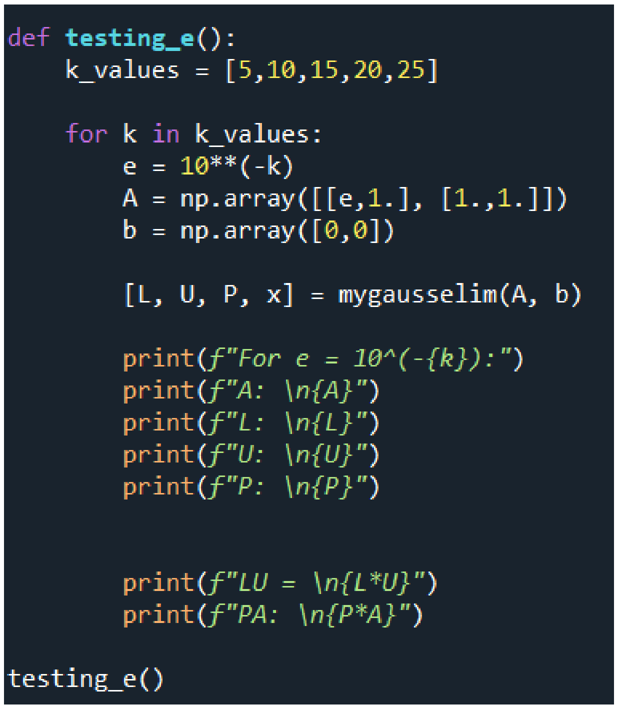
```python
for k in k_values:
    e = 10**(-k)
    A = np.array([[e,1.], [1.,1.]])
    b = np.array([0,0])

    [L, U, P, x] = mygausselim(A, b)
```

From there, I will print out the A, L, U, P, L*U and P*A values for the corresponding k value.

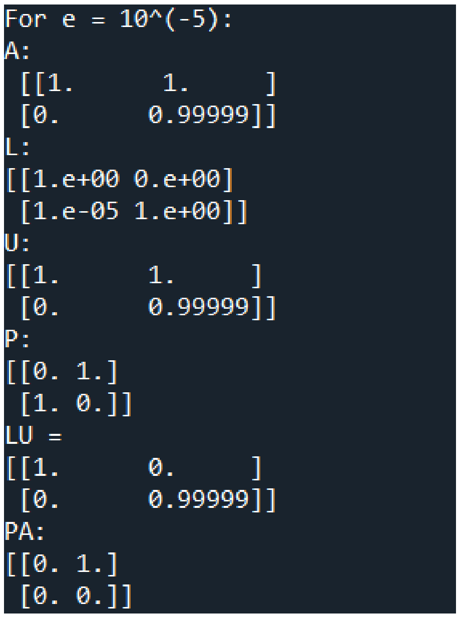


```python
print(f"For e = 10^(-{k}):")
print(f"A: \n{A}")
print(f"L: \n{L}")
print(f"U: \n{U}")
print(f"P: \n{P}")


print(f"LU = \n{L*U}")
print(f"PA: \n{P*A}")
```

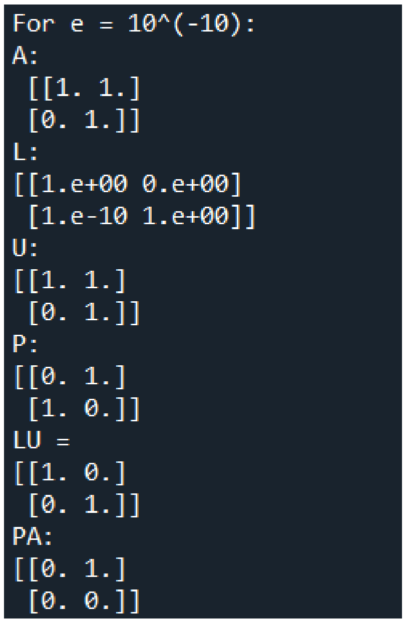
Here were the results:



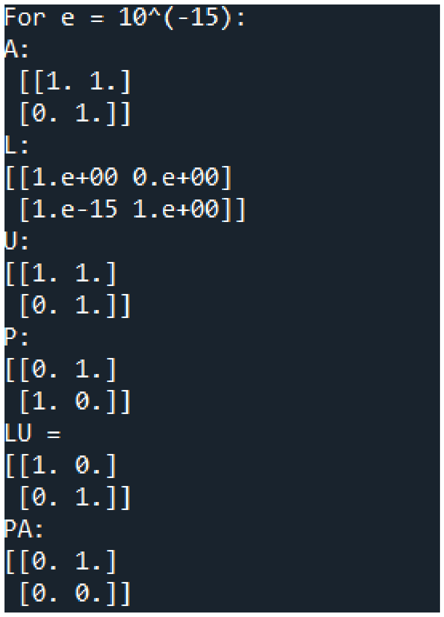
```
def testing_e():
    k_values = [5,10,15,20,25]

    for k in k_values:
        e = 10**(-k)
        A = np.array([[e,1.], [1.,1.]])
        b = np.array([0,0])

        [L, U, P, x] = mygausselim(A, b)

        print(f"For e = 10^(-{k}):")
        print(f"A: \n{A}")
        print(f"L: \n{L}")
        print(f"U: \n{U}")
        print(f"P: \n{P}")


        print(f"LU = \n{L*U}")
        print(f"PA: \n{P*A}")

testing_e()
```



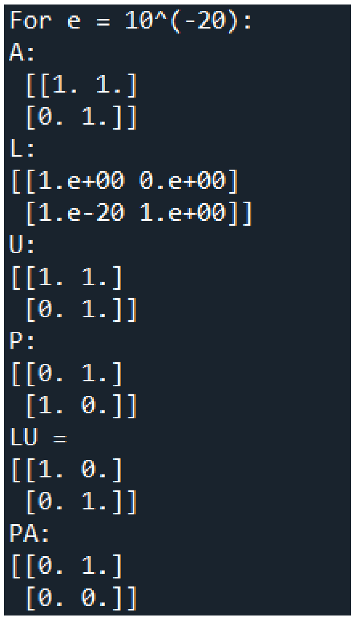
```
For e = 10^(-5):
A:
 [[1.       1.      ]
 [0.       0.99999]]
L:
[[1.e+00 0.e+00]
 [1.e-05 1.e+00]]
U:
[[1.       1.      ]
 [0.       0.99999]]
P:
[[0. 1.]
 [1. 0.]]
LU =
[[1.       0.      ]
 [0.       0.99999]]
PA:
[[0. 1.]
 [0. 0.]]
```



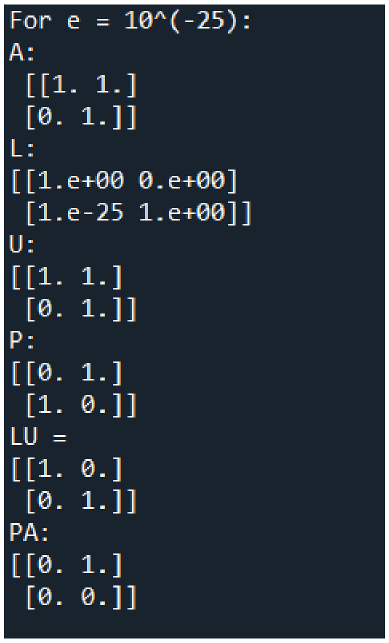
```
For e = 10^(-10):
A:
 [[1. 1.]
 [0. 1.]]
L:
[[1.e+00 0.e+00]
 [1.e-10 1.e+00]]
U:
[[1. 1.]
 [0. 1.]]
P:
[[0. 1.]
 [1. 0.]]
LU =
[[1. 0.]
 [0. 1.]]
PA:
[[0. 1.]
 [0. 0.]]
```



```
For e = 10^(-15):
A:
 [[1. 1.]
 [0. 1.]]
L:
[[1.e+00 0.e+00]
 [1.e-15 1.e+00]]
U:
[[1. 1.]
 [0. 1.]]
P:
[[0. 1.]
 [1. 0.]]
LU =
[[1. 0.]
 [0. 1.]]
PA:
[[0. 1.]
 [0. 0.]]
```



```
For e = 10^(-20):
A:
 [[1. 1.]
 [0. 1.]]
L:
[[1.e+00 0.e+00]
 [1.e-20 1.e+00]]
U:
[[1. 1.]
 [0. 1.]]
P:
[[0. 1.]
 [1. 0.]]
LU =
[[1. 0.]
 [0. 1.]]
PA:
[[0. 1.]
 [0. 0.]]
```



```
For e = 10^(-25):
A:
 [[1. 1.]
 [0. 1.]]
L:
[[1.e+00 0.e+00]
 [1.e-25 1.e+00]]
U:
[[1. 1.]
 [0. 1.]]
P:
[[0. 1.]
 [1. 0.]]
LU =
[[1. 0.]
 [0. 1.]]
PA:
[[0. 1.]
 [0. 0.]]
```

LU does *not* equate to PA for any value of k. In fact, the LU values for each k, except k = 5, are equivalent and the PA values for each k, except k = 5, are equivalent. This is due to the fact the matrix becomes incredibly ill-conditioned as the values in the first row of each matrix get closer to zero relative to the values in the second row; since k is increasing, the value of ϵ gets increasingly smaller. Without using partial pivoting, there will be more errors because the algorithm will try to solve the matrix equations by using smaller pivot elements, which will also result in LU not being equal to PA.

2.

a. Given the following equation:

$$\left\|\frac{\Delta x}{x}\right\| \leq cond(A) * \epsilon_{\text{mach}}$$

We can determine the number of decimal places to which x_1 and x_2 will be accurate.

$$\left\|\frac{\Delta x}{x}\right\| \leq 10^{10} * 10^{-16}$$

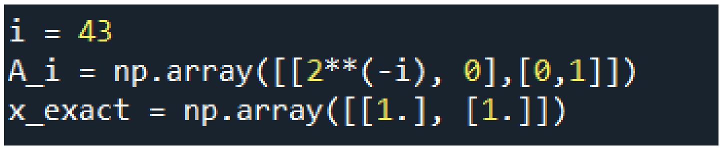
$$\left\|\frac{\Delta x}{x}\right\| \leq 10^{-6}$$

The above computation tells us that x_1 and x_2 will be accurate up to six decimal places. So, x with the underlined digits of x_1 and x_2 that will correspond to x_1 and x_2 in the solution that contains up to six decimal places is

$$x = \begin{pmatrix} 1.\underline{234567}890123456 \\ .\underline{000012}3456789012 \end{pmatrix}.$$

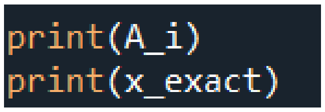
b. Given $A_i = \begin{pmatrix} 2^{-i} & 0 \\ 0 & 1 \end{pmatrix}$, with $i \geq 1$ being an integer, $x = \begin{pmatrix} 1 \\ 1 \end{pmatrix}$, and $b = A_{43}x$.

In order to compute the relative error after solving $A_{43}x = b$, we must compute the exact or given solution and the computed solution. To do so, I first made variables `A_i` and `x_exact` that will hold the given matrices from above.

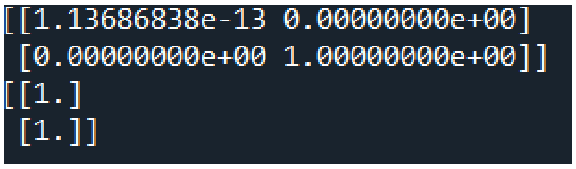


```
i = 43
A_i = np.array([[2**(-i), 0],[0,1]])
x_exact = np.array([[1.], [1.]])
```

I went ahead and printed out these two matrices just to be sure that they are accurate before I begin any computations.

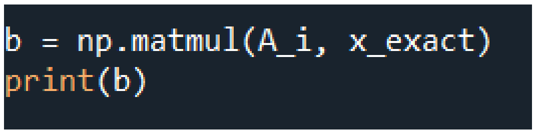


```
print(A_i)
print(x_exact)
```

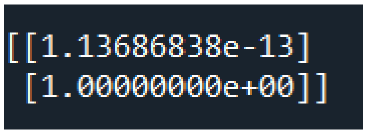


```
[[1.13686838e-13 0.00000000e+00]
 [0.00000000e+00 1.00000000e+00]]
[[1.]
 [1.]]
```

Then, I evaluated b using `A_i` and `x_exact`.

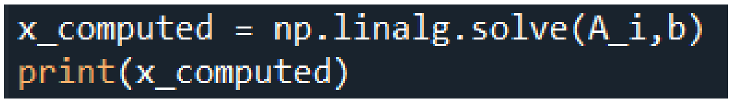


```
b = np.matmul(A_i, x_exact)
print(b)
```

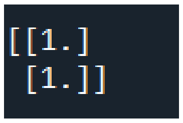


```
[[1.13686838e-13]
 [1.00000000e+00]]
```

With this b value, I was then able to retrieve the computed x value. I created a variable `x_computed` that will hold the results of $A_{43}x = b$.

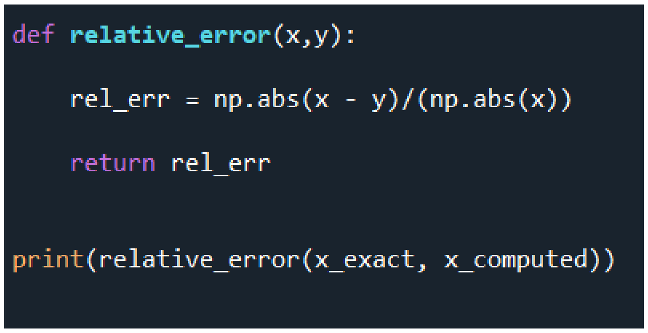


```
x_computed = np.linalg.solve(A_i,b)
print(x_computed)
```

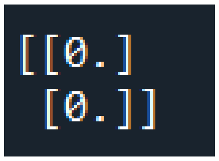


```
[[1.]
 [1.]]
```

To compute the relative error, I created a function `relative_error(x, y)` that will take in two values, the exact solution and the computed solution. The function will compute and return their relative error. The computations conclude that the relative error is 0.

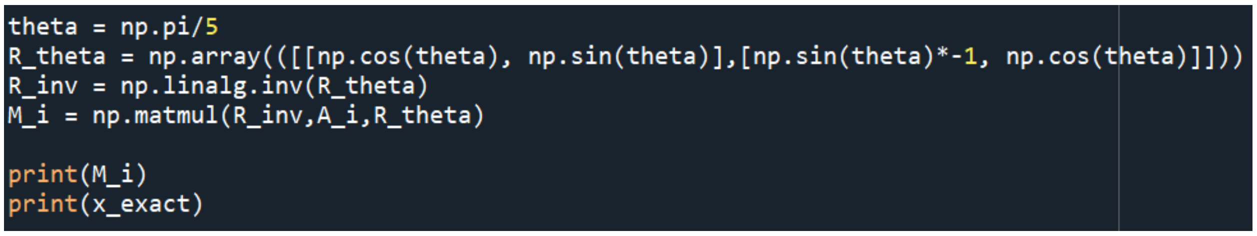


```
def relative_error(x,y):

    rel_err = np.abs(x - y)/(np.abs(x))

    return rel_err


print(relative_error(x_exact, x_computed))
```

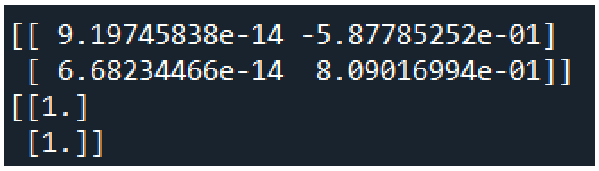


```
[[0.]
 [0.]]
```

c. Given $R(\theta) = \begin{pmatrix} \cos\theta & \sin\theta \\ -\sin\theta & \cos\theta \end{pmatrix}$ for $\theta = \frac{\pi}{5}$. To transform $R(\theta)$ to obtain $M_i = R(\theta)^{-1}A_iR(\theta)$, I first created variables theta to hold the value of $\frac{\pi}{5}$, R_theta to hold matrix $R(\theta)$, R_inv to hold the inverse of matrix R, and M_i to hold the new matrix M_i after applying similarity transformations to R.

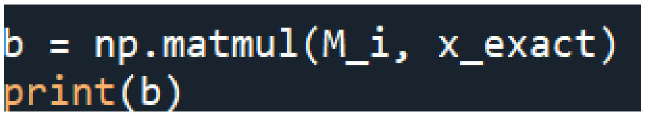


```
theta = np.pi/5
R_theta = np.array(([[np.cos(theta), np.sin(theta)],[np.sin(theta)*-1, np.cos(theta)]]))
R_inv = np.linalg.inv(R_theta)
M_i = np.matmul(R_inv,A_i,R_theta)

print(M_i)
print(x_exact)
```

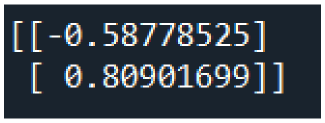


```
[[ 9.19745838e-14 -5.87785252e-01]
 [ 6.68234466e-14  8.09016994e-01]]
[[1.]
 [1.]]
```

Then, to compute the relative error for $M_{43}x = b$, I followed the same steps from above. I computed b using the given/exact x.

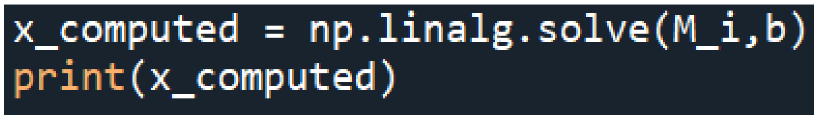


```
b = np.matmul(M_i, x_exact)
print(b)
```

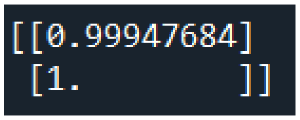


```
[[-0.58778525]
 [ 0.80901699]]
```

Then with this b value, I found the computed x.

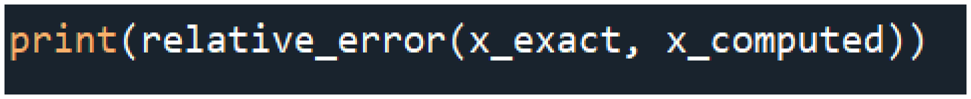


```
x_computed = np.linalg.solve(M_i,b)
print(x_computed)
```



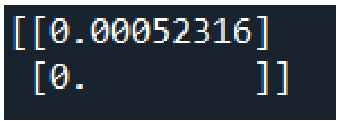
```
[[0.99947684]
 [1.        ]]
```

Lastly, I plugged the two x values in the `relative_error(x,y)` function I created in part (b).



```
print(relative_error(x_exact, x_computed))
```

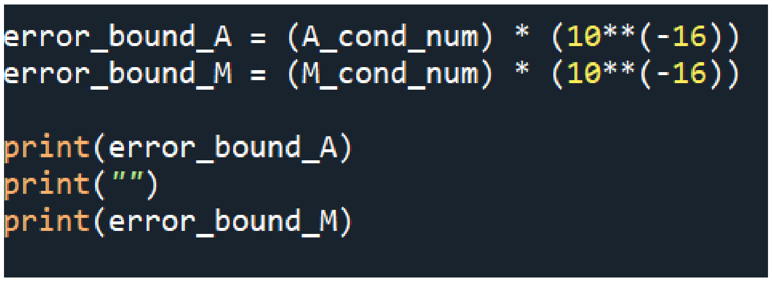
The relative error for $M_{43}x = b$ is the following:



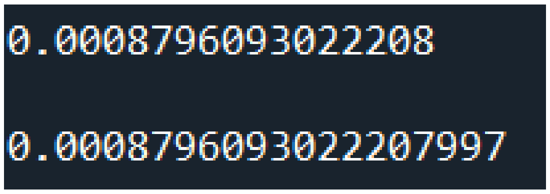
```
[[0.00052316]
 [0.        ]]
```

d. To compute the error bounds from $\mathrm{cond}(A_{43})$ and $\mathrm{cond}(M_{43})$, we need to use the machine precision, $\epsilon_{mach} = 10^{-16}$ and the values of $\mathrm{cond}(A_{43})$ and $\mathrm{cond}(M_{43})$.

To do this, I created variables `error_bound_A` and `error_bound_M` to hold the result of the error bounds.



```
error_bound_A = (A_cond_num) * (10**(-16))
error_bound_M = (M_cond_num) * (10**(-16))

print(error_bound_A)
print("")
print(error_bound_M)
```



```
0.00087960930222208

0.000879609302207997
```

The relative error results are consistent with the error bounds, as the above results state that the computed solution will deviate up to 4 decimal places from the exact solution. Since the relative error for part (b) is zero and the relative error for part (c) is approximately 5 x 10^{-4}, we can conclude that the error bound results are accurate.