Numerical Linear Algebra

Assignment 2

Abdukhomid Nurmatov

KAUST

12-10-2023

Contents

Problem

Write a code that implements the naive Gaussian elimination and the variants with column pivoting and total pivoting. The goal is to have a code that solves the small linear system of algebraic equations.

- Construct a clean and readable code.
- The code does not need to be complex and extremely general; focus on solving the aforementioned linear systems.
- Add any comments you think are relevant to explain the results you get (try to be critical).

Linear systems:

1. Fixing a value of n, define the polynomial

$$p(t) = 1 + t + t^2 + \ldots + t^{n-1} = \sum_{j=1}^{n} t^{j-1}$$

The coefficients in this polynomial are equal to 1. We want to recover these coefficients from n values of the polynomial. We use the values of $p(t)$ at the integers $t = i + 1$ for $i = 1, 2, ..., n$. If we denote the coefficients of the polynomial as $x_1, x_2, \ldots, x_n$, we have:

$$\sum_{j=1}^{n} (1+i)^{j-1} x_j = \frac{1}{i}\left[(1+i)^n - 1\right], \quad (1 \le i \le n)$$

Letting $a_{ij} = (1+i)^{j-1}$ and $b_i = \frac{1}{i}\left[(1+i)^n - 1\right]$, we have the linear system

$$\sum_{j=1}^{n} a_{ij} x_j = b_i, \quad (1 \le i \le n).$$

Solve the system for $n = 4, 5, 6, 7, 8, 9$ and compute the error in the coefficients.

2. Consider the system

$$\begin{cases} \epsilon x_1 + x_2 = 1 \\ x_1 + x_2 = 2. \end{cases}$$

Solve the system for $\epsilon = 10^2, 10^{-1}$, and 10^{-9} using double precision but only looking at the solution with 8-digit.

Could you please try to present a small example where complete pivoting is needed?

1

Solution:

I'll be using Python to solve this problem. First of all let's create a function that will generate matrix A whose elements are $a_{ij} = (1+i)^{j-1}$ and column b whose elements are $b_i = \frac{1}{i}\left((1+i)^n - 1\right)$, where $1 \leq i \leq n$, and $1 \leq j \leq n$:

```
import numpy as np

def generate_A_b(n):
    # Initializing A and b and filling them by 0
    A = np.zeros((n, n), dtype = 'float64')
    b = np.zeros(n, dtype = 'float64')

    # Generating A and b
    for i in range(1, n + 1):
        b[i - 1] =  ((1 + i)**n - 1) / i
        for j in range(1, n + 1):
            A[i - 1, j - 1] = (1 + i)**(j - 1)

    return A, b
```

Now, let's see what kind of output we'll have for matrix A and column b for $n = 4, 5, 6, 7, 8, 9$ by adding the following code:

```
list_n = [4, 5, 6, 7, 8, 9]
for n in list_n:
    A, b = generate_A_b(n)
    print(f"n = {n} \n\n A = {A} \n\n b = {b} \n-----------")
```

The output will be:

```
n = 4

 A = [[  1.   2.   4.   8.]
 [  1.   3.   9.  27.]
 [  1.   4.  16.  64.]
 [  1.   5.  25. 125.]]

 b = [ 15.  40.  85. 156.]
-----------
n = 5

 A = [[1.000e+00 2.000e+00 4.000e+00 8.000e+00 1.600e+01]
 [1.000e+00 3.000e+00 9.000e+00 2.700e+01 8.100e+01]
 [1.000e+00 4.000e+00 1.600e+01 6.400e+01 2.560e+02]
 [1.000e+00 5.000e+00 2.500e+01 1.250e+02 6.250e+02]
 [1.000e+00 6.000e+00 3.600e+01 2.160e+02 1.296e+03]]

 b = [  31.  121.  341.  781. 1555.]
-----------
n = 6

 A = [[1.0000e+00 2.0000e+00 4.0000e+00 8.0000e+00 1.6000e+01 3.2000e+01]
 [1.0000e+00 3.0000e+00 9.0000e+00 2.7000e+01 8.1000e+01 2.4300e+02]
 [1.0000e+00 4.0000e+00 1.6000e+01 6.4000e+01 2.5600e+02 1.0240e+03]
 [1.0000e+00 5.0000e+00 2.5000e+01 1.2500e+02 6.2500e+02 3.1250e+03]
 [1.0000e+00 6.0000e+00 3.6000e+01 2.1600e+02 1.2960e+03 7.7760e+03]
 [1.0000e+00 7.0000e+00 4.9000e+01 3.4300e+02 2.4010e+03 1.6807e+04]]

 b = [   63.   364.  1365.  3906.  9331. 19608.]
-----------
n = 7

 A = [[1.00000e+00 2.00000e+00 4.00000e+00 8.00000e+00 1.60000e+01 3.20000e+01
  6.40000e+01]
 [1.00000e+00 3.00000e+00 9.00000e+00 2.70000e+01 8.10000e+01 2.43000e+02
```

```
  7.29000e+02]
 [1.00000e+00 4.00000e+00 1.60000e+01 6.40000e+01 2.56000e+02 1.02400e+03
  4.09600e+03]
 [1.00000e+00 5.00000e+00 2.50000e+01 1.25000e+02 6.25000e+02 3.12500e+03
  1.56250e+04]
 [1.00000e+00 6.00000e+00 3.60000e+01 2.16000e+02 1.29600e+03 7.77600e+03
  4.66560e+04]
 [1.00000e+00 7.00000e+00 4.90000e+01 3.43000e+02 2.40100e+03 1.68070e+04
  1.17649e+05]
 [1.00000e+00 8.00000e+00 6.40000e+01 5.12000e+02 4.09600e+03 3.27680e+04
  2.62144e+05]]

 b = [1.27000e+02 1.09300e+03 5.46100e+03 1.95310e+04 5.59870e+04 1.37257e+05
 2.99593e+05]
-----------
n = 8

 A = [[1.000000e+00 2.000000e+00 4.000000e+00 8.000000e+00 1.600000e+01
  3.200000e+01 6.400000e+01 1.280000e+02]
 [1.000000e+00 3.000000e+00 9.000000e+00 2.700000e+01 8.100000e+01
  2.430000e+02 7.290000e+02 2.187000e+03]
 [1.000000e+00 4.000000e+00 1.600000e+01 6.400000e+01 2.560000e+02
  1.024000e+03 4.096000e+03 1.638400e+04]
 [1.000000e+00 5.000000e+00 2.500000e+01 1.250000e+02 6.250000e+02
  3.125000e+03 1.562500e+04 7.812500e+04]
 [1.000000e+00 6.000000e+00 3.600000e+01 2.160000e+02 1.296000e+03
  7.776000e+03 4.665600e+04 2.799360e+05]
 [1.000000e+00 7.000000e+00 4.900000e+01 3.430000e+02 2.401000e+03
  1.680700e+04 1.176490e+05 8.235430e+05]
 [1.000000e+00 8.000000e+00 6.400000e+01 5.120000e+02 4.096000e+03
  3.276800e+04 2.621440e+05 2.097152e+06]
 [1.000000e+00 9.000000e+00 8.100000e+01 7.290000e+02 6.561000e+03
  5.904900e+04 5.314410e+05 4.782969e+06]]

 b = [2.550000e+02 3.280000e+03 2.184500e+04 9.765600e+04 3.359230e+05
 9.608000e+05 2.396745e+06 5.380840e+06]
-----------
n = 9

 A = [[1.0000000e+00 2.0000000e+00 4.0000000e+00 8.0000000e+00 1.6000000e+01
  3.2000000e+01 6.4000000e+01 1.2800000e+02 2.5600000e+02]
 [1.0000000e+00 3.0000000e+00 9.0000000e+00 2.7000000e+01 8.1000000e+01
  2.4300000e+02 7.2900000e+02 2.1870000e+03 6.5610000e+03]
 [1.0000000e+00 4.0000000e+00 1.6000000e+01 6.4000000e+01 2.5600000e+02
  1.0240000e+03 4.0960000e+03 1.6384000e+04 6.5536000e+04]
 [1.0000000e+00 5.0000000e+00 2.5000000e+01 1.2500000e+02 6.2500000e+02
  3.1250000e+03 1.5625000e+04 7.8125000e+04 3.9062500e+05]
 [1.0000000e+00 6.0000000e+00 3.6000000e+01 2.1600000e+02 1.2960000e+03
  7.7760000e+03 4.6656000e+04 2.7993600e+05 1.6796160e+06]
 [1.0000000e+00 7.0000000e+00 4.9000000e+01 3.4300000e+02 2.4010000e+03
  1.6807000e+04 1.1764900e+05 8.2354300e+05 5.7648010e+06]
 [1.0000000e+00 8.0000000e+00 6.4000000e+01 5.1200000e+02 4.0960000e+03
  3.2768000e+04 2.6214400e+05 2.0971520e+06 1.6777216e+07]
 [1.0000000e+00 9.0000000e+00 8.1000000e+01 7.2900000e+02 6.5610000e+03
  5.9049000e+04 5.3144100e+05 4.7829690e+06 4.3046721e+07]
 [1.0000000e+00 1.0000000e+01 1.0000000e+02 1.0000000e+03 1.0000000e+04
  1.0000000e+05 1.0000000e+06 1.0000000e+07 1.0000000e+08]]

 b = [5.11000000e+02 9.84100000e+03 8.73810000e+04 4.88281000e+05
 2.01553900e+06 6.72560100e+06 1.91739610e+07 4.84275610e+07
 1.11111111e+08]
-----------
```

Let's implement Gaussian Elimination:
Gauss elimination consists of two parts: the elimination phase and the back substitution phase. The role of the elimination phase is to transform the equations into the form $Ux = c$, where U is upper triangular matrix. Then equations will be solved by back substitution. So, let the j-th row be a typical row below the pivot

equation that is to be transformed, meaning that the element A_{jk} is to be eliminated. We can achieve this by multiplying the pivot row by $\alpha = \frac{A_{jk}}{A_{kk}}$ and subtracting it from the j-th row. The corresponding changes in the j-th row will be:

$$A_{ik} \leftarrow A_{ik} - \alpha A_{jk}, \quad k = j, j+1, \ldots, n \tag{1.1}$$

$$b_i \leftarrow b_i - \alpha b_j \tag{1.2}$$

To transform the entire coefficient matrix to upper triangular form, j and i in (1.1) and (1.2) must have the ranges $k = 1, 2, ..., n-1$ (chooses the pivot row), $j = k+1, k+2, ..., n$ (chooses the row to be transformed). So, the code for the elimination phase will be the following:

```
for k in range(0, n - 1):
    for j in range(k + 1, n):
        alpha = A[j, k] / A[k, k]
        A[j, k:n] = A[j, k:n] - alpha * A[k, k:n]
        b[j] = b[j] - alpha * b[k]
```

After Gaussian elimination, the augmented coefficient matrix will be in the following form

$$(A \mid b) = \left(\begin{array}{ccccc|c} A_{11} & A_{12} & A_{13} & \cdots & A_{1n} & b_1 \\ 0 & A_{22} & A_{23} & \cdots & A_{2n} & b_2 \\ 0 & 0 & A_{33} & \cdots & A_{3n} & b_3 \\ \vdots & \vdots & \vdots & \ddots & \vdots & \vdots \\ 0 & 0 & 0 & \cdots & A_{nn} & b_n \end{array}\right)$$

Then first we'll solve the last equation $A_{nn}x_n = b_n \hookrightarrow x_n = \frac{b_n}{A_{nn}}$. Consider now the stage of back substitution where $x_n, x_{n-1}, \ldots, x_{k+1}$ have already been computed (in that order), and we are about to determine x_k from the k-th equation

$$A_{kk}x_k + A_{k,k+1}x_{k+1} + \ldots + A_{kn}x_n = b_k$$

The solution will be

$$x_k = \frac{1}{A_{kk}}\left(b_k - \sum_{j=k+1}^{n} A_{kj}x_j\right), \quad k = n-1, n-2, \ldots, 1$$

So, code for the back substitution phase will be the following:

```
for k in range(n - 1, -1, -1):
    b[k] = (b[k] - np.dot(A[k, k + 1:n], b[k + 1:n])) / A[k, k]
```

About the operation count: the execution time of an algorithm depends largely on the number of long operations (multiplications and divisions) performed. It can be shown that Gaussian elimination contains approximately $\frac{2n^3}{3}$ such operations (where n is the number of equations) in the elimination phase, and $\frac{n^2}{2}$ operations in back substitution. These numbers show that most of the computation time goes into the elimination phase. So, now let's present a function that will do the Gaussian Elimination:

```
def Naive_Gaussian_Elimination(A, b):
    A = A.astype('float64')
    b = b.astype('float64')
    n = len(b)
    # Elimination phase
    for k in range(0, n - 1):
        for j in range(k + 1, n):
            alpha = A[j, k] / A[k, k]
            A[j, k:n] = A[j, k:n] - alpha * A[k, k:n]
            b[j] = b[j] - alpha * b[k]

    # Back substitution phase
    for k in range(n - 1, -1, -1):
        b[k] = (b[k] - np.dot(A[k, k + 1:n], b[k + 1:n])) / A[k, k]

    return b
```

Before creating functions that will compute the error in the coefficients, let's decide in what norm we'd like to define the error. I'd like to use L_1, L_2 and L_∞ norms. So, the code for the error will be the following:

```
def coefficients_Error(true_coefficients, computed_coefficients):
    difference_Vector = abs(true_coefficients - computed_coefficients)

    # L_1 Error
    L_1_Absolute_Error = difference_Vector.sum()
    L_1_Relative_Error = L_1_Absolute_Error / abs(true_coefficients).sum()

    # L_infinity Error
    L_infinity_Absolute_Error = difference_Vector.max()
    L_infinity_Relative_Error = L_infinity_Absolute_Error / abs(true_coefficients).max()

    # L_2 Error
    sum_Of_Squared_Elements = 0
    for element in difference_Vector:
        sum_Of_Squared_Elements = sum_Of_Squared_Elements + element**2
    L_2_Absolute_Error = np.sqrt(sum_Of_Squared_Elements)
    sum_Of_Squared_Elements = 0
    for element in true_coefficients:
        sum_Of_Squared_Elements = sum_Of_Squared_Elements + element**2
    L_2_Relative_Error = L_2_Absolute_Error / np.sqrt(sum_Of_Squared_Elements)

    return L_1_Absolute_Error, L_1_Relative_Error, L_2_Absolute_Error, L_2_Relative_Error,
    L_infinity_Absolute_Error, L_infinity_Relative_Error
```

Now, we can present the full code of the Question 1 for the naive Gaussian Elimination:

```
import numpy as np

def Naive_Gaussian_Elimination(A, b):
    A = A.astype('float64')
    b = b.astype('float64')
    n = len(b)
    # Elimination phase
    for k in range(0, n - 1):
        for j in range(k + 1, n):
            alpha = A[j, k] / A[k, k]
            A[j, k:n] = A[j, k:n] - alpha * A[k, k:n]
            b[j] = b[j] - alpha * b[k]

    # Back substitution phase
    for k in range(n - 1, -1, -1):
        b[k] = (b[k] - np.dot(A[k, k + 1:n], b[k + 1:n])) / A[k, k]

    return b


def generate_A_b(n):
    # Initializing A and b and filling them by 0
    A = np.zeros((n, n), dtype = 'float64')
    b = np.zeros(n, dtype = 'float64')

    # Generating A and b
    for i in range(1, n + 1):
        b[i - 1] =  ((1 + i)**n - 1) / i
        for j in range(1, n + 1):
            A[i - 1, j - 1] = (1 + i)**(j - 1)

    return A, b


def coefficients_Error(true_coefficients, computed_coefficients):
    difference_Vector = abs(true_coefficients - computed_coefficients)

    # L_1 Error
    L_1_Absolute_Error = difference_Vector.sum()
    L_1_Relative_Error = L_1_Absolute_Error / abs(true_coefficients).sum()
```

```
41
42      # L_infinity Error
43      L_infinity_Absolute_Error = difference_Vector.max()
44      L_infinity_Relative_Error = L_infinity_Absolute_Error / abs(true_coefficients).max()
45
46      # L_2 Error
47      sum_Of_Squared_Elements = 0
48      for element in difference_Vector:
49          sum_Of_Squared_Elements = sum_Of_Squared_Elements + element**2
50      L_2_Absolute_Error = np.sqrt(sum_Of_Squared_Elements)
51      sum_Of_Squared_Elements = 0
52      for element in true_coefficients:
53          sum_Of_Squared_Elements = sum_Of_Squared_Elements + element**2
54      L_2_Relative_Error = L_2_Absolute_Error / np.sqrt(sum_Of_Squared_Elements)
55
56      return L_1_Absolute_Error, L_1_Relative_Error, L_2_Absolute_Error, L_2_Relative_Error,
        L_infinity_Absolute_Error, L_infinity_Relative_Error
57
58
59  def main():
60      list_n = [4, 5, 6, 7, 8, 9]
61      for n in list_n:
62          A, b = generate_A_b(n)
63
64          computed_coefficients = Naive_Gaussian_Elimination(A.copy(), b.copy())
65
66          true_coefficients = np.ones(n, dtype = 'float64')
67
68          error_L_1_a, error_L_1_r, error_L_2_a, error_L_2_r, error_L_infinity_a,
        error_L_infinity_r = coefficients_Error(true_coefficients, computed_coefficients)
69
70          print(f"n = {n}\n")
71          print(f"Computed Coefficients: {computed_coefficients}\n")
72          print(f"True Coefficients: {true_coefficients}\n")
73          print(f"L_1 Error: Absolute = {error_L_1_a}   Relative = {error_L_1_r}\n")
74          print(f"L_2 Error: Absolute = {error_L_2_a}   Relative = {error_L_2_r}\n")
75          print(f"L_infinity Error: Absolute = {error_L_infinity_a}   Relative = {
        error_L_infinity_r}\n-----------")
76
77  if __name__ == "__main__":
78      main()
```

The output will be:

```
n = 4

Computed Coefficients: [1. 1. 1. 1.]

True Coefficients: [1. 1. 1. 1.]

L_1 Error: Absolute = 0.0   Relative = 0.0

L_2 Error: Absolute = 0.0   Relative = 0.0

L_infinity Error: Absolute = 0.0   Relative = 0.0
-----------
n = 5

Computed Coefficients: [1. 1. 1. 1. 1.]

True Coefficients: [1. 1. 1. 1. 1.]

L_1 Error: Absolute = 0.0   Relative = 0.0

L_2 Error: Absolute = 0.0   Relative = 0.0

L_infinity Error: Absolute = 0.0   Relative = 0.0
-----------
n = 6
```

```
26
27 Computed Coefficients: [1. 1. 1. 1. 1. 1.]
28
29 True Coefficients: [1. 1. 1. 1. 1. 1.]
30
31 L_1 Error: Absolute = 0.0   Relative = 0.0
32
33 L_2 Error: Absolute = 0.0   Relative = 0.0
34
35 L_infinity Error: Absolute = 0.0   Relative = 0.0
36 -----------
37 n = 7
38
39 Computed Coefficients: [1. 1. 1. 1. 1. 1. 1.]
40
41 True Coefficients: [1. 1. 1. 1. 1. 1. 1.]
42
43 L_1 Error: Absolute = 0.0   Relative = 0.0
44
45 L_2 Error: Absolute = 0.0   Relative = 0.0
46
47 L_infinity Error: Absolute = 0.0   Relative = 0.0
48 -----------
49 n = 8
50
51 Computed Coefficients: [1. 1. 1. 1. 1. 1. 1. 1.]
52
53 True Coefficients: [1. 1. 1. 1. 1. 1. 1. 1.]
54
55 L_1 Error: Absolute = 0.0   Relative = 0.0
56
57 L_2 Error: Absolute = 0.0   Relative = 0.0
58
59 L_infinity Error: Absolute = 0.0   Relative = 0.0
60 -----------
61 n = 9
62
63 Computed Coefficients: [1. 1. 1. 1. 1. 1. 1. 1. 1.]
64
65 True Coefficients: [1. 1. 1. 1. 1. 1. 1. 1. 1.]
66
67 L_1 Error: Absolute = 0.0   Relative = 0.0
68
69 L_2 Error: Absolute = 0.0   Relative = 0.0
70
71 L_infinity Error: Absolute = 0.0   Relative = 0.0
72 -----------
```

Let's implement LU decomposition:
As been explained above most of the work in Gaussian elimination is applying row operations to arrive at the upper-triangular matrix. If we need to solve several different systems with the same A, then we would like to avoid repeating the steps of Gaussian elimination on A for every different b. This can be accomplished by the LU decomposition (U represents an upper triangular matrix, L is a lower triangular matrix), which in effect records the steps of Gaussian elimination. So, we first rewrite the equations as $LUx = b$. After using the notation $Ux = y$, the equations become

$$Ly = b. \tag{1.3}$$

Equation (1.3) can be solved for y by forward substitution. Then we'll have to solve

$$Ux = y. \tag{1.4}$$

From equation (1.4) we'll have x by the back substitution process. Let's notice that the matrix U is identical to the upper triangular matrix that results from Gauss elimination, and the off-diagonal elements of L are the pivot equation multipliers (elements along diagonal are ones) used during Gaussian Elimination, i.e. L_{jk} is the multiplier that eliminated A_{jk}. So, function for the LU decomposition will be the following:

```
def LU_Decomposition(A):
    A = A.astype('float64')
    n = len(A)
    U = A.copy()
    L = np.eye(n, dtype = 'float64')
    for k in range(0, n - 1):
        for j in range(k + 1, n):
                L[j, k] = U[j, k] / U[k, k]
                U[j, k:n] = U[j, k:n] - L[j, k] * U[k, k:n]

    return L, U
```

In the solution phase first we have to solve $Ly = b$. By solving the k-th equation for y_k we'll have

$$y_k = b_k - \sum_{j=1}^{k-1} L_{kj} y_j, \quad k = 2, 3, \ldots, n, \text{ where } y_1 = b_1$$

So, code for $Ly = b$ will be the following:

```
for k in range(1, n):
    b[k] = b[k] - np.dot(L[k, 0:k], b[0:k])
```

The back substitution phase for solving $Ux = y$ is identical to what was used in the Gaussian elimination method. So, the full code of the Question 1 for the LU decomposition will be:

```
import numpy as np

def LU_Decomposition(A):
    A = A.astype('float64')
    n = len(A)
    U = A.copy()
    L = np.eye(n, dtype = 'float64')
    for k in range(0, n - 1):
        for j in range(k + 1, n):
                L[j, k] = U[j, k] / U[k, k]
                U[j, k:n] = U[j, k:n] - L[j, k] * U[k, k:n]

    return L, U


def LU_Solver(L, U, b):
    L = L.astype('float64')
    U = U.astype('float64')
    b = b.astype('float64')
    n = len(U)
    # Forward substitution Ly = b
    for k in range(1, n):
        b[k] = b[k] - np.dot(L[k, 0:k], b[0:k])

    # Backward substitution Ux = y
    for k in range(n - 1, -1, -1):
       b[k] = (b[k] - np.dot(U[k, k + 1:n], b[k + 1:n])) / U[k, k]

    return b


def generate_A_b(n):
    # Initializing A and b and filling them by 0
    A = np.zeros((n, n), dtype = 'float64')
    b = np.zeros(n, dtype = 'float64')

    # Generating A and b
    for i in range(1, n + 1):
        b[i - 1] =  ((1 + i)**n - 1) / i
        for j in range(1, n + 1):
            A[i - 1, j - 1] = (1 + i)**(j - 1)

```

```
    return A, b


def coefficients_Error(true_coefficients, computed_coefficients):
    difference_Vector = abs(true_coefficients - computed_coefficients)

    # L_1 Error
    L_1_Absolute_Error = difference_Vector.sum()
    L_1_Relative_Error = L_1_Absolute_Error / abs(true_coefficients).sum()

    # L_infinity Error
    L_infinity_Absolute_Error = difference_Vector.max()
    L_infinity_Relative_Error = L_infinity_Absolute_Error / abs(true_coefficients).max()

    # L_2 Error
    sum_Of_Squared_Elements = 0
    for element in difference_Vector:
        sum_Of_Squared_Elements = sum_Of_Squared_Elements + element**2
    L_2_Absolute_Error = np.sqrt(sum_Of_Squared_Elements)
    sum_Of_Squared_Elements = 0
    for element in true_coefficients:
        sum_Of_Squared_Elements = sum_Of_Squared_Elements + element**2
    L_2_Relative_Error = L_2_Absolute_Error / np.sqrt(sum_Of_Squared_Elements)

    return L_1_Absolute_Error, L_1_Relative_Error, L_2_Absolute_Error, L_2_Relative_Error,
    L_infinity_Absolute_Error, L_infinity_Relative_Error


def main():
    list_n = [4, 5, 6, 7, 8, 9]
    for n in list_n:
        A, b = generate_A_b(n)

        L, U = LU_Decomposition(A)
        computed_coefficients = LU_Solver(L, U, b.copy())

        true_coefficients = np.ones(n, dtype = 'float64')

        error_L_1_a, error_L_1_r, error_L_2_a, error_L_2_r, error_L_infinity_a,
    error_L_infinity_r = coefficients_Error(true_coefficients, computed_coefficients)

        print(f"n = {n}\n")
        print(f"Computed Coefficients: {computed_coefficients}\n")
        print(f"True Coefficients: {true_coefficients}\n")
        print(f"L_1 Error: Absolute = {error_L_1_a}   Relative = {error_L_1_r}\n")
        print(f"L_2 Error: Absolute = {error_L_2_a}   Relative = {error_L_2_r}\n")
        print(f"L_infinity Error: Absolute = {error_L_infinity_a}   Relative = {
    error_L_infinity_r}\n-----------")

if __name__ == "__main__":
    main()
```

The output is the same as the output for Gaussian Elimination. Also floating point operations is the same as in Gaussian Elimination $\left(\approx \frac{2n^3}{3}\right)$.

Let's implement Gaussian Elimination with partial pivoting:
Let's emphasize that Gauss elimination fails immediately when there is the presence of the zero pivot element (or when the pivot element is very small in comparison to other elements in the pivot row, because the computer works with a fixed length, all numbers are rounded off to a finite number of significant figures), and that's why it is essential to reorder the equations during the elimination phase. So, in order to reduce instability through the process of pivoting we have to ensure that we'll use relatively large entries as our pivot elements. So, the full code of the Question 1 for the Gaussian Elimination with partial pivoting will be:

```
import numpy as np

def Gaussian_Elimination_With_Partial_Pivoting(A, b):
```

```
    A = A.astype('float64')
    b = b.astype('float64')
    n = len(b)
    for k in range(n - 1):
        p = np.argmax(np.abs(A[k: , k])) + k

        # Index interchanging if needed
        if p != k:
            A[[k, p]] = A[[p, k]]
            b[k], b[p] = b[p], b[k]

        # Elimination
        for j in range(k + 1, n):
            if A [j , k] != 0.0:
                alpha = A[j, k] / A[k, k]
                A[j, k:] = A[j, k:]- alpha * A[k, k:]
                b[j] = b[j] - alpha * b[k]

    # Back substitution phase
    for k in range(n - 1, -1, -1):
        b[k] = (b[k] - np.dot(A[k, k + 1:n], b[k + 1:n])) / A[k, k]

    return b


def generate_A_b(n):
    # Initializing A and b and filling them by 0
    A = np.zeros((n, n), dtype = 'float64')
    b = np.zeros(n, dtype = 'float64')

    # Generating A and b
    for i in range(1, n + 1):
        b[i - 1] =  ((1 + i)**n - 1) / i
        for j in range(1, n + 1):
            A[i - 1, j - 1] = (1 + i)**(j - 1)

    return A, b


def coefficients_Error(true_coefficients, computed_coefficients):
    difference_Vector = abs(true_coefficients - computed_coefficients)

    # L_1 Error
    L_1_Absolute_Error = difference_Vector.sum()
    L_1_Relative_Error = L_1_Absolute_Error / abs(true_coefficients).sum()

    # L_infinity Error
    L_infinity_Absolute_Error = difference_Vector.max()
    L_infinity_Relative_Error = L_infinity_Absolute_Error / abs(true_coefficients).max()

    # L_2 Error
    sum_Of_Squared_Elements = 0
    for element in difference_Vector:
        sum_Of_Squared_Elements = sum_Of_Squared_Elements + element**2
    L_2_Absolute_Error = np.sqrt(sum_Of_Squared_Elements)
    sum_Of_Squared_Elements = 0
    for element in true_coefficients:
        sum_Of_Squared_Elements = sum_Of_Squared_Elements + element**2
    L_2_Relative_Error = L_2_Absolute_Error / np.sqrt(sum_Of_Squared_Elements)

    return L_1_Absolute_Error, L_1_Relative_Error, L_2_Absolute_Error, L_2_Relative_Error,
    L_infinity_Absolute_Error, L_infinity_Relative_Error


def main():
    list_n = [4, 5, 6, 7, 8, 9]
    for n in list_n:
        A, b = generate_A_b(n)
```

```
71
72          computed_coefficients = Gaussian_Elimination_With_Partial_Pivoting(A.copy(), b.copy
        ())
73
74          true_coefficients = np.ones(n, dtype = 'float64')
75
76          error_L_1_a, error_L_1_r, error_L_2_a, error_L_2_r, error_L_infinity_a,
        error_L_infinity_r = coefficients_Error(true_coefficients, computed_coefficients)
77
78          print(f"n = {n}\n")
79          print(f"Computed Coefficients: {computed_coefficients}\n")
80          print(f"True Coefficients: {true_coefficients}\n")
81          print(f"L_1 Error: Absolute = {error_L_1_a}   Relative = {error_L_1_r}\n")
82          print(f"L_2 Error: Absolute = {error_L_2_a}   Relative = {error_L_2_r}\n")
83          print(f"L_infinity Error: Absolute = {error_L_infinity_a}   Relative = {
        error_L_infinity_r}\n-----------")
84
85  if __name__ == "__main__":
86      main()
```

The output is the same as for previous cases.

Let's implement LU decomposition with partial pivoting:
Now, we gonna implement LU decomposition with partial pivoting, i.e. $PA = LU$, where P is a permutation matrix according to what been explained in Gaussian Elimination with partial pivoting. So, we first rewrite the equations as $LUx = Pb$. After using the notation $Ux = y$, the equations become

$$Ly = Pb. \tag{1.5}$$

Equation (1.5) can be solved for y by forward substitution. Then we'll have to solve

$$Ux = y. \tag{1.6}$$

From equation (1.6) we'll have x by the back substitution process. So, the full code of the Question 1 for LU decomposition with partial pivoting will be:

```
1   import numpy as np
2
3   def LU_Decomposition_With_Partial_Pivoting(A):
4       A = A.astype('float64')
5       n = len(A)
6       P = np.eye(n, dtype = 'float64')
7       L = np.eye(n, dtype = 'float64')
8       U = A.copy()
9       for k in range(0, n-1):
10          p = np.argmax(abs(U[k:n, k])) + k
11
12          if p != k:
13              P_step = np.eye(n, dtype = 'float64')
14              U[[p, k], k:n] = U[[k, p], k:n]
15              L[[p, k], 0:k] = L[[k, p], 0:k]
16              P_step[[p, k]] = P_step[[k, p]]
17              P = np.dot(P_step, P)
18
19          for j in range(k+1, n):
20              L[j, k] =  U[j, k] / U[k, k]
21              U[j, k:n] = U[j, k:n] - L[j, k] * U[k, k:n]
22
23      return L, U, P
24
25
26  def LU_PP_Solver(L, U, P, b):
27      L = L.astype('float64')
28      U = U.astype('float64')
29      P = P.astype('float64')
30      b = b.astype('float64')
```

```
    n = len(U)
    # Forward substitution Ly = Pb
    b = np.dot(P, b)
    for k in range(1, n):
        b[k] = b[k] - np.dot(L[k, 0:k], b[0:k])

    # Backward substitution Ux = y
    for k in range(n - 1, -1, -1):
       b[k] = (b[k] - np.dot(U[k, k + 1:n], b[k + 1:n])) / U[k, k]

    return b

def generate_A_b(n):
    # Initializing A and b and filling them by 0
    A = np.zeros((n, n), dtype = 'float64')
    b = np.zeros(n, dtype = 'float64')

    # Generating A and b
    for i in range(1, n + 1):
        b[i - 1] =  ((1 + i)**n - 1) / i
        for j in range(1, n + 1):
            A[i - 1, j - 1] = (1 + i)**(j - 1)

    return A, b


def coefficients_Error(true_coefficients, computed_coefficients):
    difference_Vector = abs(true_coefficients - computed_coefficients)

    # L_1 Error
    L_1_Absolute_Error = difference_Vector.sum()
    L_1_Relative_Error = L_1_Absolute_Error / abs(true_coefficients).sum()

    # L_infinity Error
    L_infinity_Absolute_Error = difference_Vector.max()
    L_infinity_Relative_Error = L_infinity_Absolute_Error / abs(true_coefficients).max()

    # L_2 Error
    sum_Of_Squared_Elements = 0
    for element in difference_Vector:
        sum_Of_Squared_Elements = sum_Of_Squared_Elements + element**2
    L_2_Absolute_Error = np.sqrt(sum_Of_Squared_Elements)
    sum_Of_Squared_Elements = 0
    for element in true_coefficients:
        sum_Of_Squared_Elements = sum_Of_Squared_Elements + element**2
    L_2_Relative_Error = L_2_Absolute_Error / np.sqrt(sum_Of_Squared_Elements)

    return L_1_Absolute_Error, L_1_Relative_Error, L_2_Absolute_Error, L_2_Relative_Error,
    L_infinity_Absolute_Error, L_infinity_Relative_Error


def main():
    list_n = [4, 5, 6, 7, 8, 9]
    for n in list_n:
        A, b = generate_A_b(n)

        L, U, P = LU_Decomposition_With_Partial_Pivoting(A)
        computed_coefficients = LU_PP_Solver(L, U, P, b.copy())

        true_coefficients = np.ones(n, dtype = 'float64')

        error_L_1_a, error_L_1_r, error_L_2_a, error_L_2_r, error_L_infinity_a,
    error_L_infinity_r = coefficients_Error(true_coefficients, computed_coefficients)

        print(f"n = {n}\n")
        print(f"Computed Coefficients: {computed_coefficients}\n")
        print(f"True Coefficients: {true_coefficients}\n")
        print(f"L_1 Error: Absolute = {error_L_1_a}   Relative = {error_L_1_r}\n")
```

```
        print(f"L_2 Error: Absolute = {error_L_2_a}   Relative = {error_L_2_r}\n")
        print(f"L_infinity Error: Absolute = {error_L_infinity_a}   Relative = {
    error_L_infinity_r}\n-----------")

if __name__ == "__main__":
    main()
```

Floating point operations is the same as in Gaussian Elimination $\left(\approx \frac{2n^3}{3}\right)$, and the output is the same as before except for the case $n = 9$, which is

```
n = 9

Computed Coefficients: [0.99999997 1.00000005 0.99999996 1.00000001 1.         1.
 1.         1.         1.         ]

True Coefficients: [1. 1. 1. 1. 1. 1. 1. 1. 1.]

L_1 Error: Absolute = 1.3534761456046596e-07   Relative = 1.5038623840051773e-08

L_2 Error: Absolute = 7.295155109957717e-08   Relative = 2.4317183699859058e-08

L_infinity Error: Absolute = 5.4016709327697754e-08   Relative = 5.4016709327697754e-08
-----------
```

Let's implement Gaussian Elimination with total pivoting:
Procedure of complete pivoting takes a significant amount of time (complete pivoting incurs $O(n^3)$ comparisons, while partial pivoting incurs $O(n^2)$), and to be honest in practical it's rarely done, because the improvement in stability is marginal. So, in complete pivoting we choose largest absolute value from entries (i, j), where $i \geq k$, $j \geq k$, and then we permute the row i with row k, column j with column k. We should not forget that the position of variables also changes when the change of columns happen, and at the end we have to bring variables to the initial order. So, the full code of the Question 1 for Gaussian Elimination with total pivoting will be:

```
import numpy as np

def Gaussian_Elimination_With_Total_Pivoting(A, b):
    A = A.astype('float64')
    b = b.astype('float64')
    n = len(b)
    rootIndices = np.arange(n, dtype = int)
    for k in range(n - 1):
        p, q =  np.unravel_index(abs(A[k:, k:]).argmax(), A[k:, k:].shape)
        p = p + k
        q = q + k

        # Index interchanging if needed
        if p != k:
            A[[k, p]] = A[[p, k]]
            b[k], b[p] = b[p], b[k]

        if q != k:
            A[:, [k, q]] = A[:, [q, k]]
            rootIndices[k], rootIndices[q] = rootIndices[q], rootIndices[k]

        # Elimination
        for j in range(k + 1, n):
            alpha = A[j, k] / A[k, k]
            A[j, k:] = A[j, k:]- alpha * A[k, k:]
            b[j] = b[j] - alpha * b[k]

    # Back substitution phase
    for k in range(n - 1, -1, -1):
        b[k] = (b[k] - np.dot(A[k, k + 1:n], b[k + 1:n])) / A[k, k]

    # Bringing order of elements of x to the proper order
    x = np.zeros(n, dtype = 'float64')
    for k in range(n):
```

```
        x[k] = b[np.where(rootIndices == k)]

    return x


def generate_A_b(n):
    # Initializing A and b and filling them by 0
    A = np.zeros((n, n), dtype = 'float64')
    b = np.zeros(n, dtype = 'float64')

    # Generating A and b
    for i in range(1, n + 1):
        b[i - 1] =  ((1 + i)**n - 1) / i
        for j in range(1, n + 1):
            A[i - 1, j - 1] = (1 + i)**(j - 1)

    return A, b


def coefficients_Error(true_coefficients, computed_coefficients):
    difference_Vector = abs(true_coefficients - computed_coefficients)

    # L_1 Error
    L_1_Absolute_Error = difference_Vector.sum()
    L_1_Relative_Error = L_1_Absolute_Error / abs(true_coefficients).sum()

    # L_infinity Error
    L_infinity_Absolute_Error = difference_Vector.max()
    L_infinity_Relative_Error = L_infinity_Absolute_Error / abs(true_coefficients).max()

    # L_2 Error
    sum_Of_Squared_Elements = 0
    for element in difference_Vector:
        sum_Of_Squared_Elements = sum_Of_Squared_Elements + element**2
    L_2_Absolute_Error = np.sqrt(sum_Of_Squared_Elements)
    sum_Of_Squared_Elements = 0
    for element in true_coefficients:
        sum_Of_Squared_Elements = sum_Of_Squared_Elements + element**2
    L_2_Relative_Error = L_2_Absolute_Error / np.sqrt(sum_Of_Squared_Elements)

    return L_1_Absolute_Error, L_1_Relative_Error, L_2_Absolute_Error, L_2_Relative_Error,
    L_infinity_Absolute_Error, L_infinity_Relative_Error


def main():
    list_n = [4, 5, 6, 7, 8, 9]
    for n in list_n:
        A, b = generate_A_b(n)

        computed_coefficients = Gaussian_Elimination_With_Total_Pivoting(A.copy(), b.copy())

        true_coefficients = np.ones(n, dtype = 'float64')

        error_L_1_a, error_L_1_r, error_L_2_a, error_L_2_r, error_L_infinity_a,
    error_L_infinity_r = coefficients_Error(true_coefficients, computed_coefficients)

        print(f"n = {n}\n")
        print(f"Computed Coefficients: {computed_coefficients}\n")
        print(f"True Coefficients: {true_coefficients}\n")
        print(f"L_1 Error: Absolute = {error_L_1_a}   Relative = {error_L_1_r}\n")
        print(f"L_2 Error: Absolute = {error_L_2_a}   Relative = {error_L_2_r}\n")
        print(f"L_infinity Error: Absolute = {error_L_infinity_a}   Relative = {
    error_L_infinity_r}\n-----------")

if __name__ == "__main__":
    main()
```

The output will be

```
n = 4

Computed Coefficients: [1. 1. 1. 1.]

True Coefficients: [1. 1. 1. 1.]

L_1 Error: Absolute = 6.472600233564663e-14   Relative = 1.6181500583911657e-14

L_2 Error: Absolute = 4.082497034239384e-14   Relative = 2.041248517119692e-14

L_infinity Error: Absolute = 2.964295475749168e-14   Relative = 2.964295475749168e-14
-----------
n = 5

Computed Coefficients: [1. 1. 1. 1. 1.]

True Coefficients: [1. 1. 1. 1. 1.]

L_1 Error: Absolute = 5.922928814072748e-12   Relative = 1.1845857628145494e-12

L_2 Error: Absolute = 3.475812635880829e-12   Relative = 1.5544306661764515e-12

L_infinity Error: Absolute = 2.536526544361095e-12   Relative = 2.536526544361095e-12
-----------
n = 6

Computed Coefficients: [1. 1. 1. 1. 1. 1.]

True Coefficients: [1. 1. 1. 1. 1. 1.]

L_1 Error: Absolute = 1.8242829469272692e-10   Relative = 3.040471578212115e-11

L_2 Error: Absolute = 1.0211481054782651e-10   Relative = 4.168819683719142e-11

L_infinity Error: Absolute = 7.596612228155664e-11   Relative = 7.596612228155664e-11
-----------
n = 7

Computed Coefficients: [1. 1. 1. 1. 1. 1. 1.]

True Coefficients: [1. 1. 1. 1. 1. 1. 1.]

L_1 Error: Absolute = 1.7518053674336898e-10   Relative = 2.5025790963338425e-11

L_2 Error: Absolute = 9.58302818872881e-11   Relative = 3.622044199185454e-11

L_infinity Error: Absolute = 7.09174940993762e-11   Relative = 7.09174940993762e-11
-----------
n = 8

Computed Coefficients: [1. 1. 1. 1. 1. 1. 1. 1.]

True Coefficients: [1. 1. 1. 1. 1. 1. 1. 1.]

L_1 Error: Absolute = 7.76111697131654e-09   Relative = 9.701396214145674e-10

L_2 Error: Absolute = 3.9942162012682345e-09   Relative = 1.4121686807209702e-09

L_infinity Error: Absolute = 2.763363649016526e-09   Relative = 2.763363649016526e-09
-----------
n = 9

Computed Coefficients: [1.00000035 0.99999936 1.00000049 0.99999979 1.00000005 0.99999999
 1.         1.         1.        ]

True Coefficients: [1. 1. 1. 1. 1. 1. 1. 1. 1.]

L_1 Error: Absolute = 1.7404236292151154e-06   Relative = 1.9338040324612393e-07
```

```
L_2 Error: Absolute = 9.001989516877575e-07   Relative = 3.000663172292525e-07

L_infinity Error: Absolute = 6.367419327357737e-07   Relative = 6.367419327357737e-07
-----------
```

Let's implement LU decomposition with total pivoting:
Now, let's implement LU decomposition with total pivoting, i.e. $PAQ = LU$, where P is a permutation matrix that corresponds for rows interchange and Q is the permutation matrix that correspond for columns interchange. So, first let's denote $y = Q^{-1}x$, then we'll have the equations as $LUy = Pb$. After using the notation $Uy = z$, the equations become

$$Lz = Pb. \tag{1.7}$$

Equation (1.7) can be solved for z by forward substitution. Then we'll have to solve

$$Uy = z. \tag{1.8}$$

From equation (1.8) we'll have y by the back substitution process. And finally, we'll have that $x = Qy$. So, the full code of the Question 1 for LU decomposition with total pivoting will be:

```python
import numpy as np

def LU_Decomposition_With_Total_Pivoting(A):
    A = A.astype('float64')
    n = len(A)
    P = np.eye(n, dtype = 'float64')
    Q = np.eye(n, dtype = 'float64')
    L = np.eye(n, dtype = 'float64')
    U = A.copy()
    for k in range(0, n-1):
        p, q =  np.unravel_index(abs(U[k:, k:]).argmax(), U[k:, k:].shape)
        p = p + k
        q = q + k

        if p != k:
            P_step = np.eye(n, dtype = 'float64')
            U[[p, k], k:n] = U[[k, p], k:n]
            L[[p, k], 0:k] = L[[k, p], 0:k]
            P_step[[p, k]] = P_step[[k, p]]
            P = np.dot(P_step, P)

        if q != k:
            Q_step = np.eye(n, dtype = 'float64')
            U[:, [q, k]] = U[:, [k, q]]
            Q_step[[q, k]] = Q_step[[k, q]]
            Q = np.dot(Q, Q_step)

        for j in range(k+1, n):
            L[j, k] =  U[j, k] / U[k, k]
            U[j, k:n] = U[j, k:n] - L[j, k] * U[k, k:n]

    return L, U, P, Q


def LU_TP_Solver(L, U, P, Q, b):
    L = L.astype('float64')
    U = U.astype('float64')
    P = P.astype('float64')
    Q = Q.astype('float64')
    b = b.astype('float64')
    n = len(U)
    # Forward substitution Lz = Pb
    b = np.dot(P, b)
    for k in range(1, n):
        b[k] = b[k] - np.dot(L[k, 0:k], b[0:k])
```

```
46
47      # Backward substitution Uy = z
48      for k in range(n - 1, -1, -1):
49          b[k] = (b[k] - np.dot(U[k, k + 1:n], b[k + 1:n])) / U[k, k]
50
51      # Finally finding x = Qy
52      b = np.dot(Q, b)
53      return b
54
55  def generate_A_b(n):
56      # Initializing A and b and filling them by 0
57      A = np.zeros((n, n), dtype = 'float64')
58      b = np.zeros(n, dtype = 'float64')
59
60      # Generating A and b
61      for i in range(1, n + 1):
62          b[i - 1] =  ((1 + i)**n - 1) / i
63          for j in range(1, n + 1):
64              A[i - 1, j - 1] = (1 + i)**(j - 1)
65
66      return A, b
67
68
69  def coefficients_Error(true_coefficients, computed_coefficients):
70      difference_Vector = abs(true_coefficients - computed_coefficients)
71
72      # L_1 Error
73      L_1_Absolute_Error = difference_Vector.sum()
74      L_1_Relative_Error = L_1_Absolute_Error / abs(true_coefficients).sum()
75
76      # L_infinity Error
77      L_infinity_Absolute_Error = difference_Vector.max()
78      L_infinity_Relative_Error = L_infinity_Absolute_Error / abs(true_coefficients).max()
79
80      # L_2 Error
81      sum_Of_Squared_Elements = 0
82      for element in difference_Vector:
83          sum_Of_Squared_Elements = sum_Of_Squared_Elements + element**2
84      L_2_Absolute_Error = np.sqrt(sum_Of_Squared_Elements)
85      sum_Of_Squared_Elements = 0
86      for element in true_coefficients:
87          sum_Of_Squared_Elements = sum_Of_Squared_Elements + element**2
88      L_2_Relative_Error = L_2_Absolute_Error / np.sqrt(sum_Of_Squared_Elements)
89
90      return L_1_Absolute_Error, L_1_Relative_Error, L_2_Absolute_Error, L_2_Relative_Error,
        L_infinity_Absolute_Error, L_infinity_Relative_Error
91
92
93  def main():
94      list_n = [4, 5, 6, 7, 8, 9]
95      for n in list_n:
96          A, b = generate_A_b(n)
97
98          L, U, P, Q = LU_Decomposition_With_Total_Pivoting(A)
99          computed_coefficients = LU_TP_Solver(L, U, P, Q, b.copy())
100
101         true_coefficients = np.ones(n, dtype = 'float64')
102
103         error_L_1_a, error_L_1_r, error_L_2_a, error_L_2_r, error_L_infinity_a,
        error_L_infinity_r = coefficients_Error(true_coefficients, computed_coefficients)
104
105         print(f"n = {n}\n")
106         print(f"Computed Coefficients: {computed_coefficients}\n")
107         print(f"True Coefficients: {true_coefficients}\n")
108         print(f"L_1 Error: Absolute = {error_L_1_a}   Relative = {error_L_1_r}\n")
109         print(f"L_2 Error: Absolute = {error_L_2_a}   Relative = {error_L_2_r}\n")
110         print(f"L_infinity Error: Absolute = {error_L_infinity_a}   Relative = {
        error_L_infinity_r}\n-----------")
```

```
111
112 if __name__ == "__main__":
113     main()
```

The ouput will be

```
n = 4

Computed Coefficients: [1. 1. 1. 1.]

True Coefficients: [1. 1. 1. 1.]

L_1 Error: Absolute = 5.562217353372034e-14   Relative = 1.3905543383430086e-14

L_2 Error: Absolute = 3.328910756630253e-14   Relative = 1.6644553783151265e-14

L_infinity Error: Absolute = 2.5424107263916085e-14   Relative = 2.5424107263916085e-14
-----------
n = 5

Computed Coefficients: [1. 1. 1. 1. 1.]

True Coefficients: [1. 1. 1. 1. 1.]

L_1 Error: Absolute = 8.329004153040387e-12   Relative = 1.6658008306080775e-12

L_2 Error: Absolute = 4.8845254156258095e-12   Relative = 2.1844261734329447e-12

L_infinity Error: Absolute = 3.5654812435836902e-12   Relative = 3.5654812435836902e-12
-----------
n = 6

Computed Coefficients: [1. 1. 1. 1. 1. 1.]

True Coefficients: [1. 1. 1. 1. 1. 1.]

L_1 Error: Absolute = 2.0798729405413496e-10   Relative = 3.4664549009022494e-11

L_2 Error: Absolute = 1.1664316759670534e-10   Relative = 4.7619373765644824e-11

L_infinity Error: Absolute = 8.671507956137248e-11   Relative = 8.671507956137248e-11
-----------
n = 7

Computed Coefficients: [1. 1. 1. 1. 1. 1. 1.]

True Coefficients: [1. 1. 1. 1. 1. 1. 1.]

L_1 Error: Absolute = 7.927590806033891e-10   Relative = 1.1325129722905558e-10

L_2 Error: Absolute = 4.3164400942084877e-10   Relative = 1.63146100548341e-10

L_infinity Error: Absolute = 3.191686914760794e-10   Relative = 3.191686914760794e-10
-----------
n = 8

Computed Coefficients: [1. 1. 1. 1. 1. 1. 1. 1.]

True Coefficients: [1. 1. 1. 1. 1. 1. 1. 1.]

L_1 Error: Absolute = 3.968919859431708e-09   Relative = 4.961149824289635e-10

L_2 Error: Absolute = 2.024094911721177e-09   Relative = 7.156256189216153e-10

L_infinity Error: Absolute = 1.3357376316136538e-09   Relative = 1.3357376316136538e-09
-----------
n = 9

Computed Coefficients: [1.00000004 0.99999992 1.00000006 0.99999997 1.00000001 1.
```

```
 1.          1.          1.         ]

True Coefficients: [1. 1. 1. 1. 1. 1. 1. 1. 1.]

L_1 Error: Absolute = 2.1078019973863604e-07   Relative = 2.3420022193181783e-08

L_2 Error: Absolute = 1.0868163652735904e-07   Relative = 3.622721217578635e-08

L_infinity Error: Absolute = 7.657769063307285e-08   Relative = 7.657769063307285e-08
-----------
```

2

Solution: Before looking at the code results, let's first solve by hand the system of equations. So, we have

$$\begin{cases} \epsilon x_1 + x_2 = 1 \\ x_1 + x_2 = 2. \end{cases} \tag{2.1}$$

The solution to (2.1) will be

$$x_1 = \frac{1}{1-\epsilon} \qquad x_2 = \frac{1-2\epsilon}{1-\epsilon}$$

$$\begin{aligned} \epsilon = 10^2 : \quad & x_1 = -\frac{1}{99} & x_2 = \frac{199}{99} \\ \epsilon = 10^{-1} : \quad & x_1 = \frac{10}{9} & x_2 = \frac{8}{9} \\ \epsilon = 10^{-9} : \quad & x_1 = \frac{10^9}{10^9 - 1} & x_2 = \frac{10^9 - 2}{10^9 - 1} \end{aligned}$$

Now, let's look at results from different methods that's been reviewed in Question 1. Before showing the result, I'd like to say that most part of the code for each method will not change at all, I'll just remove from them the function $generate\_A\_b(n)$ because we don't need it for this question, and I'll also modify the $main()$ function, and it will be like the following:

```
def main():
    list_epsilon = [1e2, 1e-1, 1e-9]
    for epsilon in list_epsilon:
        A = np.array([[epsilon, 1], [1, 1]]).astype('float64')
        b = np.array([1, 2]).astype('float64')

        #(1) computed_x = Naive_Gaussian_Elimination(A.copy(), b.copy())

        #(2) L, U = LU_Decomposition(A)
        # computed_x = LU_Solver(L, U, b.copy())

        #(3) computed_x = Gaussian_Elimination_With_Partial_Pivoting(A.copy(), b.copy())

        #(4) L, U, P = LU_Decomposition_With_Partial_Pivoting(A)
        # computed_x = LU_PP_Solver(L, U, P, b.copy())

        #(5) computed_x = Gaussian_Elimination_With_Total_Pivoting(A.copy(), b.copy())

        L, U, P, Q = LU_Decomposition_With_Total_Pivoting(A)
        computed_x = LU_TP_Solver(L, U, P, Q, b.copy())

        true_x = np.array([-1 / (epsilon - 1), (2 * epsilon - 1) / (epsilon - 1)]).astype('
    float64')

        error_L_1_a, error_L_1_r, error_L_2_a, error_L_2_r, error_L_infinity_a,
    error_L_infinity_r = coefficients_Error(true_x, computed_x)

        print(f"epsilon = {epsilon}\n")
        print(f"Computed x_1 = {computed_x[0]:.8f}", f"  Computed x_2 = {computed_x[1]:.8f}"
    , "\n")
        print(f"\"True\" x_1 = {true_x[0]:.8f}", f"  \"True\" x_2 = {true_x[1]:.8f}",  "\n")
        print(f"L_1 Error: Absolute = {error_L_1_a}   Relative = {error_L_1_r}\n")
        print(f"L_2 Error: Absolute = {error_L_2_a}   Relative = {error_L_2_r}\n")
        print(f"L_infinity Error: Absolute = {error_L_infinity_a}   Relative = {
    error_L_infinity_r}\n-----------")
```

The results will be the following

```
# NAIVE GAUSSIAN ELIMINATION
epsilon = 100.0
```

```
Computed x_1 = -0.01010101   Computed x_2 = 2.01010101

"True" x_1 = -0.01010101   "True" x_2 = 2.01010101

L_1 Error: Absolute = 1.734723475976807e-18   Relative = 8.586881206085196e-19

L_2 Error: Absolute = 1.734723475976807e-18   Relative = 8.62992240267239e-19

L_infinity Error: Absolute = 1.734723475976807e-18   Relative = 8.630031362899694e-19
-----------
epsilon = 0.1

Computed x_1 = 1.11111111   Computed x_2 = 0.88888889

"True" x_1 = 1.11111111   "True" x_2 = 0.88888889

L_1 Error: Absolute = 5.551115123125783e-16   Relative = 2.7755575615628914e-16

L_2 Error: Absolute = 4.577566798522237e-16   Relative = 3.217031222497203e-16

L_infinity Error: Absolute = 4.440892098500626e-16   Relative = 3.996802888650563e-16
-----------
epsilon = 1e-09

Computed x_1 = 0.99999997   Computed x_2 = 1.00000000

"True" x_1 = 1.00000000   "True" x_2 = 1.00000000

L_1 Error: Absolute = 2.9281931768210256e-08   Relative = 1.4640965884105128e-08

L_2 Error: Absolute = 2.9281931657187954e-08   Relative = 2.070545244103864e-08

L_infinity Error: Absolute = 2.9281931657187954e-08   Relative = 2.928193162790602e-08
-----------
```

```
# LU DECOMPOSITION
epsilon = 100.0

Computed x_1 = -0.01010101   Computed x_2 = 2.01010101

"True" x_1 = -0.01010101   "True" x_2 = 2.01010101

L_1 Error: Absolute = 1.734723475976807e-18   Relative = 8.586881206085196e-19

L_2 Error: Absolute = 1.734723475976807e-18   Relative = 8.62992240267239e-19

L_infinity Error: Absolute = 1.734723475976807e-18   Relative = 8.630031362899694e-19
-----------
epsilon = 0.1

Computed x_1 = 1.11111111   Computed x_2 = 0.88888889

"True" x_1 = 1.11111111   "True" x_2 = 0.88888889

L_1 Error: Absolute = 5.551115123125783e-16   Relative = 2.7755575615628914e-16

L_2 Error: Absolute = 4.577566798522237e-16   Relative = 3.217031222497203e-16

L_infinity Error: Absolute = 4.440892098500626e-16   Relative = 3.996802888650563e-16
-----------
epsilon = 1e-09

Computed x_1 = 0.99999997   Computed x_2 = 1.00000000

"True" x_1 = 1.00000000   "True" x_2 = 1.00000000

L_1 Error: Absolute = 2.9281931768210256e-08   Relative = 1.4640965884105128e-08
```

```

L_2 Error: Absolute = 2.9281931657187954e-08   Relative = 2.070545244103864e-08

L_infinity Error: Absolute = 2.9281931657187954e-08   Relative = 2.928193162790602e-08
-----------
```

```
# GAUSSIAN ELIMINATION WITH PARTIAL PIVOTING
epsilon = 100.0

Computed x_1 = -0.01010101   Computed x_2 = 2.01010101

"True" x_1 = -0.01010101   "True" x_2 = 2.01010101

L_1 Error: Absolute = 1.734723475976807e-18   Relative = 8.586881206085196e-19

L_2 Error: Absolute = 1.734723475976807e-18   Relative = 8.62992240267239e-19

L_infinity Error: Absolute = 1.734723475976807e-18   Relative = 8.630031362899694e-19
-----------
epsilon = 0.1

Computed x_1 = 1.11111111   Computed x_2 = 0.88888889

"True" x_1 = 1.11111111   "True" x_2 = 0.88888889

L_1 Error: Absolute = 0.0   Relative = 0.0

L_2 Error: Absolute = 0.0   Relative = 0.0

L_infinity Error: Absolute = 0.0   Relative = 0.0
-----------
epsilon = 1e-09

Computed x_1 = 1.00000000   Computed x_2 = 1.00000000

"True" x_1 = 1.00000000   "True" x_2 = 1.00000000

L_1 Error: Absolute = 0.0   Relative = 0.0

L_2 Error: Absolute = 0.0   Relative = 0.0

L_infinity Error: Absolute = 0.0   Relative = 0.0
-----------
```

```
# LU DECOMPOSITION WITH PARTIAL PIVOTING
epsilon = 100.0

Computed x_1 = -0.01010101   Computed x_2 = 2.01010101

"True" x_1 = -0.01010101   "True" x_2 = 2.01010101

L_1 Error: Absolute = 1.734723475976807e-18   Relative = 8.586881206085196e-19

L_2 Error: Absolute = 1.734723475976807e-18   Relative = 8.62992240267239e-19

L_infinity Error: Absolute = 1.734723475976807e-18   Relative = 8.630031362899694e-19
-----------
epsilon = 0.1

Computed x_1 = 1.11111111   Computed x_2 = 0.88888889

"True" x_1 = 1.11111111   "True" x_2 = 0.88888889

L_1 Error: Absolute = 0.0   Relative = 0.0

L_2 Error: Absolute = 0.0   Relative = 0.0

L_infinity Error: Absolute = 0.0   Relative = 0.0
```

```
-----------
epsilon = 1e-09

Computed x_1 = 1.00000000   Computed x_2 = 1.00000000

"True" x_1 = 1.00000000   "True" x_2 = 1.00000000

L_1 Error: Absolute = 0.0   Relative = 0.0

L_2 Error: Absolute = 0.0   Relative = 0.0

L_infinity Error: Absolute = 0.0   Relative = 0.0
-----------
```

```
# GAUSSIAN ELIMINATION WITH TOTAL PIVOTING
epsilon = 100.0

Computed x_1 = -0.01010101   Computed x_2 = 2.01010101

"True" x_1 = -0.01010101   "True" x_2 = 2.01010101

L_1 Error: Absolute = 1.734723475976807e-18   Relative = 8.586881206085196e-19

L_2 Error: Absolute = 1.734723475976807e-18   Relative = 8.62992240267239e-19

L_infinity Error: Absolute = 1.734723475976807e-18   Relative = 8.630031362899694e-19
-----------
epsilon = 0.1

Computed x_1 = 1.11111111   Computed x_2 = 0.88888889

"True" x_1 = 1.11111111   "True" x_2 = 0.88888889

L_1 Error: Absolute = 1.1102230246251565e-16   Relative = 5.551115123125783e-17

L_2 Error: Absolute = 1.1102230246251565e-16   Relative = 7.802446783097574e-17

L_infinity Error: Absolute = 1.1102230246251565e-16   Relative = 9.992007221626408e-17
-----------
epsilon = 1e-09

Computed x_1 = 1.00000000   Computed x_2 = 1.00000000

"True" x_1 = 1.00000000   "True" x_2 = 1.00000000

L_1 Error: Absolute = 1.1102230246251565e-16   Relative = 5.551115123125783e-17

L_2 Error: Absolute = 1.1102230246251565e-16   Relative = 7.850462293418875e-17

L_infinity Error: Absolute = 1.1102230246251565e-16   Relative = 1.1102230235149334e-16
-----------
```

```
# LU DECOMPOSITION WITH TOTAL PIVOTING
epsilon = 100.0

Computed x_1 = -0.01010101   Computed x_2 = 2.01010101

"True" x_1 = -0.01010101   "True" x_2 = 2.01010101

L_1 Error: Absolute = 1.734723475976807e-18   Relative = 8.586881206085196e-19

L_2 Error: Absolute = 1.734723475976807e-18   Relative = 8.62992240267239e-19

L_infinity Error: Absolute = 1.734723475976807e-18   Relative = 8.630031362899694e-19
-----------
epsilon = 0.1

Computed x_1 = 1.11111111   Computed x_2 = 0.88888889
```

```
"True" x_1 = 1.11111111   "True" x_2 = 0.88888889

L_1 Error: Absolute = 1.1102230246251565e-16   Relative = 5.551115123125783e-17

L_2 Error: Absolute = 1.1102230246251565e-16   Relative = 7.802446783097574e-17

L_infinity Error: Absolute = 1.1102230246251565e-16   Relative = 9.992007221626408e-17
-----------
epsilon = 1e-09

Computed x_1 = 1.00000000   Computed x_2 = 1.00000000

"True" x_1 = 1.00000000   "True" x_2 = 1.00000000

L_1 Error: Absolute = 1.1102230246251565e-16   Relative = 5.551115123125783e-17

L_2 Error: Absolute = 1.1102230246251565e-16   Relative = 7.850462293418875e-17

L_infinity Error: Absolute = 1.1102230246251565e-16   Relative = 1.1102230235149334e-16
-----------
```

So, as can be seen the best results are from those codes which involves pivoting, and for this question it appears to be that partial pivoting even looks better than total pivoting.

Role of Pivoting

The role of pivoting is to reduce instability that is inherent in Gaussian elimination. Instability arises only if the factor L or U is relatively large in size compared to A. Pivoting reduces instability by ensuring that L and U are not too big relative to A. By keeping all of the intermediate quantities that appear in the elimination process a manageable size, we can minimize rounding errors. As a consequence of pivoting, the algorithm for computing the LU factorization is backward stable. In theory, complete pivoting is more reliable than partial pivoting because we can use it with singular matrices. When a matrix is rank-deficient, complete pivoting can be used to reveal the rank of the matrix. Suppose A is an $n \times n$ matrix such that $r(A) = r < n$. At the start of the $r + 1$ elimination, the submatrix $A_{r+1:n,r+1:n} = 0$. This means that for the remaining elimination steps, $P_k = Q_k = M_k = I$ for $r + 1 \leq k \leq n$, where

$$M_{n-1}P_{n-1}M_{n-2}P_{n-2}\dots M_2P_2M_1P_1AQ_1Q_2\dots Q_{n-1} = U.$$

So after step r of the elimination, the algorithm can be terminated with the following factorization:

$$PAQ = LU = \begin{bmatrix} L_{11} & 0 \\ L_{21} & I_{n-r} \end{bmatrix} \begin{bmatrix} U_{11} & U_{12} \\ 0 & I_{n-r} \end{bmatrix}$$

In this case, L_{11} and U_{11} have size $r \times r$, and L_{21} and U_{12}^T have size $(n - r) \times r$. Complete pivoting works in this case because we are able to identify that every entry of the submatrix $A_{r+1:n,r+1:n}$ is 0. This would not work with partial pivoting because partial pivoting only searches a single column for a pivot and would fail before recognizing that all remaining potential pivots were zero. But in practice, we know that encountering a rank-deficient matrix in numerical linear algebra is very rare. This is the reason complete pivoting does not consistently produce more accurate results than partial pivoting. Therefore, partial pivoting is usually the best pivoting strategy of the two.

GitHub

Here is the GitHub link for the codes of this Homework (just in case):