Linear Algebra and Applications
Homework #02

Submitted By: Rifat Bin Rashid

RUID: 237000174

Problem 1:

Strategy : I have done the LU decomposition by the approach of gaussian elimination by row operations. First, a pivot element in the first row is selected , then the elements that lie just below the selected pivot element (the elements from the subsequent rows that lie in the same column with the selected pivot element) are made zero by row operations. Similar operations are run until we reach the second-last row of the matrix. Subsequently, the lower triangular matrix is created by preserving the multipliers for row operations in different stages. Here is my full code to do so:

```
#Taking the input matrix(A)
N = int(input("Enter the size of the square matrix A (N x N):
"))

A = []

for i in range(N):
       # Get the row input as a string
       row_input = input(f"Enter row {i+1} (separate elements by
spaces): ")

       # Split the input string into individual elements and
convert them to integers
       row = list(map(int, row_input.split()))

       # Append the row to the matrix
       A.append(row)

print("The matrix you entered (A) is:")
for row in A:
   print(row)
```

```
#LU_decomposition function
def lu_decomposition(matrix):
   rows = len(matrix)
   cols = len(matrix[0])

   # Initialize L as an identity matrix
   L = [[1.0 if i == j else 0.0 for j in range(rows)] for i in
range(rows)]

   # Initialize U as a copy of the input matrix
   U = matrix

   # Perform Gaussian elimination to get U (upper triangular)
and L (lower triangular)
   for i in range(rows):
       pv = U[i][i]  # Pivot element
       for k in range(i + 1, rows):
           mul = U[k][i] / pv  # Multiplier for elimination
           L[k][i] = mul  # Store the multiplier in L
           for j in range(cols):
               U[k][j] = U[k][j] - (mul * U[i][j])  # Update U

   return L, U

L, U = lu_decomposition(A)

# Print the results
print("\nL (Lower Triangular Matrix):")
for row in L:
   print(row)
```

```
print("\nU (Upper Triangular Matrix):")
for row in U:
    print(row)

# Verify the decomposition by multiplying L and U
def matrix_multiply(L, U):
    result = [[0.0] * N for _ in range(N)]
    for i in range(N):
        for j in range(N):
            for k in range(N):
                result[i][j] += L[i][k] * U[k][j]
    return result

# Reconstruct the original matrix
A_reconstructed = matrix_multiply(L, U)

print("\nReconstructed A (L * U):")
for row in A_reconstructed:
    print(row)
```

Example Output :

Enter the size of the square matrix A (N x N): 3
Enter row 1 (separate elements by spaces): 2 -1 -2
Enter row 2 (separate elements by spaces): -4 6 3
Enter row 3 (separate elements by spaces): -4 -2 8
The matrix you entered (A) is:
[2, -1, -2]
[-4, 6, 3]
[-4, -2, 8]

L (Lower Triangular Matrix):
[1.0, 0.0, 0.0]
[-2.0, 1.0, 0.0]
[-2.0, -1.0, 1.0]

U (Upper Triangular Matrix):
[2, -1, -2]
[0.0, 4.0, -1.0]
[0.0, 0.0, 3.0]

Reconstructed A (L * U):
[2.0, -1.0, -2.0]
[-4.0, 6.0, 3.0]
[-4.0, -2.0, 8.0]

Problem 2 :

Extra lines added in the code of problem 1 to declare the symmetric matrix A:

```
N = int(input("Enter the size of the symmetric matrix A (N= 4,
10 or 20): "))
A= []
def create_symmetric_matrix(N):
   # Initialize an N×N matrix with zeros
   A = [[ 0.0 for j in range(N)] for i in range(N)]

   for i in range (N):
       for j in range (N):
           if i == j:
               A[i][j]= 6
           elif abs(j-i)== 1:
               A[i][j]= 1.25
```

```
            elif abs(j-i)== 2:
                A[i][j]= 1.25

    return A

# Generate A for N=10
A = create_symmetric_matrix(N)
print("The symmetric matrix (A) is:")
for row in A:
    print(row)
```

The full matrices L and U only for N = 10:

Enter the size of the symmetric matrix A (N= 4, 10 or 20): 10
The symmetric matrix (A) is:
[6, 1.25, 1.25, 0.0, 0.0, 0.0, 0.0, 0.0, 0.0, 0.0]
[1.25, 6, 1.25, 1.25, 0.0, 0.0, 0.0, 0.0, 0.0, 0.0]
[1.25, 1.25, 6, 1.25, 1.25, 0.0, 0.0, 0.0, 0.0, 0.0]
[0.0, 1.25, 1.25, 6, 1.25, 1.25, 0.0, 0.0, 0.0, 0.0]
[0.0, 0.0, 1.25, 1.25, 6, 1.25, 1.25, 0.0, 0.0, 0.0]
[0.0, 0.0, 0.0, 1.25, 1.25, 6, 1.25, 1.25, 0.0, 0.0]
[0.0, 0.0, 0.0, 0.0, 1.25, 1.25, 6, 1.25, 1.25, 0.0]
[0.0, 0.0, 0.0, 0.0, 0.0, 1.25, 1.25, 6, 1.25, 1.25]
[0.0, 0.0, 0.0, 0.0, 0.0, 0.0, 1.25, 1.25, 6, 1.25]
[0.0, 0.0, 0.0, 0.0, 0.0, 0.0, 0.0, 1.25, 1.25, 6]

L (Lower Triangular Matrix):
['1.00', '0.00', '0.00', '0.00', '0.00', '0.00', '0.00', '0.00', '0.00', '0.00']
['0.21', '1.00', '0.00', '0.00', '0.00', '0.00', '0.00', '0.00', '0.00', '0.00']
['0.21', '0.17', '1.00', '0.00', '0.00', '0.00', '0.00', '0.00', '0.00', '0.00']
['0.00', '0.22', '0.19', '1.00', '0.00', '0.00', '0.00', '0.00', '0.00', '0.00']
['0.00', '0.00', '0.22', '0.18', '1.00', '0.00', '0.00', '0.00', '0.00', '0.00']
['0.00', '0.00', '0.00', '0.23', '0.18', '1.00', '0.00', '0.00', '0.00', '0.00']
['0.00', '0.00', '0.00', '0.00', '0.23', '0.18', '1.00', '0.00', '0.00', '0.00']

['0.00', '0.00', '0.00', '0.00', '0.00', '0.23', '0.18', '1.00', '0.00', '0.00']
['0.00', '0.00', '0.00', '0.00', '0.00', '0.00', '0.23', '0.18', '1.00', '0.00']
['0.00', '0.00', '0.00', '0.00', '0.00', '0.00', '0.00', '0.23', '0.18', '1.00']

U (Upper Triangular Matrix):
['6.00', '1.25', '1.25', '0.00', '0.00', '0.00', '0.00', '0.00', '0.00', '0.00']
['0.00', '5.74', '0.99', '1.25', '0.00', '0.00', '0.00', '0.00', '0.00', '0.00']
['0.00', '0.00', '5.57', '1.03', '1.25', '0.00', '0.00', '0.00', '0.00', '0.00']
['0.00', '-0.00', '0.00', '5.54', '1.02', '1.25', '0.00', '0.00', '0.00', '0.00']
['0.00', '-0.00', '0.00', '0.00', '5.53', '1.02', '1.25', '0.00', '0.00', '0.00']
['0.00', '0.00', '0.00', '0.00', '0.00', '5.53', '1.02', '1.25', '0.00', '0.00']
['0.00', '0.00', '0.00', '0.00', '0.00', '0.00', '5.53', '1.02', '1.25', '0.00']
['0.00', '-0.00', '0.00', '0.00', '0.00', '0.00', '0.00', '5.53', '1.02', '1.25']
['0.00', '-0.00', '0.00', '0.00', '0.00', '0.00', '0.00', '0.00', '5.53', '1.02']
['0.00', '0.00', '0.00', '0.00', '0.00', '0.00', '0.00', '0.00', '0.00', '5.53']

Reconstructed A (L * U):
['6.00', '1.25', '1.25', '0.00', '0.00', '0.00', '0.00', '0.00', '0.00', '0.00']
['1.25', '6.00', '1.25', '1.25', '0.00', '0.00', '0.00', '0.00', '0.00', '0.00']
['1.25', '1.25', '6.00', '1.25', '1.25', '0.00', '0.00', '0.00', '0.00', '0.00']
['0.00', '1.25', '1.25', '6.00', '1.25', '1.25', '0.00', '0.00', '0.00', '0.00']
['0.00', '0.00', '1.25', '1.25', '6.00', '1.25', '1.25', '0.00', '0.00', '0.00']
['0.00', '0.00', '0.00', '1.25', '1.25', '6.00', '1.25', '1.25', '0.00', '0.00']
['0.00', '0.00', '0.00', '0.00', '1.25', '1.25', '6.00', '1.25', '1.25', '0.00']
['0.00', '0.00', '0.00', '0.00', '0.00', '1.25', '1.25', '6.00', '1.25', '1.25']
['0.00', '0.00', '0.00', '0.00', '0.00', '0.00', '1.25', '1.25', '6.00', '1.25']
['0.00', '0.00', '0.00', '0.00', '0.00', '0.00', '0.00', '1.25', '1.25', '6.00']

N.B: Results are shown here as a string rounded with 2 decimal places for better display purpose.

Problem 3:

a. If the LU decomposition of a matrix exists, calculating the determinant of matrix A becomes easy, because

If , **A = LU ,**

Then, det (A) = det (LU) = det (L)*det (U)

Now L is a lower triangular matrix with all diagonal elements being 1, so det (L) = 1.

And U is a upper triangular matrix , so det (U)= U_{11}*U_{22}*U_{33}….*U_{NN}

Where U_{11}, U_{22}, U_{33}….,U_{NN} are the diagonal element of U

So, **det (A)= U_{11}*U_{22}*U_{33}….*U_{NN}**

b. New lines added in Problem 1 :

```
def determinant(matrix):
   #Compute the determinant of a square matrix using LU
decomposition.
   # Initialize an determinant with one
   det=1
   _, U = lu_decomposition(matrix)
   for i in range(len(U)):
       det= det*U[i][i]

   return det

det= determinant(A)
print(det)
```

The computed determinants of A:

N=4, det (A) = 1061.625

N=10, det (A) = 30360562.294555657

N=20, det (A) = 811188987621463.1

Problem 4 :

Code for problem 4:

```
# Test cases
Ns = [3, 4, 5, 10, 20]

def create_matrix_b(N):
    B = [[ 0.0 for j in range(N)] for i in range(N)]
    for i in range (N):
        for j in range (N):
            B[i][j] = 1 / ((i + 1) + (j + 1) + 1)
    return B

def lu_decomposition(matrix):
    rows = len(matrix)
    cols = len(matrix[0])

    # Initialize L as an identity matrix
    L = [[1.0 if i == j else 0.0 for j in range(rows)] for i in
range(rows)]

    # Initialize U as a copy of the input matrix
    U = matrix

    # Perform Gaussian elimination to get U (upper triangular)
and L (lower triangular)
    for i in range(rows):
        pv = U[i][i]  # Pivot element
        for k in range(i + 1, rows):
            mul = U[k][i] / pv  # Multiplier for elimination
            L[k][i] = mul  # Store the multiplier in L
```

```
            for j in range(cols):
                U[k][j] = U[k][j] - (mul * U[i][j])  # Update U

    return L, U

def determinant(matrix):
    #Compute the determinant of a square matrix using LU
decomposition.
    # Initialize an determinant with one
    det=1
    _, U = lu_decomposition(matrix)
    for i in range(len(U)):
        det= det*U[i][i]

    return det

for N in Ns:
    print(f"\nResult for N = {N}:")
    B = create_matrix_b(N)
    if N == 10:  # Display the full matrix only for N = 10
        print("Matrix B for N = 10:")
        for row in B:
            print([f"{val:.4f}" for val in row])
    L, U = lu_decomposition(B)
    print(f"Determinant for N = {N}:")
    det = determinant(B)
    print(det)
    print("-" * 30)
```

Results of problem 4:

Result for N = 3:
Determinant for N = 3:
2.6455026455029475e-06

Result for N = 4:
Determinant for N = 4:
9.373237831298245e-11

Result for N = 5:
Determinant for N = 5:
1.9322279593586738e-16

Result for N = 10:
Matrix B for N = 10:
['0.3333', '0.2500', '0.2000', '0.1667', '0.1429', '0.1250', '0.1111', '0.1000', '0.0909', '0.0833']
['0.2500', '0.2000', '0.1667', '0.1429', '0.1250', '0.1111', '0.1000', '0.0909', '0.0833', '0.0769']
['0.2000', '0.1667', '0.1429', '0.1250', '0.1111', '0.1000', '0.0909', '0.0833', '0.0769', '0.0714']
['0.1667', '0.1429', '0.1250', '0.1111', '0.1000', '0.0909', '0.0833', '0.0769', '0.0714', '0.0667']
['0.1429', '0.1250', '0.1111', '0.1000', '0.0909', '0.0833', '0.0769', '0.0714', '0.0667', '0.0625']
['0.1250', '0.1111', '0.1000', '0.0909', '0.0833', '0.0769', '0.0714', '0.0667', '0.0625', '0.0588']
['0.1111', '0.1000', '0.0909', '0.0833', '0.0769', '0.0714', '0.0667', '0.0625', '0.0588', '0.0556']
['0.1000', '0.0909', '0.0833', '0.0769', '0.0714', '0.0667', '0.0625', '0.0588', '0.0556', '0.0526']
['0.0909', '0.0833', '0.0769', '0.0714', '0.0667', '0.0625', '0.0588', '0.0556', '0.0526', '0.0500']
['0.0833', '0.0769', '0.0714', '0.0667', '0.0625', '0.0588', '0.0556', '0.0526', '0.0500', '0.0476']
Determinant for N = 10:
3.6577221888481705e-63

Result for N = 20:
Determinant for N = 20:
-5.39602858464844e-209

Program performance if N = 20:

As we can see from the determinant values for increasing N, determinant values become very small rapidly with increasing N . Each entry becomes smaller as i, j (row and column index) increases. For larger N (like N=20) , the matrix contains many very small numbers, which contribute to a smaller overall determinant.

As N increases, the given matrix becomes increasingly ill-conditioned (meaning their rows and columns are nearly linearly dependent). When computing the determinant of an ill-conditioned matrix, small rounding errors in floating-point arithmetic can lead to large errors in the result. For N= 20 here, the determinant is so small that it is likely dominated by rounding errors.

So, the determinant value is not reliable for N= 20.

The Files of my solution to the problems can be found at:

HW2_Linear_RBR_237000174