BAHRIA UNIVERSITY KARACHI CAMPUS
Department of Software Engineering

NUMERICAL ANALYSIS

LAB EXPERIMENT # 13

LU Decomposition: Doolittle's Method and Crout's Method

OBJECTIVE:

The objective of this lab is to introduce and explore two distinct methods for **LU decomposition: Doolittle's Method** and **Crout's Method**. The focus will be on understanding the algorithms behind these techniques and their applications in solving systems of linear equations.

Introduction

In numerical analysis and linear algebra, **lower upper (LU) decomposition** or factorization factors a matrix as the product of a lower triangular matrix and an upper triangular matrix (see matrix decomposition). The product sometimes includes a permutation matrix as well. LU decomposition can be viewed as the matrix form of Gaussian elimination. Computers usually solve square systems of linear equations using LU decomposition, and it is also a key step when inverting a matrix or computing the determinant of a matrix.

The LU decomposition method finds extensive application in solving simultaneous linear equations, which commonly arise in scientific and engineering problems. For instance, in structural engineering, LU decomposition assists in analyzing complex systems of forces acting on structures. Additionally, it's utilized in computational fluid dynamics to solve differential equations, enabling the simulation of fluid flow behavior. In finance, LU decomposition aids in risk assessment models and portfolio optimization. Moreover, LU decomposition plays a crucial role in image processing, signal processing, and machine learning algorithms for matrix manipulations and optimization tasks.

There exist various methods to perform LU decomposition, including **Doolittle's method, Crout's method, Cholesky decomposition**, and more. These techniques differ in their approach to decomposing the matrix into L and U matrices. Doolittle's method and Crout's method focus on partial pivoting and rearranging matrix elements to achieve the LU factorization. Cholesky decomposition specifically deals with symmetric positive-definite matrices and factors the matrix into the product of a lower triangular matrix and its transpose.

In this lab session, we will concentrate on discussing two specific methods **Crout's method** and **Doolittle's method**. Doolittle's method offers a general approach to LU decomposition, suitable for diverse matrices, enabling the factorization into lower and upper triangular matrices for solving linear equations. Similarly, Crout's method provides another versatile technique for LU decomposition, applicable to various matrix types, without requiring specific matrix properties like symmetry or positive-definiteness.

Understanding LU decomposition and its methods is crucial in numerical analysis, providing a foundation for efficient solutions to diverse computational problems. This lab emphasizes exploring the applications and implementation of Doolittle's, and Crout's methods, aiming to demonstrate their practical use in solving matrix-related problems and systems of linear equations.

Implementation of Doolittle's Method and Crout's methods

In this section, we will delve into the implementation and exploration of LU decomposition, specifically focusing on two prominent methods: **Crout's methods** and **Doolittle's method.**

a. Doolittle's Method

Doolittle's method is an approach used in LU decomposition to factorize a square matrix A into the product of a **lower triangular matrix (L)** and an **upper triangular matrix (U).** This method starts by assuming that the diagonal elements of the lower triangular matrix are all 1. The algorithm iteratively computes the elements of L and U matrices by manipulating the original matrix A.

The LU decomposition can be represented as **A = LU**, where:

- L is a lower triangular matrix with diagonal elements equal to 1.
- U is an upper triangular matrix.

Doolittle's method for LU decomposition follows a systematic procedure to factorize a square matrix *A* into lower (*L*) and upper (*U*) triangular matrices. Initially, the method sets *L* as an identity matrix and *U* as a zero matrix. It then progresses through an iterative process, systematically computing the values of each element within the *L* and *U* matrices in accordance with the rules of the decomposition. This involves a step-by-step traversal through the elements of both matrices, where each value is calculated based on specific formulas derived for the LU factorization. Once the complete factorization of *A* into *L* and *U* is achieved, the method enables the use of **forward** and **backward substitution techniques**.

These techniques facilitate the efficient solving of linear systems, leveraging the obtained *L* and *U* matrices to swiftly solve equations derived from the original matrix, thus providing solutions to complex systems of equations in a computationally effective manner.

The formulas for computing the L and U matrices in Doolittle's method are:

For U:

$$u_{ij} = a_{ij} - \sum_{k=1}^{i-1} l_{ik}\, u_{kj} \quad \text{where i} \leq \text{j}$$

For L:

$$l_{ij} = \frac{1}{u_{ij}} (a_{ij} - \sum_{k=1}^{i-1} l_{ik}\, u_{kj}) \quad \text{where i} > \text{j}$$

Implementation in Python

```
import numpy as np

def doolittle_lu_decomposition(A):
    n = len(A)
    L = np.eye(n)           # Initializing L as an identity matrix
    U = np.zeros((n, n))    # Initializing U as a zero matrix

    for i in range(n):
        # Calculate elements of upper triangular matrix U
        for j in range(i, n):
            # Using the formula U[i][j] = A[i][j] - sum(L[i][k] *
              U[k][j] for k in range(i))
            U[i][j]= A[i][j] - sum(L[i][k] * U[k][j] for k in range(i))

        # Calculate elements of lower triangular matrix L
        for j in range(i + 1, n):
            # Using the formula L[j][i] = (A[j][i] - sum(L[j][k] *
              U[k][i] for k in range(i))) / U[i][i]
            L[j][i] = (A[j][i]- sum(L[j][k]* U[k][i] for k in range(i)))
                      / U[i][i]
    return L, U


# Example usage:
A = np.array([[2, 5],
              [1, 2]], dtype=float)

# Perform LU decomposition using Doolittle's method
L, U = doolittle_lu_decomposition(A)

# Output the results
print("Matrix L:")
print(L)
print("Matrix U:")
print(U)
```

b. Crout's methods

Crout's method is a numerical technique used for matrix decomposition, specifically for decomposing a square matrix into a lower triangular matrix (L) and an upper triangular matrix (U). It's a variation of LU decomposition, where L has ones on its diagonal, and U is an upper triangular matrix. In the context of solving linear systems of equations, Crout's method decomposes a matrix A into L and U such that **A = L * U.** This decomposition allows for efficient solving of systems of linear equations, especially when the matrix A is large or sparse.

Crout's Method works by iteratively decomposing the original matrix A into lower and upper triangular matrices. The process involves performing Gaussian elimination with pivoting on the original matrix to obtain the L and U matrices. At each step, elements of L and U are determined using forward and backward substitution techniques.

The formulation for Crout's method can be summarized as follows: considering a square matrix A of size n x n, the objective of the method is to determine matrices L (lower triangular) and U (upper triangular) such that A = L * U. The algorithm begins by assuming that A can be expressed as the product of L and U:

where:

- L is a lower triangular matrix with ones on the diagonal.
- U is an upper triangular matrix.

Then, Crout's method involves solving equations iteratively to find the elements of L and U matrices by using following defining equations:

For L:

$$l_{ij} = a_{ij} - \sum_{k=1}^{i-1} l_{ik}\, u_{kj} \quad \text{where } i \geq j$$

For U:

$$u_{ij} = \frac{1}{u_{ij}} (a_{ij} - \sum_{k=1}^{i-1} l_{ik}\, u_{kj}) \quad \text{where } i < j$$

Implementation in Python

```
import numpy as np

def crout_lu_decomposition(A):
    n = len(A)
    U = np.eye(n)           # Initializing L as an identity matrix
    L = np.zeros((n, n))    # Initializing U as a zero matrix

    for i in range(n):
        for j in range(i, n):
            # Computing elements of U matrix
            L[j][i] = A[j][i] - sum([L[j][k] * U[k][i] for k in range(i)])

        for j in range(i+1, n):
            # Computing elements of L matrix
            U[i][j] = (A[i][j] - sum([L[i][k] * U[k][j] for k in
            range(i)])) / L[i, i]

    return L, U



# Example matrix
A = np.array([[2, 5],
              [1, 2]], dtype=float)


# Perform Crout's LU Decomposition
L, U = crout_lu_decomposition(A)
print("Lower Triangular Matrix L:")
print(L)
print("\nUpper Triangular Matrix U:")
print(U)
```

Usage of LU Decomposition

LU decomposition serves multiple purposes such as **determinant computation, matrix inversion,** and solving systems of **linear equations.** While it's possible to directly find determinants, inverses, or solutions to equations, LU decomposition offers computational advantages. By decomposing a matrix into **lower (L)** and upper **(U) triangular matrices**, the original matrix can be represented as the product of L and U, simplifying **complex operations**. This decomposition provides numerical stability and efficiency, making computations less prone to rounding errors and faster to execute.

To compute determinants or inverses, one can utilize the diagonal elements of the upper triangular matrix obtained from LU decomposition. Similarly, solving systems of linear equations involves forward and backward substitution using L and U matrices, simplifying the process and making it computationally efficient. This method ensures accuracy and stability in numerical computations. The provided Python code showcases how LU decomposition can be employed to find determinants, inverses, and solutions to equations.

a. Matrix Determinant

After obtaining the lower triangular matrix U using the LU decomposition method, one can straightforwardly calculate the determinant of the matrix through the following code:

```
def determinant_from_U(U):
    # Calculate determinant from the diagonal elements of U
    det = np.prod(np.diagonal(U))
    return det
```

b. Matrix Inversion

Similarly, once you've obtained the lower triangular matrix L and U using LU decomposition, you can effortlessly find the inverse of the matrix using the code below:

```
def matrix_inverse(L, U):
    n = len(L)
    I = np.eye(n)  # Create an identity matrix of size n

    # Solve Ly = I for y using forward substitution
    Y = np.zeros((n, n))
    for i in range(n):
        Y[i][i] = 1
        for j in range(i):
            Y[i] -= L[i][j] * Y[j]

    # Solve Ux = y for x using backward substitution
    X = np.zeros((n, n))
    for i in range(n - 1, -1, -1):
        X[i] = Y[i]
        for j in range(i + 1, n):
            X[i] -= U[i][j] * X[j]
        X[i] /= U[i][i]

    return X
```

c. Solving Linear Equations

Again, once you have the lower triangular matrix L from the LU decomposition method, solving systems of linear equations becomes straightforward using the following code:

```
def solve_equations(L, U, b):
    n = len(b)
    y = np.zeros(n)
    x = np.zeros(n)

    # Solve Ly = b using forward substitution
    for i in range(n):
        y[i] = b[i]
        for j in range(i):
            y[i] -= L[i][j] * y[j]
        y[i] /= L[i][i]

    # Solve Ux = y using backward substitution
    for i in range(n - 1, -1, -1):
        x[i] = y[i]
        for j in range(i + 1, n):
            x[i] -= U[i][j] * x[j]
        x[i] /= U[i][i]

    return x
```

Lab Tasks

1. Write a Python program that can find the solution of following linear equations using LU Decomposition:

   - $2x + 5y = 21\ ,\ x + 2y = 8$
   - $2x + 3y - z = 5\ ,\ 3x + 2y + z = 10\ ,\ x - 5y + 3z = 0$

2. Write a Python program that utilizes LU Decomposition to find the inverse as well as determinant of following matrices:

   - $\begin{bmatrix} 2 & 1 & -1 \\ -3 & -1 & 2 \\ -2 & 1 & 2 \end{bmatrix}$
   - $\begin{bmatrix} 1 & 1 & 1 \\ 4 & 3 & -1 \\ -3 & 5 & 3 \end{bmatrix}$