In [ ]:
```
import numpy as np
import math as mt
import pandas as pd
import matplotlib.pyplot as plt
import copy
```

Matrix Decomposition

- For Matrix Decomposition, PA = UL P = Permutation Matrix U = Upper Triangular L = Lower Triangular Matrix
- We first use algorithm to find lower triangular matrix and simultaneously store row operations in matrix M and permutations in matrix P.
- The Algorithm looks like
  - $P_1A = L_1$ -> $M_1P_1A = L_2$ and so on upto M_n
  - Now M = $M_nM_{n-1}....M_2M_1$
  - P = $P_nP_{n-1}P_2P_1$
  - We know that MPA = L
  - Thus U = M^{-1}

In [ ]:
```
# def get_upper_permute(matrix):
#     permute = []
#     upper = copy.deepcopy(matrix)
#     n = len(matrix)
#     for i in range(n):
#         j = 0
#         while(j < n and upper[i][j] == 0):
#             j += 1
#         permute.append(j)
#         for k in range(i+1,n):
#             m = upper[k][j]/upper[i][j]
#             upper[k][j] = 0
#             for l in range(j+1,n):
#                 upper[k][l] -= m*upper[i][l]
#     new_upper = copy.deepcopy(upper)
#     new_upper = upper[permute]
#     permutation = [[0]*n for _ in range(n)]
#     permute = np.array(permute)
#     # print(permute)
#     print(permutation)
#     for i,a in enumerate(permute):
#         print(i,a)
#         permutation[i][a] = 1
#     print(permutation)
#     return new_upper,np.array(permutation)

# def get_upper(matrix,permute):
#     upper = np.dot(permute,matrix)
#     n = len(upper)
#     for i in range(n):
#             for j in range(i+1,n):
#                 m = upper[j][i]/upper[i][i]
#                 upper[j][i] = 0
```

```
#                   for k in range(i+1,n):
#                       upper[j][k] = upper[j][k] - m*upper[i][k]
#       return upper

# def get_permute(matrix):
#       n = len(matrix)
#       permutation = np.eye(n)
#       new_m = copy.deepcopy(matrix)
#       for i in range(n):
#           maxrow = i
#           maxval = new_m[i][i]
#           for j in range(i+1,n):
#               if(maxval < new_m[j][i]):
#                   maxval = new_m[j][i]
#                   maxrow = j
#           permutation[[i,maxrow]] = permutation[[maxrow,i]]
#           new_m[[i,maxrow]] = new_m[[maxrow,i]]
#       return np.array(permutation)

# def get_lower(matrix,upper,permute):
#       pa = np.dot(permute,matrix)
#       l = np.dot(pa,np.linalg.inv(upper))
#       return l

# def myUL(matrix):
#       permute = get_permute(matrix)
#       upper = get_upper(matrix,permute)
#       lower = get_lower(matrix,upper,permute)
#       print(upper)
#       print(matrix)
#       print(lower)
#       print(permute)
#       #return upper,lower,permute
```

In [ ]:

```
def myUL(matrix):
    n = len(matrix)
    lowmat = copy.deepcopy(matrix)
    permute = np.eye(n)
    identity = np.eye(n)
    operation = identity
    for i in range(n-1, -1, -1):
        if lowmat[i][i] == 0:
            print('enter')
            j = i - 1
            while j > -1 and lowmat[j][i] == 0:
                j -= 1
            if j == -1:
                continue
            else:
                permute[[i, j]] = permute[[j, i]]
                lowmat[[i,j]] = lowmat[[j,i]]
        op = copy.deepcopy(identity)
        for j in range(i - 1, -1, -1):
            m = lowmat[j][i] / lowmat[i][i]
            op[j][i] = -m
            lowmat[j][i] = 0
            for k in range(i - 1, -1, -1):
                x1 = m * lowmat[i][k]
                x2 = lowmat[j][k]
                lowmat[j][k] = x2 - x1
```

```
        #print(i,op)
        operation = np.dot(op,operation)
    upper = get_inv(operation)
    return upper,lowmat,permute
def get_inv(matrix):
    matinv = np.linalg.inv(matrix)
    return matinv

def printList(lst):
    for a in lst:
        print(a)
```

In [ ]:
```
mat = np.array([[2,1,1],[4,3,3],[8,7,9]],dtype = float)
upper,lower,permute = myUL(mat)
print('Upper Triangular :')
printList(upper)
print('Lower Triangular :')
printList(lower)
print('Permutation Matrix :')
printList(permute)
print('PA = UL')
printList((np.dot(upper,lower)))
```

```
Upper Triangular :
[1.         0.33333333 0.11111111]
[0.         1.         0.33333333]
[0. 0. 1.]
Lower Triangular :
[0.66666667 0.         0.        ]
[1.33333333 0.66666667 0.        ]
[8. 7. 9.]
Permutation Matrix :
[1. 0. 0.]
[0. 1. 0.]
[0. 0. 1.]
PA = UL
[2. 1. 1.]
[4. 3. 3.]
[8. 7. 9.]
```

Gaussian Elimination

- We need to solve $Ax = b$
- In Guassian Elimination, We follow similar procedure followed in matrix decomposition
- We attach right side b with matrix A
- Now we convert matrix A into row-echelon form i.e. try to make it upper triangular.We do same operations on vector b.
- After doing row operation $A- > R$ and $b- > c$
- Comparing Rx = c, we can easily compute x

In [ ]:
```
class LinearSolve:
    def __init__(self,matrix,vector) -> None:
        self.matrix = matrix
        self.vector = vector
```

```python
        self.res = np.array([])
        self.err = 1e-4

    def get_upper_permute(self,matrix):
        n = len(matrix)
        upper = copy.deepcopy(matrix)
        permute = np.eye(n)
        for i in range(n):
            if(upper[i][i] == 0):
                j = i+1
                while(j < n and upper[j][i] == 0):
                    j += 1
                if(j == n):
                    continue
                else:
                    permute[[i,j]] = permute[[j,i]]
                    upper = np.dot(permute,upper)
            for j in range(i+1,n):
                m = upper[j][i]/upper[i][i]
                upper[j][i] = 0
                for k in range(i+1,n+1):
                    upper[j][k] = upper[j][k] - m*upper[i][k]
        return upper,permute

    def get_inv(self,matrix):
        matinv = np.linalg.inv(matrix)
        return matinv

    def get_lower(self,matrix,upper,permute):
        pa = np.dot(permute,matrix)
        upp_inv = self.get_inv(upper)
        return np.dot(pa,upp_inv)

    def gauss(self,A_n,b):
        n = len(A_n)
        A = copy.deepcopy(A_n)
        #st = deque()
        for i in range(n):
            A[i].append(b[i])

        # for i in range(n):
        #     if(A[i][i] == 0):
        #         A[i][i] = self.err
        #     for j in range(i+1,n):
        #         m = A[j][i]/A[i][i]
        #         A[j][i] = 0
        #         for k in range(i+1,n+1):
        #             A[j][k] = A[j][k] - m*A[i][k]

        A,_ = self.get_upper_permute(A)
        x = [0]*n
        #print(A)
        for i in range(n-1,-1,-1):
            x[i] = A[i][n]
            k = 0
            for j in range(n-1,i,-1):
                x_j = x[j]*A[i][j]
                if(abs(x_j) < self.err):
                    continue
                x[i] -= x[j]*A[i][j]
```

```
                x[i] /= A[i][i]
                if(abs(x[i]) < self.err):
                    x[i] = 0
            #x.reverse()
            self.res = x
            return np.array(x)

    def inbuilt(self,A_n,b):
        A = copy.deepcopy(A_n)
        A = np.array(A)
        b = np.array(b)
        #print(A)
        x = np.dot(np.linalg.inv(A),b)
        x = [0 if abs(a) < self.err else a for a in x]
        x = np.array(x)
        return x
    def get_roots(self,method = 'None'):
        match method:
            case 'gauss':
                return self.gauss(self.matrix,self.vector)
            case 'numpy':
                return self.inbuilt(self.matrix,self.vector)
            case 'None':
                return self.gauss(self.matrix,self.vector)
```

In [ ]:

```
A = [[1,2,1],[2,2,3],[-1,-3,0]]
b = [0,3,2]
ls = LinearSolve(A,b)
val_x = ls.get_roots(method = 'gauss')
print('Roots are: ',val_x)
print('Roots using numpy: ',ls.get_roots(method='numpy'))
```

```
Roots are:  [ 1. -1.  1.]
Roots using numpy:  [ 1. -1.  1.]
```

In [ ]:

```
A = [[4,3,2,1],[3,4,3,2],[2,3,4,3],[1,2,3,4]]
b = [1,1,-1,-1]
#A_n = copy.deepcopy(A)
ls = LinearSolve(A,b)
val_x = ls.get_roots(method = 'gauss')
print('Roots are: ',val_x)
print('Roots using numpy: ',ls.get_roots(method='numpy'))
```

```
Roots are:  [ 0.  1. -1.  0.]
Roots using numpy:  [ 0.  1. -1.  0.]
```