Q.1

Triangular matrices:

$$\begin{bmatrix} 1 & 0 & 0 \\ 0 & 1 & 0 \\ \frac{1}{3} & -\frac{1}{3} & 1 \end{bmatrix} \qquad \begin{bmatrix} 3 & 2 & 0 \\ 0 & 5 & 1 \\ 0 & 0 & \frac{1}{3} \end{bmatrix}$$

Lower　　　　　　　　　　Upper

Permutation matrix:

$$\begin{bmatrix} 1 & 0 & 0 \\ 0 & 0 & 1 \\ 0 & 1 & 0 \end{bmatrix}$$

Meaning of permutation matrix:

Sometimes we obtain a matrix that it has 0 values in its pivot elements. However, the pivot elements must not as 0 value in order to generate multipliers. So we introduce Permutation matrix in order to generate other non-zero pivot elements, and it satisfies PA = LU.

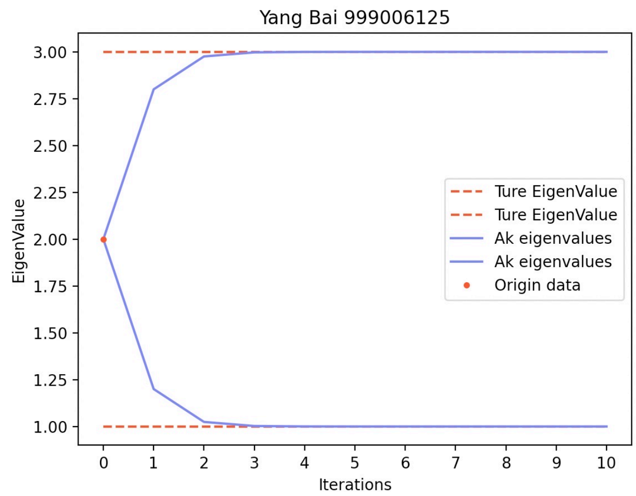
Demonstration:

```
Permutation Matrix :
[[1. 0. 0.]
 [0. 0. 1.]
 [0. 1. 0.]]
Lower Triangular Matrix :
[[ 1.          0.          0.        ]
 [ 0.          1.          0.        ]
 [ 0.33333333 -0.33333333  1.        ]]
Upper Triangular Matrix :
[[3.         2.         0.        ]
 [0.         5.         1.        ]
 [0.         0.         0.33333333]]
Result of the matrixs' product :
[[ 3.  2.  0.]
 [ 1. -1.  0.]
 [ 0.  5.  1.]]
```

```python
from scipy import linalg

matrix_Given = [[3,2,0],
               [1,-1,0],
               [0,5,1]]

luResults = linalg.lu(matrix_Given)

print("Permutation Matrix : ")
print(luResults[0])

print("Lower Triangular Matrix : ")
print(luResults[1])

print("Upper Triangular Matrix : ")
print(luResults[2])

print("Result of the matrixs' product : ")
print((luResults[0].dot(luResults[1])).dot(luResults[2]))
```

Codes of Q1

Q.2



After 10 iterations of QR decompositions, we obtain A10 = ~ 0

```
[[ 3.00000000e+00 -1.12900585e-05]
 [-1.12900585e-05  1.00000000e+00]]
```

~ 0

So we can consider eigenvalue is [3,1] and eigenvector is [1,1] and [-1,1] by solving the equation (A0 - λI)x = 0 | λ = 3 & 1

```python
matrix_Given = np.array([[2,1],
                [1,2]])

def getEigenValue(qrmatrix = np.array(0),depth = 1):
    if depth == 0:
        #return qrmatrix
        return qrmatrix.diagonal()
    mt = linalg.qr(qrmatrix)
    return getEigenValue(np.dot(mt[1],mt[0]),depth-1)

#print(getEigenValue(matrix_Given,11))

eigenValueList = []
for i in range(0,11):
    eigenValueList.append(getEigenValue(matrix_Given,i))

xValues = range(0, 11)
plt.title("Yang Bai 999006125")
plt.xlabel("Iterations")
plt.ylabel("EigenValue")
plt.xticks(range(0,11,1))
plt.plot(xValues, [3]*11, '--', label = 'Ture EigenValue',color='#F54325')
plt.plot(xValues, [1]*11, '--', label = 'Ture EigenValue',color='#F54325')
plt.plot(xValues,eigenValueList, color = '#6D73F2', label = 'Ak eigenvalues')
plt.plot(0 , 2, '.', color = '#F54325', label = 'Origin data')
plt.legend()
plt.show()
```

Codes of Q2

PS: Calculations of eigenvectors are done by hand

Q. 3

In classical Newton-Raphson method, The idea is to start with an initial guess(X_0), then to approximate the function by its tangent line, and finally to compute the x-intercept of this tangent line(X_1). This x-intercept will typically be a better approximation to the original function's root than the first guess, and the method can be iterated. ($X_n \to X_{n+1}$)
For the tangent line to the curve f(X) at X=Xn, intercepts the x-axis at xn+1, we can write the slope as:
$f'(X_n) = f(X_n) - 0 / X_n - X_{n+1}$
So we can solve X_{n+1} by
$X_{n+1} = X_n - f(X_n) / f'(X_n)$

We notice that everytime we want to iterate in order to get a better solution, we need to obtain the derivative of last fuction (which is fn). But sometimes we will meet a situation that the first derivative of the function is not given. So we use the secant of two points to replace tangent.
To calculate the secant of two points (X_{n-1}, $f(X_{n-1})$) , (X_n, $f(X_n)$):
secant = $f(X) - f(X_{n-1}) / X_n - X_{n-1}$
replace secant with the tangent($f'(X_n)$) in Newton-Raphson method:
$X_{n+1} = X_n - (X_n - X_{n-1})f(X_n) / f(X_n) - f(X_{n-1})$

That is the connection between the secant method and the Newton method.