Homework 3

Author: Wang Haozhe

Date: 2024/4/3

Problem 3.1

Code:

```python
#!/usr/local/bin/python3.11
# -*- coding: UTF-8 -*-
# @Project : Computational_Physics
# @File    : Problem3_1.py
# @Author  : Albert Wang
# @Time    : 2024/4/1
# @Brief   : None

import numpy as np


def lu_decomposition(A):
    """
    LU decomposition of a matrix A.

    :param A: a square matrix.
    :return: the result of the decomposition. i.e L, U.
    """
    if A.shape[0] != A.shape[1]:
        raise ValueError('Input Matrix must be square')  # Error handle

    dimension = A.shape[0]
    U_ = np.copy(A)  # Separate the following operations from the original matrix A
    L_ = np.zeros_like(U_)  # Initialize L

    for i in range(dimension):
        for j in range(i, dimension):
            L_[j, i] = U_[j, i]  # Assemble L

        U_[i] /= U_[i, i]  # Divide by the diagonal element
        for j in range(i + 1, dimension):
            U_[j, :] -= U_[j, i] * U_[i, :]  # Subtract from the lower rows

    return L_, U_


def lu_solution(A, v):
    """
```

```
    Solve Ax=v.
    :param A: a square matrix.
    :param v: a vector which has the same dimension as A.
    :return: the solution vector.
    """
    L_, U_ = lu_decomposition(A)

    dimension = A.shape[0]
    y_ = np.zeros(dimension)
    x_ = np.zeros(dimension)
    u_ = np.copy(v)

    # Calculate vector y
    for i in range(dimension):
        y_[i] = u_[i]
        for j in range(i):
            y_[i] -= L_[i, j] * y_[j]
        y_[i] /= L_[i, i]

    # Calculate vector x
    for i in range(dimension):
        x_[dimension - i - 1] = y_[dimension - i - 1]
        for j in range(i):
            x_[dimension - i - 1] -= x_[dimension - j - 1] * U_[dimension - i - 1, dime
        x_[dimension - i - 1] /= U_[dimension - i - 1, dimension - i - 1]

    return x_


if __name__ == '__main__':
    M = np.array([[2, 1, 4, 1],
                  [3, 4, -1, -1],
                  [1, -4, 1, 5],
                  [2, -2, 1, 3]], float)
    v = np.array([-4, 3, 9, 7])

    L, U = lu_decomposition(M)
    print("L:", L)
    print("U:", U)
    print(np.dot(L, U))  # Verify

    x = lu_solution(M, v)
```

```python
print("x:", x)
print(np.linalg.solve(M, v))  # Verify
```

Result:

The LU decomposition given by the function `lu_decomposition(A)` are

```
L: [[  2.    0.    0.    0. ]
 [  3.    2.5   0.    0. ]
 [  1.   -4.5 -13.6   0. ]
 [  2.   -3.  -11.4  -1. ]]
U: [[ 1.   0.5  2.   0.5]
 [ 0.   1.  -2.8 -1. ]
 [-0.  -0.   1.  -0. ]
 [-0.  -0.  -0.   1. ]]
```

Verify this result by multiplying them:

```
[[ 2.  1.  4.  1.]
 [ 3.  4. -1. -1.]
 [ 1. -4.  1.  5.]
 [ 2. -2.  1.  3.]]
```

Solve Ax=v and verify the result by using `np.linalg.solve()` :

```
x: [ 2. -1. -2.  1.]
[ 2. -1. -2.  1.]
```

Problem 3.1(c)

Code:

```
#!/usr/local/bin/python3.11
# -*- coding: UTF-8 -*-
# @Project : Computational_Physics
# @File    : Problem3_1_c.py
# @Author  : Albert Wang
# @Time    : 2024/4/3
# @Brief   : None

import numpy as np


def lu_decomposition(A):
    """
    LU decomposition of a matrix A.

    :param A: a square matrix.
    :return: the result of the decomposition and the permutation matrix. i.e L, U and P
    """
    if A.shape[0] != A.shape[1]:
        raise ValueError('Input Matrix must be square')  # Error handle

    dimension = A.shape[0]
    U_ = np.copy(A)  # Separate the following operations from the original matrix A
    L_ = np.zeros_like(U_)  # Initialize L
    P_ = np.eye(dimension)  # Initialize P

    for i in range(dimension):
        max_ = np.argmax(U_[i:dimension, i]) + i  # Find the largest number in each col
        # Swap rows
        temp = np.copy(U_[i])
        U_[i] = U_[max_]
        U_[max_] = temp

        temp = np.copy(P_[i])
        P_[i] = P_[max_]
        P_[max_] = temp

        for j in range(i, dimension):
```

```python
            L_[j, i] = U_[j, i]  # Assemble L

        U_[i] /= U_[i, i]  # Divide by the diagonal element
        for j in range(i + 1, dimension):
            U_[j, :] -= U_[j, i] * U_[i, :]  # Subtract from the lower rows

    return L_, U_, P_


def lu_solution(A, v):
    """
    Solve Ax=v.
    :param A: a square matrix.
    :param v: a vector which has the same dimension as A.
    :return: the solution vector.
    """
    L_, U_, P_ = lu_decomposition(A)

    dimension = A.shape[0]
    y_ = np.zeros(dimension)
    x_ = np.zeros(dimension)
    u_ = np.copy(v)
    u_ = np.dot(P_, u_)

    # Calculate vector y
    for i in range(dimension):
        y_[i] = u_[i]
        for j in range(i):
            y_[i] -= L_[i, j] * y_[j]
        y_[i] /= L_[i, i]

    # Calculate vector x
    for i in range(dimension):
        x_[dimension - i - 1] = y_[dimension - i - 1]
        for j in range(i):
            x_[dimension - i - 1] -= x_[dimension - j - 1] * U_[dimension - i - 1, dime
        x_[dimension - i - 1] /= U_[dimension - i - 1, dimension - i - 1]

    return x_


if __name__ == '__main__':
    M = np.array([[0, 1, 4, 1],
```

```
                  [3, 4, -1, -1],
                  [1, -4, 1, 5],
                  [2, -2, 1, 3]], float)
    v = np.array([-4, 3, 9, 7])

    L, U, P = lu_decomposition(M)
    print("L:", L)
    print("U:", U)
    print("P:", P)
    print(np.dot(L, U))  # Verify

    x = lu_solution(M, v)
    print("x:", x)
    print(np.linalg.solve(M, v))  # Verify
```

Result

LU decomposition with partial pivoting by using the new `lu_decomposition()` function:

```
L: [[ 3.          0.          0.          0.        ]
 [ 0.          1.          0.          0.        ]
 [ 1.         -5.33333333 22.66666667  0.        ]
 [ 2.         -4.66666667 20.33333333 -1.23529412]]
U: [[ 1.          1.33333333 -0.33333333 -0.33333333]
 [ 0.          1.          4.          1.        ]
 [ 0.          0.          1.          0.47058824]
 [-0.         -0.         -0.          1.        ]]
P: [[0. 1. 0. 0.]
 [1. 0. 0. 0.]
 [0. 0. 1. 0.]
 [0. 0. 0. 1.]]
```

Matrix P represents the permutation process during partial pivoting.

Verify this result by multiplying L and U:

```
[[ 3.  4. -1. -1.]
 [ 0.  1.  4.  1.]
 [ 1. -4.  1.  5.]
 [ 2. -2.  1.  3.]]
```

Solve Ax=v and verify the result by using `np.linalg.solve()` :

```
x: [ 1.61904762 -0.42857143 -1.23809524  1.38095238]
[ 1.61904762 -0.42857143 -1.23809524  1.38095238]
```

Problem 3.2(a)

It's self-evident that when $i = j, < q_i, q_j >= 1$. The follows proves that when $i \neq j, < q_i, q_j >= 0$.

When $k = 1$,

$$\begin{aligned} < q_1, q_0 > &= \frac{1}{|u_1|}(< a_1, q_0 > - < q_0, a_1 > \cdot < q_0, q_0 >) \\ &= \frac{1}{|u_1|}(< a_1, q_0 > - < q_0, a_1 > \cdot 1) \\ &= 0 \end{aligned}$$

When $k = i$,

$$\begin{aligned} < q_i, q_{i-1} > &= \frac{1}{|u_i|} < a_i - \sum_{j=0}^{i-1} < q_j, a_i > \cdot q_j, q_{i-1} > \\ &= \frac{1}{|u_i|}(< a_i, q_{i-1} > - \sum_{j=0}^{i-1} < q_j, a_i > \cdot < q_j, q_{i-1} >) \\ &= \frac{1}{|u_i|}(< a_i, q_{i-1} > - < q_{i-1}, a_i > \cdot < q_{i-1}, q_{i-1} >) \\ &= \frac{1}{|u_i|}(< a_i, q_{i-1} > - < a_i, q_{i-1} >) \\ &= 0 \end{aligned}$$

, which shows that q_i and q_{i-1} are orthogonal. Thus, $q_i, q_j (i \neq j)$ are orthogonal. Therefore, $< q_i, q_j >= 0$ when $i \neq j$.

Problem 3.2(b)(c)

Code:

```python
#!/usr/local/bin/python3.11
# -*- coding: UTF-8 -*-
# @Project : Computational_Physics
# @File    : Problem3_2.py
# @Author  : Albert Wang
# @Time    : 2024/4/3
# @Brief   : None

import numpy as np
import scipy


def qr_decomposition(A):
    """
    QR decomposition of a matrix A.

    :param A: a square matrix.
    :return: the result of the decomposition. i.e Q, R.
    """
    if A.shape[0] != A.shape[1]:
        raise ValueError('Input Matrix must be square')  # Error handle

    dimension = A.shape[0]
    A_ = np.copy(A)  # Separate the following operations from the original matrix A
    Q_ = np.zeros_like(A_)  # Initialize Q
    R_ = np.zeros_like(A_)  # Initialize R

    for i in range(dimension):
        # Gram-Schmidt Orthogonalization
        u_ = np.copy(A_[:, i])
        for j in range(i):
            u_ -= np.dot(Q_[:, j], A_[:, i]) * Q_[:, j]
        Q_[:, i] = u_ / np.linalg.norm(u_)

        # Get R
        R_[i, i] = np.linalg.norm(u_)
        for j in range(i):
            R_[j, i] = np.dot(Q_[:, j], A_[:, i])
```

```python
    return Q_, R_


def qr_eigens(A, max_iter):
    """
    Get eigenvalues and eigenvectors of a matrix A by solving QR decomposition.

    :param A: a square matrix.
    :param max_iter: max iteration number
    :return: An array contains the eigenvalues and a matrix contains eigenvectors.
    """
    dimension = A.shape[0]
    A_ = np.copy(A)  # Separate the following operations from the original matrix A
    eigenvector_ = np.zeros_like(A_)

    for i in range(int(max_iter)):
        Q_, R_ = qr_decomposition(A_)
        A_ = np.dot(R_, Q_)

        # Get the max off-diag element
        off_diagonal = []
        for j in range(1, dimension):
            off_diagonal.append(np.max(np.diag(A_, k=j)))

        # Stop the iteration when the off-diagonal elements are smaller than 1e-6
        if max(off_diagonal) < 1e-6:
            eigenvalue_ = np.diag(A_)
            # Calculate the corresponding eigenvector
            for j in range(dimension):
                K_ = A - eigenvalue_[j] * np.eye(dimension)
                null_space = scipy.linalg.null_space(K_)
                for k in range(null_space.shape[0]):
                    eigenvector_[k, j] = null_space[k, 0]
            return eigenvalue_, eigenvector_

    raise ValueError("QR iteration did not converge")  # Not converge handle


if __name__ == '__main__':
    M = np.array([[1, 4, 8, 4],
                  [4, 2, 3, 7],
                  [8, 3, 6, 9],
```

```
                 [4, 7, 9, 2]], float)
Q, R = qr_decomposition(M)
print("Q:", Q)
print("R:", R)
print(np.dot(Q, R))  # Verify

eigenvalue, eigenvector = qr_eigens(M, 1e6)
print("Eigenvalues:", eigenvalue)
print("Eigenvectors:", eigenvector)
```

Result:

QR decomposition by using `qr_decomposition()` :

```
Q: [[ 0.10153462  0.558463    0.80981107  0.1483773 ]
 [ 0.40613847 -0.10686638 -0.14147555  0.8964462 ]
 [ 0.81227693 -0.38092692  0.22995024 -0.37712564]
 [ 0.40613847  0.72910447 -0.5208777  -0.17928924]]
R: [[ 9.8488578   6.49821546 10.55960012 11.37187705]
 [ 0.          5.98106979  8.4234836  -0.484346  ]
 [ 0.          0.          2.74586406  3.27671222]
 [ 0.          0.          0.          3.11592335]]
```

Verify this result by multiplying Q and R:

```
[[1. 4. 8. 4.]
 [4. 2. 3. 7.]
 [8. 3. 6. 9.]
 [4. 7. 9. 2.]]
```

Solve eigenvalues and eigenvectors by using `qr_eignes` :

```
Eigenvalues: [21. -8. -3.  1.]
Eigenvectors: [[ 0.43151697 -0.38357064  0.77459667  0.25819889]
 [ 0.38357064  0.43151697  0.25819889 -0.77459667]
 [ 0.62330229  0.52740963 -0.25819889  0.51639778]
 [ 0.52740963 -0.62330229 -0.51639778 -0.25819889]]
```