THÔNG TIN SINH VIÊN

Mã số sinh viên: 23127447
Họ và tên: Nguyễn Thanh Owen
Lớp: 23CLC06
Email: ntowen23@clc.fitus.edu.vn

Phần 1: Tìm ma trận chéo P, P^{-1}, và ma trận đường chéo D, biết rằng $A = PDP^{-1}$

Thuật toán tìm ma trận chéo P, P^{-1}, và ma trận đường chéo D, biết rằng $A = PDP^{-1}$

Các bước thực hiện

Bước 1: Tìm trị riêng λ của ma trận A

Phương trình đặc trưng được viết là:

$$\det(A - \lambda I) = 0$$

Để giải phương trình này, cần tính định thức $\det(A - \lambda I)$ và tìm nghiệm λ. Có thể áp dụng các cách tính định thức tùy theo kích thước ma trận:

1. Ma trận 2×2:

$$A - \lambda I = \begin{bmatrix} a - \lambda & b \\ c & d - \lambda \end{bmatrix} \quad \Rightarrow \quad \det = (a - \lambda)(d - \lambda) - bc$$

Phương trình trên là đa thức bậc hai theo λ, có thể giải bằng công thức nghiệm bậc hai.

2. Ma trận 3×3:

a. Quy tắc Sarrus:

$$\begin{vmatrix} a & b & c \\ d & e & f \\ g & h & i \end{vmatrix} = aei + bfg + cdh - ceg - bdi - afh$$

b. Khai triển Laplace theo hàng hoặc cột:

Ví dụ khai triển theo hàng đầu tiên:

$$\det = a_{11}\det(M_{11}) - a_{12}\det(M_{12}) + a_{13}\det(M_{13})$$

với M_{1j} là ma trận con loại bỏ dòng 1 và cột j.

3. Ma trận $n \times n$ với $n > 3$:

Khai triển Laplace đệ quy:

$$\det(A) = \sum_{j=1}^{n} (-1)^{i+j} a_{ij} \det(M_{ij})$$

$$(\text{với } i \text{ là 1 dòng cố định thứ } i \text{ bất kỳ})$$

Bước 2: Tìm vector riêng ứng với từng trị riêng

Sau khi tìm được một trị riêng λ_i của ma trận A, ta tiến hành tìm vector riêng tương ứng bằng cách giải hệ:

$$(A - \lambda_i I)\vec{v} = 0$$

trong đó:

- I là ma trận đơn vị cùng kích thước với A
- $\vec{v} \neq 0$ là vector cột cần tìm (vector riêng ứng với λ_i)

Giải thích:

- Ma trận $(A - \lambda_i I)$ là một ma trận mới phụ thuộc vào trị riêng λ_i.

- Phương trình $(A - \lambda_i I)\vec{v} = 0$ là một **hệ phương trình tuyến tính đồng nhất**.
- Nghiệm $\mathbf{v}$ của hệ này là tất cả các vector bị ánh xạ thành vector 0 khi nhân với ma trận $(A - \lambda_i I)$.
- Vì hệ đồng nhất luôn có nghiệm tầm thường $\vec{v} = 0$ nhưng **ta chỉ quan tâm đến nghiệm không tầm thường**, tức là vector $\vec{v} \neq 0$.

1. Tính $A - \lambda_i I$

2. Giải hệ phương trình tuyến tính:

$$(A - \lambda_i I)\vec{v} = \vec{0}$$

3. Dùng phương pháp khử Gauss hoặc biến đổi sơ cấp để đưa hệ về dạng bậc thang rút gọn.

4. Biểu diễn nghiệm tổng quát dưới dạng:

$$\vec{v} = \alpha_1 \vec{v}_1 + \alpha_2 \vec{v}_2 + \cdots$$

   Trong đó:

   - $\vec{v}_1, \vec{v}_2, \ldots$ là các nghiệm cơ sở của hệ
   - $\alpha_1, \alpha_2, \ldots$ là các tham số tự do (hằng số vô hướng)

Bước 3: Kiểm tra ma trận có thể chéo hóa hay không

Kiểm tra tổng số vector riêng tìm được có bằng kích thước ma trận n hay không.

- Nếu tổng số vector riêng đủ n, ma trận có thể chéo hóa.
- Nếu không đủ, ma trận không thể chéo hóa.

Bước 4: Xây dựng ma trận P

$$P = [\vec{v}_1 \quad \vec{v}_2 \quad \ldots \quad \vec{v}_n]$$

Bước 5: Xây dựng ma trận D

$$D = \text{diag}(\lambda_1, \lambda_2, \ldots, \lambda_n)$$

Bước 6: Tính ma trận nghịch đảo P^{-1}

Dùng thuật toán Gauss–Jordan để tìm P^{-1} thỏa mãn:

$$P^{-1}P = I$$

Bước 7: Kiểm tra kết quả $A = PDP^{-1}$

Các hàm hỗ trợ

In [120…

```python
# Hàm in ma trận
def printMatrix(A):
    for row in A:
        print(row)


# Hàm nhân ma trận với một số
def scalarMultipleMatrix(k, matrix):
    result = [[k * element for element in row] for row in matrix]
    return result

# Hàm nhân hai ma trận
def matmul(A, B):
    rows_A = len(A)
    cols_A = len(A[0])
    rows_B = len(B)
    cols_B = len(B[0])


    # Khởi tạo ma trận kết quả với 0
    result = [[0] * cols_B for _ in range(rows_A)]

    for i in range(rows_A):
        for j in range(cols_B):
            s = 0
            for k in range(cols_A):
                s += A[i][k] * B[k][j]
            result[i][j] = s
    return result
```

```python
# Hàm tổng hai ma trận
def sumMatrix(matrix1, matrix2):
    if len(matrix1) != len(matrix2) or len(matrix1) == 0 or len(matrix2) == 0:
        return None
    for row1, row2 in zip(matrix1, matrix2):
        if len(row1) != len(row2):
            return None

    result = [
            [matrix1[i][j] + matrix2[i][j] for j in range(len(matrix1[0]))] for i in range(len(matrix1))
        ]
    return result

# Hàm hiệu hai ma trận
def subtractMatrix(matrix1, matrix2):
    if len(matrix1) != len(matrix2) or len(matrix1) == 0 or len(matrix2) == 0:
        return None
    for row1, row2 in zip(matrix1, matrix2):
        if len(row1) != len(row2):
            return None

    result = [
            [matrix1[i][j] - matrix2[i][j] for j in range(len(matrix1[0]))] for i in range(len(matrix1))
        ]
    return result


# Hàm tính A - lamb * I
def subtract_lambda_identity(A, lamb):
    n = len(A)
    return [[A[i][j] - (lamb if i == j else 0) for j in range(n)] for i in range(n)]
```

Bước 1 Tìm trị riêng λ của ma trận A

In [120…

```python
from sympy import expand
# Hàm tính định thức
def determinant(matrix):
    n = len(matrix)
    if n == 1:
```

```
        return matrix[0][0]
    if n == 2:
        return matrix[0][0] * matrix[1][1] - matrix[0][1] * matrix[1][0]

    det = 0
    for j in range(n):
        sub_matrix = [row[:j] + row[j + 1:] for row in matrix[1:]]
        det += (-1) ** j * matrix[0][j] * determinant(sub_matrix)
    det = expand(det)
    return det
```

Tìm trị riêng với ma trận 2x2

In [120…

```
def remove_duplicate_eigenvalues(eigenvalues, tolerance=1e-6):
    unique = []
    for val in eigenvalues:
        # So sánh từng trị riêng đã thêm với trị riêng hiện tại, nếu khác biệt nhỏ hơn sai số cho phép thì coi là trù
        if not any(abs(val - u) < tolerance for u in unique):
            unique.append(val)
    return unique

# Tìm trị riêng với ma trận 2x2
def find_eigenvalues_2x2(A):
    a, b = A[0]
    c, d = A[1]
    B = a * (-1) + d * (-1)
    det = a * d - b * c
    delta = B ** 2 - 4 * det
    if delta < 0:
        return []

    sqrt_delta = delta ** 0.5
    lamb1 = (B * (-1) + sqrt_delta) / 2
    lamb2 = (B * (-1) - sqrt_delta) / 2
    eigenValues = [lamb1, lamb2]

    return eigenValues
```

Tìm trị riêng với ma trận 3x3

In [120…

```python
# Tìm trị riêng với ma trận 3x3
import math

def cbrt(x):
    #Căn bậc 3 với số âm
    if x >= 0:
        return x ** (1/3)
    else:
        return -(-x) ** (1/3)

def find_eigenvalues_3x3(A):
    a, b, c = A[0]
    d, e, f = A[1]
    g, h, i = A[2]

    # Hệ số của phương trình đặc trưng: λ^3 + aλ^2 + bλ + c = 0
    A3 = -1
    A2 = -(a + e + i)
    A1 = (a*e + a*i + e*i) - (b*d + c*g + f*h)
    A0 = -(a*e*i + b*f*g + c*d*h) + c*e*g + b*d*i + a*f*h

    # Dạng chuẩn: x^3 + a x^2 + b x + c = 0
    a_ = -A2 / A3
    b_ = -A1 / A3
    c_ = -A0 / A3

    p = b_ - a_**2 / 3
    q = 2 * a_**3 / 27 - a_ * b_ / 3 + c_
    delta = (q**2) / 4 + (p**3) / 27

    eigenvalues = []

    if abs(delta) < 1e-9:
        delta = 0

    if delta > 0:
        # 1 nghiệm thực
        sqrt_delta = math.sqrt(delta)
        u = cbrt(-q/2 + sqrt_delta)
        v = cbrt(-q/2 - sqrt_delta)
        root = u + v - a_ / 3
```

```
        eigenvalues.append(round(root, 4))

    elif delta == 0:
        if abs(q) < 1e-9:
            root = -a_ / 3
            eigenvalues.extend([round(root, 4)] * 3)
        else:
            u = cbrt(-q / 2)
            root1 = 2*u - a_/3
            root2 = -u - a_/3
            eigenvalues.extend([round(root1, 4), round(root2, 4), round(root2, 4)])
    else:
        # 3 nghiệm thực
        r = (-p**3 / 27) ** (1/2)
        phi = math.acos(-q / (2 * r))
        t = 2 * (-p / 3) ** (1/2)
        for k in range(3):
            angle = (phi + 2 * math.pi * k) / 3
            root = t * math.cos(angle) - a_ / 3
            eigenvalues.append(round(root, 4))

    return eigenvalues
```

Tìm trị riêng cho ma trận $n \times n$ với $n > 3$

In [120…

```
from sympy import Symbol, Poly

def poly_eval(coeffs, x):
    result = 0
    for coeff in coeffs:
        result = result * x + coeff
    return result

def poly_derivative(coeffs):
    n = len(coeffs)
    return [coeffs[i] * (n - i - 1) for i in range(n - 1)]

def newton_raphson(poly_coeffs, initial_guess, tolerance=1e-10, max_iterations=1000):
    deriv_coeffs = poly_derivative(poly_coeffs)
    x = initial_guess
    for _ in range(max_iterations):
```

```
        fx = poly_eval(poly_coeffs, x)
        f_prime_x = poly_eval(deriv_coeffs, x)
        if abs(f_prime_x) < 1e-14:
            break
        x_new = x - fx / f_prime_x
        if abs(x_new - x) < tolerance:
            return x_new
        x = x_new
    return x

def poly_divide(poly, root):
    new_coeffs = []
    temp = 0
    for coeff in poly:
        temp = temp * root + coeff
        new_coeffs.append(temp)
    new_coeffs.pop()  # Bỏ phần dư
    return new_coeffs

def find_all_roots(poly_coeffs, tolerance=1e-10):
    roots = []
    poly = poly_coeffs[:]
    degree = len(poly) - 1
    while degree > 0:
        guess = 0.0
        root = newton_raphson(poly, guess, tolerance)
        roots.append(root)
        poly = poly_divide(poly, root)
        degree = len(poly) - 1
    return roots

def find_eigenvalues_nxn(expr, var):
    round_digits= 4
    poly = Poly(expr, var)
    coeffs = [float(c) for c in poly.all_coeffs()]
    roots = find_all_roots(coeffs)
    return [round(r, round_digits) for r in roots]
```

Hàm tìm trị riêng cho tất cả trường hợp

In [ ]:
```python
def solve_eigenvalues(A):
    n = len(A)

    if n == 2:
        eigenvals = find_eigenvalues_2x2(A)
    elif n == 3:
        eigenvals = find_eigenvalues_3x3(A)
    else:
        lam = Symbol('λ')
        B = subtract_lambda_identity(A, lam)
        det_expr = determinant(B)
        eigenvals = find_eigenvalues_nxn(det_expr, lam)

    return remove_duplicate_eigenvalues(eigenvals)
```

Bước 2: Tìm vector riêng ứng với từng trị riêng

Hàm đưa ma trận về dạng bậc thang rút gọn

In [ ]:
```python
def swap_rows(A, row1, row2):
    A[row1], A[row2] = A[row2], A[row1]

def Gauss_elimination(A, rowLength, colLength):
    numRow = rowLength
    numCol = colLength

    leftMost = 0 # cột trái nhất không chứa toàn số 0
    for row in range(numRow):

        # Xác định cột trái nhất không chứa toàn số 0
        while leftMost < numCol:
            check = True
            for i in range(row, numRow):
                if abs(A[i][leftMost]) > 1e-10:
                    check = False
                    break
            if check == True:
                leftMost += 1
            else:
```

```
                break

        if leftMost >= numCol:
            A = to_RREF(A)
            return A

        # Kiểm tra phần tử hiện tại của dòng đầu tiên có = 0 hay không
            # Nếu có thì swap với 1 dòng khác
        if A[row][leftMost] == 0:
            for i in range(row + 1, numRow):
                if A[i][leftMost] != 0:
                    swap_rows(A, row, i)
                    break

        # Nhân dòng "row" nghịch đảo với 1/A[row][leftMost] để chuyển phần tử đầu tiên thành 1
        pivot = A[row][leftMost]
        for j in range(leftMost, numCol):
            A[row][j] /= pivot

        # Cộng bội số thích hợp của dòng đầu cho từng dòng bên dưới để biến các số hạng bên dưới thành 0
        for i in range(row + 1, numRow):
            factor = A[i][leftMost]
            for j in range(leftMost, numCol):
                A[i][j] = A[i][j] - A[row][j] * factor

        # Tăng chỉ số leftmost và lặp lại quá trình
        leftMost += 1

    A = to_RREF(A)
    return A

# Hàm chuyển ma trận bậc thang về ma trận bậc thang rút gọn
def to_RREF(A):
    numRow = len(A)
    if numRow == 0:
        return A
    numCol = len(A[0])

    for i in range(numRow - 1, -1, -1):
        # Tìm pivot trong hàng i (phần tử xấp xỉ 1)
        pivot_col = -1
        for j in range(numCol):
```

```
            if abs(A[i][j] - 1.0) < 1e-10:
                pivot_col = j
                break
        if pivot_col == -1:
            continue

        # Loại bỏ các phần tử phía trên pivot về 0
        for k in range(i - 1, -1, -1):
            factor = A[k][pivot_col]
            for j in range(pivot_col, numCol):
                A[k][j] -= factor * A[i][j]
    return A
```

Hàm tìm các vector riêng tương ứng với trị riêng của nó

In [120…

```
def find_eigenvector(A, lam):

    n = len(A)
    M = subtract_lambda_identity(A, lam)

    # Thêm cột hệ số 0 để giải hệ đồng nhất
    for row in M:
        row.append(0.0)

    rref = Gauss_elimination(M, len(M), len(M[0]))

    pivots = [-1] * n
    for i in range(n):
        for j in range(n):
            if abs(rref[i][j]) > 1e-8:
                pivots[i] = j
                break

    pivot_cols = set([j for j in pivots if j != -1])
    free_vars = [j for j in range(n) if j not in pivot_cols]

    if not free_vars:
        return []

    eigenvectors = []
    for fv in free_vars:
```

```
        vec = [0.0] * n
        vec[fv] = 1.0
        for i in reversed(range(n)):
            col = pivots[i]
            if col == -1:
                continue
            # Tính hệ số phụ thuộc của pivot theo free variable
            total = 0.0
            for j in free_vars:
                total += rref[i][j] * vec[j]
            if total == 0.0:
                vec[col] = 0.0
            else:
                vec[col] = round(-total, 4)  # làm tròn 4 chữ số sau dấu phẩy

        eigenvectors.append(vec)

    return eigenvectors



def get_all_eigenvectors(A, eigenvalues):
    eigenvalues = remove_duplicate_eigenvalues(eigenvalues)
    all_vectors = []
    for lam in eigenvalues:
        eigenvectors = find_eigenvector([row[:] for row in A], lam)
        all_vectors.append( eigenvectors)
    return all_vectors
```

Bước 3: Kiểm tra ma trận có thể chéo hóa được hay không

In [120…

```
def can_diagonalize(eigenvectors, n):
    # Số vector riêng thực sự tìm được
    count_vectors = 0
    for row in eigenvectors:
        count_vectors += len(row)

    # Nếu số vector riêng đủ n thì ma trận có thể chéo hóa
    if count_vectors == n:
        return True
```

```
    else:
        return False
```

Bước 4: Xây dựng ma trận P

In [121…

```
def build_P_matrix(eigenVectors, n):
    P = [[0.0 for _ in range(n)] for _ in range(n)]

    col_index = 0
    for vectors in eigenVectors:
        for vec in vectors:
            if col_index >= n:
                break  # Không vượt quá số cột
            for row in range(n):
                P[row][col_index] = vec[row]
            col_index += 1
    return P
```

Bước 5: Xây dựng ma trận D

In [121…

```
def build_D_matrix(eigenvalues, n):
    D = [[0.0 for _ in range(n)] for _ in range(n)]

    i = 0
    for lam in eigenvalues:
        if i >= n:
            break
        D[i][i] = lam  # Gán trị riêng vào đường chéo
        i += 1
    return D
```

Bước 6: Tính ma trận nghịch đảo P^{-1}

In [121…

```
def identity_matrix(n):
    return [[1 if i == j else 0 for j in range(n)] for i in range(n)]

def combine_matrix_horizontal(A, B):
    return [row_A + row_B for row_A, row_B in zip(A, B)]
```

```
def extract_inverse(augmented_matrix, split_index):
    return [row[split_index:] for row in augmented_matrix]

def build_P1_matrix(P):
    n = len(P)
    I = identity_matrix(n)
    augmented = combine_matrix_horizontal(P, I)
    Gauss_elimination(augmented, n, 2*n)
    inverse = extract_inverse(augmented, n)
    return inverse
```

Bước 7: Kiểm tra lại kết quả $A = PDP^{-1}$

In [121…

```
def multiply_three_matrices(A, B, C):
    AB = matmul(A, B)
    ABC = matmul(AB, C)
    result = [[round(elem, 4) for elem in row] for row in ABC]
    return result
```

Ví dụ với ma trận A (4x4)

In [121…

```
A = [
    [4, -9, 6, 12],
    [0, -1, 4, 6],
    [2, -11, 8, 16],
    [-1, 3, 0, -1],
     ]


lamb = Symbol('λ')

# Tìm ma trận A - λI
B = subtract_lambda_identity(A, lamb)
print('A - λI = ')
printMatrix(B)

# Kết quả định thức
print('\n|A - λI| = ', end='')
```

```
det = determinant(B)
print(det, end=' = 0\n')

# Tìm trị riêng của A
eigenValues = solve_eigenvalues(A)
print("Các trị riêng tìm được:", end='')
print(eigenValues)

# Tìm các vector riêng tương ứng với trị riêng
eigenVectors = get_all_eigenvectors(A, eigenValues)
print("\nCác vector riêng:")
for i, row in enumerate(eigenVectors):
    print(f"Vector riêng ứng với trị riêng {eigenValues[i]:.2f}: {row}")

# Kiểm tra xem A có chéo hóa được hay không
if can_diagonalize(eigenVectors, len(A)): # Nếu A chéo hóa được
    #Xây dựng ma trận P
    P = build_P_matrix(eigenVectors, len(A))
    print('\nKết quả của ma trận P là: \n P = ')
    printMatrix(P)

    #Xây dựng ma trận D
    D = build_D_matrix(eigenValues, len(A))
    print('\nKết quả của ma trận D là: \n D = ')
    printMatrix(D)

    #Xây dựng ma trận nghịch đảo của P (P^-1)
    P1 = build_P1_matrix(P)
    print('\nKết quả của ma trận P^-1 là: \n P^-1 = ')
    printMatrix(P1)

else: #A không chéo hóa được
    print('Ma trận đã cho:\n A =')
    printMatrix(A)
    print('Không chéo hóa được.')

#Kiểm tra kết quả
A_reconstructed = multiply_three_matrices(P, D, P1)
print('\nKiểm tra kết quả: \n A * D * P^-1 = ')
printMatrix(A_reconstructed)
```

```
A - λI =
[4 - λ, -9, 6, 12]
[0, -λ - 1, 4, 6]
[2, -11, 8 - λ, 16]
[-1, 3, 0, -λ - 1]

|A - λI| = λ**4 - 10*λ**3 + 35*λ**2 - 50*λ + 24 = 0
Các trị riêng tìm được:[1.0, 2.0, 3.0, 4.0]

Các vector riêng:
Vector riêng ứng với trị riêng 1.00: [[1.0, 1.0, -1.0, 1.0]]
Vector riêng ứng với trị riêng 2.00: [[3.0, 2.0, 0.0, 1.0]]
Vector riêng ứng với trị riêng 3.00: [[3.0, 1.0, 1.0, 0.0]]
Vector riêng ứng với trị riêng 4.00: [[1.0, 2.0, 1.0, 1.0]]

Kết quả của ma trận P là:
 P =
[1.0, 3.0, 3.0, 1.0]
[1.0, 2.0, 1.0, 2.0]
[-1.0, 0.0, 1.0, 1.0]
[1.0, 1.0, 0.0, 1.0]

Kết quả của ma trận D là:
 D =
[1.0, 0.0, 0.0, 0.0]
[0.0, 2.0, 0.0, 0.0]
[0.0, 0.0, 3.0, 0.0]
[0.0, 0.0, 0.0, 4.0]

Kết quả của ma trận P^-1 là:
 P^-1 =
[1.0, -5.0, 2.0, 7.0]
[-1.0, 6.0, -3.0, -8.0]
[1.0, -4.0, 2.0, 5.0]
[0.0, -1.0, 1.0, 2.0]

Kiểm tra kết quả:
 A * D * P^-1 =
[4.0, -9.0, 6.0, 12.0]
[0.0, -1.0, 4.0, 6.0]
[2.0, -11.0, 8.0, 16.0]
[-1.0, 3.0, 0.0, -1.0]
```

Giải thích thư viện đã sử dụng

1. `Symbol` (từ `sympy`)

- Tạo biến đại số λ để biểu diễn biểu thức như $\lambda^2 + 3\lambda + 2$.

2. `Poly` (từ `sympy`)

Lấy hệ số của từng bậc trong biểu thức đa thức đại số:

- Với đa thức $2\lambda^3 - 4\lambda + 5$, dùng `Poly` để trích hệ số là $[2, 0, -4, 5]$.

3. `expand` (từ `sympy`)

Hàm `expand` giúp triển khai biểu thức $(\lambda + 2) \times (\lambda + 3)$ thành $\lambda^2 + 5\lambda + 6$.

4. `math` (thư viện chuẩn Python)

Thư viện này hỗ trợ các hàm toán học cơ bản:

- Tính căn bậc hai: `math.sqrt(9)` trả về 3.
- Tính hàm cosin: `math.cos(math.pi)` trả về -1.
- Tính căn bậc ba theo cách thủ công bằng hàm lũy thừa.

Giải thích các hàm đã viết

Hàm `printMatrix(A)`

In ra từng hàng của ma trận `A`. Mỗi hàng được in trên một dòng để dễ quan sát kết quả.

Hàm `scalarMultipleMatrix(k, matrix)`

Thực hiện phép nhân vô hướng: nhân tất cả phần tử trong ma trận `matrix` với một số thực `k`.
Trả về ma trận mới sau khi nhân.

Hàm `sumMatrix(matrix1, matrix2)`

Tính tổng của hai ma trận `matrix1` và `matrix2` nếu chúng cùng kích thước.
Trả về ma trận tổng, hoặc `None` nếu kích thước không hợp lệ.

Hàm `subtractMatrix(matrix1, matrix2)`

Tính hiệu của hai ma trận `matrix1 - matrix2` nếu chúng cùng kích thước.
Trả về ma trận hiệu, hoặc `None` nếu kích thước không hợp lệ.

Hàm `subtract_lambda_identity(A, lamb)`

Tính hiệu giữa ma trận `A` và tích `λI`, trong đó `I` là ma trận đơn vị cùng kích thước với `A`.
Chức năng chính: thực hiện phép `A - λI` bằng cách trừ `λ` khỏi các phần tử trên đường chéo chính của `A`.

Hàm `determinant(matrix)`

Tính định thức của một ma trận vuông bằng phương pháp khai triển Laplace.

- Với ma trận 1×1 hoặc 2×2, dùng công thức trực tiếp.
- Với ma trận lớn hơn, đệ quy bằng cách bỏ từng cột ở hàng đầu.

Hàm `remove_duplicate_eigenvalues(eigenvalues)`

Loại bỏ các trị riêng trùng nhau trong danh sách `eigenvalues`.

Hàm `find_eigenvalues_2x2(A)`

Tìm trị riêng của ma trận 2×2 dựa vào nghiệm của phương trình đặc trưng:

$$\lambda^2 - \text{tr}(A)\lambda + \det(A) = 0$$

Trả về 2 nghiệm thực (hoặc rỗng nếu $\Delta < 0$).

Hàm `find_eigenvalues_3x3(A)`

Tìm trị riêng của ma trận 3×3 bằng cách giải phương trình bậc ba đặc trưng:

$$\lambda^3 + a\lambda^2 + b\lambda + c = 0$$

- Tự động phân biệt giữa các trường hợp $\Delta > 0$, $\Delta = 0$, hoặc $\Delta < 0$ để xác định số nghiệm thực.

Hàm `cbrt(x)`

Trả về căn bậc 3 của số thực `x`, xử lý đúng cho cả số âm.

Hàm `swap_rows(A, row1, row2)`

Hoán đổi hai dòng trong ma trận `A`.

- Dùng để xử lý trường hợp phần tử pivot bằng 0, cần đưa dòng khác lên để tiếp tục khử Gauss.

Hàm `Gauss_elimination(A, rowLength, colLength)`

Biến đổi ma trận `A` về dạng **ma trận bậc thang (REF)** rồi **bậc thang rút gọn (RREF)** bằng phép biến đổi sơ cấp:

- Tìm cột trái nhất không toàn 0 (`leftMost`).
- Đưa phần tử đầu dòng về 1 bằng cách chia cả dòng cho pivot.
- Khử các phần tử dưới pivot về 0.
- Cuối cùng gọi `to_RREF(A)` để đưa ma trận về dạng rút gọn.

Hàm `to_RREF(A)`

Biến đổi ma trận bậc thang về dạng **bậc thang rút gọn (Reduced Row Echelon Form)**:

- Với mỗi dòng từ dưới lên, tìm pivot (phần tử có giá trị gần 1).
- Dùng phép biến đổi sơ cấp để đưa các phần tử phía trên pivot về 0.

Hàm `find_eigenvector(A, lam)`

Tìm các **vector riêng (eigenvectors)** ứng với trị riêng `λ`:

1. Tính ma trận $A - \lambda I$.
2. Giải hệ phương trình đồng nhất $(A - \lambda I)\vec{v} = 0$ bằng phương pháp Gauss–Jordan.
3. Xác định các **ẩn tự do** trong hệ và tạo các vector riêng tương ứng.

Hàm `get_all_eigenvectors(A, eigenvalues)`

Trả về danh sách tất cả các vector riêng ứng với các trị riêng đã cho:

- Loại bỏ trị riêng trùng.
- Với mỗi trị riêng, gọi `find_eigenvector` để tìm các vector riêng.
- Kết quả là danh sách các danh sách vector riêng cho từng trị riêng.

Hàm `can_diagonalize(eigenvectors, n)`

Kiểm tra xem một ma trận có **chéo hóa được** hay không:

- Đếm tổng số **vector riêng** tìm được từ danh sách `eigenvectors`.
- Nếu số lượng vector riêng bằng đúng kích thước ma trận `n`, thì ma trận có thể chéo hóa được.

Hàm `build_P_matrix(eigenVectors, n)`

Tạo ma trận P từ các vector riêng:

- Gộp tất cả các vector riêng theo cột vào một ma trận kích thước $n \times n$.
- P có dạng:

$$P = [\vec{v}_1 \quad \vec{v}_2 \quad \dots \quad \vec{v}_n]$$

Hàm `build_D_matrix(eigenvalues, n)`

Tạo ma trận **chéo** D chứa các **trị riêng** trên đường chéo chính:

- D có dạng:

$$D = \text{diag}(\lambda_1, \lambda_2, \dots, \lambda_n)$$

Hàm `identity_matrix(n)`

Tạo ma trận đơn vị I_n kích thước $n \times n$:

- Dùng để hỗ trợ quá trình nghịch đảo ma trận.

Hàm `combine_matrix_horizontal(A, B)`

Nối hai ma trận `A` và `B` theo chiều ngang (cột):

- Dùng để tạo ma trận bổ sung $(P|I)$ khi tìm P^{-1} bằng phép Gauss–Jordan.

Hàm `extract_inverse(augmented_matrix, split_index)`

Trích xuất phần bên phải (từ `split_index` trở đi) của ma trận ghép:

- Sau khi biến đổi $(P|I)$ thành $(I|P^{-1})$, lấy phần bên phải để thu được P^{-1}.

Hàm `build_P1_matrix(P)`

Tìm **ma trận nghịch đảo** P^{-1} bằng phương pháp Gauss–Jordan:

1. Nối ma trận P với ma trận đơn vị I: $(P|I)$.
2. Biến đổi về dạng $(I|P^{-1})$ bằng khử Gauss.
3. Trích phần bên phải để lấy P^{-1}.

Hàm `poly_eval(coeffs, x)`

Tính giá trị đa thức tại điểm `x` dựa trên hệ số `coeffs` theo quy tắc Horner:

- `coeffs` là danh sách các hệ số từ bậc cao đến bậc thấp.
- Trả về giá trị đa thức tại `x`.

Hàm `poly_derivative(coeffs)`

Tính đa thức đạo hàm của đa thức cho trước dưới dạng hệ số:

- Trả về danh sách hệ số đa thức đạo hàm.

Hàm `newton_raphson(poly_coeffs, initial_guess, tolerance=1e-10, max_iterations=1000)`

Giải phương trình đa thức bằng phương pháp Newton-Raphson để tìm nghiệm gần với `initial_guess`.

- Dừng khi sai số nhỏ hơn `tolerance` hoặc vượt quá `max_iterations`.
- Trả về nghiệm tìm được.

Hàm `poly_divide(poly, root)`

Chia đa thức `poly` cho đa thức bậc nhất `(x - root)` sử dụng quy tắc Horner (synthetic division).

- Trả về hệ số đa thức sau phép chia (bỏ phần dư).

Hàm `find_all_roots(poly_coeffs, tolerance=1e-10)`

Tìm tất cả các nghiệm thực của đa thức bậc `n` dựa trên thuật toán Newton-Raphson và chia đa thức lần lượt cho các nghiệm tìm được.

- Trả về danh sách các nghiệm.

Hàm `find_eigenvalues_nxn(expr, var)`

Tìm trị riêng (nghiệm đa thức đặc trưng) cho ma trận kích thước lớn (bất kỳ `n x n`) bằng thư viện SymPy:

- Dựng đa thức đặc trưng `det(A - λI)` dưới dạng biểu thức.
- Chuyển biểu thức sang đa thức và lấy hệ số.
- Dùng `find_all_roots` để tìm nghiệm.
- Làm tròn trị riêng đến 4 chữ số thập phân.

Hàm `solve_eigenvalues(A)`

Giải phương trình đặc trưng để tìm trị riêng của ma trận vuông `A`:

- Với ma trận 2x2, gọi hàm `find_eigenvalues_2x2`.

- Với ma trận 3x3, gọi hàm `find_eigenvalues_3x3`.
- Với các kích thước lớn hơn giải đa thức bằng phương pháp Newton-Raphson.
- Loại bỏ trị riêng trùng nhau và trả về danh sách trị riêng.

Phần 2: Mở rộng

Ứng dụng thư viện NumPy trong chéo hóa ma trận

Thư viện **NumPy** cung cấp các hàm mạnh mẽ để xử lý đại số tuyến tính, trong đó có các hàm giúp tính trị riêng (eigenvalues), vector riêng (eigenvectors), và các phép toán ma trận cần thiết cho việc chéo hóa ma trận.

Các hàm chính liên quan đến chéo hóa:

- **`numpy.linalg.eig`**
  - Mục đích: Tính toán trị riêng và vector riêng của ma trận vuông.
  - Cách dùng:
    ```
    eigenvalues, eigenvectors = np.linalg.eig(A)
    ```
  - Giải thích:
    - `eigenvalues`: mảng chứa các trị riêng λ_i.
    - `eigenvectors`: ma trận mà mỗi cột là vector riêng tương ứng với trị riêng cùng vị trí trong `eigenvalues`.
- **`numpy.diag`**
  - Mục đích: Tạo ma trận đường chéo từ một danh sách các phần tử hoặc lấy đường chéo của ma trận.
  - Cách dùng:
    ```
    D = np.diag(eigenvalues)
    ```
  - Giải thích: Tạo ma trận D là ma trận đường chéo với các trị riêng trên đường chéo chính:

$$D = \text{diag}(\lambda_1, \lambda_2, \ldots, \lambda_n)$$

- **`numpy.linalg.inv`**
  - Mục đích: Tính ma trận nghịch đảo của ma trận vuông.

- Cách dùng:
  `P^{-1} = np.linalg.inv(P)`
- Giải thích: Tính P^{-1}, nghịch đảo của ma trận vector riêng P.

Tổng kết quy trình

1. Sử dụng `np.linalg.eig` để lấy trị riêng và vector riêng:

   $A\vec{v}_i = \lambda_i \vec{v}_i$

2. Dùng `np.diag` để tạo ma trận đường chéo D:

   $D = \text{diag}(\lambda_1, \lambda_2, \ldots, \lambda_n)$

3. Dùng `np.linalg.inv` để tính ma trận nghịch đảo P^{-1}.

4. Áp dụng công thức chéo hóa:

   $A = PDP^{-1}$

In [121…

```
import numpy as np

A2 = [
    [4, -9, 6, 12],
    [0, -1, 4, 6],
    [2, -11, 8, 16],
    [-1, 3, 0, -1],
]

# Tính trị riêng (eigenvalues) và vector riêng (eigenvectors)
eigenValues2, eigenVectors2 = np.linalg.eig(A2)

# Ma trận P là các vector riêng (cột)
P2 = eigenVectors2

# Ma trận D là đường chéo với các trị riêng
D2 = np.diag(eigenValues2)

# P^-1 là nghịch đảo của P
```

```
P3 = np.linalg.inv(P2)

# Tính lại A
A2_reconstructed = P2 @ D2 @ P3
```

In [121…

```
print("Các trị riêng:")
print(eigenValues2)

print("\nCác vector riêng:")
for i, row in enumerate(eigenVectors2):
    print(f"Vector riêng ứng với trị riêng {eigenValues2[i]:.2f}: {row}")

print('\nKết quả của ma trận P là: \n P = ')
printMatrix(P2)

print('\nKết quả của ma trận D là: \n D = ')
printMatrix(D2)

print('\nKết quả của ma trận P^-1 là: \n P^-1 = ')
printMatrix(P3)

print('\nKiểm tra kết quả: \n A * D * P^-1 = ')
printMatrix(A2_reconstructed)
```

```
Các trị riêng:
[1. 2. 3. 4.]

Các vector riêng:
Vector riêng ứng với trị riêng 1.00: [-0.5         0.80178373  0.90453403  0.37796447]
Vector riêng ứng với trị riêng 2.00: [-0.5         0.53452248  0.30151134  0.75592895]
Vector riêng ứng với trị riêng 3.00: [ 5.00000000e-01 -1.83499310e-14  3.01511345e-01  3.77964473e-01]
Vector riêng ứng với trị riêng 4.00: [-5.00000000e-01  2.67261242e-01 -4.61395833e-15  3.77964473e-01]

Kết quả của ma trận P là:
 P =
[-0.5         0.80178373  0.90453403  0.37796447]
[-0.5         0.53452248  0.30151134  0.75592895]
[ 5.00000000e-01 -1.83499310e-14  3.01511345e-01  3.77964473e-01]
[-5.00000000e-01  2.67261242e-01 -4.61395833e-15  3.77964473e-01]

Kết quả của ma trận D là:
 D =
[1. 0. 0. 0.]
[0. 2. 0. 0.]
[0. 0. 3. 0.]
[0. 0. 0. 4.]

Kết quả của ma trận P^-1 là:
 P^-1 =
[ -2.  10.  -4. -14.]
[ -3.74165739  22.44994432 -11.22497216 -29.93325909]
[  3.31662479 -13.26649916   6.63324958  16.58312395]
[ 1.07703714e-15 -2.64575131e+00  2.64575131e+00  5.29150262e+00]

Kiểm tra kết quả:
 A * D * P^-1 =
[ 4. -9.  6. 12.]
[-3.3148784e-15 -1.0000000e+00  4.0000000e+00  6.0000000e+00]
[  2. -11.   8.  16.]
[-1.00000000e+00  3.00000000e+00  5.25991751e-15 -1.00000000e+00]
```

So sánh chi tiết: Bài làm thủ công vs. Sử dụng thư viện NumPy

Tiêu chí	Bài làm thủ công	Sử dụng NumPy (`np.linalg.eig`)
Cách tiếp cận	Tự xây dựng thuật toán từ đầu, giải hệ tuyến tính bằng Gauss	Gọi hàm có sẵn trả về trị riêng và vector riêng
Tìm trị riêng	Tính đa thức đặc trưng, sau đó giải tìm trị riêng	Thuật toán số học tối ưu (QR, Schur decomposition)
Tìm vector riêng	Giải hệ ((A - λI)v = 0) bằng phép biến đổi Gauss-Jordan, xử lý trực tiếp	Tự động giải và chuẩn hóa vector riêng
Làm tròn kết quả	Làm tròn 4 chữ số ở vector riêng khi tính phụ thuộc	Mặc định sử dụng độ chính xác kép (khoảng 15 chữ số)
Xử lý trị riêng trùng	Loại trùng bằng hàm tự viết `remove_duplicate_eigenvalues`	Xử lý tự động bên trong hàm
Tính ma trận (P^{-1})	Khử Gauss-Jordan trên ma trận mở rộng P	Dùng `np.linalg.inv` với thuật toán LU decomposition
Tái tạo lại ma trận (A)	Tính thủ công $(A = PDP^{-1})$ bằng hàm nhân ma trận tự viết	Dùng phép nhân ma trận NumPy: `A = P @ D @ P_inv`
Dễ tùy chỉnh / can thiệp	Dễ tùy chỉnh, thêm kiểm tra, xử lý chi tiết	Khó can thiệp sâu, chỉ dùng hàm có sẵn
Hiệu quả số học	Dễ sai số nếu làm tròn quá nhiều hoặc cài đặt thuật toán chưa chuẩn	Ổn định và chính xác cao, sai số máy rất nhỏ
Vector riêng thu được	Không chuẩn hóa, dễ kiểm tra bằng tay	Chuẩn hóa độ dài 1, khó kiểm tra trực tiếp

Kết luận

- **Cách thủ công** thích hợp cho học tập và giảng dạy từng bước rõ ràng.
- **Cách dùng NumPy** phù hợp cho ứng dụng thực tế, khi cần tốc độ và độ chính xác cao.

Một số ứng dụng của chéo hóa ma trận

1. Phân tích thành phần chính (PCA) trong học máy

PCA (Principal Component Analysis) là kỹ thuật giảm chiều dữ liệu sử dụng chéo hóa ma trận để xác định các thành phần chính. Giúp giảm độ phức tạp của dữ liệu trong khi vẫn giữ lại thông tin quan trọng, hỗ trợ phân loại và nhận dạng mẫu.
Nguồn: numberanalytics.com

2. Nén ảnh và xử lý tín hiệu

Chéo hóa ma trận được sử dụng để nén ảnh bằng cách giữ lại các trị riêng quan trọng nhất, giúp giảm kích thước tệp mà không làm mất nhiều chất lượng ảnh.
Nguồn: numberanalytics.com

3. Phân tích dao động trong kỹ thuật

Chéo hóa ma trận khối lượng và độ cứng giúp xác định tần số tự nhiên và dạng mode của hệ thống dao động. Đây là bước quan trọng trong thiết kế và kiểm tra độ bền cấu trúc kỹ thuật.
Nguồn: fastercapital.com

4. Phân tích chuỗi Markov

Ma trận chuyển trạng thái trong chuỗi Markov được chéo hóa để tìm phân phối xác suất ổn định, giúp dự đoán hành vi dài hạn của hệ thống xác suất.
Nguồn: numberanalytics.com