Zadanie Numeryczne 03

Mateusz Wojtyna

1. Wstęp

Należało wyznaczyć dwie największe (na moduł) wartości własne macierzy

$$\mathbf{A} = \begin{bmatrix} \frac{19}{12} & \frac{13}{12} & \frac{5}{6} & \frac{5}{6} & \frac{13}{12} & -\frac{17}{12} \\ \frac{13}{12} & \frac{13}{12} & \frac{5}{6} & \frac{5}{6} & -\frac{11}{12} & \frac{13}{12} \\ \frac{5}{6} & \frac{5}{6} & \frac{5}{6} & -\frac{1}{6} & \frac{5}{6} & \frac{5}{6} \\ \frac{5}{6} & \frac{5}{6} & -\frac{1}{6} & \frac{5}{6} & \frac{5}{6} & \frac{5}{6} \\ \frac{13}{12} & -\frac{11}{12} & \frac{5}{6} & \frac{5}{6} & \frac{13}{12} & \frac{13}{12} \\ -\frac{17}{12} & \frac{13}{12} & \frac{5}{6} & \frac{5}{6} & \frac{13}{12} & \frac{19}{12} \end{bmatrix}.$$

Wykorzystano metodę potęgową do ich iteracyjnego wyznaczenia.

2. Opis

2.1. Największa wartość własna

Można zauważyć, że $\mathbf{A} \in \mathbb{R}^{n\times n}$ jest symetryczna, więc wiadomo że jest diagonalizowalna, ma rzeczywiste wartości własne i jej unormowane wektory własne tworzą bazę ortonormalną w $\mathbb{R}^n$. Oznaczając poszczególne wektory w tej bazie przez $\mathbf{e}_i$, mamy $\mathbf{A}\mathbf{e}_i = \lambda_i \mathbf{e}_i$. Weźmy teraz dowolny wektor $\mathbf{y}$ należący do tej bazy, $\mathbf{y} = \sum_{i=1}^{n} \alpha_i \mathbf{e}_i$. Następnie kolejno obliczamy

$$\begin{aligned} \mathbf{A}\mathbf{y} &= \mathbf{A}\sum_{i=1}^{n} \alpha_i \mathbf{e}_i = \sum_{i=1}^{n} \alpha_i \mathbf{A}\mathbf{e}_i = \sum_{i=1}^{n} \alpha_i \lambda_i \mathbf{e}_i \\ \mathbf{A}^2\mathbf{y} &= \mathbf{A}^2\sum_{i=1}^{n} \alpha_i \mathbf{e}_i = \sum_{i=1}^{n} \alpha_i \lambda_i^2 \mathbf{e}_i \\ &\vdots \\ \mathbf{A}^k\mathbf{y} &= \sum_{i=1}^{n} \alpha_i \lambda_i^k \mathbf{e}_i \end{aligned}$$

Widać, że dla $k \gg 1$, czynnik sumy z największym λ_i będzie dominował nad innymi, więc prawa strona równości będzie dążyć do wektora proporcjonalnego do wektora własnego **największej** wartości własnej.

Zatem k-ty krok iteracji wygląda następująco (początkowo wybieramy dowolny unormowany wektor $\mathbf{y}_1$):

$$\begin{aligned} \mathbf{z}_k &\leftarrow \mathbf{A}\mathbf{y}_k \\ \mathbf{y}_{k+1} &\leftarrow \frac{\mathbf{z}_k}{||\mathbf{z}_k||} \end{aligned}$$

Algorytm zatrzymujemy gdy $||\mathbf{y}_{k+1} - \mathbf{y}_k|| \leq \epsilon$ lub gdy przekroczono limit kroków.

2.2. Druga największa wartość własna

Na początku oznaczmy wektor własny odpowiadający największej wartości własnej przez $\mathbf{e}_1$. Jeżeli chcemy otrzymać **drugą co do wielkości** wartość własną λ_2, musimy w jakiś sposób wyzerować czynnik α_1 przy $\mathbf{e}_1$ (w definicji $\mathbf{y}$). Można to zrobić przez wymuszenie ortogonalności $\mathbf{y}$ względem $\mathbf{e}_1$, ponieważ dla wektorów niezerowych[1] ortogonalność implikuje liniową niezależność.

Zatem w tym przypadku algorytm jest lekko zmieniony; początkowo wybieramy dowolny unormowany **oraz ortogonalny** względem $\mathbf{e}_1$ wektor $\mathbf{y}_1$:

$$\begin{aligned} \mathbf{z}_k &\leftarrow \mathbf{A}\mathbf{y}_k \\ \mathbf{z}_k &\leftarrow \mathbf{z}_k - \underbrace{\mathbf{e}_1(\mathbf{z}_k \cdot \mathbf{e}_1)}_{\text{proj}_{\mathbf{e}_1}\mathbf{z}_k} \\ \mathbf{y}_{k+1} &\leftarrow \frac{\mathbf{z}_k}{||\mathbf{z}_k||} \end{aligned}$$

Dodatkowo w każdym kroku ortogonalizujemy $\mathbf{z}_k$ względem $\mathbf{e}_1$. Jest to konieczne ze względu na błędy numeryczne, które mogą spowodować, że kolejne $\mathbf{z}_k$ nie będą ortogonalne do $\mathbf{e}_1$.

[1] $\mathbf{e}_1$ jest niezerowe z definicji wektora własnego, a $\mathbf{y}$ musi mieć normę równą 1.

3. Kod

Program napisano w Pythonie z użyciem pakietu *NumPy*.

```python
import numpy as np
from numpy.typing import NDArray

# Dla przejrzystości kodu
array = NDArray[np.float64]
vector = NDArray[np.float64]
matrix = NDArray[np.float64]

N = 6


def power_method(A: matrix, eps: float, limit: int):
    # wylosuj wektor i unormuj
    e1 = np.random.rand(N)
    norm_y = np.linalg.norm(e1)
    e1 /= norm_y

    z = np.zeros(N, dtype=np.float64)
    for i in range(limit):
        z = A @ e1
        e1_new = z / np.linalg.norm(z)

        if np.linalg.norm(e1_new - e1) <= eps:
            return e1, np.linalg.norm(z), i + 1

        e1 = e1_new

    return e1, np.linalg.norm(z), limit


def power_method_second(A: matrix, e1: vector, eps: float, limit: int):
    # wylosuj wektor i unormuj
    e2 = np.random.rand(N)
    norm_y = np.linalg.norm(e2)
    e2 /= norm_y

    # ortogonalizuj względem e1 i unormuj
    e2 -= e1 * np.dot(e1, e2)

    # zabezpieczenie przed wylosowaniem wektora prawie identycznego do e1
    while np.linalg.norm(e2) < 1e-14:
        e2 = np.random.rand(N)
        e2 -= e1 * np.dot(e1, e2)
    e2 /= np.linalg.norm(e2)

    z = np.zeros(N, dtype=np.float64)
    for i in range(limit):
        z = A @ e2
        z -= e1 * np.dot(e1, z)
        e2_new = z / np.linalg.norm(z)
```

```
        if np.linalg.norm(e2_new - e2) <= eps:
            return e2, np.linalg.norm(z), i + 1

        e2 = e2_new

    return e2, np.linalg.norm(z), limit


def vec_error(v1, v2):
    return min(np.linalg.norm(v1 - v2), np.linalg.norm(v1 + v2))


def main():
    np.set_printoptions(linewidth=np.inf)  # pyright: ignore[reportArgumentType]

    A = np.array(
        [
            [19 / 12, 13 / 12, 5 / 6, 5 / 6, 13 / 12, -17 / 12],
            [13 / 12, 13 / 12, 5 / 6, 5 / 6, -11 / 12, 13 / 12],
            [5 / 6, 5 / 6, 5 / 6, -1 / 6, 5 / 6, 5 / 6],
            [5 / 6, 5 / 6, -1 / 6, 5 / 6, 5 / 6, 5 / 6],
            [13 / 12, -11 / 12, 5 / 6, 5 / 6, 13 / 12, 13 / 12],
            [-17 / 12, 13 / 12, 5 / 6, 5 / 6, 13 / 12, 19 / 12],
        ],
        dtype=np.float64,
    )

    e1, lam1, steps1 = power_method(A, 1e-12, 100)
    print(f"{lam1}, {e1} after {steps1} steps")

    e2, lam2, steps2 = power_method_second(A, e1, 1e-12, 100)
    print(f"{lam2}, {e2} after {steps2} steps")

    expected = np.linalg.eig(A)
    print(f"\n|lamda1 - expected lambda1|: {abs(lam1 - expected.eigenvalues[0])}")
    print(f"||e1 - expected e1||: {vec_error(e1, expected.eigenvectors[:, 0])}")
    print(f"\n|lamda2 - expected lambda2|: {abs(lam2 - expected.eigenvalues[1])}")
    print(f"||e2 - expected e2||: {vec_error(e2, expected.eigenvectors[:,1])}")


if __name__ == "__main__":
    main()
```

4. Wyniki

Otrzymano zbieżność λ_1 średnio po 88 krokach, λ_2 po 76.

$$\lambda_1 = 4, \qquad \mathbf{e}_1 = \begin{bmatrix} 0.40824829 \\ 0.40824829 \\ 0.40824829 \\ 0.40824829 \\ 0.40824829 \\ 0.40824829 \end{bmatrix}$$

$$\lambda_2 = 3, \qquad \mathbf{e}_2 = \begin{bmatrix} -0.707106781 \\ 2.20420136 \times 10^{-12} \\ 1.33704316 \times 10^{-12} \\ 1.33704316 \times 10^{-12} \\ 1.04250099 \times 10^{-12} \\ 0.707106781 \end{bmatrix}$$

5. Podsumowanie

W przeprowadzonych obliczeniach zastosowano metodę potęgową do wyznaczenia dwóch największych (co do modułu) wartości własnych macierzy $\mathbf{A}$. Dzięki własnościom macierzy symetrycznych metoda potęgowa okazała się stabilna i szybko zbieżna.

Poprawnie wyznaczono wartości własne $\lambda_1 = 4$, $\lambda_2 = 3$ oraz ich wektory własne poprzez zastosowanie metody potęgowej dla λ_1 oraz zmodyfikowanej wersji z ortogonalizacją względem $\mathbf{e}_1$ dla λ_2. Uzyskane wyniki są zgodne z rozwiązaniem otrzymanym przez *NumPy*, co potwierdza poprawność implementacji.