Zadanie Numeryczne 04

Mateusz Wojtyna

1. Wstęp

Należało sprowadzić macierz symetryczną

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

do postaci trójdiagonalnej, a następnie znaleźć jej wartości własne.

2. Opis

2.1. Transformacja Householdera

Macierz symetryczna $\mathbf{A} \in \mathbb{R}^{n \times n}$ zostaje przekształcona do macierzy trójdiagonalnej $\mathbf{T}$ za pomocą transformacji Householdera:

$$\mathbf{T} = \mathbf{Q}\mathbf{A}\mathbf{Q}^T$$

Krok k algorytmu iteracyjnego wygląda następująco ($\mathbf{A}_1 = \mathbf{A}$):

$$\begin{aligned} \mathbf{x} &= \mathbf{A}_k[k+1:n, k] && \text{(elementy kolumny pod diagonalą)} \\ \mathbf{u} &= \mathbf{x} + \operatorname{sgn}(\mathbf{x}[1]) \cdot ||\mathbf{x}|| \cdot \mathbf{e}_1 \\ \mathbf{H}_k &= \mathbb{I} - 2\frac{\mathbf{u}\mathbf{u}^T}{||\mathbf{u}||^2} && \text{(macierz Householdera)} \\ \mathbf{Q}_k &= \begin{bmatrix} \mathbb{I}^{k \times k} & 0 \\ 0 & \mathbf{H}_k \end{bmatrix} \\ \mathbf{A}_{k+1} &= \mathbf{Q}_k \mathbf{A}_k \mathbf{Q}_k^T \end{aligned}$$

Po $n-2$ krokach otrzymujemy macierz trójdiagonalną:

$$\mathbf{T} = \mathbf{A}_{n-1} = \mathbf{Q}_{n-2}\mathbf{A}_{n-2}\mathbf{Q}_{n-2}^T = \mathbf{Q}_{n-2} \dots \mathbf{Q}_1 \mathbf{A}_1 \mathbf{Q}_1^T \dots \mathbf{Q}_{n-2}^T = \mathbf{Q}\mathbf{A}\mathbf{Q}^T$$

2.2. Algorytm QR

Jeżeli macierz $\mathbf{A} \in \mathbb{R}^{n \times n}$ posiada faktoryzację QR, to macierz $\mathbf{A}' = \mathbf{RQ} = \mathbf{Q}^T\mathbf{AQ}$ jest ortogonalną transformacją podobieństwa macierzy $\mathbf{A}$.

Iterując, algorytm wygląda następująco ($\mathbf{A}_1 = \mathbf{A}$):

$$\begin{aligned}
\mathbf{A}_1 &= \mathbf{Q}_1\mathbf{R}_1 \\
\mathbf{A}_2 &= \mathbf{R}_1\mathbf{Q}_1 = \mathbf{Q}_2\mathbf{R}_2 \\
\mathbf{A}_3 &= \mathbf{R}_2\mathbf{Q}_2 = \mathbf{Q}_3\mathbf{R}_3 \\
&\vdots
\end{aligned}$$

Iteracja zbiega się do macierzy trójkątnej górnej, w której na diagonali znajdują się wartości własne. W kodzie zatrzymujemy iterację jeżeli $||\text{diag}(\mathbf{A}_{k+1}) - \text{diag}(\mathbf{A}_k)|| \leq \epsilon$.

Dla macierzy trójdiagonalnych symetrycznych faktoryzację QR występującą w każdym kroku algorytmu można zoptymalizować do czasu liniowego za pomocą obrotów Givensa:

$$\mathbf{A}_k = \mathbf{R}_{k-1}\mathbf{Q}_{k-1} = \mathbf{G}_{n-1} \dots \mathbf{G}_1\mathbf{A}_{k-1}\mathbf{G}_1^T \dots \mathbf{G}_{n-1}^T$$

3. Kod

Program napisano w Pythonie z użyciem pakietu *NumPy*.

```
# pyright: reportConstantRedefinition=false

import numpy as np
from numpy.typing import NDArray

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


def householder_vector(x: vector) -> vector:
    u = x.copy()
    u[0] = x[0] + np.sign(x[0]) * np.linalg.norm(x)
    return u


def householder_tridiagonalize(A: matrix) -> matrix:
    n = len(A[0])
    T = A.copy()

    for k in range(n - 2):
        x = T[k + 1 :, k]
        u = householder_vector(x)
        H = np.eye(n - k - 1) - 2.0 * (np.outer(u, u) / (np.linalg.norm(u) **
            2))

        Q = np.eye(n)
```

```
        Q[k + 1 :, k + 1 :] = H

        T = Q @ T @ Q.transpose()

    return T


# Działa in place
def givens_row(T: matrix, i: int, c: float, s: float):
    n = len(T[0])
    left = max(0, i - 1)
    right = min(n - 1, i + 2)

    for col in range(left, right + 1):
        # Należy zapisać te elementy wcześniej, ponieważ się zmieniają
        A_i_j = T[i, col]
        A_ip1_j = T[i + 1, col]

        T[i, col] = c * A_i_j + s * A_ip1_j
        T[i + 1, col] = -s * A_i_j + c * A_ip1_j


# Działa in place
def givens_col(T: matrix, i: int, c: float, s: float):
    n = len(T[0])

    top = max(0, i - 1)
    bottom = min(n - 1, i + 2)

    for row in range(top, bottom + 1):
        # Należy zapisać te elementy wcześniej, ponieważ się zmieniają
        A_i_j = T[row, i]
        A_i_jp1 = T[row, i + 1]

        T[row, i] = c * A_i_j + s * A_i_jp1
        T[row, i + 1] = -s * A_i_j + c * A_i_jp1


def qr_algorithm(T: matrix, limit: int, eps: float) -> tuple[array, int]:
    n = len(T[0])
    diag = np.diag(T)
    T = T.copy()

    for k in range(limit):
        diag_old = np.diag(T).copy()
        for i in range(n - 1):
            # Należy obliczyć te elementy wcześniej, bo givens_row() je
               zmienia
            # a givens_col() musi mieć takie same jak givens_row()
            a = T[i, i]
            b = T[i + 1, i]
            r = np.hypot(a, b)
            c = a / r
            s = b / r
```

```
            givens_row(T, i, c, s)
            givens_col(T, i, c, s)
        diag = np.diag(T)

        if np.linalg.norm(diag - diag_old) <= eps:
            return diag, k + 1

    return diag, limit


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

    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,
    )

    T = householder_tridiagonalize(A)
    eigenvalues, steps = qr_algorithm(T, limit=1000, eps=1e-12)

    # Sortujemy malejąco, żeby łatwo można było porównać
    eigenvalues = -np.sort(-eigenvalues)
    expected = -np.sort(-np.linalg.eig(A).eigenvalues)
    print(f"Result after {steps} steps:", eigenvalues)
    print("Expected:", expected)
    print(
        "Error:",
        np.linalg.norm(expected - eigenvalues),
    )


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

4. Wyniki

Po 74 krokach otrzymano zbieżność ($\epsilon = 10^{-12}$):

$$\lambda_1 = 4,\ \lambda_2 = 3,\ \lambda_3 = 2,\ \lambda_4 = 1,\ \lambda_5 = -1,\ \lambda_6 = -2$$

Różnica względem rozwiązania policzonego z *NumPy* była rzędu 10^{-13}, czyli w praktyce 0.

5. Podsumowanie

- Zastosowano transformację Householdera aby sprowadzić macierz symetryczną do postaci trójdiagonalnej w czasie $\mathcal{O}(n^3)$.
- Zoptymalizowano algorytm QR dzięki postaci trójdiagonalnej macierzy używając obrotów Givensa, dzięki czemu jeden krok algorytmu QR wykonuje się w czasie $\mathcal{O}(n)$.