Metody Numeryczne - Zadania II

Matusz Kamiński

Listopad 2025

1 Wstęp

Celem ćwiczenia jest numeryczne rozwiązanie układu równań liniowych

$$Ax = e,$$

w którym $A \in R^{128\times 128}$ ma postać siedmiodiagonalną (pasmową o szerokości $p = 4$), a $e = (1, \dots, 1)^\top$. Struktura A jest następująca:

$$A_{ii} = 4, \qquad A_{i,i\pm1} = 1, \qquad A_{i,i\pm4} = 1,$$

zaś elementy wykraczające poza zakres indeksów są równe 0. Macierz jest symetryczna i dodatnio określona, a jej przechowywanie oraz mnożenie $y \leftarrow Ax$ mogą być zrealizowane w czasie i pamięci $O(n)$ dzięki reprezentacji pasmowej (wektory przekątnych D, L_1, L_4).

2 Macierz układu

Macierz A ma postać

$$A = \begin{bmatrix} 4 & 1 & 0 & 0 & 1 & 0 & 0 & 0 & \cdots & 0 \\ 1 & 4 & 1 & 0 & 0 & 1 & 0 & 0 & \cdots & 0 \\ 0 & 1 & 4 & 1 & 0 & 0 & 1 & 0 & \cdots & 0 \\ 0 & 0 & 1 & 4 & 1 & 0 & 0 & 1 & \cdots & 0 \\ 1 & 0 & 0 & 1 & 4 & 1 & 0 & 0 & \cdots & 0 \\ 0 & 1 & 0 & 0 & 1 & 4 & 1 & 0 & \cdots & 0 \\ 0 & 0 & 1 & 0 & 0 & 1 & 4 & 1 & \cdots & 0 \\ \vdots & \vdots & \vdots & \vdots & \vdots & \vdots & \vdots & \vdots & \ddots & \vdots \\ 0 & 0 & 0 & 0 & 0 & 0 & 0 & 1 & \cdots & 4 \end{bmatrix}.$$

W opisie indeksowym elementy niezerowe występują tylko na diagonalach $i, i\pm1$ oraz $i, i\pm4$:

$$A_{i,i} = 4, \qquad A_{i,i\pm1} = 1, \qquad A_{i,i\pm4} = 1.$$

Dzięki temu macierz jest przechowywana w sposób oszczędny za pomocą trzech wektorów o długości 128:

$$D = (4, 4, \dots, 4), \quad L_1 = (1, 1, \dots), \quad L_4 = (1, 1, \dots).$$

3 Metoda Gaussa–Seidela

Macierz A zawiera niezerowe elementy na diagonalach oddalonych o 1 oraz 4 po obu stronach przekątnej. Współczynniki tych diagonal zapisywane są w wektorach L_1 oraz L_4, odpowiednio dla przesunięć ± 1 i ± 4.

Dla ogólnego przypadku iteracja metody Gaussa–Seidela przyjmuje postać:

$$x_i^{(k+1)} = \frac{1}{D[i]} \left(e[i] - L_1[i-1]\, x_{i-1}^{(k+1)} - L_1[i]\, x_{i+1}^{(k)} - L_4[i-4]\, x_{i-4}^{(k+1)} - L_4[i]\, x_{i+4}^{(k)} \right),$$

W naszym zadaniu:

$$\forall i \qquad L_1[i] = 1, \qquad L_4[i] = 1 \qquad e[i] = 1$$

dlatego wzór iteracyjny przyjmuje postać

$$x_i^{(k+1)} = \frac{1}{4} \left(1 - x_{i-1}^{(k+1)} - x_{i+1}^{(k)} - x_{i-4}^{(k+1)} - x_{i+4}^{(k)} \right),$$

z pominięciem tych składników, dla których odpowiedni indeks nie należy do zakresu $0, \ldots, n-1$.

Ze względu na różne zależności brzegowe, implementacja została rozbita na kilka pętli, z których każda obejmuje zakres indeksów posiadających taki sam układ sąsiadów:

- $i = 0$ — pierwszy element ma sąsiadów tylko po prawej stronie,
- $i = 1, 2, 3$ — elementy przy brzegu posiadają $i-1$ i $i+1$, lecz brak $i-4$,
- $i = 4, \ldots, n-5$ — pełna zależność od pięciu punktów siatki,
- $i = n-4, n-3, n-2$ — brak sąsiada w pozycji $i+4$,
- $i = n-1$ — ostatni element korzysta wyłącznie z $i-1$ oraz $i-4$.

4 Metoda gradientów sprzężonych

Metoda gradientów sprzężonych (CG) Używamy jej iteracyjną wersję do rozwiązywania układów liniowych

$$Ax = e,$$

gdzie macierz A jest symetryczna oraz dodatnio określona. Metoda minimalizuje błąd i generuje kolejne kierunki przeszukiwania $p^{(k)}$ tak, aby były one wzajemnie sprzężone względem macierzy A (tzn. $p^{(i)} A p^{(j)} = 0$ dla $i \neq j$).

Algorytm rozpoczyna się od wyboru przybliżenia początkowego, np.

$$x^{(0)} = 0.$$

Następnie obliczamy:

$$r^{(0)} = e - Ax^{(0)}, \qquad p^{(0)} = r^{(0)}.$$

Dla $k = 0, 1, 2, \ldots$ wykonujemy kolejne iteracje:

$$\alpha_k = \frac{r^{(k)} r^{(k)}}{p^{(k)} A p^{(k)}},$$

$$x^{(k+1)} = x^{(k)} + \alpha_k \, p^{(k)},$$

$$r^{(k+1)} = r^{(k)} - \alpha_k \, A p^{(k)},$$

$$\beta_k = \frac{r^{(k+1)} r^{(k+1)}}{r^{(k)} r^{(k)}},$$

$$p^{(k+1)} = r^{(k+1)} + \beta_k \, p^{(k)}.$$

Iteracje zatrzymujemy, gdy spełniony jest warunek zbieżności:

$$\|r^{(k)}\| < \varepsilon.$$

W implementacji wykorzystano funkcję `multiply_A`, która realizuje działanie Ax bez budowania macierzy A, korzystając wyłącznie z jej pasmowej struktury. Dzięki temu każda iteracja metody CG ma koszt obliczeniowy rzędu $O(n)$.

Zastosowano implementację z wykorzystaniem operacji mnożenia przez A zoptymalizowanych pod strukturę pasmową macierzy (wykorzystano funkcję `multiply_A`, działającą w czasie $O(n)$).

5 Implementacja

Wszystkie obliczenia zostały wykonane w języku Python z wykorzystaniem pakietów `NumPy`, `Matplotlib` oraz `Numba`

```
import numpy as np
import matplotlib.pyplot as plt
from numba import njit
from numpy.typing import NDArray

vector = NDArray[np.float64]

@njit(fastmath=True)
def gauss_seidel_step(x: vector, x_old: vector, D: vector):
    n = len(x)

    # 1) i = 0
    i = 0
    x[i] = (1.0 - x_old[i+1] - x_old[i + 4]) / D[i]

    # 2) i = 1, 2, 3
```

```
    for i in range(1, 4):
        x[i] = (1.0 - x[i-1] - x_old[i+1] - x_old[i+4]) / D[i]

    # 3) i = 4 .. n-5  (pełna formuła)
    for i in range(4, n - 4):
        x[i] = (1.0 - x[i-1] - x_old[i+1] - x[i-4] - x_old[i+4]) /
            D[i]

    # 4) i = n-4, n-3, n-2
    for i in range(n - 4, n - 1):
        x[i] = (1.0 - x[i-1] - x[i-4] - x_old[i+1]) / D[i]

    # 5) i = n-1
    i = n - 1
    x[i] = (1.0 - x[i-1] - x[i-4]) / D[i]

def conjugate_gradient(x0:vector, D: vector, iterations: int,
    tolerance: np.float64) -> vector:
    x = x0.copy()
    r = 1.0 - multiply_A(x, D)
    p = r.copy()
    rTr_old = np.dot(r, r)

    history = [] # lista norm

    for k in range(iterations):
        Ap = multiply_A(p, D)
        alpha = rTr_old / np.dot(p, Ap)

        x_new = x + alpha * p
        r_new = r - alpha * Ap

        history.append(np.linalg.norm(x_new - x))

        if np.linalg.norm(r) < tolerance:
            x = x_new
            break

        rTr_new = np.dot(r_new, r_new)
        beta = rTr_new / rTr_old
        p = r_new + beta * p

        x, r, rTr_old = x_new, r_new, rTr_new
    return x, history

@njit(fastmath=True)
def multiply_A(x: vector, D: vector) -> vector:
    y = D * x
    n = x.shape[0]

    for i in range(n - 4):
        y[i + 1] += x[i]
        y[i] += x[i + 1]

        y[i + 4] += x[i]
        y[i] += x[i + 4]
```

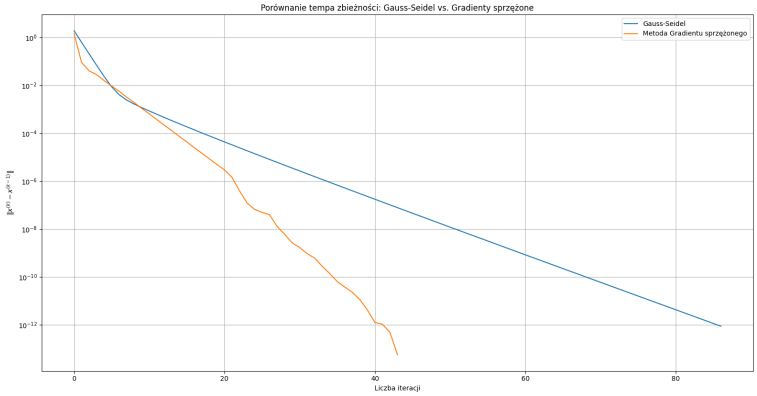
```
    for i in range(n - 4 , n - 1):
        y[i + 1] += x[i]
        y[i] += x[i + 1]

    return y

def main() -> None:
    n = 128

    ITERATION_LIMIT = 100
    TOLERANCE = 1e-12

    D: vector = np.full(n, 4.0)
    # --- Niepotrzebne wektory ----
    # L1: vector = np.ones(n - 1, dtype=np.float64)
    # L4: vector = np.ones(n - 4, dtype=np.float64)
    # e: vector = np.ones(n, dtype=np.float64)

    # ---- Gauss-Seidel ----
    x: vector = np.zeros(n, dtype=np.float64)
    x_old: vector = np.zeros(n, dtype=np.float64)
    gauss_history = []

    for iterations in range(1, ITERATION_LIMIT + 1):
        x_old[:] = x
        gauss_seidel_step(x, x_old, D)
        gauss_history.append(np.linalg.norm(x - x_old))

        if np.linalg.norm(x - x_old) < TOLERANCE:
            print(f"Gauss-Seidel Zbieżny po {iterations} iteracjach
                .")
            break

    # ---- Conjugate Gradient ----
    x0 = np.zeros(n, dtype=np.float64)
    x, cg_history = conjugate_gradient(x0, D, ITERATION_LIMIT,
        TOLERANCE)
    print(f"CG zakończone po {len(cg_history)} iteracjach.")


    # ---- Rysowanie wykresu ----
    plt.figure()
    plt.plot(gauss_history, label="Gauss-Seidel")
    plt.plot(cg_history, label="Metoda Gradientu sprzężonego")
    plt.yscale("log")
    plt.xlabel("Liczba iteracji")
    plt.ylabel(r"$\|x^{(k)} - x^{(k-1)}\|$")
    plt.title("Porównanie tempa zbieżności: Gauss-Seidel vs.
        Gradienty sprzężone")
    plt.legend()
    plt.grid(True)
    plt.show()

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

6 Porównanie złożoności

- Gauss–Seidel: koszt jednej iteracji $O(n)$, liczba iteracji $\approx 80-90$.
- Gradienty sprzężone: koszt jednej iteracji $O(n)$, liczba iteracji $\approx 40-50$.
- Rozkład Choleskiego dla macierzy pasmowej o szerokości $p = 4$:

  $O(np^2) = O(16n)$, rozwiązanie układu po rozkładzie: $2{*}O(7n) = O(14n)$.

GS: $\approx 80 \cdot O(n)$, CG: $\approx 40 \cdot O(n)$, Cholesky: $\approx 30 \cdot O(n)$.



Rysunek 1: Zbieżność metod w kolejnych iteracjach.

7 Wnioski

Dzięki wykorzystaniu struktury pasmowej macierzy możliwe było wykonywanie operacji Ax w czasie liniowym, co znacząco obniżyło koszt obliczeń. Metoda Gaussa–Seidela okazała się zbiegać wolniej i wymagała większej liczby iteracji. Metoda gradientów sprzężonych osiągnęła dokładne rozwiązanie znacznie szybciej, przy zachowaniu tego samego kosztu pojedynczej iteracji. Rozkład Choleskiego jest najbardziej efektywny przy wielokrotnym rozwiązywaniu tego samego układu, natomiast metoda gradientów sprzężonych stanowi rozwiązanie najbardziej praktyczne dla pojedynczego przebiegu.