

# Metody Numeryczne - Zadanie II

Matusz Kamiński

Listopad 2025

## 1 Wstęp

Celem ćwiczenia jest numeryczne rozwiązywanie układu równań liniowych

$$\mathbf{Ax} = \mathbf{e},$$

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

$$A_{ii} = 4, \quad A_{i,i\pm 1} = 1, \quad A_{i,i\pm 4} = 1,$$

zaś elementy wykraczające poza zakres indeksów są równe 0. Macierz jest symetryczna i dodatnio określona, a jej przechowywanie oraz mnożenie  $\mathbf{y} \leftarrow \mathbf{Ax}$  mogą być zrealizowane w czasie i pamięci  $O(n)$  dzięki reprezentacji pasmowej (wektory przekątnych  $\mathbf{D}, \mathbf{L}_1, \mathbf{L}_4$ ).

## 2 Macierz układu

Macierz  $\mathbf{A}$  ma postać

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

W opisie indeksowym elementy niezerowe występują tylko na diagonalach  $i, i \pm 1$  oraz  $i, i \pm 4$ :

$$A_{i,i} = 4, \quad A_{i,i\pm 1} = 1, \quad A_{i,i\pm 4} = 1.$$

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

$$\mathbf{D} = (4, 4, \dots, 4), \quad \mathbf{L}_1 = (1, 1, \dots), \quad \mathbf{L}_4 = (1, 1, \dots).$$

### 3 Metoda Gaussa–Seidela

Macierz  $\mathbf{A}$  zawiera niezerowe elementy na diagonalach oddalonych o 1 oraz 4 po obu stronach przekątnej. Współczynniki tych diagonal zapisywane są w wektorach  $\mathbf{L}_1$  oraz  $\mathbf{L}_4$ , odpowiednio dla przesunięć  $\pm 1$  i  $\pm 4$ .

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:

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

Dlatego wzór iteracyjny upraszcza się do:

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

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

### 4 Metoda gradientów sprzężonych

Metoda gradientów sprzężonych (CG) służy do rozwiązywania układów liniowych

$$\mathbf{Ax} = \mathbf{e},$$

gdzie  $\mathbf{A}$  jest symetryczna oraz dodatnio określona. Minimalizuje błąd i generuje kolejne kierunki przeszukiwania  $\mathbf{p}^{(k)}$  tak, aby były one wzajemnie sprzężone względem  $\mathbf{A}$ .

$$\begin{aligned} \mathbf{x}^{(0)} &= 0, \\ \mathbf{r}^{(0)} &= \mathbf{e} - \mathbf{Ax}^{(0)}, \\ \mathbf{p}^{(0)} &= \mathbf{r}^{(0)}. \end{aligned}$$

Dla  $k = 0, 1, 2, \dots$ :

$$\begin{aligned} \alpha_k &= \frac{\mathbf{r}^{(k)} \mathbf{r}^{(k)}}{\mathbf{p}^{(k)} \mathbf{Ap}^{(k)}}, \\ \mathbf{x}^{(k+1)} &= \mathbf{x}^{(k)} + \alpha_k \mathbf{p}^{(k)}, \\ \mathbf{r}^{(k+1)} &= \mathbf{r}^{(k)} - \alpha_k \mathbf{Ap}^{(k)}, \\ \beta_k &= \frac{\mathbf{r}^{(k+1)} \mathbf{r}^{(k+1)}}{\mathbf{r}^{(k)} \mathbf{r}^{(k)}}, \\ \mathbf{p}^{(k+1)} &= \mathbf{r}^{(k+1)} + \beta_k \mathbf{p}^{(k)}. \end{aligned}$$

Iteracje kończymy, gdy  $\|\mathbf{r}^{(k)}\| < \varepsilon$ .

Każda iteracja ma koszt  $\mathbf{O}(n)$  dzięki funkcji `multiply_A`, która wykorzystuje pasmową strukturę  $\mathbf{A}$ .

## 5 Implementacja

Wszystkie obliczenia wykonano w języku Python z użyciem 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()

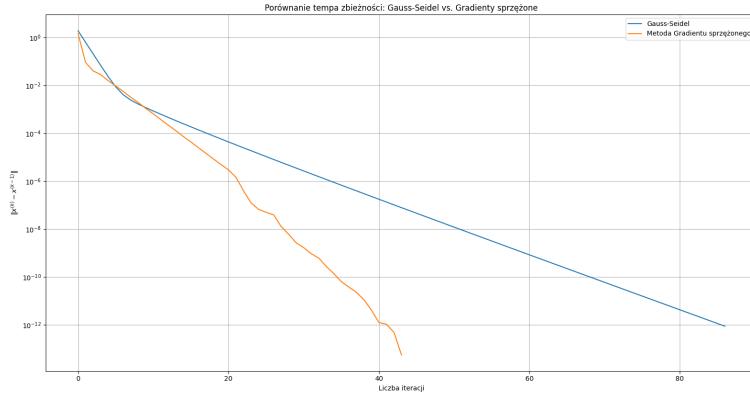
```

## 6 Porównanie złożoności

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

$$\mathbf{O}(\mathbf{n}\mathbf{p}^2) = \mathbf{O}(16\mathbf{n}), \quad \text{rozwiązanie układu: } 2 \times \mathbf{O}(7\mathbf{n}) = \mathbf{O}(14\mathbf{n}).$$

$$\text{GS: } \approx 80 \mathbf{O}(\mathbf{n}), \quad \text{CG: } \approx 40 \mathbf{O}(\mathbf{n}), \quad \text{Cholesky: } \approx 30 \mathbf{O}(\mathbf{n}).$$



Rysunek 1: Zbieżność metod w kolejnych iteracjach.

## 7 Wnioski

Wykorzystanie struktury pasmowej macierzy pozwoliło zredukować koszt pamięciowy i czasowy z  $\mathbf{O}(\mathbf{n}^2)$  do  $\mathbf{O}(\mathbf{n})$ . Metoda Gaussa–Seidela wymaga większej liczby iteracji, natomiast metoda gradientów sprzężonych zbiega szybciej przy tym samym koszcie pojedynczego kroku. Rozkład Choleskiego jest najefektywniejszy przy wielokrotnym rozwiązywaniu tego samego układu, podczas gdy CG jest najbardziej praktyczna przy pojedynczym obliczeniu.