

Zadanie Numeryczne 02

Mateusz Wojtyna

1. Wstęp

Należało rozwiązać układ równań $\mathbf{Ax} = \mathbf{e}$, gdzie $\mathbf{A} \in \mathbb{R}^{128 \times 128}$, $\mathbf{x} \in \mathbb{R}^{128 \times 1}$, natomiast $\mathbf{e} \in \mathbb{R}^{128 \times 1}$ jest wektorem o wszystkich składowych równych 1. Macierz \mathbf{A} jest symetryczna i dodatnio określona. Układ rozwiązyano za pomocą metody Gaussa-Seidela oraz metody gradientów sprzężonych, następnie zanalizowano wydajność faktoryzacji Cholesky'ego na tym układzie. Dzięki uwzględnieniu struktury macierzy, algorytmy działają w czasie liniowym.

2. Opis

2.1. Metoda Gaussa-Seidela

Metoda Gaussa-Seidela jest iteracyjną metodą rozwiązywania układów równań dla macierzy symetrycznych i dodatnio określonych. Metod iteracyjnych używa się najczęściej gdy zastosowanie faktoryzacji prowadziłoby do wypełnienia macierzy rzadkiej.

Po wybraniu początkowego przybliżenia $\mathbf{x}^{(0)}$, wzór na i -tą niewiadomą w $k+1$ -ym kroku w ogólności wygląda następująco:

$$x_i^{(k+1)} = \left(b_i - \sum_{j=1}^{i-1} a_{ij}x_j^{(k+1)} - \sum_{j=i+1}^N a_{ij}x_j^{(k)} \right) / a_{ii}$$

Dzięki strukturze \mathbf{A} możemy ten wzór uprościć, zauważając że macierz ma zera wszędzie oprócz $a_{ii} = 4$, $a_{i,i\pm 1} = 1$, $a_{i,i\pm 4} = 1$:

$$x_i^{(k+1)} = \left(e_i - a_{i,i-4}x_{i-4}^{(k+1)} - a_{i,i-1}x_{i-1}^{(k+1)} - a_{i,i+1}x_{i+1}^{(k)} - a_{i,i+4}x_{i+4}^{(k)} \right) / a_{ii}$$

Dla zwartości wzoru przyjmijmy, że wyrażenie $a_{ij}x_j = 0$ dla $j < 1$ lub $j > 128$. W kodzie jest to podzielone na kilka pętli po odpowiednich przedziałach.

Dzięki wykorzystaniu uproszczonego wzoru, jeden krok wykonuje się w czasie $\mathcal{O}(n)$. Algorytm zatrzymujemy, gdy $\|\mathbf{x}_{k+1} - \mathbf{x}_k\| \leq \epsilon$ lub gdy przekroczony został limit kroków.

2.2. Metoda gradientów sprzężonych (CG)

W ogólności CG minimalizuje funkcję

$$f(\mathbf{x}) = \frac{1}{2} \mathbf{x}^T \mathbf{A} \mathbf{x} - \mathbf{b}^T \mathbf{x} + c$$

gdzie $\mathbf{A} = \mathbf{A}^T \in \mathbb{R}^{n \times n}$ jest dodatnio określona, $\mathbf{x} \in \mathbb{R}^{n \times 1}$, $c \in \mathbb{R}$. Sprowadza się to do rozwiązywania układu równań

$$\mathbf{Ax} = \mathbf{b}$$

Wynika stąd, że metoda gradientów sprzężonych to kolejny sposób na rozwiązywanie układów równań z macierzą symetryczną oraz dodatnio określona. Dla efektywności obliczeniowej wykorzystujemy wersję iteracyjną CG, która z każdym krokiem coraz bardziej zbliża się do wyniku dokładnego.

Po wybraniu przybliżenia \mathbf{x}_0 , obliczamy początkowe wartości: $\mathbf{r}_0 = \mathbf{b} - \mathbf{Ax}_0$, $\mathbf{p}_0 = \mathbf{r}_0$. Następnie algorytm wyznaczenia \mathbf{x}_{k+1} wygląda następująco:

$$\begin{aligned}\alpha_k &= \frac{\mathbf{r}_k \cdot \mathbf{r}_k}{\mathbf{p}_k \cdot \mathbf{Ap}_k} \\ \mathbf{r}_{k+1} &= \mathbf{r}_k - \alpha_k \mathbf{Ap}_k \\ \beta_k &= \frac{\mathbf{r}_{k+1} \cdot \mathbf{r}_{k+1}}{\mathbf{r}_k \cdot \mathbf{r}_k} \\ \mathbf{p}_{k+1} &= \mathbf{r}_{k+1} + \beta_k \mathbf{p}_k \\ \mathbf{x}_{k+1} &= \mathbf{x}_k + \alpha_k \mathbf{p}_k\end{aligned}$$

Algorytm zatrzymujemy, gdy $\|\mathbf{r}_k\| \leq \epsilon$ lub gdy przekroczył został limit kroków.

Korzystając ze struktury macierzy możemy zmniejszyć dominujący koszt operacji mnożenia \mathbf{A} przez wektor do $\mathcal{O}(n)$, dzięki czemu jeden krok algorytmu będzie się wykonywał w czasie liniowym. Zostało to zaimplementowane w funkcji `mult_vector`.

3. Kod

Program napisano w Pythonie z użyciem pakietu *NumPy* oraz *Matplotlib*.

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

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

class DiagMatrix:
    diag: array
    subdiag1: array
    subdiag4: array
    n: int

    def __init__(self, diag: array, subdiag: array, subdiag2: array, n: int):
        self.diag = diag
        self.subdiag1 = subdiag
        self.subdiag4 = subdiag
        self.n = n

    def mult_vector(self, x: vector) -> vector:
        N = self.n
        y = np.zeros(N, dtype=np.float64)

        for i in range(N):
            y[i] += self.diag[i] * x[i]

            if i > 0:
                y[i] += self.subdiag1[i - 1] * x[i - 1]
            if i < N - 1:
                y[i] += self.subdiag1[i] * x[i + 1]
            if i >= 4:
                y[i] += self.subdiag4[i - 4] * x[i - 4]
            if i < N - 4:
                y[i] += self.subdiag4[i] * x[i + 4]
        return y

def gauss_seidel(
    x: vector,
    A: DiagMatrix,
    b: vector,
    eps: float,
    limit: int,
) -> tuple[int, list[float]]:
    steps = 0
    x_old = x.copy()
    diffs = []
```

```

while steps < limit:
    N = A.n
    x[0] = (b[0] - A.subdiag1[0] * x_old[1] - A.subdiag4[0] * x_old[4]) / A.diag[0]

    for i in range(1, 3 + 1):
        x[i] = (
            b[i]
            - A.subdiag1[i - 1] * x[i - 1]
            - A.subdiag1[i] * x_old[i + 1]
            - A.subdiag4[i] * x_old[i + 4]
        ) / A.diag[i]

    for i in range(4, N - 4):
        x[i] = (
            b[i]
            - A.subdiag4[i - 4] * x[i - 4]
            - A.subdiag1[i - 1] * x[i - 1]
            - A.subdiag1[i] * x_old[i + 1]
            - A.subdiag4[i] * x_old[i + 4]
        ) / A.diag[i]

    for i in range(N - 4, N - 1):
        x[i] = (
            b[i]
            - A.subdiag4[i - 4] * x[i - 4]
            - A.subdiag1[i - 1] * x[i - 1]
            - A.subdiag1[i] * x_old[i + 1]
        ) / A.diag[i]

    x[N - 1] = (
        b[0]
        - A.subdiag4[N - 1 - 4] * x[N - 1 - 4]
        - A.subdiag1[N - 1 - 1] * x[N - 1 - 1]
    ) / A.diag[N - 1]

    diff = np.linalg.norm(x - x_old)
    diffss.append(diff)

    if diff <= eps:
        break

    x_old = x.copy()
    steps += 1

return steps, diffss

def conjugate_gradient(
    A: DiagMatrix,
    x: vector,
    b: vector,
    eps: float,
    limit: int,
) -> tuple[int, list[float]]:

```



```

if norm_diff < 1e-8:
    print("\nWynik poprawny - różnice są pomijalne.")
else:
    print("\nWynik może być niepoprawny.")


def main():
    np.set_printoptions(linewidth=np.inf) # pyright: ignore[reportArgumentType]
    N = 128

    diag = np.full(N, 4, dtype=np.float64)
    subdiag1 = np.ones(N - 1, dtype=np.float64)
    subdiag4 = np.ones(N - 4, dtype=np.float64)
    A = DiagMatrix(diag, subdiag1, subdiag4, N)
    e = np.ones(N, dtype=np.float64)

    x = np.zeros(N, dtype=np.float64)
    steps_gs, diffs_gs = gauss_seidel(x, A, e, eps=1e-12, limit=100)
    print("Wynik po", steps_gs, "iteracjach:")
    print(x)
    check_solution(A, x, e)

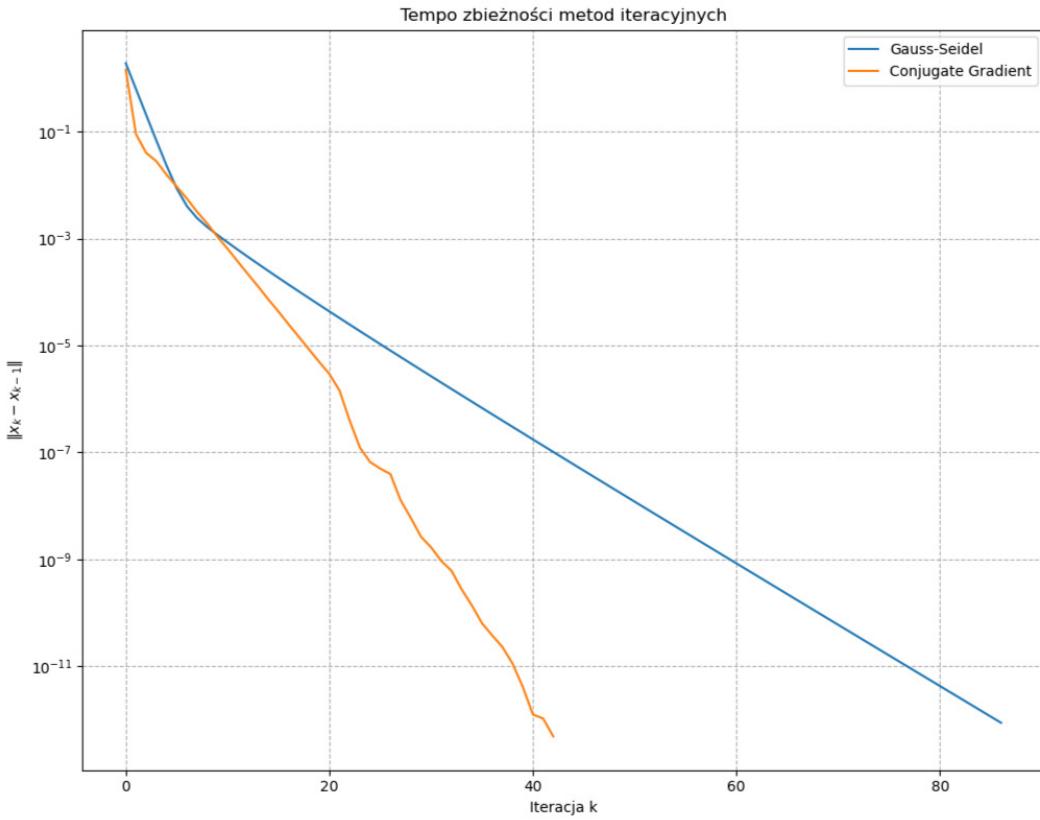
    x = np.zeros(N, dtype=np.float64)
    steps_cg, diffs_cg = conjugate_gradient(A, x, e, eps=1e-12, limit=100)
    print("Wynik po", steps_cg, "iteracjach:")
    print(x)
    check_solution(A, x, e)

    plt.figure(figsize=(8, 5))
    plt.semilogy(diffs_gs, label="Gauss-Seidel")
    plt.semilogy(diffs_cg, label="Conjugate Gradient")
    plt.xlabel("Iteracja k")
    plt.ylabel(r"$\|x_k - x_{k-1}\|$")
    plt.title("Tempo zbieżności metod iteracyjnych")
    plt.legend()
    plt.grid(True, which="both", ls="--")
    plt.show()

if __name__ == "__main__":
    main()

```

4. Porównanie złożoności



Rysunek 1: Porównanie zbieżności metod iteracyjnych dla $\epsilon = 10^{-12}$

- Metoda Gaussa-Seidela: koszt iteracji $\mathcal{O}(n)$, liczba iteracji 86, całkowity koszt $\mathcal{O}(86n)$
- Metoda gradientów sprzężonych: koszt iteracji $\mathcal{O}(n)$, liczba iteracji 42, całkowity koszt $\mathcal{O}(42n)$
- Faktoryzacja Cholesky'ego: skoro macierz jest pasmowa (szerokość pasma to $4 \ll 128$), to koszt faktoryzacji wynosi $\mathcal{O}(16n)$ ¹. Następnie koszt *backsubstitution* jest równy kosztowi *forward substitution* i wynosi $\mathcal{O}(n)$. Łącznie mamy $\mathcal{O}(18n)$

5. Wyniki

Metoda Gaussa-Seidela oblicza wynik z najgorszą dokładnością 10^{-13} , zaś CG z najgorszą dokładnością 10^{-14} .

¹Faktoryzacja Cholesky'ego macierzy pasmowej o szerokości pasma p wynosi $\mathcal{O}(p^2n)$.

6. Wnioski

W metodzie Gaussa-Seidela oraz gradientów sprzężonych wykorzystanie pasmowej struktury macierzy pozwoliło obniżyć koszt pojedynczej iteracji do rzędu $\mathcal{O}(n)$. CG potrzebowało dwukrotnie mniej iteracji do otrzymania tej samej dokładności.

Zaskakujący jest fakt, że faktoryzacja Cholesky'ego okazała się w tym przypadku najwydajniejsza, gdyż ta metoda nie zawsze jest optymalna dla macierzy rzadkich. Przykładem jest macierz o strukturze:

$$\begin{bmatrix} \bullet & \bullet & \bullet & \bullet & \bullet & \cdots \\ \bullet & \bullet & & & & \\ \bullet & & \bullet & & & \\ \bullet & & & \bullet & & \\ \bullet & & & & \bullet & \\ \bullet & & & & & \ddots \end{bmatrix}$$

której rozkład Cholesky'ego jest macierzą pełną.