Zadanie 2:

2 O. Dane jest macierz $\mathbf{A} \in \mathbb{R}^{128\times128}$ o następującej strukturze

$$\mathbf{A} = \begin{bmatrix} 4 & 1 & 0 & 0 & 1 & 0 & 0 & 0 & 0 & 0 & 0 & \dots \\ 1 & 4 & 1 & 0 & 0 & 1 & 0 & 0 & 0 & 0 & 0 & \dots \\ 0 & 1 & 4 & 1 & 0 & 0 & 1 & 0 & 0 & 0 & 0 & \dots \\ 0 & 0 & 1 & 4 & 1 & 0 & 0 & 1 & 0 & 0 & 0 & \dots \\ 1 & 0 & 0 & 1 & 4 & 1 & 0 & 0 & 1 & 0 & 0 & \dots \\ 0 & 1 & 0 & 0 & 1 & 4 & 1 & 0 & 0 & 1 & 0 & \dots \\ 0 & 0 & 1 & 0 & 0 & 1 & 4 & 1 & 0 & 0 & 1 & \dots \\ \dots & \dots & \dots & \dots & \dots & \dots & \dots & \dots & \dots & \dots & \dots & \dots \\ \dots & 0 & 0 & 1 & 0 & 0 & 1 & 4 & 1 & 0 & 0 & 1 \\ \dots & 0 & 0 & 0 & 1 & 0 & 0 & 1 & 4 & 1 & 0 & 0 \\ \dots & 0 & 0 & 0 & 0 & 1 & 0 & 0 & 1 & 4 & 1 & 0 \\ \dots & 0 & 0 & 0 & 0 & 0 & 1 & 0 & 0 & 1 & 4 & 1 \\ \dots & 0 & 0 & 0 & 0 & 0 & 0 & 1 & 0 & 0 & 1 & 4 \end{bmatrix} \tag{3}$$

Rozwiązać równanie $\mathbf{Ax} = \mathbf{e}$, gdzie $\mathbf{A}$ jest macierzą (3), natomiast $\mathbf{e}$ jest wektorem, którego wszystkie składowe są równe 1, za pomocą

(a) metody Gaussa-Seidela,

(b) metody gradientów sprzężonych.

Algorytmy **muszą** uwzględniać strukturę macierzy (3) — w przeciwnym razie zadanie nie będzie zaliczone!
Oba algorytmy proszę zastartować z tego samego przybliżenia początkowego. Porównać graficznie tempo zbieżności tych metod, to znaczy jak zmieniaję się normy $\|\mathbf{x}_k - \mathbf{x}_{k-1}\|$, gdzie $\mathbf{x}_k$ oznacza k-ty iterat. Porównać efektywną złożoność obliczeniową ze złożonością obliczeniową rozkładu Cholesky'ego dla tej macierzy.

Kod w języku Python:

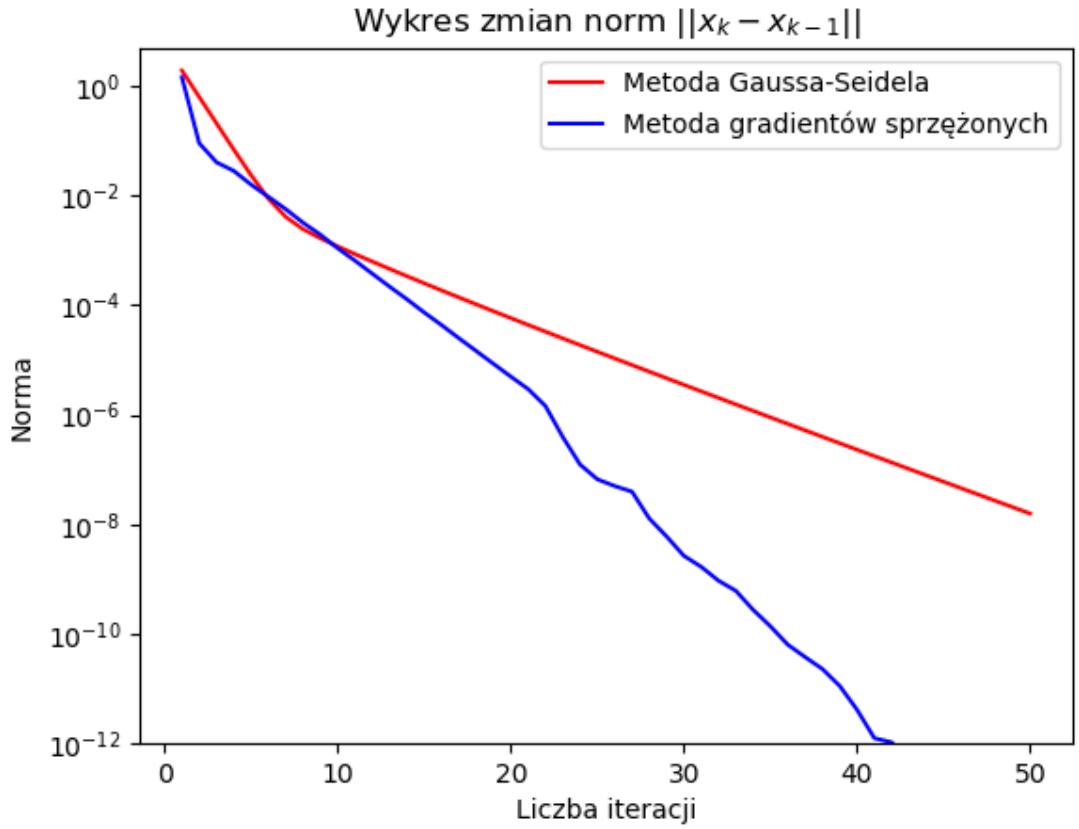
```python
import numpy as np
import copy
import math
import time
import matplotlib.pyplot as plt


def fill(A, x, b):
    for i in range(N):
        x[i] = 0
        b[i] = 1
        for j in range(N):
            if i == j:
                A[i][j] = 4
            elif (j == i + 1) or (j == i - 1) or (j == i + 4) or (j == i - 4):
                A[i][j] = 1


def calcNorm(x, prev_x):
    result = 0
    for i in range(len(x)):
        tmp = x[i] - prev_x[i]
        result += pow(tmp, 2)
```

```
    result = math.sqrt(result)
    return result


def seidel(A, x, b, iters, gaussNorms):
    length = len(A)
    prev_x = []
    for i in range(iters):
        prev_x = copy.deepcopy(x)
        for row in range(length):
            d = b[row]
            if row - 4 >= 0:
                d -= A[row][row - 4] * x[row - 4]
            if row - 1 >= 0:
                d -= A[row][row - 1] * x[row - 1]
            if row + 1 < length:
                d -= A[row][row + 1] * x[row + 1]
            if row + 4 < length:
                d -= A[row][row + 4] * x[row + 4]
            x[row] = d / A[row][row]
        gaussNorms.append(calcNorm(x, prev_x))


def calcAPK(A, p):
    length = len(A)
    vec = [0] * len(p)
    for i in range(len(p)):
        tmp = 0
        if i - 4 >= 0:
            tmp += A[i][i - 4] * p[i - 4]
        if i - 1 >= 0:
            tmp += A[i][i - 1] * p[i - 1]
        if i + 1 < length:
            tmp += A[i][i + 1] * p[i + 1]
        if i + 4 < length:
            tmp += A[i][i + 4] * p[i + 4]
        tmp += A[i][i] * p[i]
        vec[i] = tmp
    return vec


def gradients(A, x, b, iters, gradientNorms):
    r = copy.deepcopy(b)
    p = copy.deepcopy(r)

    for i in range(iters):
        prev_r = copy.deepcopy(r)
        Apk = calcAPK(A, p)
        alfa = np.dot(r, r) / np.dot(p, Apk)

        for j in range(len(r)):
            r[j] = prev_r[j] - alfa * Apk[j]

        beta = np.dot(r, r) / np.dot(prev_r, prev_r)
        prev_p = copy.deepcopy(p)

        for j in range(len(p)):
            p[j] = r[j] + beta * prev_p[j]
```

```
        prev_x = copy.deepcopy(x)

        for j in range(len(x)):
            x[j] = prev_x[j] + alfa * prev_p[j]

        gradientsNorms.append(calcNorm(x, prev_x))


iters = 50
N = 128
A = [[0] * N for i in range(N)]
b = [1] * N
x = [0] * N
gradientsNorms = []
gaussNorms = []

# Wypelnienie poczatkowymi wartosciami
fill(A, x, b)

# Metoda Gaussa-Seidela
start = time.time()
seidel(A, x, b, iters, gaussNorms)
end = time.time()
print("Metoda Gaussa-Seidela")
print("Zajelo to ", (end - start), " sekundy")
print("x:", x)
print()

# Ponowne wypelnianie poczatkowyni wartosciami
fill(A, x, b)

# Metoda Gradientów sprzężonych
start = time.time()
gradients(A, x, b, iters, gradientsNorms)
end = time.time()
print("Metoda Gradientów sprzężonych")
print("Zajelo to ", (end - start), " sekundy")
print("x:", x)


# Rysowanie wykresu
xRange = [0] * iters
for j in range(iters):
    xRange[j] = j+1

lower = pow(10, -12)
plt.ylim(lower, 5)
plt.yscale("log")
plt.title('Wykres zmian norm $|| x_k - x_{k-1} ||$')
plt.xlabel("Liczba iteracji")
plt.ylabel("Norma")
plt.plot(xRange, gaussNorms, 'r', xRange, gradientsNorms, 'b')
plt.legend(('Metoda Gaussa-Seidela', 'Metoda gradientów sprzężonych'))
plt.savefig('asd.png')
plt.show()
```

Wyniki:

```
Metoda Gaussa-Seidela
Zajelo to  0.008977890014648438  sekundy
Pierwsze 5 wyrazów:
x:  0.19427679547980875 0.1309302026080166 0.14679490804047404 0.16231131562715603 0.09196261556104166

Metoda Gradientów sprzężonych
Zajelo to  0.05712437629699707  sekundy
Pierwsze 5 wyrazów:
x:  0.19427679544419685 0.13093020247814458 0.14679490834352368 0.16231131528577905 0.09196261574506784
```



Obie metody są metodami iteracyjnymi. Z każdą kolejną iteracją wektor x będzie coraz bliżej wartości dokładnej. Z pomiaru widocznego na pierwszym obrazku, wykonanego dla 50 iteracji widać, że metoda gradientów sprzężonych jest wolniejsza. Daje ona jednak dokładniejsze wyniki, co widać na wykresie. Na przykład dla dokładności 10^{-6} metoda gradientów sprzężonych wymaga około 20 iteracji, podczas gdy metoda Gaussa-Seidela potrzebuje ich prawie 2 razy więcej.

Efektywne złożoności obliczeniowe:

Metoda Gaussa-Seidela – dla każdej iteracji należy obliczyć „N” elementów wektora x. W naszym przypadku obliczenie pojedynczego elementu składowego wektora x nie jest zależne od wielkości macierzy ponieważ zawsze mnożymy maksymalnie 4 elementy. Dla macierzy podanej w zadaniu efektywność metody Gaussa-Seidela wynosi O(N*k), gdzie „k” oznacza liczbę iteracji.

Metoda gradientów sprzężonych - dla każdej iteracji należy obliczyć „N” elementów wektora x. Obliczenie pojedynczego x[i] wymaga obliczenia Apk, alfy, bety, r oraz p. Obliczenie Apk wymaga wyliczenia „N” elementów wektora. Koszt obliczenia pojedynczego elementu wektora nie zależy od jego wielkości ponieważ zawsze wymnażamy maksymalnie 5 elementów. Aby wyliczyć alfę i betę musimy obliczyć po dwa iloczyny skalarne. Koszt takiej operacji to 2N. Koszt wyliczenia p oraz r to N. Złożoność obliczeniowa metody wynosi $O(k*N^2)$

Metoda Cholesky’ego – macierz podana w zadaniu jest macierzą pięciodiagonalną. Czynnik Cholesky’ego wyznaczymy w czasie liniowym. Forward i backsubstitution również wykonamy w czasie liniowym. Koszt całkowity metody Cholesky’ego wynosi O(N)