METODY NUMERYCZNE

ZADANIE 2

Artur Guniewicz

2 O. Dane jest macierz $\mathbf{A} \in \mathbb{R}^{128\times 128}$ 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.

Oba podpunkty wykonałem w jednym programie.

Kod programu:

```
/*
************************************************************
*                                                          *
*                Artur Guniewicz - Zadanie 2               *
*                                                          *
************************************************************
*/

#include <iostream>
#include <vector>  // vector
#include <fstream> // fstream
#include <math.h>  // sqrt
#include <numeric> // inner_product
#include <chrono>  // chrono::high_resolution_clock
#include <stdio.h>
#include <stdlib.h>
#include <iomanip> // zawiera funkcję setprecision()

using namespace std;

#define SIZE 128
#define Vector vector<long double>
#define Matrix vector<vector<long double>>

void fill(Matrix &A, Vector &x, Vector &b);
void GaussSeidel(const Matrix &A, Vector &x, const Vector &b, const int
no_of_iters);
void ConjugateGradients(const Matrix &A, Vector &x, const Vector &b,
const int no_of_iters);
long double sum(int i, const Matrix &A, const Vector &x);
long double normCalculation(const Vector &prev_x, const Vector &x);
void writeToFile(const Vector &x, const char *f);
long double scalarProduct(const Vector &a, const Vector &b);

int main(int argc, char const *argv[])
{
   // kontrola liczby argumentów progamu
   if (argc != 2)
```

```
    {
        cout << "Niepoprawna liczba argumentów wywołania programu!" <<
endl;
        cout << "Prawidlowe wywolanie programu: ./Zadanie2 <liczba
iteracji>" << endl;
        exit(-1);
    }

    Matrix A(SIZE);

    for (int i = 0; i < SIZE; i++)
        A[i].resize(SIZE); // zmiana wielkości

    Vector x(SIZE);
    Vector b(SIZE);

    cout << "Czas wykonania dla metod: " << endl;

    // wypełnianie wektorów
    fill(A, x, b);

    auto start = chrono::high_resolution_clock::now();
    GaussSeidel(A, x, b, stoi(argv[1]));
    auto end = chrono::high_resolution_clock::now();

    cout << "a) Gaussa-Seidela:"            << "           " <<
chrono::duration<double>(end-start).count() << endl;

    // wypełnianie od nowa dla drugiej metody
    fill(A, x, b);

    start = chrono::high_resolution_clock::now();
    ConjugateGradients(A, x, b, stoi(argv[1]));
    end = chrono::high_resolution_clock::now();

    cout << "b) Gradientów Sprzezonych:"    << "  " <<
chrono::duration<double>(end-start).count() << endl;

    return 0;
}
```

```cpp
void fill(Matrix &A, Vector &x, Vector &b)
{
   for (int i = 0; i < x.size(); i++)
   {
       x[i] = -1;
       b[i] = 1;

       if (i == 0)
           A[i][i + 1] = 1;

       else if (i == x.size() - 1)
           A[i][i - 1] = 1;

       else
       {
           A[i][i - 1] = 1;
           A[i][i + 1] = 1;
       }

       A[i][i] = 4;

       if (i < x.size() - 4)
       {
           A[i][i + 4] = 1;
           A[i + 4][i] = 1;
       }
   }
}

// metoda Gaussa-Seidela dla <no_of_iters> iteracji
void GaussSeidel(const Matrix &A, Vector &x, const Vector &b, const int
no_of_iters)
{
   int iter_num = 0; // ile już było iteracji
   Vector prev_x;    // wektor poprzednich wartości x
   Vector norms;     // wektor dla obliczonych norm
   prev_x.resize(x.size());
```

```
    for (int i = 0; i < x.size(); i++) // dodawanie nowego elementu na
koniec wektora
        prev_x.push_back(x[i]);

    for (int it = 0; it < no_of_iters; it++)
    {
        // obliczamy kolejną iterację x
        for (int i = 0; i < x.size(); i++)
        {
            prev_x[i] = x[i];
            x[i] = (b[i] - sum(i, A, x)) / A[i][i];
        }

        // każdorazowo obliczamy normę
        norms.push_back(normCalculation(prev_x, x));
    }

    // zapisujemy wektor norm do pliku (do późniejszego wykresu)
    writeToFile(norms, "Zadanie2_GaussSeidel.txt");
}

// suma składników mnożenia macierzy i wektora
long double sum(int i, const Matrix &A, const Vector &x)
{
    long double sum = 0;

    if (i == 0)
        sum = +A[0][1] * x[1] + A[0][4] * x[4];

    else if (i < 4)
        sum = A[i][i - 1] * x[i - 1] + A[i][i + 1] * x[i + 1] + A[i][i +
4] * x[i + 4];

    else if (i < A.size() - 4)
        sum = A[i][i - 4] * x[i - 4] + A[i][i - 1] * x[i - 1] + A[i][i +
1] * x[i + 1] + A[i][i + 4] * x[i + 4];

    else if (i < A.size() - 1)
        sum = A[i][i - 4] * x[i - 4] + A[i][i - 1] * x[i - 1] + A[i][i +
1] * x[i + 1];
```

```
    else
        sum = A[i][i - 4] * x[i - 4] + A[i][i - 1] * x[i - 1];

    return sum;
}

//  norma różnicy wektorów
long double normCalculation(const Vector &prev_x, const Vector &x)
{
    long double norm;
    long double res = 0;

    for (int i = 0; i < x.size(); i++)
    {
        norm = x[i] - prev_x[i];
        res += norm * norm;
    }

    res = sqrt(res);
    return res;
}

// zapis do pliku (do stworzenia wykresu)
void writeToFile(const Vector &x, const char *f)
{
    ofstream file;

    file.open(f);

    for (int i = 0; i < x.size(); i++)
        file << i + 1 << ". " << x[i] << endl;

    file.close();
}

// metoda Gradientów Sprzężonych wykonana <no_of_iters> razy
void ConjugateGradients(const Matrix &A, Vector &x, const Vector &b,
const int no_of_iters)
{
```

```
    Vector r = b;
    Vector p = r;
    Vector prev_x = x; //wektor poprzednich wartosci x
    Vector norms;      //wektor dla obliczonych norm
    Vector Apk(x.size());

    for (int i = 0; i < no_of_iters; i++)
    {
        Vector prev_r = r;

        // obliczanie Apk
        for (int j = 0; j < x.size(); j++)
            Apk[j] = scalarProduct(A[j], p);

        long double alfa = scalarProduct(r, r) / scalarProduct(p, Apk);

        // następny wynik x
        for (int j = 0; j < x.size(); j++)
        {
            prev_x[j] = x[j];
            x[j] = x[j] + alfa * p[j];
        }

        for (int j = 0; j < x.size(); j++)
        {
            r[j] = r[j] - alfa * Apk[j];
        }

        long double beta = scalarProduct(r, r) / scalarProduct(prev_r,
prev_r);

        //kolejne p
        for (int j = 0; j < x.size(); j++)
        {
            p[j] = r[j] + beta * p[j];
        }

        //obliczenie normy
        norms.push_back(normCalculation(prev_x, x));
    }
```

```cpp
    // zapisujemy wektor norm do pliku (do późniejszego wykresu)
    writeToFile(norms, "Zadanie2_ConjGrad.txt");
}

// oblicza iloczyn skalarny wektorow a i b
long double scalarProduct(const Vector &a, const Vector &b)
{
    return inner_product(a.begin(), a.end(), b.begin(), (long 
double)0.0);
}
```

Kompilacja:

g++ Zadanie2.cpp -o Zadanie2 && ./Zadanie2 <liczba iteracji>

Wyniki:

-> dla 45 iteracji:

```
Czas wykonania dla metod:
a) Gaussa-Seidela:           0.000933919
b) Gradientów Sprzezonych:   0.0152403
```

-> dla 100 iteracji:

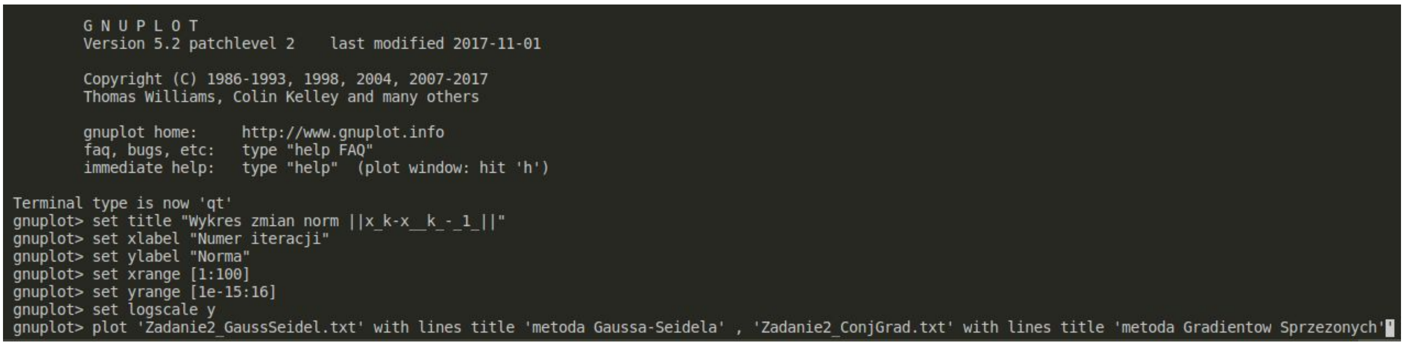
```
Czas wykonania dla metod:
a) Gaussa-Seidela:           0.00176955
b) Gradientów Sprzezonych:   0.0413558
```

Wykres:

uruchomienie z terminalu: gnuplot --persist

skrypt do stworzenia wykresu:

gnuplot> set title "Wykres zmian norm ||x_k-x__k_-_1_||"
gnuplot> set xlabel "Numer iteracji"
gnuplot> set ylabel "Norma"
gnuplot> set xrange [1:100]
gnuplot> set yrange [1e-15:16]
gnuplot> set logscale y
gnuplot> plot 'Zadanie2_GaussSeidel.txt' with lines title 'metoda Gaussa-Seidela' ,
'Zadanie2_ConjGrad.txt' with lines title 'metoda Gradientow Sprzezonych'

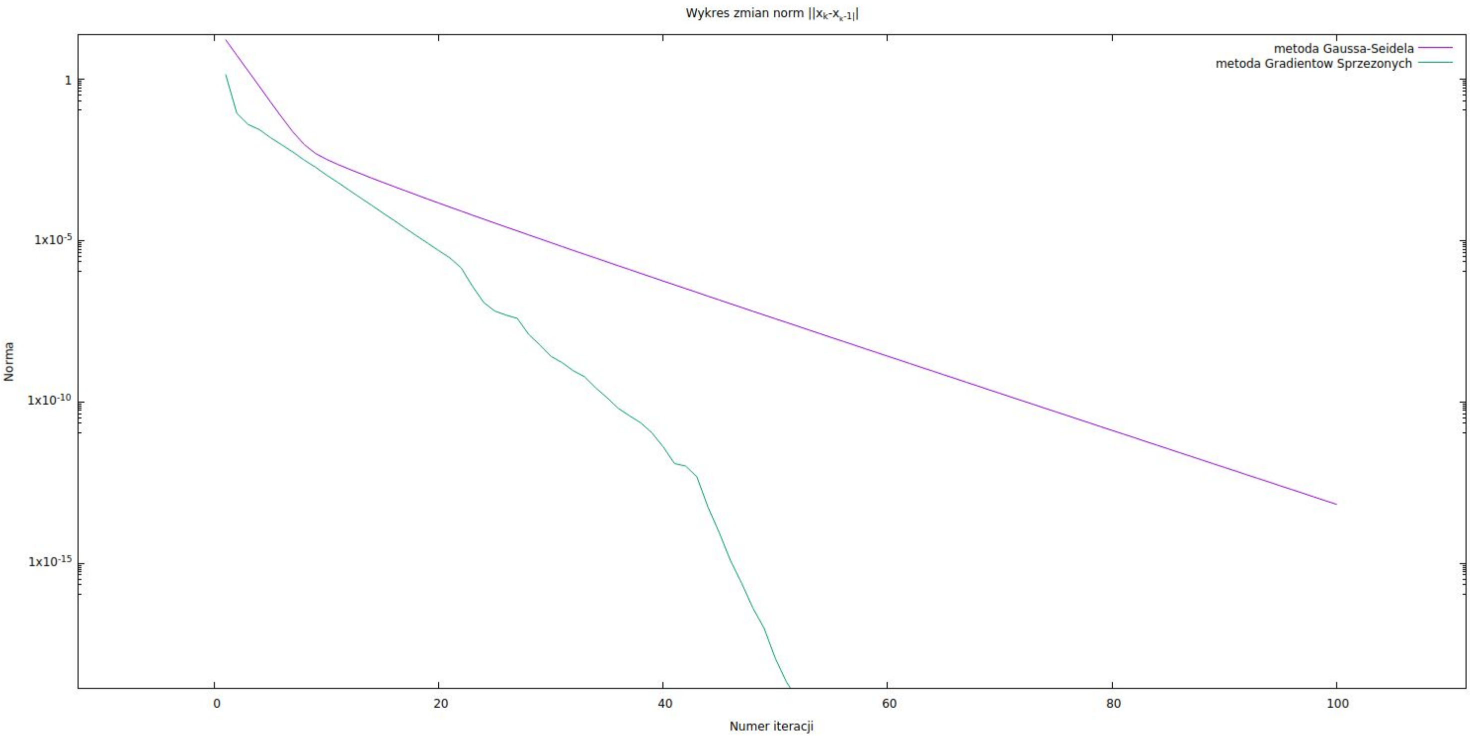


```
        G N U P L O T
        Version 5.2 patchlevel 2    last modified 2017-11-01

        Copyright (C) 1986-1993, 1998, 2004, 2007-2017
        Thomas Williams, Colin Kelley and many others

        gnuplot home:     http://www.gnuplot.info
        faq, bugs, etc:   type "help FAQ"
        immediate help:   type "help"  (plot window: hit 'h')

Terminal type is now 'qt'
gnuplot> set title "Wykres zmian norm ||x_k-x__k_-_1_||"
gnuplot> set xlabel "Numer iteracji"
gnuplot> set ylabel "Norma"
gnuplot> set xrange [1:100]
gnuplot> set yrange [1e-15:16]
gnuplot> set logscale y
gnuplot> plot 'Zadanie2_GaussSeidel.txt' with lines title 'metoda Gaussa-Seidela' , 'Zadanie2_ConjGrad.txt' with lines title 'metoda Gradientow Sprzezonych'
```



Komentarz:

Zarówno metoda Gaussa-Seidela jak i Gradientów Sprzężonych są metodami iteracyjnymi, co oznacza, że przy każdej iteracji otrzymamy coraz dokładniejszy wynik. Hipotetycznie rzecz ujmując, dla nieskończonej liczby iteracji otrzymamy wynik dokładny. Z wykresu można odczytać że metoda Gaussa-Seidela jest metodą szybszą. Dysproporcja prędkości wykonania programu rośnie wraz ze wzrostem iteracji. Metoda Gradientów Sprzężonych rekompensuje to jednak dając dokładniejsze wyniki dla mniejszej liczby iteracji.

Jak wygląda sprawa ze złożonościami obliczeniowymi?

1. Metoda Gaussa-Seidela

   W każdej iteracji liczone jest N elementów wektora x.
   Złożoność: O(k*N)

2. Metoda Gradientów Sprzężonych

   W każdej iteracji obliczane są Apk, alfa, beta, r, p. Dla Apk liczone jest N elementów wektora, dla alfy i bety 2N, natomiast dla r i p -> po N dla każdego.
   Złożoność: około O(k*12N*N)

3. Metoda Choleskiego

   Macierz ta posiada 3 pasma niezerowe więc łącznie faktoryzacja Choleskiego zajmie 3N czasu. Potem, w celu obliczenia współczynników wektora x, wykonane zostaną backsubstitution oraz forwardsubstitution, po N czasu każde.
   Złożoność: O(5N)