Metody numeryczne - N5

DAMIAN PORADYŁO

Zaimplementować metodę:

- relaksacyjną (Richardsona)

$$x^{(n+1)} = x^{(n)} + \gamma \left(b - Ax^{(n)} \right),$$

- Jacobiego:

$$x^{(n+1)} = D^{-1} \left(b - Rx^{(n)} \right),$$

- Gausa-Seidla

$$x^{(n+1)} = L^{-1} \left(b - Ux^{(n)} \right),$$

- Successive OverRelaxation

$$x_i^{(n+1)} = (1-\omega)x_i^{(n)} + \frac{\omega}{a_{ii}} \left(b_i - \sum_{j<i} a_{ij} x_j^{(n+1)} - \sum_{j>i} a_{ij} x_j^{(n)} \right), \quad i = 1, 2, \ldots, N.$$

Znaleźć rozwiązania układu z zadania N3 z dokładnością 10^{-10}. Która metoda jest najszybsza? Proszę uwzględnić strukturę układu równań.
Dlaczego z tym układem są problemy? Jak "naprawić" układ aby dało sie go rozwązać iteracyjnie?
Jeśli nie znasz odpowiedzi na powyższe pytania znajdź rozwiązanie dla układu *prawie* takiego jak w zadaniu N3, zastępując równania $(D_2 y)_n + y_n = 0$ równaniami $-(D_2 y)_n + y_n = 0$.
Uwaga: w razie dalszych problemów z wykorzystaniem macierzy z zadania N3 można skorzystać z macierzy

$$A = \begin{pmatrix} 4 & -1 & 0 \\ -1 & 4 & -1 \\ 0 & -1 & 4 \end{pmatrix}, \qquad b = \begin{pmatrix} 2 \\ 6 \\ 2 \end{pmatrix}$$

i walczyć o 3 punkty.

1 OPIS METODY

W swoim rozwiązaniu skupiłem się na rozwiązaniu układu (z N3) poprzez zastąpienie równania:

$(D_2y)_n + y_n = 0$ równaniem $-(D_2y)_n + y_n = 0$

$$\begin{cases} y_0 = 1 & (0) \\ -(D_2y)_n + y_n = 0 \quad n = 1 \dots (N-1) & (1) \\ y_{N-1} - 2y_N + y_0 = 0 & (2) \end{cases}$$

gdzie $N = 1000,\ h = 0.01$, natomiast

$$(D_2y)_n = \frac{y_{n-1} - 2y_n + y_{n+1}}{h^2}$$

Zauważmy że w układzie równań w ostatnim równaniu (2) występuje y_0 którego wartość jest nam znana z pierwszego równania (1). Dzięki temu możemy przenieść je na prawą stronę równania otrzymując wówczas taką postać:

$$\begin{cases} y_0 = 1 & (3) \\ -(D_2y)_n + y_n = 0 \quad n = 1 \dots (N-1) & (4) \\ y_{N-1} - 2y_N = -1 & (5) \end{cases}$$

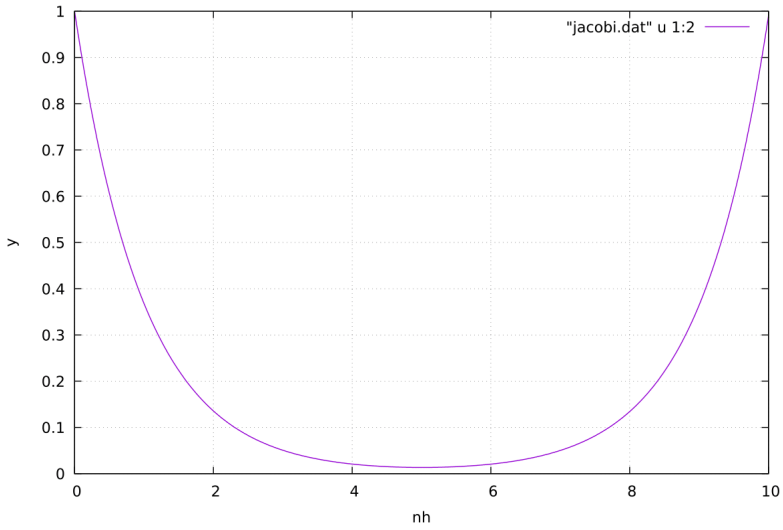
Układ równań po przekształceniu do macierzy przedstawia się następująco:

$$\begin{bmatrix} 1 & 0 & 0 & 0 & 0 & \cdots & 0 \\ -1 & h^2+2 & -1 & 0 & 0 & \cdots & 0 \\ 0 & -1 & h^2+2 & -1 & 0 & \cdots & 0 \\ 0 & 0 & -1 & h^2+2 & -1 & \cdots & 0 \\ \vdots & \vdots & \vdots & \ddots & \ddots & \ddots & \vdots \\ 0 & 0 & 0 & 0 & -1 & h^2+2 & -1 \\ 0 & 0 & 0 & 0 & 0 & 1 & -2 \end{bmatrix} \begin{bmatrix} y_0 \\ y_1 \\ y_2 \\ y_3 \\ \vdots \\ y_{N-1} \\ y_N \end{bmatrix} = \begin{bmatrix} 1 \\ 0 \\ 0 \\ 0 \\ \vdots \\ 0 \\ -1 \end{bmatrix}$$

2 WYNIKI

2.1 METODA JACOBIEGO

$$x^{(n+1)} = D^{-1}\left(b - Rx^{(n)}\right)$$



Wykres 1 Metoda Jacobiego(y,nh)

Liczba iteracji: **264015**

Jak widzimy, metoda ta nie jest optymalna. Rozwiązanie układu z dokładnością 10-10 udaje się nam otrzymać aż po **264015** iteracjach.

D w naszym równaniu to macierz diagonalna, natomiast macierz R jest macierzą z elementami macierzy A bez diagonali na której znajduja się zera ($A = D + R$)

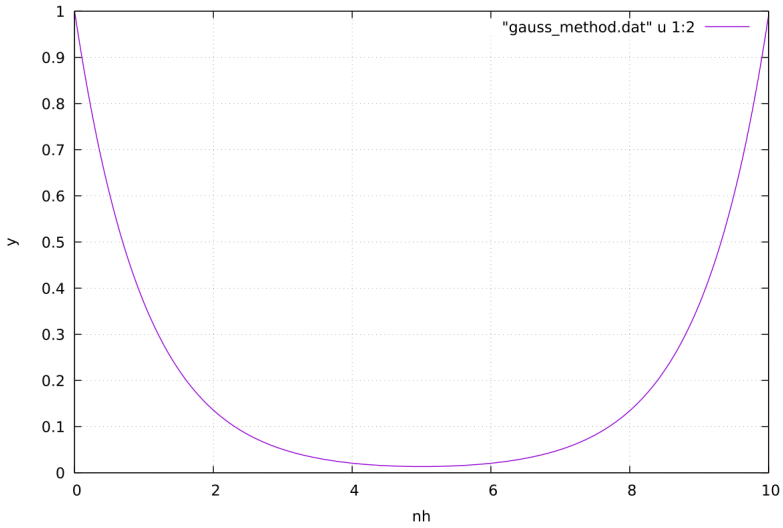
Nasze równanie, możemy zapisać w taki sposób:

$$x_i^{(k+1)} = \frac{1}{a_{ii}}\left(b_i - \sum_{j \neq i} a_{ij} x_j^{(k)}\right), \quad i = 1, 2, \ldots, n.$$

Dzięki temu unikamy redundantnych operacji, które są związane ze znaczną przewagą zer w naszej macierzy.

2.2 METODA GAUSA-SEIDLA

$$x^{(n+1)} = L^{-1}\left(b - Ux^{(n)}\right)$$



Wykres 2 Metoda Gausa-Seidla(y,nh)

Liczba iteracji: **135168**

Liczba iteracji potrzebna do uzyskania wyników w tym przypadku jest o około połowę mniejsza od **metody Jacobiego.**

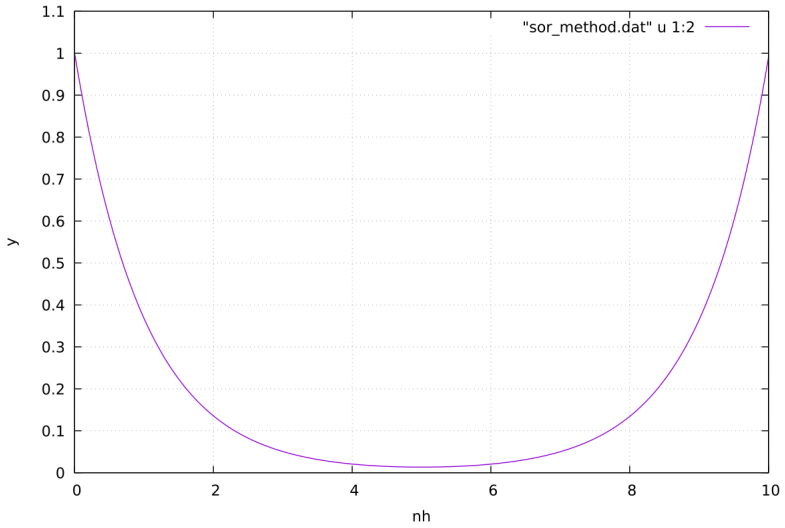
W naszym wzorze L jest dolnym trójkątem macierzy A wraz z diagonalą, natomiast U to górny trójkąt tej macierzy jednak już bez diagonali.

Powyższy wzór możemy rozpisać na składowe, otrzymując postać używaną w implementacjach numerycznych:

$$x_i^{(k+1)} = \frac{1}{a_{ii}}\left(b_i - \sum_{j=1}^{i-1} a_{ij}x_j^{(k+1)} - \sum_{j=i+1}^{n} a_{ij}x_j^{(k)}\right), \quad (i = 1, 2, \ldots, n)$$

2.3 METODA SUCCESSIVE OVER-RELAXATION

$$x_i^{(n+1)} = (1-\omega)x_i^{(n)} + \frac{\omega}{a_{ii}}\left(b_i - \sum_{j<i} a_{ij}x_j^{(n+1)} - \sum_{j>i} a_{ij}x_j^{(n)}\right), \quad i = 1, 2, \ldots, N$$



Liczba iteracji: **4013**
Parametr $\omega \sim \mathbf{1.99375}$

Przy odpowiednim doborze parametru ω jesteśmy w stanie wygenerować nasze rozwiązania już po **4013** iteracjach, co wśród metod iteracyjnych z tego zadania daje najlepszy wynik.

Do wyznaczenia odpowiedniego ω, tzn. optymalnego parametru relaksacji posłużyłem się metodą[1]:

$$\omega_{opt} = \frac{2}{1+\sin(\pi * h)} \qquad \text{gdzie} \qquad h = \frac{1}{N+1}$$

Metoda ta jest można powiedzieć przyspieszeniem metody Gaussa-Seidela z racji dodania wspomnianego wyżej współczynnika ω zwanego parametrem relaksacji.

Należy również dodać, że jeśli macierz A ma niezerową diagonale to parametr ω musi znajdować się w przedziale $(0; 2)$ aby metoda SOR była zbieżna. Jeżeli przyjmiemy $\omega > 1$ to możemy mówić o *nadrelaksacji.*

[1] https://www.sciencedirect.com/science/article/pii/S0893965908001523

2.4 Metoda relaksacyjna (Richardsona)

$$x^{(n+1)} = x^{(n)} + \gamma(b - Ax^{(n)})$$

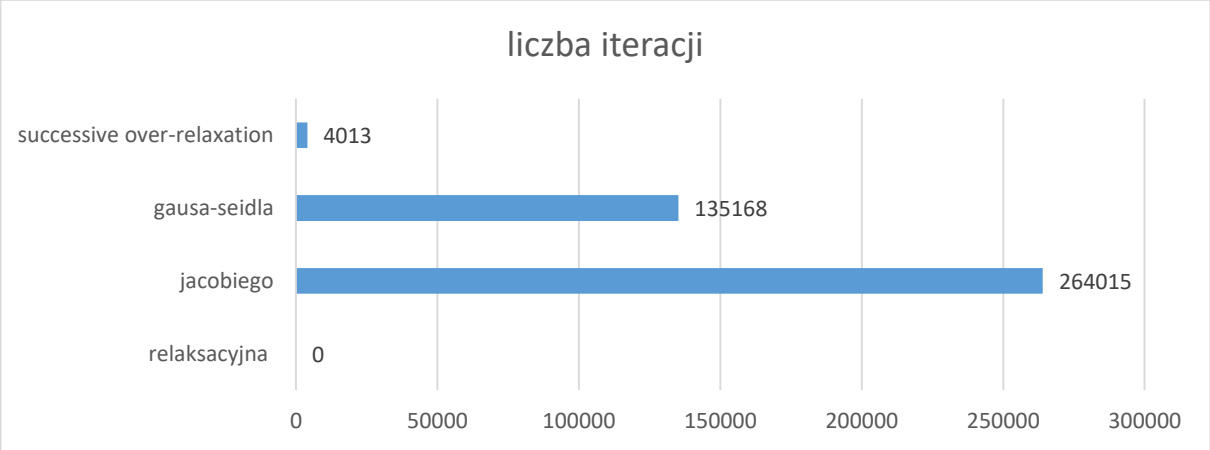
Niestety, w tej metodzie zauważyłem iż nie da się osiągnąć zbieżności dla podanego w zadaniu układu NxN. Zatem posłużyłem się tutaj inną macierzą podaną w zadaniu, mianowicie macierzą $3x3$.

$$A = \begin{pmatrix} 4 & -1 & 0 \\ -1 & 4 & -1 \\ 0 & -1 & 4 \end{pmatrix} \qquad b = \begin{pmatrix} 2 \\ 6 \\ 2 \end{pmatrix}$$

Wybrałem $\gamma = 0.25$, która pozwoliła na otrzymanie poniższych wyników:

$$x = \begin{pmatrix} 1 \\ 2 \\ 1 \end{pmatrix}$$

Przy pomocy 24 iteracji.



Wykres 3 Porównanie liczby iteracji dla poszczególnych metod dla macierzy NxN

3 KOD PROGRAMU

```cpp
#include <iostream>
#include <iomanip>
#include <vector>
#include <cmath>
#include <numeric>

constexpr unsigned N = 1001;
constexpr double h = 0.01;
constexpr double h_sqare = h*h;

double errorNorm(std::vector<double>error) {
    double norm = 0.0;
    for(int i=0; i<error.size(); i++) {
        norm += error[i]*error[i];
    }
    return std::sqrt(norm);
}

std::vector<double> errorResult(std::vector<double> a, std::vector<double> b) {
    std::vector<double> result(N);
    for(int i=0; i<N; i++) {
        result[i] = a[i] - b[i];
    }
    return result;
}

void initMatrix(double *lower, double *main, double *upper, double *b) {
    main[0] = b[0] = 1.0;
    upper[0] = 0;
    lower[0] = -1.0;
    for(int i=1; i<N; i++) {
        main[i] = h_sqare + 2;
        upper[i] = -1.0;
        b[i] = 0;
        if(i<N-2) lower[i] = -1.0;
    }
    b[N-1] = -1.0;
    lower[N-2] = 1.0;
    main[N-1] = -2.0;
}

std::vector<double> jacobi_Method(double *lower, double *main, double *upper, double *b, double precision) {
    std::vector<double> x(N, 0.0);
    std::vector<double> x_new(N, 0.0);
    std::vector<double> error(N, 0.0);
    double actualPrecisionValue = 1.0;
    int iteration = 0;

    while(actualPrecisionValue >= precision) {
        x_new[0] = b[0];
        for(int i=1; i<N-1; i++)   x_new[i] = (1/main[i]) * (b[i] - (lower[i-1] * x[i-1] + upper[i] * x[i+1]));

        x_new[N-1] = (1/main[N-1]) * (b[N-1] - (lower[N-2] * x[N-2]));

        error = errorResult(x, x_new);
        actualPrecisionValue = errorNorm(error);
        x = x_new;
        iteration++;
    }
    //std::cout << iteration << std::endl;

    return x;
}
std::vector<double> gauss_Seidel_Method(double *lower, double *main, double *upper, double *b, double precision) {
    std::vector<double> x(N, 0.0);
    std::vector<double> x_new(N, 0.0);
    std::vector<double> error(N, 0.0);
    double actualPrecisionValue = 1.0;
    int iteration = 0;

    while(actualPrecisionValue >= precision) {
        x_new[0] = b[0];
        x_new[N-1] =  (1/main[N-1]) * (b[N-1] - lower[N-2] * x_new[N-2]);

        for(int i=1; i<N-1; i++)    x_new[i] = (1/main[i]) * (b[i] - lower[i-1] * x_new[i-1] - upper[i] * x[i+1]);

        error = errorResult(x, x_new);
        actualPrecisionValue = errorNorm(error);
        x = x_new;
        iteration++;
    }
    //std::cout << iteration << std::endl;
    return x;
}
std::vector<double> successive_OverRelaxation_Method (double *lower, double *main, double *upper, double *b, double precision) {
    double w_optimal = 2.0 / (1.0 + sin(M_PI * (1.0 / (N + 1))));
    std::vector<double> x(N, 0.0);
    std::vector<double> x_new(N, 0.0);
    std::vector<double> error(N, 0.0);
    double actualPrecisionValue = 1.0;
    int iteration = 0;
    while(actualPrecisionValue >= precision) {
        x_new[0] = ((1.0 - w_optimal) * x[0] + (w_optimal * b[0]));
        for(int i=1; i<N-1; i++) {
            double x_new_Gauss = ((1.0 / main[i]) * (b[i] - lower[i-1] * x_new[i-1] - upper[i] * x[i+1]));
            x_new[i] = ((1.0 - w_optimal) * x[i]) + w_optimal * x_new_Gauss;
        }
        x_new[N-1] = ((1.0 - w_optimal) * x[N-1]) + ((w_optimal / main[N-1]) * (b[N-1] - lower[N-2] * x_new[N-2]));

        error = errorResult(x, x_new);
```

```
        actualPrecisionValue = errorNorm(error);
        x = x_new;
        iteration++;
    }
    //std::cout << iteration << std::endl;
    return x;
}
std::vector<double>relaxation_Richardson_Method(double A[][3], double *b, double precision) {
    std::vector<double> x(3, 1.0);
    std::vector<double> x_new(3, 0.0);
    std::vector<double> tempValues(3, 0.0);
    std::vector<double> error(3, 0.0);
    double omega = 0.25;
    double actualPrecisionValue = 1.0;
    int iteration = 0;

    while(actualPrecisionValue >= precision) {
        for(int i=0; i<3; i++)     tempValues[i] = b[i] - ((A[i][0] * x[0]) + (A[i][1] * x[1]) + (A[i][2] * x[2]));

        for(int i=0; i<3; i++)     x_new[i] = x[i] + (omega * tempValues[i]);

        for(int i=0; i<3; i++)     error[i] = x_new[i] - x[i];


        actualPrecisionValue = errorNorm(error);
        x = x_new;
        iteration++;
    }
    //std::cout << iteration << std::endl;
    return x;

}
void printResult(std::vector<double>wynik) {
    for(int i=0; i<wynik.size(); i++) {
       std::cout << std::fixed << std::setprecision(10) << (i*h) << " " << wynik[i] << std::endl;
    }
}

int main(int argc, char * argv[]) {
    double lower[N-1], main[N], upper[N-1], b[N];
    int arg = atoi(argv[1]);
    double relaxationA[3][3]{{4.0, -1.0, 0.0}, {-1.0, 4.0, -1.0}, {0.0, -1.0, 4.0}};
    double relaxationB[3]{2,6,2};
    std::vector<double>wynik;

    initMatrix(lower, main, upper, b);

    if(arg == 1) {
        wynik = relaxation_Richardson_Method(relaxationA, relaxationB, 1e-10);
        for(int i=0; i<wynik.size(); i++) {
            std::cout<< (i) << std::fixed << std::setprecision(10)  << " " << wynik[i] << std::endl;
        }
    } else {
        if(arg == 2) wynik = jacobi_Method(lower, main, upper, b, 1e-10);
        if(arg == 3) wynik = gauss_Seidel_Method(lower, main, upper, b, 1e-10);
        if(arg == 4) wynik = successive_OverRelaxation_Method(lower, main, upper, b, 1e-10);
        printResult(wynik);
    }

}
```

4 URUCHOMIENIE

Należy wpisać w konsoli polecenie

make run{numer}

{numer} – liczba z zakresu 1-4

1 – metoda Richardsona

2 – metoda Jacobiego

3 – metoda Gausa-Seidla

4 – metoda successive over-relaxation

W skład zestawu wchodzą:

- opracowanie
- pliki z wykresami (gauss_graph.pdf, jacobi_graph.pdf, sor_graph.pdf)
- pliki z wynikami (jacobi_method.dat, gauss_method.dat, sor_method.dat, richardson.dat)
- makefile
- kod programu