Implementacja sposobów rozwiązywania układów równań

Sebastian Kwaśniak

2024-04-22

Wstęp

W ramach projektu wymagane jest zaimplementowanie trzech sposobów rozwiązywania układów równań liniowych. Metody to: Jacobiego (iteracyjna), Gaussa-Seidla (iteracyjna), oraz faktoryzacja LU (bezpośrednia).

Metody te były już opisane w ramach wykładu oraz trzeciego laboratorium. We wszystkich trzech metodach, najpierw dzielimy macierz na macierz trójkątną dolną i górną, w metodach iteracyjnych potrzebujemy także diagonalną.

- Metoda LU opiera się na rozwiązaniu dwóch układów równań liniowych $Ly = b$ oraz $Ux = y$, gdzie $Ux = y$ to wektor pomocniczy, L to macierz trójkątna dolna, a U to górna.
- Metoda Jacobiego opiera się na rozbiciu macierzy, tak aby $A = D - L - U$, gdzie D to diagonalna. Z każdą następną iteracją obliczane są nowe przybliżenia rozwiązania.
- Metoda Gaussa-Seidla jest zbliżona, lecz korzysta z najnowszych wartości x^{k+1}

Opis rozwiązywanego równania macierzowego

W moim wypadku $a_1 = 13$, $f = 9$.

$$\begin{bmatrix} 13 & -1 & -1 & 0 & 0 & 0 & 0 & \dots & 0 \\ -1 & 13 & -1 & -1 & 0 & 0 & 0 & \dots & 0 \\ -1 & -1 & 13 & -1 & -1 & 0 & 0 & \dots & 0 \\ 0 & -1 & -1 & 13 & -1 & -1 & 0 & \dots & 0 \\ \vdots & \vdots & \vdots & \vdots & \vdots & \vdots & \vdots & \vdots & \vdots \\ 0 & 0 & 0 & 0 & 0 & 0 & -1 & -1 & 13 \end{bmatrix} \begin{bmatrix} x_1 \\ x_2 \\ x_3 \\ x_4 \\ \vdots \\ x_{907} \end{bmatrix} = \begin{bmatrix} \sin(9) \\ \sin(18) \\ \sin(27) \\ \sin(36) \\ \vdots \\ \sin(8163) \end{bmatrix}$$

Zadanie A

Mając wcześniej zadeklarowane 3 macierze A, x, oraz b, uzupełnianie ich następuje w ten sposób:

```
void exercise_A() {
    A.init(a1A, a2, a3);
    for (int i = 0; i < N; i++) {
        x[i][0] = 1.0;
        double elem = sin(i  * (f + 1));
        b[i][0] = elem;
    }
}
```

Gdzie `A.init` to ta funkcja:

```
void init(int a1, int a2, int a3) {
    for (int i = 0; i < rows - 2; i++) {
        for (int j = 0; j < cols - 2; j++) {
            if (i == j) {
                mat[i][j] = a1;
                mat[i + 1][j] = mat[i][j + 1] = a2;
```

```
                mat[i + 2][j] = mat[i][j + 2] = a3;
            }
        }
    }
    mat[rows - 2][cols - 2] = mat[rows - 1][cols - 1] = a1;
    mat[rows - 1][cols - 2] = mat[rows - 2][cols - 1] = a2;
}
```

Zadanie B

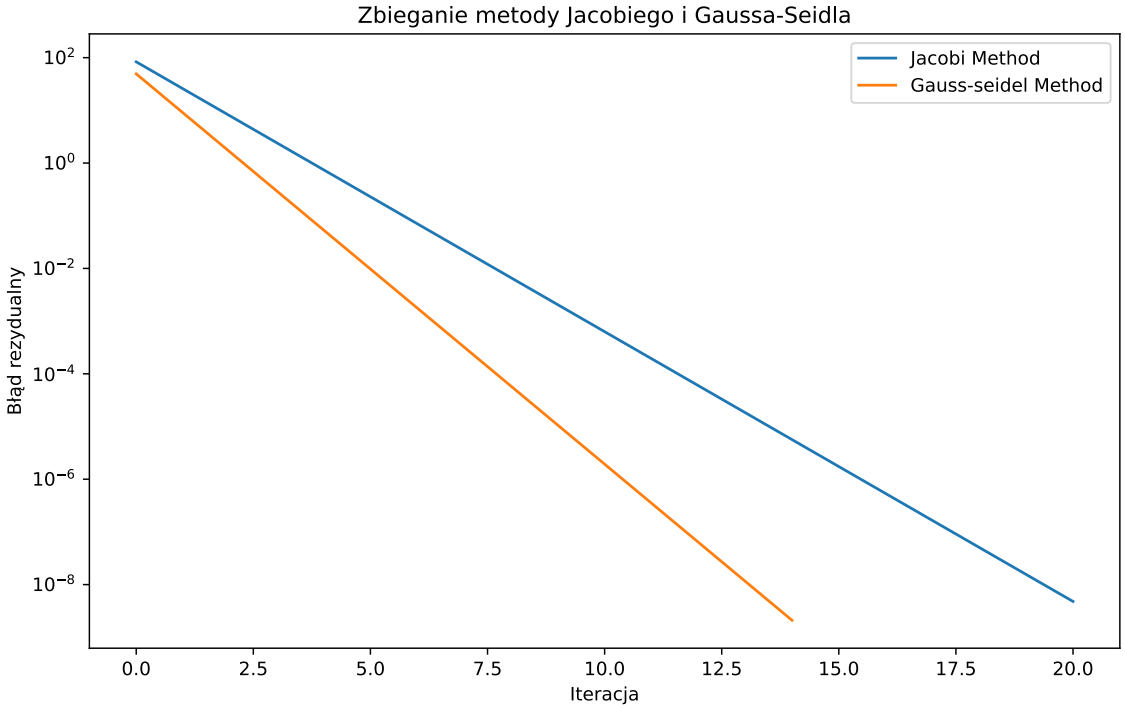
Jacobi wymagał 21 iteracji, zajęło mu to 0.197s, a Gauss-seidel 15 iteracji, czas wykonania 0.139s.

Kod do metody Jacobiego:

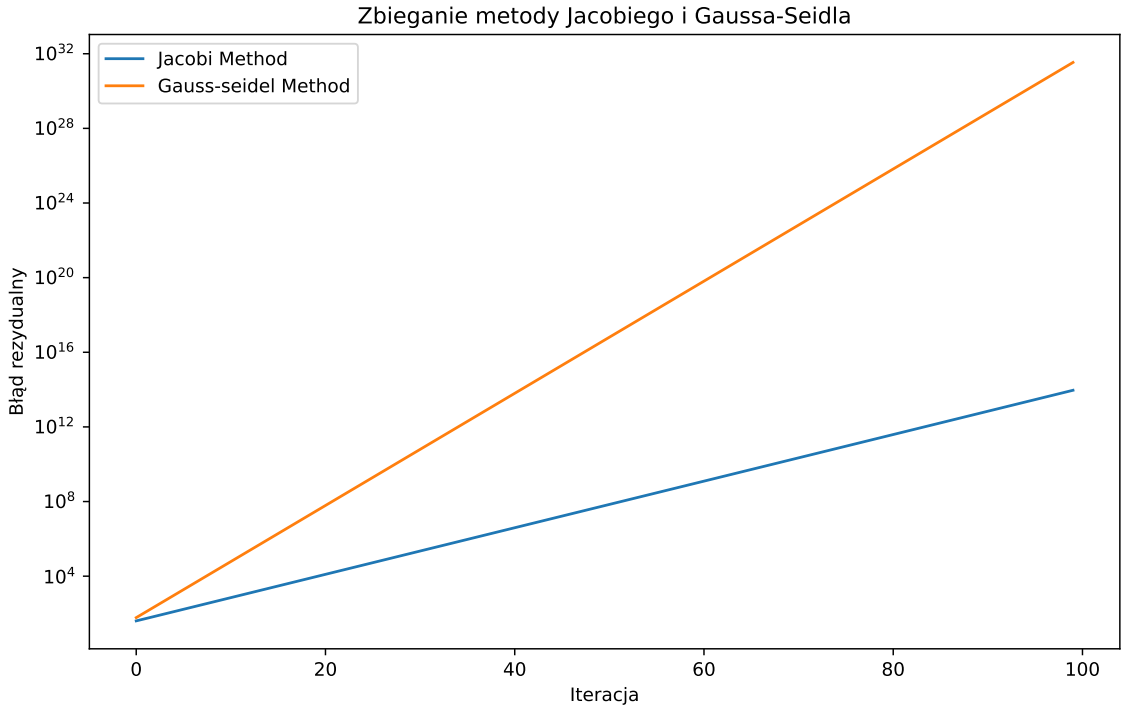
```
double Jacobi() {
    Matrix x_clone(x);
    for (int t = 1; ; t++) {
        for (int i = 0; i < N; i++) {
            double val = b[i][0];
            for (int k = 0; k < N; k++) {
                if (k != i)
                    val -= A[i][k] * x[k][0];
            }
            val /= A[i][i];
            x_clone[i][0] = val;
        }
        x = x_clone;
        auto norm = Matrix::norm(A * x - b);
        if (norm <= e || t >= 1000)
            return t;
    }
}
```

Kod do metody Gaussa-Seidla:

```
double Gauss_Seidl() {
    for (int t = 1; ; t++) {
        for (int i = 0; i < N; i++) {
            double val = b[i][0];
            for (int k = 0; k < N; k++) {
                if (k != i)
                    val -= A[i][k] * x[k][0];
            }
            val /= A[i][i];
            x[i][0] = val;
        }
        auto norm = Matrix::norm(A * x - b);
        if (norm <= e || t >= 1000)
            return t;
    }
}
```


Zbieganie metody Jacobiego i Gaussa-Seidla
Jacobi Method
Gauss-seidel Method
Błąd rezydualny
Iteracja
10^{2}
10^{0}
10^{-2}
10^{-4}
10^{-6}
10^{-8}
0.0
2.5
5.0
7.5
10.0
12.5
15.0
17.5
20.0

Zadanie C


Zbieganie metody Jacobiego i Gaussa-Seidla
Jacobi Method
Gauss-seidel Method
Błąd rezydualny
Iteracja
10^{32}
10^{28}
10^{24}
10^{20}
10^{16}
10^{12}
10^{8}
10^{4}
0
20
40
60
80
100

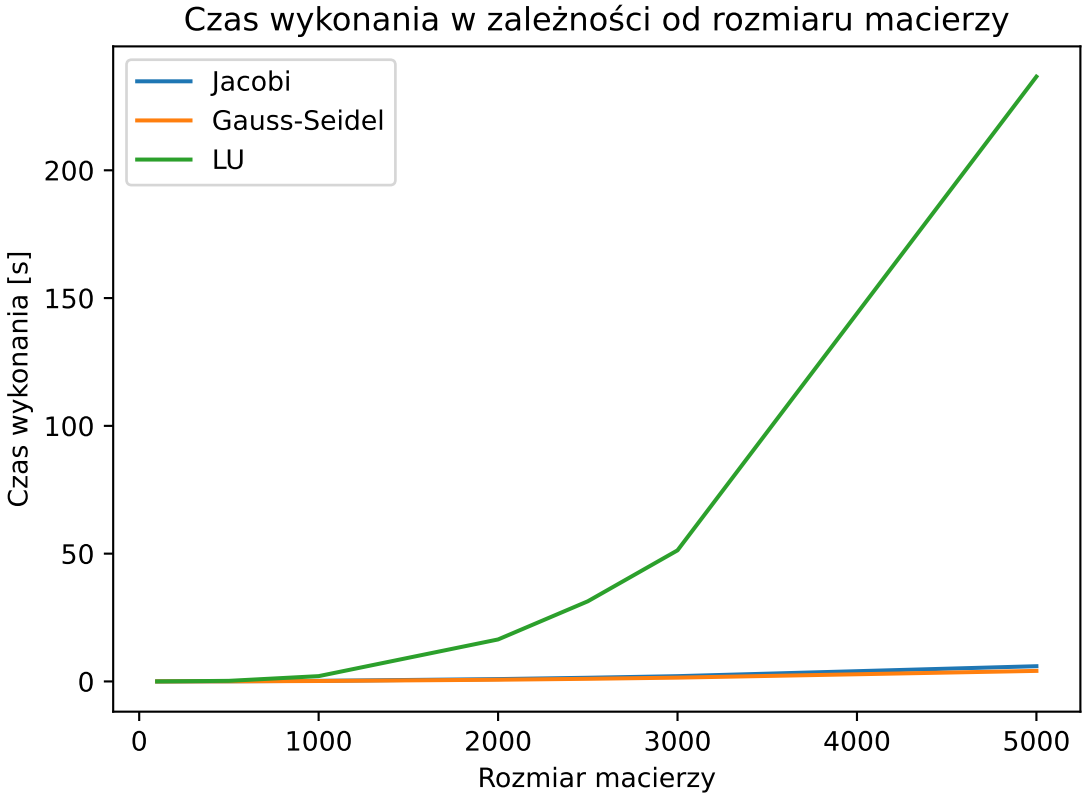
Zadanie D

Wykorzystany kod:

```
double LU() {
    Matrix L(N, N);
    Matrix U(N, N);
    for (int i = 0; i < N; i++)
        L[i][i] = 1.0;
    for (int j = 0; j < N; j++) {
        for (int i = 0; i <= j; i++) {
            U[i][j] += A[i][j];
            for (int k = 0; k <= i - 1; k++)
                U[i][j] -= L[i][k] * U[k][j];
        }
        for (int i = j+1; i < N; i++) {
            for (int k = 0; k <= j - 1; k++)
                L[i][j] -= L[i][k] * U[k][j];
            L[i][j] += A[i][j];
            L[i][j] /= U[j][j];
        }
    }
    Matrix y(N, 1);
    for (int i = 0; i < N; i++) {
        double val = b[i][0];
        for (int j = 0; j < i; j++) {
            if (j != i) val -= L[i][j] * y[j][0];
        }
        y[i][0] = val / L[i][i];
    }
    for (int i = N - 1; i >= 0; i--) {
        double val = y[i][0];
        for (int j = i; j < N; j++) {
            if (j != i) val -= U[i][j] * x[j][0];
        }
        x[i][0] = val / U[i][i];
    }
    return Matrix::norm(A * x - b);
}
```

Błąd rezydualny: $1.38612 * 10^{-13}$. Błąd ten jest bardzo niski, więc wynik jest bardzo dokładny. Poprzednie metody zawiodły, a tutaj mamy bardzo dobry wynik.

Zadanie E



Wnioski

Metoda faktoryzacji LU daje najdokładniejsze wyniki z tych trzech sposobów, ale jest znacząco wolniejsza i bardzo źle się skaluje. Metody iteracyjne mają jedną znaczącą wadę: nie zawsze się zbiegają. Trzeba wtedy użyć innej metody, która jest w stanie obliczyć dla nas wynik.