Metody Obliczeniowe w Nauce i Technice

Laboratorium 6 – raport

Mateusz Kocot

1. Wstęp

Rozpatrzę trzy iteracyjne metody rozwiązywania równań liniowych $\boldsymbol{AX} = \boldsymbol{B}$: metodę Jacobiego, metodę Gaussa-Seidela i metodę SOR. Metod tych można użyć pod warunkiem, że macierz $\boldsymbol{A}$ ma dominującą przekątną, tj. $\forall i: |a_{ii}| \geq \sum_{j \neq i} |a_{ij}|$. Wszystkie metody wykorzystują rozkład macierzy $\boldsymbol{A}$ na trzy macierze: diagonalną $\boldsymbol{D}$, ściśle dolną trójkątną $\boldsymbol{L}$ i ściśle górną trójkątną $\boldsymbol{U}$. Wówczas $\boldsymbol{A} = \boldsymbol{L} + \boldsymbol{D} + \boldsymbol{U}$. Każda z tych metod zaczyna od ustalenia wartości początkowych $\boldsymbol{X^{(0)}}$, a następnie w trakcie każdej iteracji poprawia dokładność rozwiązania. Kroki wykonywane w trakcie poszczególnych iteracji można zapisać macierzowo albo jako układ równań.

Metody będę testować na sześciu poniższych przykładach:

$$\begin{pmatrix} 2 & 1 \\ 5 & 7 \end{pmatrix} \boldsymbol{X_1} = \begin{pmatrix} 11 \\ 13 \end{pmatrix} \tag{1}$$

$$\begin{pmatrix} 4 & 1 & 2 \\ 0 & 2 & 1 \\ -4 & 2 & 8 \end{pmatrix} \boldsymbol{X_2} = \begin{pmatrix} -2 \\ -5 \\ 0 \end{pmatrix} \tag{2}$$

$$\begin{pmatrix} -37 & 9 & 22 \\ 2 & 14 & -9 \\ 1 & -3 & 5 \end{pmatrix} \boldsymbol{X_3} = \begin{pmatrix} 5.6 \\ -6.5 \\ 22.37 \end{pmatrix} \tag{3}$$

$$\begin{pmatrix} 10 & -1 & 2 & 0 \\ -1 & 11 & -1 & 3 \\ 2 & -1 & 10 & -1 \\ 0 & 3 & -1 & 8 \end{pmatrix} \boldsymbol{X_4} = \begin{pmatrix} 6 \\ 25 \\ -11 \\ 15 \end{pmatrix} \tag{4}$$

$$\begin{pmatrix} -9 & 1 & 5 & 1 \\ 1 & 11 & -3 & 4 \\ 3 & 6 & 22.997 & 4 \\ 2.1 & 2.2 & 2.3 & 32.22 \end{pmatrix} \boldsymbol{X_5} = \begin{pmatrix} 5 \\ -1 \\ 8 \\ 2 \end{pmatrix} \tag{5}$$

$$\begin{pmatrix} 13 & 8 & -1 & 1 & -1 \\ 2 & 20 & 7 & -1 & 1 \\ 4 & 22 & 36.6 & -3 & 2 \\ 1 & -5 & 0 & -19.91 & 1 \\ 3 & 25 & -2 & -3.1 & 52 \end{pmatrix} \boldsymbol{X_6} = \begin{pmatrix} 3.1 \\ 28 \\ 0 \\ -6.45 \\ 5.46 \end{pmatrix} \tag{6}$$

Należy oczywiście wybrać maksymalną różnicę dwóch liczb, dla której wciąż liczby te uznamy za równe. Ja wybiorę $EPS = 10^{-6}$. Z taką też dokładnością będę zapisywał wszystkie liczby.

Z wykorzystaniem pakietu *NumPy* w języku *Python* obliczono rozwiązania powyższych układów. Dzięki temu będzie można później wyznaczyć dokładność poszczególnych iteracji.

$$\boldsymbol{X_1} = \begin{pmatrix} 7.111111 \\ -3.222222 \end{pmatrix}$$

$$\boldsymbol{X_2} = \begin{pmatrix} -0.117647 \\ -2.823529 \\ 0.647059 \end{pmatrix}$$

$$\boldsymbol{X_3} = \begin{pmatrix} 3.514818 \\ 2.37322 \\ 5.194968 \end{pmatrix}$$

$$\boldsymbol{X_4} = \begin{pmatrix} 1 \\ 2 \\ -1 \\ 1 \end{pmatrix}$$

$$\boldsymbol{X_5} = \begin{pmatrix} -0.33776 \\ 0.022211 \\ 0.376450 \\ 0.055698 \end{pmatrix}$$

$$\boldsymbol{X_6} = \begin{pmatrix} -1.036275 \\ 1.878502 \\ -0.992324 \\ -0.239557 \\ -0.790789 \end{pmatrix}$$

W trakcie implementacji wszystkich metod korzystano z klasy *Matrix*, będącej zmodyfikowaną wersją klasy *AGHMatrix* z laboratorium nr 2.

2. Metoda Jacobiego

W tej metodzie, krok iteracji wygląda następująco:

$$\boldsymbol{X^{(k+1)} = D^{-1}\left(B - (L + U)X^{(k)}\right)}$$

Dla współrzędnych:

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

Zaimplementowano funkcję *jacobi* realizującą metodę Jacobiego. Funkcja oprócz macierzy $\boldsymbol{A}$ i $\boldsymbol{B}$ oraz liczby iteracji *iters* przyjmuje także macierz służącą do zainicjalizowania wartości początkowych rozwiązania. Przeładowana wersja tej funkcji przyjmuje jedną liczbę, którą zostanie zainicjalizowana każda współrzędna rozwiązania; domyślnie jest to 0. Funkcja zwraca macierz $\boldsymbol{X}$ obliczonych rozwiązań.

```cpp
template<typename T>
Matrix<double> jacobi(const Matrix<T> &A, const Matrix<T> &B, int iters,
                      const Matrix<double> &init)
{
    if (!A.isDiagonallyDominant())
        throw std::invalid_argument("matrix is not diagonally dominant");

    int n = A.getRows();

    Matrix<double> X = init;
    Matrix<double> Xprev;

    for (int it = 1; it <= iters; it++)
    {
        Xprev = X;
        for (int i = 0; i < n; i++)
        {
            double sum = 0;
            for (int j = 0; j < n; j++)
            {
                if (j == i)
                    continue;
                sum += A(i, j) * Xprev(j);
            }
            X(i) = (B(i) - sum) / A(i, i);
        }
    }

    return X;
}

template<typename T>
Matrix<double> jacobi(const Matrix<T> &A, const Matrix<T> &B, int iters,
                      double init = 0)
{
    Matrix<double> X(A.getRows(), 1, init);
    return jacobi(A, B, iters, X);
}
```

Funkcja ta wykonuje metodę Jacobiego bez operacji na macierzach – liczy po prostu kolejne współrzędne. Uznałem, że będzie to czytelniejsze oraz obędzie się potrzeby wykorzystania dodatkowej pamięci, którą pochłonęłyby macierze.

Funkcję przetestowano na testowych układach równań. Liczbę iteracji wymaganą do osiągnięcia założonej dokładności przedstawiono w poniższej tabeli.

nr równania	1	2	3	4	5	6
liczba iteracji	31	18	30	17	13	24

3. Metoda Gaussa-Seidela

Metoda ta jest bardzo podobna do poprzedniej. Kod wygląda niemal identycznie. Jedyną różnicą jest to, że licząc rozwiązania w trakcie danej iteracji, wykorzystywane są także rozwiązania policzone już przy okazji tej iteracji. Wzory na kolejne kroki wyglądają następująco:

$$\boldsymbol{X^{(k+1)} = (L + D)^{-1}\left(B - UX^{(k)}\right)}$$

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

Zaimplementowano funkcję *gaussSeidel* realizującą metodę Gaussa-Seidela. Argumenty oraz zwracana wartość są analogiczne jak w poprzedniej metodzie.

```
template<typename T>
Matrix<double> gaussSeidel(const Matrix<T> &A, const Matrix<T> &B, int iters,
                           const Matrix<double> &init)
{
    if (!A.isDiagonallyDominant())
        throw std::invalid_argument("matrix is not diagonally dominant");

    int n = A.getRows();

    Matrix<double> X = init;

    for (int it = 0; it < iters; it++)
    {
        for (int i = 0; i < n; i++)
        {
            double sum = 0;
            for (int j = 0; j < n; j++)
            {
                if (i == j)
                    continue;
                sum += A(i, j) * X(j, 0);
            }
            X(i) = (B(i) - sum) / A(i, i);
        }
    }
    return X;
}

template<typename T>
Matrix<double> gaussSeidel(const Matrix<T> &A, const Matrix<T> &B, int iters,
                           double init = 0)
{
    Matrix<double> X(A.getRows(), 1, init);
    return gaussSeidel(A, B, iters, X);
}
```

Implementacja ta jest nawet prostsza od implementacji metody Jacobiego. Nie musimy już trzymać macierzy rozwiązań z poprzedniej iteracji (*Xprev)*; cały czas do obliczeń można wykorzystywać jedną macierz (*X)*.

Funkcję przetestowano na testowych układach równań. Liczbę iteracji wymaganą do osiągnięcia założonej dokładności przedstawiono w poniższej tabeli.

nr równania	1	2	3	4	5	6
liczba iteracji	15	12	10	7	8	10

4. Metoda SOR (Successive Over Relaxation)

Metoda SOR wprowadza modyfikację do metody Gaussa-Seidela. Nowe przybliżenie liczone jest jako kombinacja przybliżenia z poprzedniej iteracji $\boldsymbol{X^{(k)}}$ oraz nowego przybliżenia $\boldsymbol{X^{\sim(k+1)}}$ (obliczanego metodą Gaussa-Seidela). Wzór wygląda następująco.

$$x_i^{(k+1)} = (1-\omega)x_i^{(k)} + \omega x_i^{\sim(k+1)}$$

Stała ω nazywana jest współczynnikiem relaksacji i powinno spełnione być $\omega \in (0,2)$.

Pełne wzory przedstawiono poniżej:

$$\boldsymbol{X^{(k+1)} = (D + \omega L)^{-1}\left(\omega B - [\omega U + (\omega - 1)D]X^{(k)}\right)}$$

$$x_i^{(k+1)} = (1-\omega)x_i^{(k)} + \frac{\omega}{a_{ii}}\left(b_i - \sum_{j<i} a_{ij}x_j^{(k+1)} - \sum_{j>i} a_{ij}x_j^{(k)}\right)$$

Zaimplementowano funkcję *sor* realizującą metodę SOR. Funkcja oprócz argumentów wymienianych w poprzednich przypadkach przyjmuje także współczynnik relaksacji *omega*.

```cpp
template<typename T>
Matrix<double> sor(const Matrix<T> &A, const Matrix<T> &B, double omega,
                   int iters, const Matrix<double> &init)
{
    if (!A.isDiagonallyDominant())
        throw std::invalid_argument("matrix is not diagonally dominant");

    int n = A.getRows();

    Matrix<double> X = init;

    for (int it = 0; it < iters; it++)
    {
        for (int i = 0; i < n; i++)
        {
            double sum = 0;
            for (int j = 0; j < n; j++)
            {
                if (i == j)
                    continue;
                sum += A(i, j) * X(j, 0);
            }
            X(i) = (1 - omega) * X(i) + omega * (B(i) - sum) / A(i, i);
        }
    }
    return X;
}

template<typename T>
Matrix<double> sor(const Matrix<T> &A, const Matrix<T> &B, double omega,
                   int iters, double init = 0)
{
    Matrix<double> X(A.getRows(), 1, init);
    return sor(A, B, omega, iters, X);
}
```

Jak widać, implementacja tej metody tylko nieznacznie różni się od funkcji *gaussSeidel*.

Niestety, nie istnieje sposób wyznaczenia ω takiego, by szybkość zbieżności była największa w każdym przypadku. Wyznaczono więc optymalne ω z zakresu $(0,2)$ (próbkowanie co 0.01) oraz minimalną liczbę iteracji potrzebną do osiągnięcia wymaganej dokładności. Wyniki zamieszczono w poniższej tabeli.

nr równania	1	2	3	4	5	6
liczba iteracji	7	11	10	6	6	8
optymalne ω	1.13	0.93	0.99	1.02	0.94	1.02

5. Porównanie teoretyczne

Najprostszą metodą wydaje się być metoda Jacobiego. Wykorzystuje ona przybliżenia rozwiązań obliczone w trakcie poprzedniej iteracji do obliczenia aktualnego, dokładniejszego przybliżenia. Zależność ta jest spełniona, gdy macierz $\boldsymbol{A}$ ma dominującą przekątną. W trakcie działania algorytmu, by nie nadpisać informacji o poprzednim przybliżeniu, należy je tymczasowo gdzieś zapisać, co sprawia, że wymagana jest większa ilość pamięci. W notacji asymptotycznej jest to jednak wciąż $O(n)$.

Metoda Gaussa-Seidela w sprytny sposób rozwiązuje ten problem. Nie ma potrzeby bowiem wykorzystywania części przybliżeń z poprzedniej iteracji, skoro obliczono już dokładniejsze przybliżenia w trakcie trwania aktualnej iteracji. Jeśli więc liczymy współrzędną o indeksie i, wykorzystamy i $(0,1,\dots,i-1)$ współrzędnych z aktualnej iteracji oraz $n-i-1$ $(i+1,\dots,n-1)$ współrzędnych z poprzedniej iteracji. W związku z tym, dodatkowa pamięć nie jest już potrzebna i sam algorytm jeszcze się upraszcza. Metoda ta, podobnie jak poprzednia, jest zbieżna, gdy macierz $\boldsymbol{A}$ ma dominującą przekątną. Będzie ona jednak zbieżna także dla macierzy symetrycznej lub dodatnio określonej. Dodatkowo, do otrzymania wymaganej dokładności powinna wystarczyć mniejsza liczba iteracji niż w przypadku metody Jacobiego.

Metodę Gaussa-Seidela można jeszcze rozwinąć, wprowadzając współczynnik relaksacji ω i otrzymując metodę SOR. Przybliżenia wyznacza się tu jako kombinację liniową (z wagami określonymi przez ω) przybliżeń z poprzedniej iteracji oraz aktualnych przybliżeń obliczonych metodą Gaussa-Seidela. Można więc lepiej dostosować sposób postępowania do danego układu. Niestety, nie można wyznaczyć takiego ω, że będzie optymalnie dla każdego przypadku. Dysponując jednak odpowiednim współczynnikiem relaksacji, można jeszcze przyspieszyć szybkość zbieżności i zmniejszyć liczbę iteracji potrzebnych do uzyskania określonej dokładności. Metoda ta ma takie same warunki zbieżności jak metoda Gaussa-Seidela.

Warto jeszcze zaznaczyć, że wszystkie metody polegają na odpowiednim przedstawieniu macierzy $\boldsymbol{A}$. Potrzebna jest część łatwo odwracalna $\boldsymbol{M}$ oraz reszta $\boldsymbol{Z} = \boldsymbol{M} - \boldsymbol{A}$. Części te buduje się poprzez odpowiednie kombinacje macierzy $\boldsymbol{B}$ oraz macierzy $\boldsymbol{L}$, $\boldsymbol{D}$ i $\boldsymbol{U}$, na które, według wcześniejszego opisu, dzielona jest macierz $\boldsymbol{A}$. Ważnym faktem, który wykorzystałem w trakcie implementacji, jest to, że dzięki odpowiednim postaciom macierzy odwracanych, rozwiązania można liczyć także sekwencyjnie (tak jak zwykły układ równań) bez powtarzania obliczeń.

6. Porównanie praktyczne

Minimalne liczby iteracji dla poszczególnych metod i układów przedstawiono poniżej. Dla metody SOR podano także optymalną wartość parametru ω.

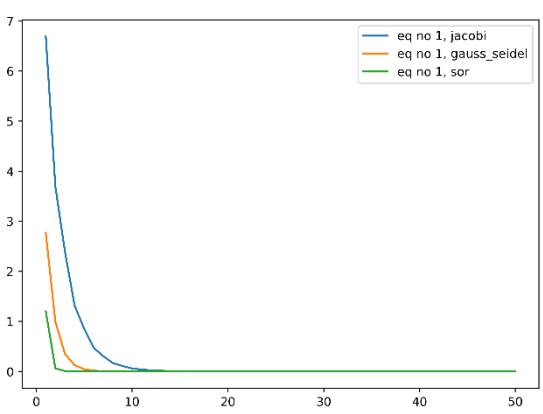
	nr równania	1	2	3	4	5	6
metoda Jacobiego	liczba iteracji	31	18	30	17	13	24
metoda Gaussa-Seidela	liczba iteracji	15	12	10	7	8	10
metoda SOR	liczba iteracji	7	11	10	6	6	8
	optymalne ω	1.13	0.93	0.99	1.02	0.94	1.02

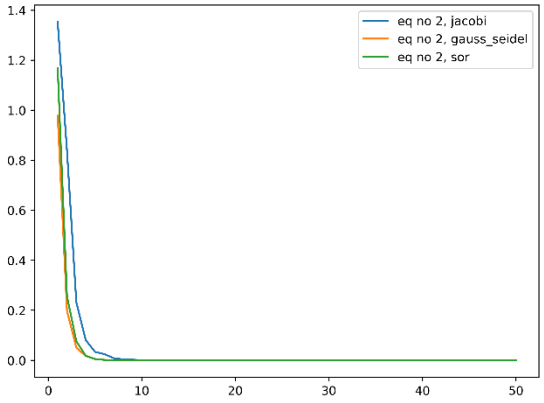
Metoda Gaussa-Seidela mimo tylko niewielkiej zmiany (lub nawet uproszczenia) w stosunku do metody Jacobiego oblicza rozwiązanie z dokładnością $EPS = 10^{-6}$ dużo szybciej. Średnio jest to ok. 2 razy szybciej. Dla 3 równania jest to nawet 3 razy szybciej, ale już dla 2 – 1.5 razy szybciej. W związku z tym

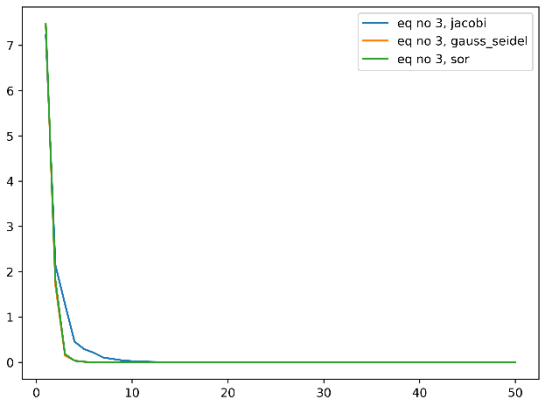
oraz mniejszym narzutem pamięciowym, metoda Gaussa-Seidela jest pod każdym względem lepsza od metody Jacobiego.

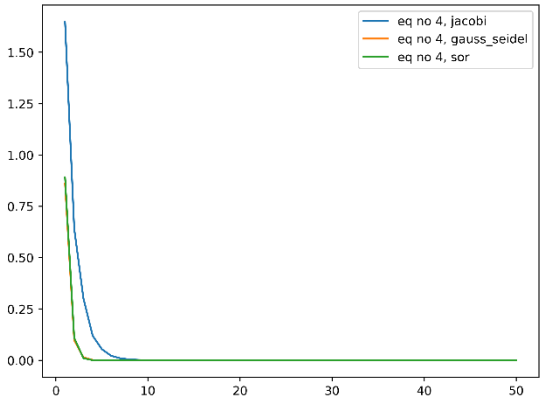
Nieco problematyczna jest metoda SOR. Co prawda, przyspiesza ona proces znalezienia dokładnego rozwiązania, natomiast w większości przypadków przyspiesza nieznacznie (albo wcale dla układu nr 3). Tylko dla układu nr 1, który jest układem zaledwie dwóch zmiennych, przyspiesza ona proces znalezienia dokładnego rozwiązania znacznie. Zastanawiające jest więc, czy opłaca się w ogóle stosować tę metodę. Innym problemem jest znalezienie optymalnego parametru ω. Nie udało mi się znaleźć takiego ω, żeby dla każdego z układów metoda SOR działała najszybciej. Zapewne więc zastosowanie tej metody warto ograniczyć do układów, dla których możemy oszacować optymalną wartość parametru relaksacji i mocno przyspieszyć poszukiwanie rozwiązania.

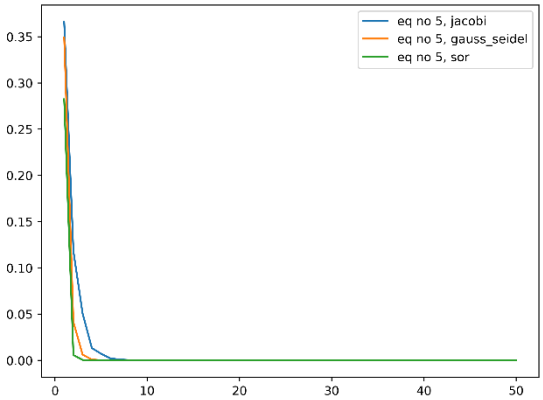
Porównam teraz na wykresach tempo zbiegania do rozwiązania. Wykresy utworzono za pomocą pakietu *Matplotlib* w języku *Python*. Wykresy są funkcją wielkości błędu od numeru iteracji. Wielkość błędu liczona jest jako suma wartości bezwzględnych różnic rozwiązań przybliżonych i rozwiązań prawdziwych (suma po wszystkich współrzędnych). Wykresy dla wszystkich układów testowych, dla liczby iteracji od 1 do 50, przedstawiono poniżej.

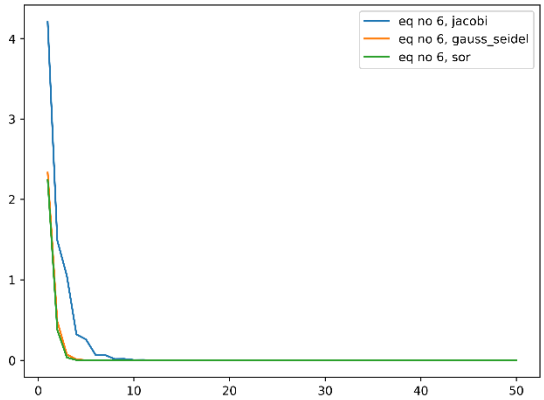












Zgodnie z przypuszczeniami metoda Gaussa-Seidela zbiega do prawdziwego rozwiązania dużo szybciej niż metoda Jacobiego w każdym przypadku. Ciekawsza jest relacja między metodą Gaussa-Seidela a metodą SOR. W większości przypadków zielona linia (SOR) jest lekko pod linią żółtą (Gauss-Seidel). W kilku przypadkach krzywe te niemal się pokrywają. Dzieje się tak mimo zastosowania optymalnego parametru ω (tak jak poprzednio, szukany co 0.01). Potwierdza to moje wątpliwości co do stosowania metody SOR.

Na wykresach można także zaobserwować, że wykresy mają podobny, hiperboliczny kształt. Fakt ten jest zgodny z oczekiwaniami, gdyż wszystkie metody działają w bardzo podobny sposób.