Laboratorium 6: Iteracyjne metody rozwiązywania równań liniowych

Metody Obliczeniowe w Nauce i Technice

Wiktor Tarsa

1. Układy równań.

Poniżej zamieszczam układy równań, na których będę testował implementowane metody.
Dla każdego z układów wyznaczyłem rozwiązanie korzystając z biblioteki numpy języka python:

1) $2x + y = 11$
   $5x + 7y = 13$

```
[[ 7.11111111]
 [-3.22222222]]
```

2) $10x - y + 2z = 6$
   $-x + 11y - z + 3t = 25$
   $2x - y + 10z - t = (-11)$
   $3y - z + 8t = 15$

```
[[ 1.]
 [ 2.]
 [-1.]
 [ 1.]]
```

3) $16x + 3y = 11$
   $7x - 11y = 13$

```
[[ 0.81218274]
 [-0.66497462]]
```

4) $4x - y - z = 3$
   $-2x + 6y + z = 9$
   $-x + y + 7z = -6$

```
[[ 1.]
 [ 2.]
 [-1.]]
```

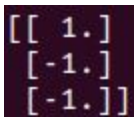
5) $5x - y + 2z = 12$
   $3x + 8y - 2y = -25$
   $x + y + 4z = 6$

```
[[ 1.]
 [-3.]
 [ 2.]]
```

6) $20x + y - 2z = 17$
   $3x + 20y - z = -18$
   $2x - 3y + 20z = 25$

```
[[ 1.]
 [-1.]
 [ 1.]]
```

7) $4x + y + z = 2$
$x + 5y + 2z = -6$
$x + 2y + 3z = -4$



2. Metoda Jacobiego.

Zaimplementowałem funkcję, która rozwiązuje układ równań liniowych korzystając z metody Jacobiego. Implementacja opiera się na wzorze podanym w instrukcji. W metodzie Jacobiego można uzyskać określoną dokładność poprzez wykonanie odpowiedniej ilości iteracji algorytmu. Początkowo założyłem, że liczba iteracji k = 25 będzie wystarczająca. Rozwiązania wyznaczałem z dokładnością do czterech miejsc po przecinku. W przypadku ostatniego testowanego układu potrzeba było zwiększyć ilość operacji do 27 aby otrzymać dokładny wynik.

```
std::vector<double> jacobiMethod(std::vector<std::vector<double>> matrix,
int iterations){
   int size = matrix.size();
   std::vector<double> X;
   X.resize(size, 0.0);
   std::vector<double> tmp;
   tmp.resize(size, 0.0);
   std::vector<std::vector<double>> LU = matrixAddition(getLMatrix(matrix),
getUMatrix(matrix));

   for (int i = 0; i < size; i++) {
       for (int j = 0; j < size; j++) {
           if (i == j) {
               LU[i][j] = 0;
           } else {
               LU[i][j] = -(matrix[i][j] / matrix[i][i]);
           }
       }
   }

   for (int k = 0; k < iterations; k++) {
       for (int i = 0; i < size; i++) {
           tmp[i] = (1/matrix[i][i]) * matrix[i][size];
           for (int j = 0; j < size; j++) {
               tmp[i] += LU[i][j] * X[j];
           }
       }
       for (int i = 0; i < size; i++) {
```

```
            X[i] = tmp[i];
        }
    }
    return X;
}
```

Funkcja używa wielu funkcji pomocniczych takich jak getUMatrix, matrixAddition itp. Ze względu na prostotę tych funkcji zdecydowałem się nie umieszczać ich kodu źródłowego w sprawozdaniu. Dokładna implementacja znajduje się w pliku main.cpp dołączonym do tego sprawozdania.

Funkcja została przetestowana dla wszystkich siedmiu układów równań i jej wyniki są zgodne z wynikami wyznaczonymi przez funkcje biblioteki numpy.

3. Metoda Gaussa-Seidela.

Implementacja tej metody jest podobna do metody Jacobiego. Różnica polega na korzystaniu z aktualnych współrzędnych wyliczonych w danym kroku. Jeśli w kroku k+1 obliczyliśmy już j-tą współrzędną i jesteśmy pewni, że nie ulegnie ona zmianie, możemy użyć jej do wyznaczenia pozostałych współrzędnych(zamiast używać j-tej współrzędnej wyliczonej w kroku k).

```
std::vector<double> gaussSeidelMethod(std::vector<std::vector<double>>
matrix, int iterations){
    int size = matrix.size();
    std::vector<double> X;
    X.resize(size, 0.0);
    std::vector<double> tmp;
    tmp.resize(size, 0.0);
    std::vector<std::vector<double>> LU = matrixAddition(getLMatrix(matrix),
getUMatrix(matrix));

    for (int i = 0; i < size; i++) {
        for (int j = 0; j < size; j++) {
            if (i == j) {
                LU[i][j] = 0;
            } else {
                LU[i][j] = -(matrix[i][j] / matrix[i][i]);
            }
        }
    }

    for (int k = 0; k < iterations; k++) {
        for (int i = 0; i < size; i++) {
            tmp[i] = (1 / matrix[i][i]) * matrix[i][size];
            for (int j = 0; j < size; j++) {
                if(j < i){
                    tmp[i] += LU[i][j] * tmp[j];
```

```
                } else if(j > i){
                    tmp[i] += LU[i][j] * X[j];
                }
            }
        }
        for (int i = 0; i < size; i++) {
            X[i] = tmp[i];
        }
    }
    return X;
}
```

Funkcja została przetestowana dla wszystkich siedmiu układów równań podanych na początku tego sprawozdania. Dla każdego z nich zwróciła takie same rozwiązania jak metoda Jacobiego i funkcja biblioteki numpy.

4. Metoda SOR (Successive Over Relaxation).

Metoda SOR jest udoskonaleniem metody Gaussa-Seidela. Został wprowadzony parametr relaksacji ω dzięki któremu można wyznaczać nowe przybliżenie poprzez kombinację poprzedniego i nowego przybliżenia. Odpowiedni dobór parametru ω powoduje przyśpieszenie zbieżności otrzymanych wyników.

Zmiany względem metody Gaussa-Seidela, jakie dokonałem w implementacji tej metody to dodanie parametru w do argumentów funkcji oraz modyfikacja ostatniej pętli funkcji. Implementacja wygląda następująco:

```
std::vector<double> SORMethod(std::vector<std::vector<double>> matrix, int
iterations, double w){
    int size = matrix.size();
    std::vector<double> X;
    X.resize(size, 0.0);
    std::vector<double> tmp;
    tmp.resize(size, 0.0);
    std::vector<std::vector<double>> LU = matrixAddition(getLMatrix(matrix),
getUMatrix(matrix));

    for (int i = 0; i < size; i++) {
        for (int j = 0; j < size; j++) {
            if (i == j) {
                LU[i][j] = 0;
            } else {
                LU[i][j] = -(matrix[i][j] / matrix[i][i]);
            }
        }
```

```
    }

    for (int k = 0; k < iterations; k++) {
        for (int i = 0; i < size; i++) {
            tmp[i] = (1 / matrix[i][i]) * matrix[i][size];
            for (int j = 0; j < size; j++) {
                if(j < i){
                    tmp[i] += LU[i][j] * tmp[j];
                } else if(j > i){
                    tmp[i] += LU[i][j] * X[j];
                }
            }
        }
        for (int i = 0; i < size; i++) {
            X[i] = w*tmp[i] + (1-w)*X[i];
        }
    }
    return X;
}
```

Przetestowałem metodę dla siedmiu układów równań. Tym razem zmieniałem parametr relaksacji. Dla 25 iteracji algorytmu i wartości ω z przedziału <0.5, 1.5> otrzymane wyniki są równe wynikom z poprzednich metod.

5. Porównanie teoretyczne metod.

Metoda Jacobiego jest najprostszą z trzech metod. Wymaga jednak, by macierz współczynników była dominująca. Jest to warunek konieczny uzyskania poprawnych wyników równania. W przeciwieństwie do metody Gaussa-Seidela, algorytm potrzebuje co najmniej dwie tablice o długości n (gdzie n to liczba niewiadomych) by wyznaczyć rozwiązanie. Jest tak dlatego, że do wyznaczenia j-tej współrzędnej z k+1 kroku potrzebujemy wszystkich współrzędnych z kroku poprzedniego. W metodzie Gaussa-Seidela potrzebujemy tylko tych współrzędnych z poprzedniego kroku, które mają większy indeks od obecnie wyznaczanej(ponieważ te o mniejszym indeksie zastępujemy współrzędnymi z tego samego kroku). Odpowiednia implementacja umożliwia zoptymalizowanie złożoności pamięciowej algorytmu do O(n), co może być szczególnie przydatne w przypadku operacji na dużej ilości danych.
Metoda Gaussa-Seidela podobnie jak metoda Jacobiego daje poprawne wyniki, gdy macierz współczynników jest dominująca. Można ją jednak zastosować także dla macierzy symetrycznej oraz dodatnio określonej.

Metoda SOR(Successive over-relaxation) jest tak naprawdę zmodyfikowaną metodą Gaussa-Seidela. Różnica polega na wprowadzeniu parametru relaksacji ω. Współrzędne obliczane w kroku k+1 są kombinacją współrzędnych z obecnego i poprzedniego kroku. Wprowadzenie parametru relaksacji ma na celu przyśpieszenie zbieżności metody Gaussa-Seidela.

6. Porównanie tempa zbiegania metod: Jacobiego, Gaussa-Seidela i SOR.

Aby porównać tempo zbiegania wszystkich trzech metod należy znać najlepszą wartość parametru relaksacji w metodzie SOR dla konkretnego równania. Do wyznaczenia tej wartości zaimplementowałem następującą funkcję:

```
double findRelaxationFactor(std::vector<std::vector<double>> matrix,
std::vector<double> X){
   double error = DBL_MAX;
   double relaxationFactor = -1.0;
   double interval = 0.01;
   int iterations = 0;
   for(int k = 1; k < 25; k++) {
       for (int i = 0; i < 200; i++) {
           double tmp_w = i * interval;
           double tmp_error = 0.0;
           std::vector<double> results = SORMethod(matrix, k, tmp_w);
           for (int j = 0; j < results.size(); j++) {
               tmp_error += std::abs(X[j] - results[j]);
           }
           if (tmp_error < error) {
               iterations = k;
               error = tmp_error;
               relaxationFactor = tmp_w;
           }
       }
   }
   printf("Liczba iteracji: %d\n", iterations);
   return relaxationFactor;
}
```

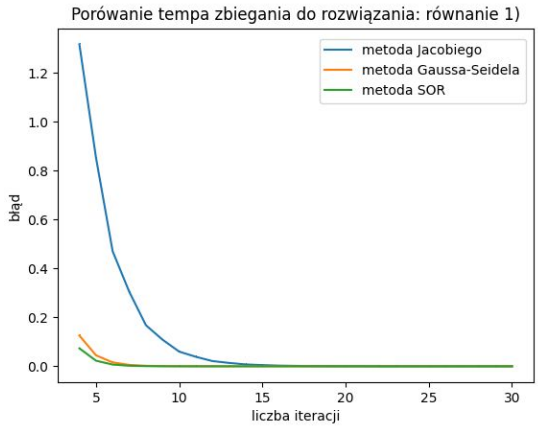
Funkcja wyznacza rozwiązanie metodą SOR dla 200 różnych wartości (z przedziału od 0 do 2, krok 0.01) i różnej liczby iteracji. Funkcja wypisuje liczbę iteracji w której znaleziono najmniejszy błąd(różnicę między wynikiem a wynikiem otrzymanym korzystając z biblioteki numpy) oraz zwraca wartość parametru relaksacji.

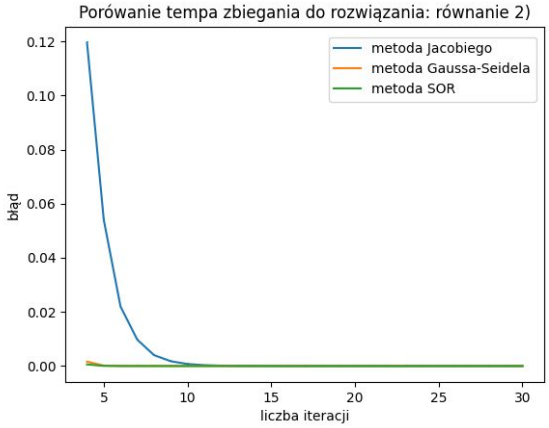
Wyznaczyłem wartość parametru dla testowych układów równań. Otrzymane wartości parametru relaksacji wahały się od 0,94 do 1,1 z liczbą wykonanych iteracji od 11 do 19.

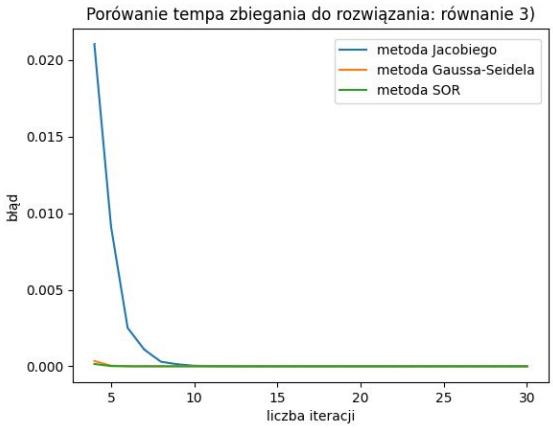
Aby porównać tempo zbiegania metod do rozwiązania zaimplementowałem funkcję, która oblicza sumę różnic bezwzględnych pomiędzy wyznaczonymi wartościami, a wartościami prawdziwymi. W tym celu użyłem wyników wyznaczonych przez bibliotekę numpy.

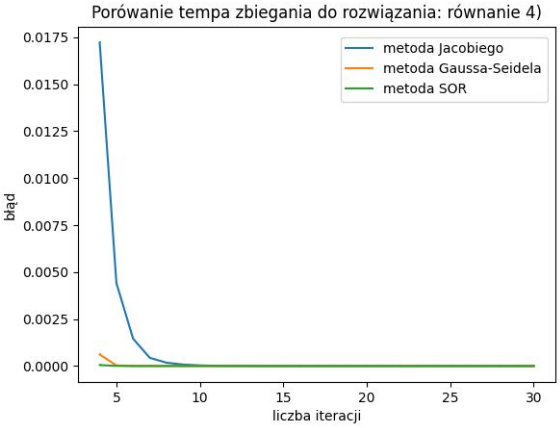
```
double findError(std::vector<std::vector<double>> matrix,
std::vector<double> solutions, int iterations){
   std::vector<double> X = jacobiMethod(matrix, iterations);
  // std::vector<double> X = gaussSeidelMethod(matrix, iterations);
  // std::vector<double> X = SORMethod(matrix, iterations, 1.03);

   double error = 0.0;
   for(int i = 0; i < X.size(); i++){
       error += std::abs(solutions[i]-X[i]);
   }
   return error;
}
```

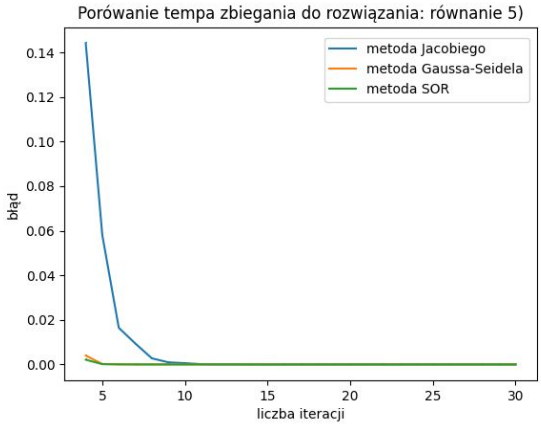
Następnie wygenerowałem sumy różnic dla każdej metody i dla każdego równania. Sumy generowałem od 4 do 30 iteracji.
Na przedstawionych poniżej wykresach numery równań są zgodne z numerami z punktu pierwszego tego sprawozdania.

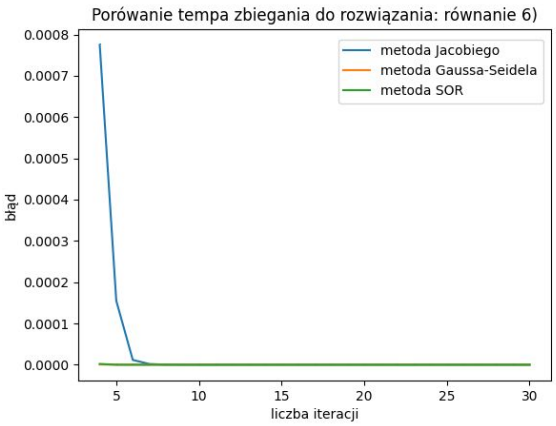


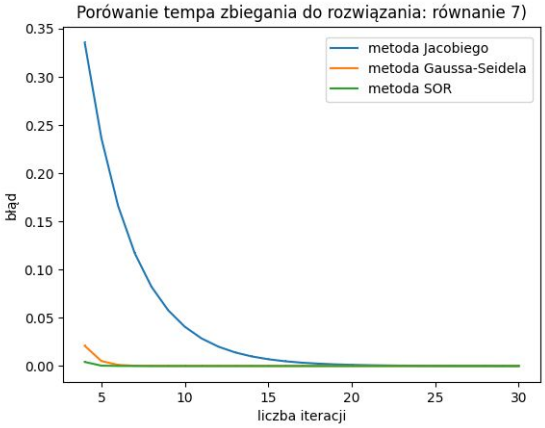












Na wszystkich wykresach widać, że metoda SOR zbiega do rozwiązania najszybciej. Najwolniej zbiega metoda Jacobiego.
Metoda Gaussa-Seidela jest tak naprawdę zoptymalizowaną metodą Jacobiego. Natomiast metoda SOR - to ulepszona metoda Gaussa-Seidela(bo metoda SOR z $\omega = 1$ to metoda Gaussa-Seidela). Powyższe wykresy potwierdzają, że optymalizowanie metod ma realny wpływ na szybkość zbiegania metody do rozwiązania, czyli ma też wpływ na dokładność wyniku. Używanie bardziej optymalnych metod przyśpiesza proces uzyskania wyniku.

Powyższe wykresy są też potwierdzeniem poprawności mojej implementacji.