Maciej Kozub

Laboratorium 6 – iteracyjne metody rozwiązywania równań liniowych

1. Metoda Jacobiego

Zaimplementowana funkcja rozwiązuje układ równań wykorzystując metodę Jacobiego opartą na wzorze podanym w instrukcji. Dokładność tej metody uzależniona jest od liczby wykonanych iteracji algorytmu.

```
<T>
AGHMatrix<T> AGHMatrix<T>::Jacoby(const AGHMatrix<T> &inputMatrix, unsigned iterations) const {

    AGHMatrix<double> X(inputMatrix);
    AGHMatrix<double> Xcopy(inputMatrix);

    if (X.get_cols() != 1) {
        std::cout << "input matrix must have one column";
        exit( _Code: -1);
    }
    if (this->rows != this->cols || matrix.size() != inputMatrix.get_rows()) {
        std::cout << "matrix must be square, and have as many rows as input matrix";
        exit( _Code: -1);
    }

    for (unsigned i = 0; i < iterations; i++) {
        for (unsigned row = 0; row < this->rows; row++) {
            double sum = 0;
            for (unsigned col = 0; col < this->cols; col++) {
                if (col != row) {
                    sum += matrix[row][col] * Xcopy(col,  col: 0);
                }
            }
            X(row,  col: 0) = (inputMatrix(row, 0) - sum) / matrix[row][row];
        }
        Xcopy = X;
    }
    return X;
}
```

W celu implementacji wykorzystałem klasę AGHMatrix, która zawiera funkcje pomocnicze, takie jak get_rows() czy konstruktory nowych macierzy.

Liczba iteracji wynosiła 20. Dla tej liczby iteracji metoda Jacobiego nie dała wyników dokładnych, były one obarczone niewielkim błędem.

Układy równań dla których została przetestowana metoda:

```
std::vector<std::vector<double>> init1a{{3.0,  -1.0, 1.0},
                                        {-1.0, 3.0,  -1.0},
                                        {1.0,  -1.0, 3.0}};

std::vector<std::vector<double>> init1b{{-1.0},
                                        {7.0},
                                        {-7.0}};

std::vector<std::vector<double>> sol1{{1.0},
                                      {2.0},
                                      {-2.0}};

std::vector<std::vector<double>> nit2a{{5.0, -3.0},
                                       {1.0, -2.0}};

std::vector<std::vector<double>> init2b{{21.0},
                                        {7.0}};

std::vector<std::vector<double>> sol2{{3.0},
                                      {-2.0}};
```

```
std::vector<std::vector<double>> init3a{{4.0,  -1.0, -0.2, 2.0},
                                        {-1.0, 5.0,  0.0,  -2.0},
                                        {0.2,  1.0,  10.0, -1.0},
                                        {0.0,  -2.0, -1.0, 4.0}};

std::vector<std::vector<double>> init3b{{21.6},
                                        {36.0},
                                        {-20.6},
                                        {-11.0}};

std::vector<std::vector<double>> sol3{{7.0},
                                      {9.0},
                                      {-3.0},
                                      {1.0}};

std::vector<std::vector<double>> init4a{{5.02, 2.01,  -0.98},
                                        {3.03, 6.95,  3.04},
                                        {1.01, -3.99, 5.98}};

std::vector<std::vector<double>> init4b{{2.05},
                                        {-1.02},
                                        {0.98}};

std::vector<std::vector<double>> sol4{{0.50774},
                                      {-0.31141},
                                      {-0.12966}};
```

```
std::vector<std::vector<double>> init5a{{1.0, 0.0, 0.0, 0.0, 0.0},
                                        {0.0, 1.0, 0.0, 0.0, 0.0},
                                        {0.0, 0.0, 1.0, 0.0, 0.0},
                                        {0.0, 0.0, 0.0, 1.0, 0.0},
                                        {0.0, 0.0, 0.0, 0.0, 1.0}};

std::vector<std::vector<double>> init5b{{1.0},
                                        {1.0},
                                        {1.0},
                                        {1.0},
                                        {1.0}};

std::vector<std::vector<double>> sol5{{1.0},
                                      {1.0},
                                      {1.0},
                                      {1.0},
                                      {1.0}};
```

Wyniki uzyskane metodą Jacobiego, dla kolejnych układów testowych:

```
Jacoby:
0.998797,
2.0012,
-2.0012,
```

```
Jacoby:
3.00011,
-1.99995,
```

```
Jacoby:
6.9993,
9.00059,
-2.99984,
0.999453,
```

```
Jacoby:
0.507743,
-0.311412,
-0.129653,
```

```
Jacoby:
1,
1,
1,
1,
1,
```

2. Metoda Gaussa-Seidela

Implementacja tej metody ponownie bazuje na instrukcji. Metoda jest podobna do metody Jacobiego, różnica polega na wykorzystywaniu aktualnie wyliczanych współrzędnych w danym kroku. Jeżeli k-ta wartość była już obliczona w n+1 kroku to możemy jej użyć do wyznaczenia pozostałych współrzędnych, bez konieczności używania k-tej współrzędnej wyliczonej w n-tym kroku.

```
<T>
AGHMatrix<T> AGHMatrix<T>::GaussSiedel(const AGHMatrix<T> &inputMatrix, unsigned iterations) const {

    AGHMatrix<double> X(inputMatrix);
    AGHMatrix<double> Xcopy(inputMatrix);

    if (X.get_cols() != 1) {
        std::cout << "input matrix must have one column";
        exit( _Code: -1);
    }
    if (this->rows != this->cols || matrix.size() != inputMatrix.get_rows()) {
        std::cout << "matrix must be square, and have as many rows as input matrix";
        exit( _Code: -1);
    }

    for (unsigned i = 0; i < iterations; i++) {
        for (unsigned row = 0; row < this->rows; row++) {
            double sum = 0;
            for (unsigned col = 0; col < this->cols; col++) {
                if (col > row) {
                    sum += matrix[row][col] * Xcopy(col,  col: 0);
                }
                if (col < row) {
                    sum += matrix[row][col] * X(col,  col: 0);
                }
            }
            X(row,  col: 0) = (inputMatrix(row, 0) - sum) / matrix[row][row];
        }
        Xcopy = X;
    }
    file.close();
    fileCounter++;
    return X;
}
```

Funkcja została przetestowana dla tych samych układów równań co metoda Jacobiego, również dla 20 iteracji.

Tym razem otrzymane wyniki były dokładne. Otrzymane rozwiązania:

```
Gauss-Siedel:
1,
2,
-2,
```

```
Gauss-Siedel:
3,
-2,
```

```
Gauss-Siedel:
7,
9,
-3,
1,
```

```
Gauss-Siedel:
0.507743,
-0.311411,
-0.129657,
```

```
Gauss-Siedel:
1,
1,
1,
1,
1,
```

3. Metoda SOR

Metoda ta jest ulepszoną metodą Gaussa-Seidela, został do niej dodany parametr ω, który umożliwia wyznaczanie nowego przybliżenia używając kombinacji poprzedniego i następnego przybliżenia. Parametr został ustawiony na 1,25.

```
<T>
AGHMatrix<T> AGHMatrix<T>::SOR(const AGHMatrix<T> &inputMatrix, unsigned iterations) const {

    double omega = 1.25;
    AGHMatrix<double> X(inputMatrix);
    AGHMatrix<double> Xcopy(inputMatrix);

    if (X.get_cols() != 1) {
        std::cout << "input matrix must have one column";
        exit( _Code: -1);
    }
    if (this->rows != this->cols || matrix.size() != inputMatrix.get_rows()) {
        std::cout << "matrix must be square, and have as many rows as input matrix";
        exit( _Code: -1);
    }

    for (unsigned i = 0; i < iterations; i++) {
        for (unsigned row = 0; row < this->rows; row++) {
            double sum = 0;
            for (unsigned col = 0; col < this->cols; col++) {
                if (col > row) {
                    sum += matrix[row][col] * Xcopy(col,  col: 0);
                }
                if (col < row) {
                    sum += matrix[row][col] * X(col,  col: 0);
                }
            }
            X(row,  col: 0) = (1 - omega) * Xcopy(row,  col: 0) + omega * (inputMatrix(row, 0) - sum) / matrix[row][row];
        }
        Xcopy = X;
    }
    file.close();
    fileCounter++;
    return X;
}
```

Funkcja została przetestowana dla tych samych układów równań co poprzednie metody.

Otrzymane wyniki są dokładniejsze od metody Jacobiego ale nie od metody Gaussa-Seidela.

Otrzymane wyniki:

```
SOR:
1,
2,
-2,
```

```
SOR:
3,
-2,
```

```
SOR:
7,
9,
-3,
1,
```

```
SOR:
0.508141,
-0.310629,
-0.128763,
```

```
SOR:
1,
1,
1,
1,
1,
```

4. Porównanie teoretyczne metod.

Metoda Jacobiego jest najprostszą z tych trzech metod. Wymaga ona aby macierz była dominująca. W tej metodzie do wyznaczenia i-tej współrzędnej w k+1 kroku potrzebujemy wszystkich współrzędnych z kroku poprzedniego.

W metodzie Gaussa-Seidela do wyznaczenia i-tej współrzędnej w k+1 kroku potrzebujemy tylko tych współrzędnych z kroku k, których indeks jest większy od obecnie wyznaczanego, inne zastępujemy współrzędnymi dopiero wyznaczanymi. Metoda wymaga aby macierz była dominująca, symetryczna lub dodatnio określona do prawidłowego wyznaczenia rozwiązania.

Metoda SOR (Successive over relaxation) jest zmodyfikowaną metodą Gaussa-Seidela, dzięki wprowadzeniu parametru ω, współrzędne obliczane w korku k+1 są kombinacją współrzędnych z obecnego i poprzedniego kroku. Przyspiesza to zbieżność algorytmu.

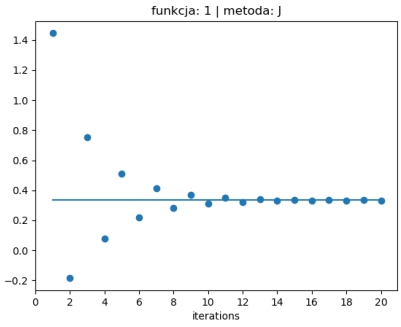
5. Porównanie tempa zbieżności metod.

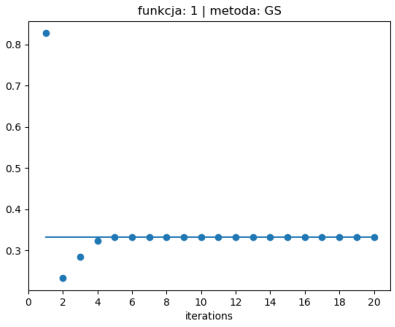
Aby zilustrować tempo zbieżności do implementacji każdej metody dodałem fragment kodu zapisujący do pliku średnią wyciągniętą z wartości będącymi rozwiązaniami układu po każdym przejściu iteracyjnym funkcji. Następnie w używając pythona utworzyłem wykresu ukazujące różnicę pomiędzy tą wartością, a wartością średniej uzyskanej z rzeczywistych rozwiązań danego równania.

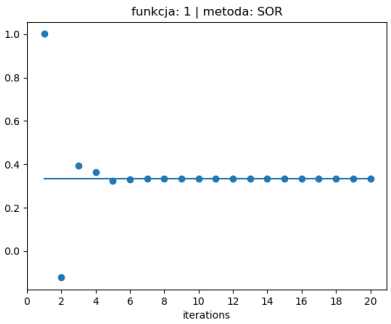
W pętli iterującej:

```
double sum = 0;
for (unsigned j = 0; j < this->rows; j++) {
    sum += X(j,  col: 0);
}
sum /= this->rows;
file << sum << std::endl;
```

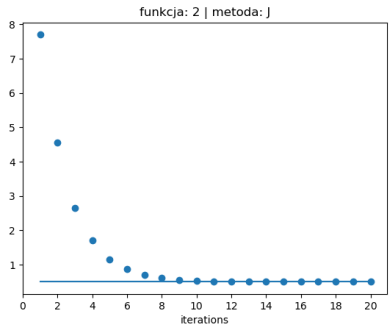
Zbieżność dla pierwszego testowanego układu:

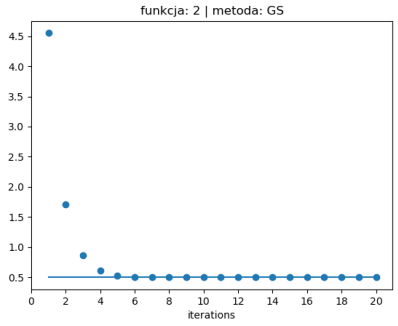


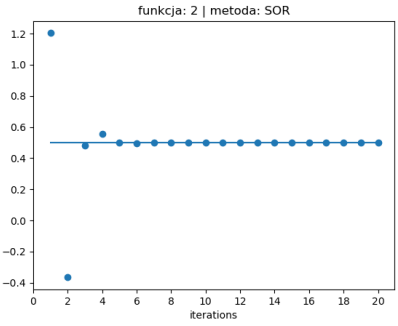




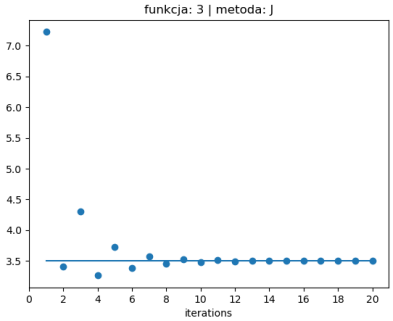
Zbieżność dla drugiego testowanego układu:

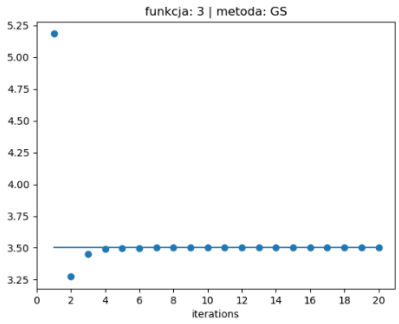


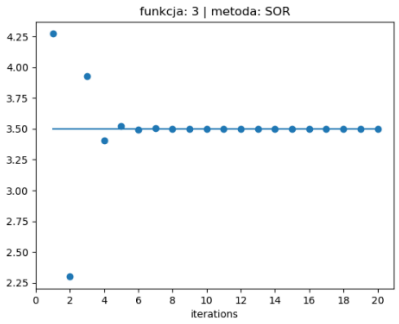




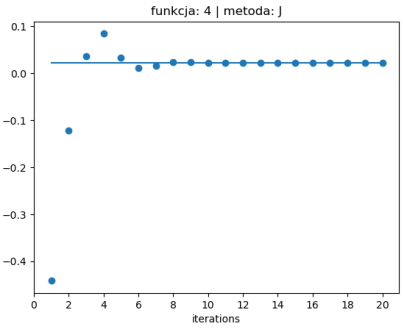
Zbieżność dla trzeciego testowanego układu:

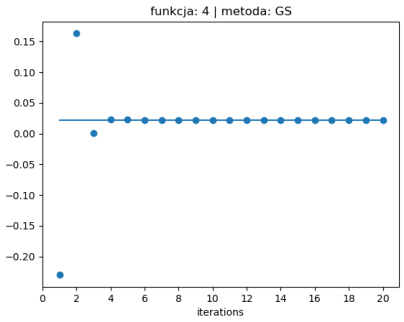


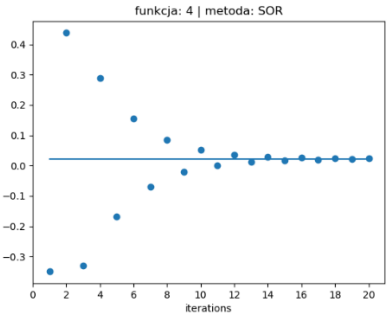




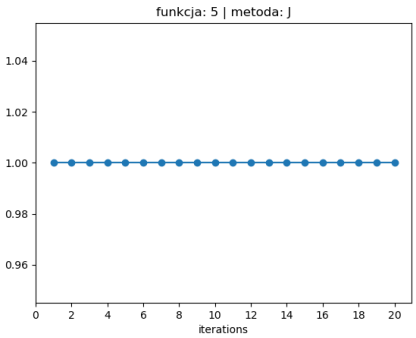
Zbieżność dla czwartego testowanego układu:

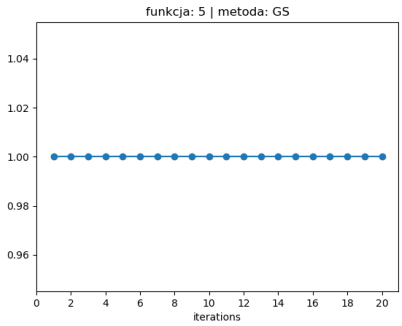


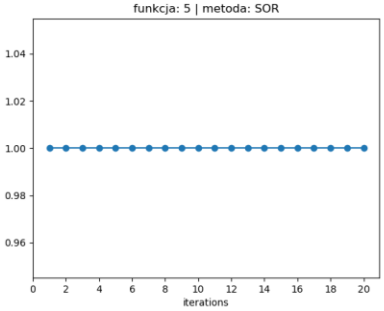




Zbieżność dla piątego testowanego układu:







Na wykresach widać że metoda Gaussa-Seidela i SOR są zdecydowanie szybciej zbiegające do rozwiązania niż metoda Jacobiego.