AGH

AKADEMIA GÓRNICZO-HUTNICZA
IM. STANISŁAWA STASZICA
W KRAKOWIE

Metody Obliczeniowe w Nauce i Technice:

Iteracyjne metody rozwiązywania równań liniowych

Laboratorium 6

Przemysław Lechowicz

1. Testowe układy równań:

W celu testowania implementowanych metod utworzyłem następujące układy równań:

a.

$$-6x = 5$$

Rozwiązaniem tego układu równań jest $x = -\frac{5}{6} \approx 0,8333$

b.

$$10x - 6y = -6$$
$$x + 8y = -2$$

Rozwiązaniem tego układu równań jest
$x = -0.69767442, y = -0.1627907$

c.

$$10x - y + 5z = -6$$
$$-4x + 7y + z = -7$$
$$5x + 2y + 8z = 7$$

Rozwiązaniem tego układu równań jest
$x = -2.4375, y = -2.4375 - 2.83680556, z = 3.10763889$

d.

$$8x + 6y - 6z = 6$$
$$-6x + 10y + 7z - 9v = 9$$
$$-5x - 3y + 8z - 3v = 9$$
$$7x - 6y + 3z + 8v = 8$$

Rozwiązaniem tego układu równań jest
$x = 2.45203906, y = -0.74439977, z = 1.52498564, v = -2.27570362$

e.

$$10x + 7y + 8z - 6v = -2$$
$$2x + 10y - 10z + 9v = -9$$
$$x - 5y + 10z = -3$$
$$x - 4y - 2z + 10v = -9$$

Rozwiązaniem tego układu równań jest
$x = -0.30476673, y = -0.29208925, z = -0.41556795, v = -1.06947262,$

Wszystkie powyższe układy są ułożone w ten sposób, by po utworzeniu z nich macierzy była ona przekątniowo dominująca. Wszystkie układy wraz z rozwiązaniami zostały zapisane w pliku equations.h.

2. Zadanie 1: metoda Jacobiego

Proszę zaimplementować metodę Jacobiego oraz przetestować jej działanie na kilku znalezionych przez siebie układach równań (nie mniej niż 5 układów, nie więcej niż 10, w miarę możliwości różnorodne).

Metoda Jacobiego jest iteracyjnym algorytmem służącym do określania rozwiązań układu równań liniowych. Każdy element diagonalny jest rozwiązany i i określana jest jego przybliżona wartość. Proces jest następnie powtarzany.

Została zaimplementowana klasa Matrix:

Matrix.h:

```
class Matrix {
 private:
  std::vector<std::vector<double>> matrix;
  int x;
  int y;

 public:
  Matrix(int size);
  Matrix(std::vector<std::vector<double>> equation);
  int getx() const { return x; }
  int gety() const { return y; }
  std::vector<std::vector<double>> getMatrix() { return matrix; }
  void setMatrix(int i, int j, double val) { this->matrix[i][j] = val; }
  Matrix getLMatrix();
  Matrix getUMatrix();
  Matrix operator+(Matrix rhs);
};
```

Matrix.cpp:

```
Matrix::Matrix(int size) {
  std::vector<double> row;
  row.resize(size + 1, 0.0);
  std::vector<std::vector<double>> matrix;
  matrix.resize(size, row);
  this->matrix = matrix;
  this->x = size + 1;
  this->y = size;
}

Matrix::Matrix(std::vector<std::vector<double>> equation) {
  this->matrix = equation;
  this->x = equation.size();
  this->y = equation[0].size();
}

Matrix Matrix::getLMatrix() {
  int size = this->getMatrix().size();
  Matrix L(size);
  for (int i = 0; i < size; i++) {
    for (int j = 0; j < i; j++) {
      L.setMatrix(i, j, this->getMatrix()[i][j]);
    }
  }
  return L;
}

Matrix Matrix::getUMatrix() {
  int size = this->getMatrix().size();
  Matrix U(size);
  for (int i = 0; i < size; i++) {
    for (int j = i + 1; j < size; j++) {
      U.setMatrix(i, j, this->getMatrix()[i][j]);
    }
  }
  return U;
}

Matrix Matrix::operator+(Matrix rhs) {
  int size = this->getMatrix().size();
  Matrix result(size);
  for (int i = 0; i < size; i++) {
    for (int j = 0; j < size; j++) {
      result.setMatrix(i, j, this->getMatrix()[i][j] + rhs.getMatrix()[i][j]);
    }
  }
  return result;
}
```

Oraz została zaimplementowana funkcja **`jacobiMethod`**, zwracająca wektor z rozwiązaniem układu równań. Metoda była iterowana do momentu, gdy dla każdego rozwiązania wartość bezwzględna z różnicy dokładnego rozwiązania, a rozwiązania zwracanego przez kolejną iterację był mniejszy niż **`0.0001`**, lub gdy liczba iteracji będzie większa niż 1000**.**

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

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

  while (current_iteration < iterations) {
    if (difference(X, solutions)) {
      break;
    }
    for (int i = 0; i < size; i++) {
      tmp[i] = (1 / matrix.getMatrix()[i][i]) * matrix.getMatrix()[i][size];
      for (int j = 0; j < size; j++) {
        tmp[i] += LU.getMatrix()[i][j] * X[j];
      }
    }
    for (int i = 0; i < size; i++) {
      X[i] = tmp[i];
    }
    current_iteration++;
  }
  std::cout << current_iteration << std::endl;

  return X;
}
```

Ilość potrzebnych iteracji do rozwiązania układu równań z dokładnością do **`0.0001`** przedstawia poniższa tabela:

Numer równania	Potrzebna ilość iteracji:
1	1
2	7
3	32
4	198
5	16

Maksymalną liczbą iteracji było 1000, jednak ta granica nie została osiągnięta podczas rozwiązywania żadnego równania, zatem każde równanie zostało rozwiązane poprawnie.

Uwagę zwraca bardzo duża liczba iteracji dla czwartego równania. Jest to prawdopodobnie spowodowane tym, że czwarte równanie jest najbardziej skomplikowane i ma najbardziej skomplikowane rozwiązania.

3. Zadanie 2: Metoda Gaussa-Seidela

Proszę zaimplementować metodę Gaussa-Seidela oraz przetestować jej działanie na układach równań z poprzedniego zadania

Metoda Gaussa-Seidla bazuje na metodzie Jacobiego, w której krok iteracyjny zmieniono w ten sposób, by każda modyfikacja rozwiązania próbnego korzystała ze wszystkich aktualnie dostępnych przybliżonych składowych rozwiązania. Pozwala to zmniejszyć ok. dwukrotnie liczbę obliczeń niezbędnych do osiągnięcia zadanej dokładności rozwiązania.

Została zaimplementowana metoda Gaussa-Seidela:

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

  for (int i = 0; i < size; i++) {
    for (int j = 0; j < size; j++) {
      if (i == j) {
```

```
        LU.setMatrix(i, j, 0);
      } else {
        LU.setMatrix(i, j,
                     -(matrix.getMatrix()[i][j] / matrix.getMatrix()[i][i]));
      }
    }
  }
  int current_iteration = 0;

  while (current_iteration < iterations) {
    if (difference(X, solutions)) {
      break;
    }
    for (int i = 0; i < size; i++) {
      tmp[i] = (1 / matrix.getMatrix()[i][i]) * matrix.getMatrix()[i][size];
      for (int j = 0; j < size; j++) {
        if (j < i) {
          tmp[i] += LU.getMatrix()[i][j] * tmp[j];
        } else if (j > i) {
          tmp[i] += LU.getMatrix()[i][j] * X[j];
        }
      }
    }
    for (int i = 0; i < size; i++) {
      X[i] = tmp[i];
    }
    current_iteration++;
  }
  std::cout << current_iteration << std::endl;
  return X;
}
```

Również dla tej metody warunkiem końcowym było 1000 iteracji, lub osiągnięcie dokładności 0.0001. Ilość iteracji dla poszczególnych równań prezentuje tabela:

Numer równania	Potrzebna ilość iteracji:
1	1
2	4
3	15
4	18
5	8

Nadal najwięcej iteracji potrzebowało równanie czwarte, jednak tym razem zostało wykonane 11 razy mniej iteracji niż w przypadku metody Jakobiego. Dla pozostałych równań liczba iteracji została zmiejszona około dwukrotnie.
Również liczba iteracji nie przekroczyła 1000, zatem wszystkie układy równań zostały rozwiązane poprawnie.

4. Zadanie 3: Metoda SOR

Proszę zaimplementować metodę SOR oraz przetestować jej działanie na układach równań z poprzedniego zadania.

Metoda SOR jest wariantem metody Gaussa-Seidla do rozwiązywania liniowego układu równań, powodując szybszą zbieżność poprzez wprowadzenie parametru relaksacji ω. Dzięki temu kolejne współrzędne nowego przybliżenia możemy wyznaczać poprzez kombinacje poprzedniego przybliżenia oraz współrzędną nowego przybliżenia.

Została zaimplementowana metoda SOR:

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

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

  while (current_iteration < iterations) {
    if (difference(X, solutions)) {
      break;
    }
    for (int i = 0; i < size; i++) {
      tmp[i] = (1 / matrix.getMatrix()[i][i]) * matrix.getMatrix()[i][size];
      for (int j = 0; j < size; j++) {
        if (j < i) {
          tmp[i] += LU.getMatrix()[i][j] * tmp[j];
        } else if (j > i) {
          tmp[i] += LU.getMatrix()[i][j] * X[j];
        }
      }
    }
    for (int i = 0; i < size; i++) {
      X[i] = w * tmp[i] + (1 - w) * X[i];
```

```
    }
    current_iteration++;
  }
  std::cout << current_iteration << std::endl;
  return X;
}
```

Podobnie jak w poprzednich metodach warunkiem końcowym było albo 1000 iteracjii albo odpowiednio dokładne wyniki. Dla ω = 0.75 liczba iteracji dla kolejnych równań przedstawia tabela:

Numer równania	Potrzebna ilość iteracji:
1	7
2	8
3	23
4	25
5	13

Wyniki są gorsze niż w przypadku metody Gaussa-Seidela.

Natomiast dla najlepszego wyznaczonego ω (wyjaśnienie wyznaczenia i wartości są przedstawione w następnym punkcie).

Numer równania	Potrzebna ilość iteracji dla najlepszego ω:
1	1
2	4
3	10
4	17
5	6

Dla dobrze wyznaczonego ω wyniki są w każdym przypadku lepsze niż dla metodu Gaussa-Seidela.

5. Zadanie 4: Porównanie powyższych metod

Metoda Jacobiego opiera się na rozwiązywaniu każdej zmiennej lokalnie w odniesieniu do innych zmiennych - jedna iteracja metody odpowiada jednokrotnemu rozwiązaniu każdej zmiennej. Aby metoda działała macierz musi być dominująca. Powstała metoda jest łatwa do zaimplementowania, jednak metoda działa najwolniej.

Metoda Gaussa-Seidela jest podobna do metody Jacobiego, z tym wyjątkiem, że wykorzystuje zaktualizowane wartości, gdy tylko będą one dostępne. W każdym przypadku metoda Gaussa-Seidela potrzebowała dwukrotnie mniejszej liczby iteracji aby osiągnąć taką samą dokładność jak rozwiązanie za pomocą metody Jacobiego. Dodatkowo metoda potrzebuje dwa razy mniej pamięci, ponieważ przechowujemy współrzędne tylko z jednego kroku wstecz. Aby metoda działała poprawnie macierz musi być, tak jak wprzypadku metody Jacobiego, dominująca.

Metoda SOR jest zmodyfikowaną metodą Gaussa-Seidela - został wprowadzony parametr relaksacji. Współrzędne ω są kombinacją współrzędnych z obecnego i poprzedniego kroku. Dla nieoptymalnego wyboru SOR może zbiegać wolniej niż metoda Gaussa-Seidelai.

6. Zadanie 5: Tempo zbieżności

Proszę dla powyższych metod porównać tempo zbiegania do rozwiązania (na wykresie). Co można zaobserwować i o czym to może świadczyć?

W celu porównania zbieżności wszystkich metod należy wyznaczyć najbardziej optymalny współczynnik ω dla metody SOR. W tym celu modyfikuję tę metodę aby zamiast wektora z wynikami zwracała ilość potrzebnych iteracji do uzyskania wystarczająco dokładnego wyniku. Następnie zaimplementowana została funkcja, która szuka takiego współczynnika relaksacji dla którego liczba iteracji jest najmniejsza

```
double findRelaxationFactor(std::vector<std::vector<double>> matrix,
                            std::vector<double> solutions) {
  int min = 10000;
  double omega = 0;
  double i = 0;
  while (i <= 2) {
    if (SORMethod(matrix, 1000, i, solutions) < min) {
      min = SORMethod(matrix, 1000, i, solutions);
      omega = i;
    }
    i += 0.01;
  }
  return omega;
}
```

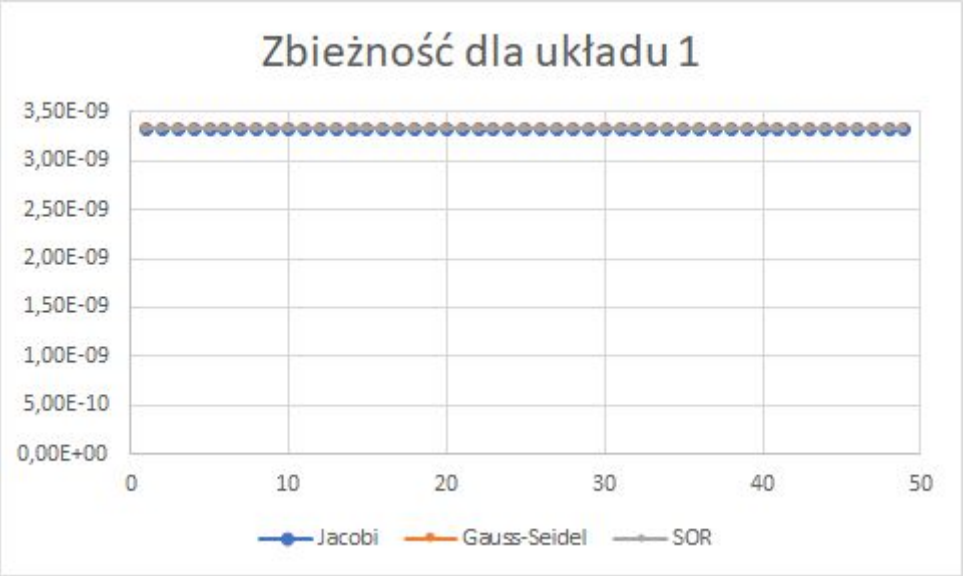
Najlepsze współczynniki dla każdego równania są przedstawione w tabeli:

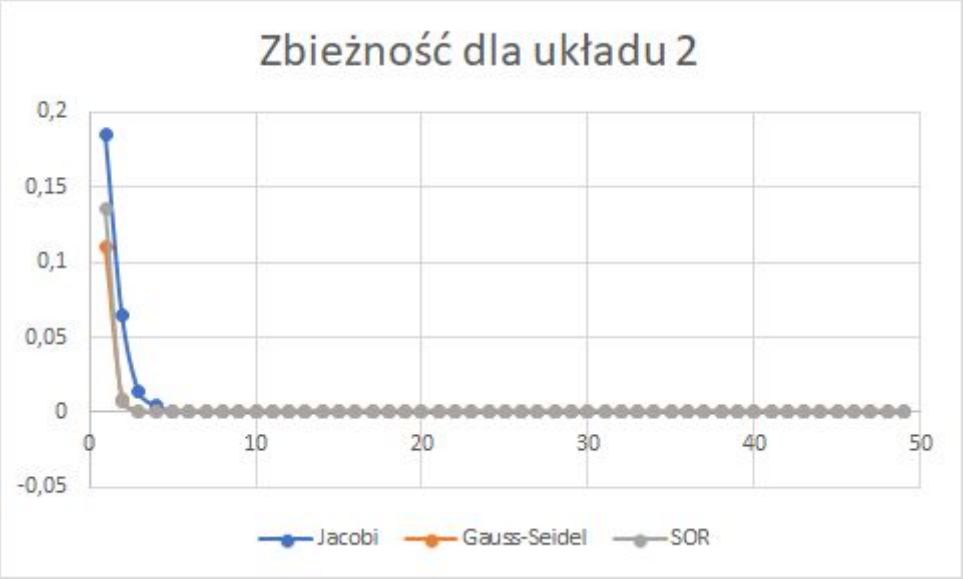
Numer równania	Współczynnik relaksacji
1	1
2	0.94
3	1.32
4	1.02
5	1.08

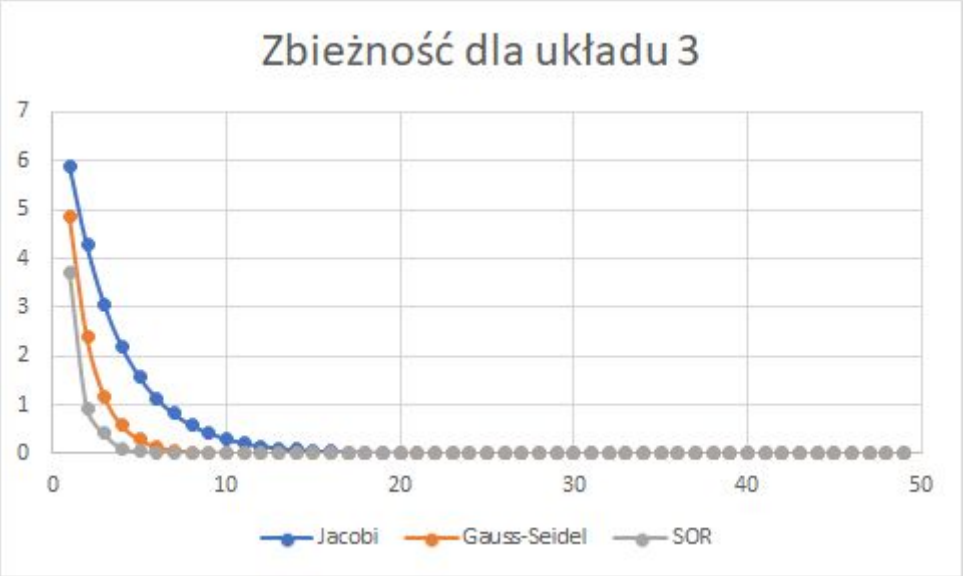
Zaimplementowana została funkcja, która dla liczby iteracji od 1 do 1000 wylicza błąd, a więc i tempo zbiegania dla każdej z trzech metod. Dane zostały zapisane do plików.

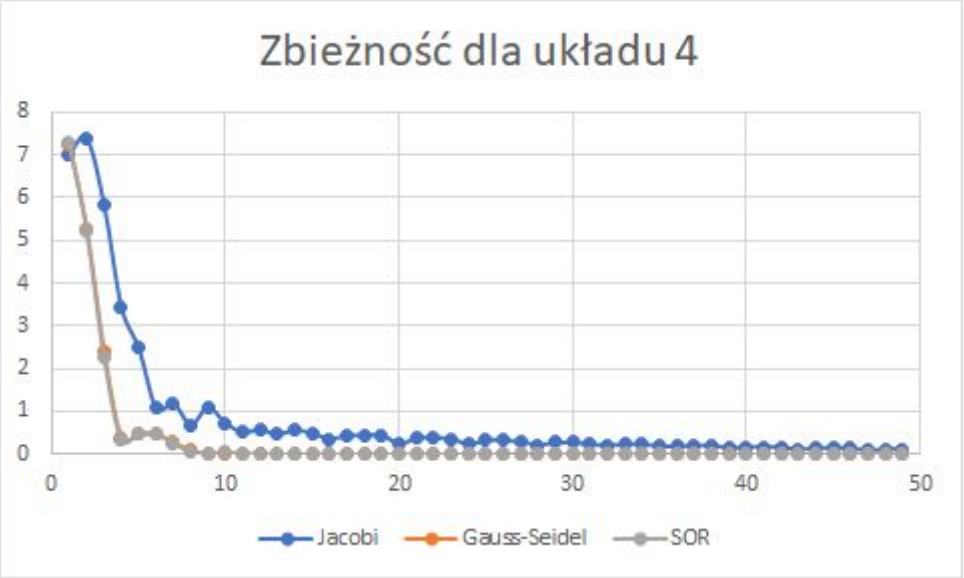
```
void findError(std::vector<std::vector<double>> matrix,
               std::vector<double> solutions,
               double relax,
               int num) {
  std::ofstream plik;
  std::string name = "zbieznosc" + std::to_string(num) + ".txt";
  plik.open(name);
  for (int i = 1; i < 1000; i++) {
    std::vector<double> jac = jacobiMethod(matrix, i, solutions);
    std::vector<double> gau = gaussSeidelMethod(matrix, i, solutions);
    std::vector<double> sor = SORMethod(matrix, i, relax, solutions).first;

    double err_jac = 0;
    double err_gau = 0;
    double err_sor = 0;

    for (int j = 0; j < (int)jac.size(); j++) {
      err_jac += fabs(jac[j] - solutions[j]);
      err_gau += fabs(gau[j] - solutions[j]);
      err_sor += fabs(sor[j] - solutions[j]);
    }
    plik << i << ":" << err_jac << ":" << err_gau << ":" << err_sor
         << std::endl;
    err_jac = 0;
    err_gau = 0;
    err_sor = 0;
  }
}
```

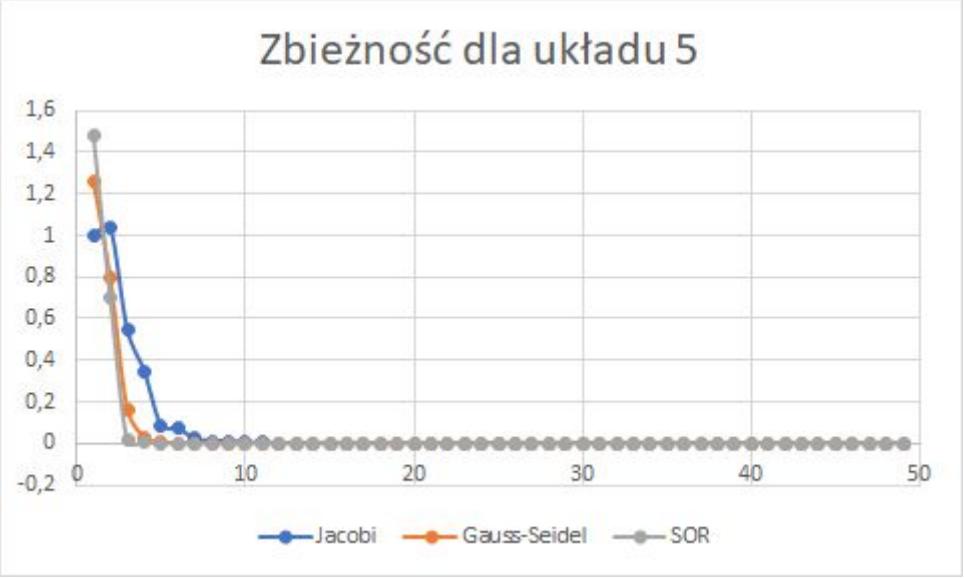
Tempo zbiegania dla poszczególnych układów prezentują poniższe wykresy:






Zbieżność dla układu 3
7
6
5
4
3
2
1
0
0
10
20
30
40
50
Jacobi
Gauss-Seidel
SOR


Zbieżność dla układu 4
8
7
6
5
4
3
2
1
0
0
10
20
30
40
50
Jacobi
Gauss-Seidel
SOR


Zbieżność dla układu 5
1,6
1,4
1,2
1
0,8
0,6
0,4
0,2
0
-0,2
0
10
20
30
40
50
Jacobi
Gauss-Seidel
SOR

Z wykresów wynika, że najszybciej zbiega do rozwiązania zbiega metoda SOR, natomiast najwolniej metoda Jacobiego. Także dla każdego układu 10 iteracji dla metod Gaussa-Seidela i SOR są wystarczające aby uzyskać dokładny wynik, podczas gdy metoda Jacobiego w wielu przypadkach, na przykład dla układu 4, potrzebuje ich dużo więcej. Wynika z tego, że warto korzystać z zoptymalizowanych metod, ponieważ ich działanie jest szybsze.