Julia Plewa
MOwNiT
Laboratorium 3

1. **Proszę w załączonym do laboratorium kodzie napisać funkcję realizującą dodawanie oraz mnożenie macierzy. Po krótce opisać obie metody.**

Kod:

```
// Addition of two matrices
template<typename T>
AGHMatrix<T> AGHMatrix<T>::operator+(const AGHMatrix<T>& rhs)
{
 if ((get_cols() != rhs.cols) || (get_rows() != rhs.rows))
 {
       throw exceptions::MismatchedMatricesException();
 }
 AGHMatrix<T> result(*this);
 for (int i = 0; i < get_rows(); i++)
 {
   for (int j = 0; j < get_cols(); j++){
     result.matrix[i][j] = matrix[i][j] + rhs.matrix[i][j];
   }
 }
 return(result);
}
```

```
// Left multiplication of this matrix and another
template<typename T>
AGHMatrix<T> AGHMatrix<T>::operator*(const AGHMatrix<T>& rhs)
{
 // the number of columns in the first matrix needs to be equal to the number of rows
in the second one
   if ((get_cols() != rhs.rows))
   {
         throw exceptions::MismatchedMatricesException();
   }
   // the resulting matrix has the same number of rows as the first one and the same
number of columns as the second one
   AGHMatrix<T> result(get_rows(), rhs.cols, 0);
```

```
   for (int i = 0; i < get_rows(); i++)
   {
     for (int j = 0; j < rhs.cols; j++)
     {
       for (int k = 0; k < get_cols(); k++)
       {
         result.matrix[i][j] += matrix[i][k] * rhs.matrix[k][j];
       }
     }
   }
   return(result);
}
```

Opis:

Suma macierzy $A_{N\times M}$ i $B_{N\times M}$ to macierz $C_{N\times M}$, której elementy mają następującą postać:

$$\forall\ i \in [1,\ N]\ \ \forall\ j \in [1,\ M]\ :\ c_{ij} = a_{ij} + b_{ij}\,,$$

gdzie x_{ij} oznacza i-ty wiersz i j-tą kolumnę macierzy X.

Wynik mnożenia macierzy $A_{N\times K}$ i $B_{K\times M}$ to macierz $C_{N\times M}$, której elementy mają następującą postać:

$$\forall\ i \in [1,\ N]\ \ \forall\ j \in [1,\ M]\ :\ c_{ij} = \sum_{k=1}^{K} a_{ik} \times b_{kj}\,.$$

W implementacji warto zwrócić uwagę na to, czy wykorzystywany język programowania przechowuje tablice wielowymiarowe w pamięci w postaci *row-* czy *column-major*. W przypadku języka C/C++ jest to kolejność *row-major*, czyli wzdłuż wierszy, zatem iteracja po macierzach odbywa się w takiej właśnie kolejności, co przyspiesza działanie programu.

Przykłady działania:

➢ dodawanie macierzy:

Kod	Output

`void ex1_addition()`
`{`
`  AGHMatrix<double> m1(5, 5, 1.2);`
`  AGHMatrix<double> m2(5, 5, 2.8);`
`  AGHMatrix<double> m3 = m1 + m2;`
`  std::cout << m1 << std::endl;`
`}`
	`4, 4, 4, 4, 4,`
	`4, 4, 4, 4, 4,`
	`4, 4, 4, 4, 4,`
	`4, 4, 4, 4, 4,`
	`4, 4, 4, 4, 4,`

➢ mnożenie macierzy o tych samych wymiarach:

Kod	Output

`void ex1_multiplication_1()`
`{`
`  std::vector<std::vector<double>> m1_v { { 1, 2 },`
`  { 3, 4 } };`
`  std::vector<std::vector<double>> m2_v { { 2, 0 },`
`  { 1, 2 } };`
`  AGHMatrix<double> m1(m1_v);`
`  AGHMatrix<double> m2(m2_v);`
`  AGHMatrix<double> m3 = m1 * m2;`
`  std::cout << m3 << std::endl;`
`}`
	`4, 4,`
	`10, 8,`

➢ mnożenie macierzy o różnych wymiarach:

Kod	Output

`void ex1_multiplication_2()`
`{`
`  std::vector<std::vector<double>> m1_v { { 3, 4, 2 } };`
`  std::vector<std::vector<double>> m2_v { { 13, 9, 7, 15 },`
`  { 8, 7, 4, 6 },`
`  { 6, 4, 0, 3 } };`
`  AGHMatrix<double> m1(m1_v);`
`  AGHMatrix<double> m2(m2_v);`
`  AGHMatrix<double> m3 = m1 * m2;`
`  std::cout << m3 << std::endl;`
`}`
	`83, 63, 37, 75,`

2. Proszę zaimplementować:

2.1. Funkcję/metodę, która sprawdzi czy macierz jest symetryczna.

Kod:

```
template<typename T>
bool AGHMatrix<T>::isSymmetric()
{
 if (get_cols() != get_rows())
 {
   return false;
 }
 for (int i = 0; i < get_rows(); i++)
 {
   for (int j = 0; j < i; j++)
   {
     if (matrix[i][j] != matrix[j][i])
     {
       return false;
     }
   }
 }
 return true;
}
```

Opis:

Macierz symetryczna to macierz $A_{N\times N}$, której elementy spełniają warunek

$$\forall\ i \in [1,\ N]\ \ \forall\ j \in [1,\ N]\ :\ a_{ij} = a_{ji}\,.$$

W powyższej implementacji w iteracjach pomijane są elementy na przekątnej, a każda para indeksów sprawdzana jest tylko raz.

Przykłady działania:

`matrix`	`matrix.isSymmetric()`
13, 9, 7, 15, 8, 7, 4, 6, 6, 4, 0, 3,	false
2, 0, 1, 2,	false
2, 1, 1, 7,	true
2, 1, 3, 13, 8, 1, 2, 7, 0, -3, 3, 7, -7, 10, 12, 13, 0, 10, 3, 19, 8, -3, 12, 19, 100,	true
2, 1, 8, 13, 8, 1, 2, 7, 0, -3, 3, 7, -7, 10, 12, 13, 0, 10, 3, 19, 8, -3, 12, 19, 100,	false
13,	true

2.2. Funkcję/metodę, która obliczy wyznacznik macierzy.

Kod:

```
// Laplace expansion of the matrix along the first column
template<typename T>
T AGHMatrix<T>::laplaceExpansion(int start_col, std::vector<int> rows)
{
 if (start_col + 1 == get_cols() && rows.size() == 1)
 {
   return matrix[rows[0]][start_col];
 }
 T det = 0;
 for (int i = 0; i < rows.size(); i++)
 {
   std::vector<int> new_rows(rows);
   new_rows.erase(std::remove(new_rows.begin(), new_rows.end(), rows[i]),
                  new_rows.end());
   det += matrix[rows[i]][start_col] * pow(-1, i + 2) *
          laplaceExpansion(start_col + 1, new_rows);
 }
 return det;
}

template<typename T>
T AGHMatrix<T>::getDeterminant()
{
 // the matrix needs to be square
 if (get_cols() != get_rows())
 {
   throw exceptions::MatrixNotSquareException();
 }
 // to avoid having to copy matrix values we shall always use Laplace
expansion along the first column
 // this way we need to remember the column we're starting from and the rows
that compose the matrix in question
 std::vector<int> rows(get_rows());
 std::iota(rows.begin() + 1, rows.end(), 1);
 return laplaceExpansion(0, rows);
}
```

Opis:

Wyznacznik macierzy $A_{N\times N}$ można obliczyć przy pomocy rozwinięcia Laplace'a. Rozwinięcie Laplace'a macierzy $A_{N\times N}$ wzdłuż kolumny j ma następujący rekurencyjny wzór:

$$det\ A\ =\ \begin{cases} a_{11}, & n = 1 \\ \sum_{i=1}^{N} (-1)^{i+j}\ a_{ij}\ det\ A_{ij}, & n\ >\ 1 \end{cases}$$

gdzie j jest dowolną liczbą z zakresu $1 \leq j \leq N$, a A_{ij} to macierz stopnia $N-1$ powstała poprzez skreślenie z macierzy A i-tego wiersza oraz j-tej kolumny. Analogicznie definiuje się rozwinięcie Laplace'a wzdłuż wiersza.

Przykłady działania:

matrix	**matrix.getDeterminant()**
2,	2
2, 0, 1, 2,	4
-5, 0, -1, 1, 2, -1, -3, 4, 1,	-40
0, 1, 2, 7, 1, 2, 3, 4, 5, 6, 7, 8, -1, 1, -1, 1,	-64
0, 1, 0, -2, 1, 1, 0, 3, 1, 1, 1, -1, 1, 1, 1, 2, 2, 1, 0, 1, 3, 1, 1, 1, 2,	4

2.3. (*) Metodę transpose().

Kod:

```
template<typename T>
AGHMatrix<T> AGHMatrix<T>::transpose()
{
 AGHMatrix<T> transposed(get_cols(), get_rows(), 0);
 for (int i = 0; i < get_rows(); i++)
 {
   for (int j = 0; j < get_cols(); j++)
   {
     (transposed.matrix)[j][i] = matrix[i][j];
   }
 }
 return transposed;
}
```

Opis:

Dla danej macierzy $A_{N\times M}$ jej macierz transponowana A^T to macierz $B_{M\times N}$, w której

$$\forall\ i \in [1,\ M]\ \ \forall\ j \in [1,\ N]\ :\ b_{ij} = a_{ji}\,.$$

Przykłady działania:

matrix	**matrix.transpose()**
2,	2
2, 7, 1, 2,	2, 1, 7, 2,
0, 1, 2, 7, 1, 2, 3, 4, 5, 6, 7, 8,	0, 1, 5, 1, 2, 6, 2, 3, 7, 7, 4, 8,
2, 1, 3, 13, 8, 1, 2, 7, 0, -3, 3, 7, -7, 10, 12, 13, 0, 10, 3, 19, 8, -3, 12, 19, 100,	2, 1, 3, 13, 8, 1, 2, 7, 0, -3, 3, 7, -7, 10, 12, 13, 0, 10, 3, 19, 8, -3, 12, 19, 100,

3. Proszę zaimplementować algorytm faktoryzacji LU macierzy (można to zrobić przy użyciu kodu dostarczonego do laboratorium lub stworzyć własną strukturę macierzy i na niej działać). Algorytm przetestować na przykładzie z wikipedii lub korzystając z poniższego kodu.

Kod:

```
template<typename T>
std::pair<AGHMatrix<T>, AGHMatrix<T>> AGHMatrix<T>::luDecomposition()
{
 if (get_cols() != get_rows()){
   throw exceptions::MatrixNotSquareException();
 }
 T tmp_U = 0;
 T tmp_L = 0;
 AGHMatrix<T> U(get_rows(), get_cols(), 0);
 AGHMatrix<T> L(get_rows(), get_cols(), 0);
 for (int i = 0; i < get_rows(); i++)
 {
   L.matrix[i][i] = 1;
   tmp_U = 0;
   for (int k = 0; k < i; k++)
   {
     tmp_U += L.matrix[i][k] * U.matrix[k][i];
   }
   U.matrix[i][i] = matrix[i][i] - tmp_U;
   for (int j = i + 1; j < get_rows(); j++)
   {
     tmp_U = 0;
     tmp_L = 0;
     for (int k = 0; k < i; k++)
     {
       tmp_U += L.matrix[i][k] * U.matrix[k][j];
       tmp_L += L.matrix[j][k] * U.matrix[k][i];
     }
     U.matrix[i][j] = matrix[i][j] - tmp_U;
     L.matrix[j][i] = (matrix[j][i] - tmp_L) / U.matrix[i][i];
   }
 }
 return std::make_pair(L, U);
}
```

Opis:

Metoda LU pozwala na rozwiązywanie układu równań liniowych. Nazwa pochodzi od użytych w tej metodzie macierzy trójkątnych: dolnotrójkątnej (*lower*) i górnotrójkątnej (*upper*).

Dla układu równań liniowych

$$A \cdot x = y,$$

gdzie A jest macierzą współczynników, x wektorem niewiadomych, a y wektorem danych, *metoda LU* polega na zapisaniu macierzy A jako iloczynu pewnej macierzy dolnej L oraz pewnej macierzy górnej U.

Układ równań przyjmuje wtedy postać

$$L \cdot U \cdot x = y,$$

a jego rozwiązanie polega na rozwiązaniu dwóch układów równań z macierzami trójkątnymi:

$$L \cdot z = y,$$
$$U \cdot x = z.$$

Realizację metody LU umożliwia m.in. *Algorytm Doolittle'a*, w którym elementy macierzy L i U przyjmują następującą postać:

$\forall\ 1 \le i \le N$:

$$u_{ij} = a_{ij} - \sum_{k=1}^{i-1} l_{ik} \cdot u_{kj}, \quad \forall\ i \le j \le N,$$

$$l_{ji} = \frac{a_{ji} - \sum_{k=1}^{i-1} l_{jk} \cdot u_{ki}}{u_{ii}}, \quad \forall\ i+1 \le j \le N.$$

Przykład działania:

A	L	U
5, 3, 2, 1, 2, 0, 3, 0, 4,	1, 0, 0, 0.2, 1, 0, 0.6, -1.28571, 1,	5, 3, 2, 0, 1.4, -0.4, 0, 0, 2.28571,

4. Proszę zaimplementować algorytm faktoryzacji Cholesky'ego macierzy. Jego test można analogicznie do poprzedniego zadania oprzeć o przykład z wikipedii. Po zakończeniu tego zadania proszę porównać oba algorytmy faktoryzacyjne i opisać różnice w ich konstrukcji.

Kod:

```
template<typename T>
AGHMatrix<T> AGHMatrix<T>::cholskyDecomposition()
{
 if (get_cols() != get_rows()){
   throw exceptions::MatrixNotSquareException();
 }
 T tmp_L = 0;
 AGHMatrix<T> L(get_rows(), get_cols(), 0);
 for (int i = 0; i < get_rows(); i++){
   tmp_L = 0;
   for (int k = 0; k < i; k++)
   {
     tmp_L += pow(L.matrix[i][k], 2);
   }
   L.matrix[i][i] = sqrt(matrix[i][i] - tmp_L);
   for (int j = i + 1; j < get_rows(); j++)
   {
     tmp_L = 0;
     for (int k = 0; k < i; k++)
     {
       tmp_L = L.matrix[j][k] * L.matrix[i][k];
     }
     L.matrix[j][i] = (matrix[j][i] - tmp_L) / L.matrix[i][i];
   }
 }
 return L;
}
```

Opis:

Rozkład Choleskiego polega na zapisaniu macierzy A w postaci:

$$A = LL^T ,$$

gdzie L jest dolną macierzą trójkątną, a L^T jej transpozycją.

Macierz L wyznaczona przy pomocy algorytmu Choleskiego-Crouta ma następującą postać:

$\forall \ 1 \le i \le N$:

$$l_{ii} = \sqrt{a_{ii} - \sum_{k=1}^{i-1} l_{ik}^2} \ ,$$

$$l_{ji} = \frac{a_{ji} - \sum_{k=1}^{i-1} l_{jk} l_{ik}}{l_{ii}} \ , \ \forall \ i < j \le N$$

Po sprowadzeniu układu równań $Ax = y$ do postaci $LL^T x = y$, rozwiązanie takiego układu sprowadza się do rozwiązania $Lz = y$ względem z przy pomocy *forward substitution* oraz $L^T x = z$ przy pomocy *backward substitution*.

Porównanie z metodą LU:

Rozkład Choleskiego jest specjalnym przypadkiem rozkładu LU i może być stosowany w przypadku macierzy symetrycznych i dodatnio określonych. Dzięki temu, że obliczamy elementy tylko jednej macierzy (drugą możemy otrzymać przez transpozycję), a nie dwóch, jak w przypadku rozkładu LU, czas działania rozkładu Choleskiego jest około dwukrotnie krótszy.

Przykład działania:

A	L	L^T
4, 12, -16, 12, 37, -43, -16, -43, 98,	2, 0, 0, 6, 1, 0, -8, 5, 3,	2, 6, -8, 0, 1, 5, 0, 0, 3,

5. **Proszę napisać funkcję (lub klasę wraz z metodami), która realizuje eliminacje Gaussa. Proszę starannie opisać kod, który ją realizuje. Test algorytmu jest najłatwiej zrealizować przy pomocy języka python oraz pakietu numpy (poniższy kod). (*) Dla chętnych - można napisać prosty TestCase, który porówna dwie macierze - poprawną znaleźć przy pomocy pythona. Środowisk testowych w C++ jest kilka - ja polecam GoogleTest.**

Kod:

```
// transform matrix into row echelon form
template<typename T>
void AGHMatrix<T>::gaussianElimination()
{
 // underdetermined systems have no single solutions
 if (get_cols() > (get_rows()) + 1)
 {
     throw exceptions::UnderdeterminedSystemException();
 }
 // overdetermined systems might have a solution, but we're not considering them
 else if (get_cols() != (get_rows()) + 1)
 {
     throw exceptions::OverdeterminedSystemException();
 }
 int h = 0;  // pivot row
 int k = 0;  // pivot column
 while(h < get_rows() && k < get_cols())
 {
   // looking for the k-th partial pivot
   int i_max = h;
   for (int i = h + 1; i < get_rows(); i++)
   {
     if (abs(matrix[i][k]) > abs(matrix[i_max][k])){
       i_max = i;
     }
   }
   // if no pivot is found in the k-th column, go to the next column
   if (matrix[i_max][k] == 0)
   {
     k += 1;
   }
```

```
    else
    {
      // swap pivot row with the current row
      swap_rows(h, i_max);
      // all rows below the pivot
      for (int i = h + 1; i < get_rows(); i++)
      {
        T coeff = matrix[i][k] / matrix[h][k];
        // fill the lower part of pivot column with zeros
        matrix[i][k] = 0;
        // continue going down the current row
        for (int j = k + 1; j < get_cols(); j++)
        {
          matrix[i][j] -= matrix[h][j] * coeff;
        }
      }
      h += 1;
      k += 1;
    }
  }
}

template<typename T>
AGHMatrix<T> AGHMatrix<T>::backwardSubstitution(){
 AGHMatrix<T> results(get_rows(), 1, 0);
 for (int i = get_rows() - 1; i >= 0; i--)
 {
   results.matrix[i][0] = matrix[i][get_cols() - 1];
   for (int j = get_cols() - 2; j > i; j--)
   {
     results.matrix[i][0] -= matrix[i][j] * results.matrix[j][0];
   }
   // nan -> infinite solutions
   // inf -> no solutions
   results.matrix[i][0] /= matrix[i][i];
 }
 return results;
}
```

Opis:

Metoda eliminacji Gaussa pozwala na sprowadzenie danej rozszerzonej macierzy układu równań liniowych do **postaci schodkowej** przy pomocy trzech operacji elementarnych: zamiany dwóch wierszy miejscami, mnożenia wiersza przez współczynnik różny od 0 oraz dodawania wiersza pomnożonego przez współczynnik do innego wiersza.

W powyższej implementacji uwzględniono wspomnianą wcześniej zamianę wierszy, dzięki czemu unikamy potencjalnych problemów wynikających z sytuacji wymagających dzielenia przez 0. Dodatkowo zastosowano **częściowy wybór elementu głównego w kolumnie** (*partial pivoting*) - jako wiersz do zamiany wybieramy taki, którego element w rozważanej kolumnie ma największą wartość bezwzględną, co znacząco **zwiększa stabilność numeryczną algorytmu**.

Warto wspomnieć, że istnieje również analogiczna metoda częściowego wyboru elementu głównego w wierszu oraz metoda pełnego wyboru (*complete pivoting*), w której elementu głównego poszukujemy w całej rozważanej podmacierzy - choć prowadzi to do zwiększenia stabilności w pewnym stopniu, to dodatkowe porównania między wartościami znacząco zwiększają złożoność obliczeń i metoda ta rzadko jest stosowana.

W niektórych sytuacjach eliminacja Gaussa z częściowym wyborem elementu głównego może prowadzić do błędnych wyników - w szczególności, gdy współczynniki równań mają bardzo różne rzędy wielkości, ponieważ w metodzie tej nie rozważamy wartości wzdłuż wierszy, a jedynie wartości w danej kolumnie. Pomocna w tej sytuacji może być **eliminacja ze skalowalnym wyborem wierszy głównych**, w której wybieramy wiersz na podstawie stosunku aktualnie rozważanego elementu do maksymalnej bezwzględnej wartości współczynnika z danego wiersza. Inna potencjalna optymalizacja wiąże się z rezygnacją z faktycznej zamiany wierszy na rzecz wprowadzenia **tablicy indeksującej** opisującej aktualną pozycję wierszy.

W powyższej implementacji nie są rozważane układy niedookreślone, które nigdy nie posiadają pojedynczego rozwiązania (nie posiadają go wcale lub posiadają nieskończenie wiele rozwiązań), oraz układy nadokreślone, które mogą mieć pojedyncze rozwiązanie w sytuacji, gdy w ich skład wchodzą “redundantne” równania.

Po doprowadzeniu macierzy do postaci schodkowej, rozwiązanie układu jest możliwe przy pomocy **back substitution**, czyli obliczania kolejnych wartości od końca. W przypadku układu sprzecznego w trakcie obliczeń natrafimy na wartość ∞ wynikającą z dzielenia liczby przez 0. W przypadku układu nieoznaczonego natrafimy na wartość NaN wynikającą z dzielenia 0 przez 0.

Testy:

Przykładowe testy zostały przeprowadzone przy pomocy GoogleTest.

```
TEST(LinearEquationSolver, uniqueSolution1) {
   std::vector<double> expected { 0.21602477, -0.00791511,  0.63524333, 0.74617428 };
   std::vector<std::vector<double>> m_v {{ 0.0001, -5.0300, 5.8090, 7.8320, 9.5740 },
                                         { 2.2660, 1.9950,  1.2120, 8.0080, 7.2190 },
                                         { 8.8500, 5.6810,  4.5520, 1.3020, 5.7300 },
                                         { 6.7750, -2.253,  2.9080, 3.9700, 6.291 }};
   AGHMatrix<double> m(m_v);
   m.gaussianElimination();
   std::vector<double> result(m.backwardSubstitution().transpose().matrix[0]);
   EXPECT_EQ(result.size(), expected.size());
   for (int i = 0; i < result.size(); i++){
       EXPECT_NEAR(expected[i], result[i], 1e-8);
   }
}
TEST(LinearEquationSolver, uniqueSolution2) {
   std::vector<double> expected { 2, 3, -1 };
   std::vector<std::vector<double>> m_v  { {  2,  1, -1,   8 },
                                           { -3, -1,  2, -11 },
                                           { -2,  1,  2,  -3 } };
   AGHMatrix<double> m(m_v);
   m.gaussianElimination();
   std::vector<double> result(m.backwardSubstitution().transpose().matrix[0]);
   EXPECT_EQ(result.size(), expected.size());
   for (int i = 0; i < result.size(); i++){
       EXPECT_NEAR(expected[i], result[i], 1e-8);
   }
}
TEST(LinearEquationSolver, uniqueSolutionPivotingRequired) {
   std::vector<double> expected { -10, 4, 11 };
   std::vector<std::vector<double>> m_v { { 1, -1,   2,   8 },  { 0,  0,  -1, -11 },
                                          { 0,  2,  -1,  -3 } };
   AGHMatrix<double> m(m_v);
   m.gaussianElimination();
   std::vector<double> result(m.backwardSubstitution().transpose().matrix[0]);
   EXPECT_EQ(result.size(), expected.size());
   for (int i = 0; i < result.size(); i++){
       EXPECT_NEAR(expected[i], result[i], 1e-8);
   }
}
```

```
TEST(LinearEquationSolver, infinitelyManySolutions) {
   std::vector<std::vector<double>> m_v  { {  2,  1, -1,  8 },
                                           {  2,  1, -1,  8 },
                                           { -2,  1,  2, -3 } };
   AGHMatrix<double> m(m_v);
   m.gaussianElimination();
   std::vector<double> result(m.backwardSubstitution().transpose().matrix[0]);
   EXPECT_TRUE(isnan(result[result.size() - 1]));
}

TEST(LinearEquationSolver, noSolutions) {
   std::vector<std::vector<double>> m_v { {  2,  1, -1,  8 },
                                          {  2,  1, -1,  7 },
                                          { -2,  1,  2, -3 } };
   AGHMatrix<double> m(m_v);
   m.gaussianElimination();
   std::vector<double> result(m.backwardSubstitution().transpose().matrix[0]);
   EXPECT_TRUE(isinf(result[result.size() - 1]));
}
```

6. (*)Implementacja metody Jackobiego - tworzenie i wymagania analogicznie do Zad.4.

Kod:

```
#define K_MAX 100
#define EPSILON 1e-10

template<typename T>
AGHMatrix<T> AGHMatrix<T>::jacobiMethod()
{
 // result vector - the initial estimation is 0
 AGHMatrix<T> results(get_rows(), 1, 0);
 // the next estimation of the result
 std::vector<double> y(get_rows(), 0);
 // K_MAX denotes the maximum number of iterations
 for (int k = 0; k < K_MAX; k++)
 {
   // copy current result estimations into a temporary vector
```

```
  for (int i = 0; i < get_rows(); i++)
  {
    y[i] = results.matrix[i][0];
  }
  // calculate a new estimation of the variables based on the previous one
  for (int i = 0; i < get_rows(); i++)
  {
    T sum = matrix[i][get_cols() - 1];
    T diag = matrix[i][i];
    for (int j = 0; j < get_rows(); j++)
    {
      if (j != i)
      {
        sum -= matrix[i][j] * y[j];
      }
    }
    results.matrix[i][0] = sum / diag;
  }
  std::cout << "ITERATION #" << k << std::endl << results << std::endl;
  double norm = 0;
  for (int i = 0; i < get_rows(); i++)
  {
    if (fabs(y[i] - results.matrix[i][0]) > norm)
    {
      norm = fabs(y[i] - results.matrix[i][0]);
    }
  }
  // return if convergence was reached
  if (norm < EPSILON)
  {
      return results;
  }
 }
 std::cout << "ITERATION LIMIT REACHED" << std::endl;
 return results;
}
```

Opis:

Metoda Jacobiego to metoda przybliżania rozwiązania układu równań liniowych dla macierzy przekątniowo dominujących, czyli takich, dla których

$$\forall\ 1 \le i \le N\ :\ \left|a_{ii}\right| \ge \sum_{j=1, j\neq i}^{N} \left|a_{ij}\right| \ .$$

W powyższej implementacji poczyniono założenie, że macierz na wejściu spełnia ten warunek. Oprócz tego metoda Jacobiego wymaga podania początkowego oszacowania wartości niewiadomych. W powyższej implementacji wykorzystywany jest do tego wektor wypełniony zerami. Na podstawie oszacowanych wyników w każdej kolejnej iteracji obliczane są nowe szacunki - algorytm kończy działanie w przypadku, gdy przekroczony zostanie limit iteracji lub gdy osiąga zbieżność, czyli kiedy norma tego wektora (najczęściej euklidesowa lub maksimowa) stanie się mniejsza od pewnego ε.

Przykład działania:

➢ na wejściu:

```
std::vector<std::vector<double>> init_Jacobi_2 { { 10, -1,  2,  0,   6 },
                                                 { -1, 11, -1,  3,  25 },
                                                 {  2, -1, 10, -1, -11 },
                                                 {  0,  3, -1,  8,  15 } };
```

➢ na wyjściu:

iteracja	oszacowanie	$\|\|x-y\|\|_\infty$
0	0.6, 2.27273, -1.1, 1.875,	2.27273
1	1.04727, 1.71591, -0.805227, 0.885227,	0.989773
2	0.932636, 2.05331, -1.04934, 1.13088,	0.337397
3	1.0152, 1.9537, -0.968109, 0.973843,	0.157038

4	0.988991, 2.01141, -1.01029, 1.02135,	0.057719
. . .	. . .	. . .
14	0.999998, 2, -1, 1,	1.09436e-05
15	1, 2, -0.999999, 0.999999,	4.82589e-06
. . .	. . .	. . .
19	1, 2, -1, 1,	1.58236e-07
20	1, 2, -1, 1,	6.67057e-08