AKADEMIA GÓRNICZO-HUTNICZA
IM. STANISŁAWA STASZICA
W KRAKOWIE

Metody Obliczeniowe w Nauce i Technice:

Układy równań liniowych

Laboratorium 2

Przemysław Lechowicz

SPIS TREŚCI

1. W załączonym do laboratorium kodzie napisz funkcje realizujące dodawanie oraz mnożenie macierzy.

1.1. Dodawanie

```
template <typename T>
AGHMatrix<T> AGHMatrix<T>::operator+(const AGHMatrix<T>& rhs) {
  if (this->get_cols() == rhs.get_cols() &&
      this->get_rows() == rhs.get_rows()) {
    AGHMatrix<T> res(this->get_rows(), this->get_cols(), 0);
    for (int i = 0; i < this->get_rows(); i++) {
      for (int j = 0; j < this->get_cols(); j++) {
        res.matrix[i][j] = this->matrix[i][j] + rhs.matrix[i][j];
      }
    }
    return res;
  } else {
    throw std::invalid_argument(
        "Matrixes does not have equal number of columns or rows");
  }
}
```

Funkcja zwraca macierz res jako wynik. Jeśli dodawania nie da się wykonać funkcja zwraca wyjątek. Dla macierzy:

$$\begin{pmatrix} 1.2 & 1.2 & 1.2 \\ 1.2 & 1.2 & 1.2 \\ 1.2 & 1.2 & 1.2 \end{pmatrix} + \begin{pmatrix} 2.8 & 2.8 & 2.8 \\ 2.8 & 2.8 & 2.8 \\ 2.8 & 2.8 & 2.8 \end{pmatrix}$$

Zwracane są wartości w macierzy:

```
4, 4, 4,
4, 4, 4,
4, 4, 4,
```

Więc metoda działa poprawnie.

1.2. Mnożenie

```cpp
template <typename T>
AGHMatrix<T> AGHMatrix<T>::operator*(const AGHMatrix<T>& rhs) {
  if (this->get_cols() == rhs.get_rows()) {
    AGHMatrix<T> res(this->get_rows(), rhs.get_cols(), 0);
    for (int i = 0; i < this->get_rows(); i++) {
      for (int j = 0; j < rhs.get_cols(); j++) {
        for (int k = 0; k < this->get_cols(); k++) {
          res.matrix[i][j] += this->matrix[i][k] * rhs.matrix[k][j];
        }
      }
    }
    return res;
  } else {
    throw std::invalid_argument(
        "Number of columns of first matrix is not equal to number of rows of 
second matrix");
  }
}
```

Funkcja zwraca macierz res jako wynik. Jeśli mnożenia nie da się wykonać funkcja zwraca wyjątek. Dla macierzy:

$$\begin{pmatrix} 1 & 2 & 3 \\ 4 & 5 & 6 \\ 7 & 8 & 9 \end{pmatrix} + \begin{pmatrix} 2.8 & 2.8 & 2.8 \\ 2.8 & 2.8 & 2.8 \\ 2.8 & 2.8 & 2.8 \end{pmatrix}$$

Otrzymujemy macierz:

```
16.8, 16.8, 16.8,
42, 42, 42,
67.2, 67.2, 67.2,
```

Więc metoda działa poprawnie.

2. Zaimplementuj:

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

```
template <typename T>
bool AGHMatrix<T>::isSymmetrical() {
  if (this->get_rows() != this->get_cols()) {
    return false;
  }

  for (int i = 0; i < this->get_rows(); i++) {
    for (int j = 0; j < this->get_cols(); j++) {
      if (this->matrix[i][j] != this->matrix[j][i]) {
        return false;
      }
    }
  }
  return true;
}
```

Funkcja sprawdza czy macierz jest macierzą kwadratową, jeśli to nie macierz nie jest symetryczna. Jeśli jest macierzą kwadratową sprawdzana jest zależność $a_{ij} = a_{ji}$.
Dla macierzy:

$$\begin{pmatrix} 1 & 2 & 3 \\ 2 & 3 & 1 \\ 3 & 1 & 2 \end{pmatrix}$$

Metoda zwraca wartość true, natomiast dla macierzy:

$$\begin{pmatrix} 1 & 3 & 2 \\ 2 & 3 & 1 \\ 3 & 1 & 2 \end{pmatrix}$$

metoda zwraca wartość false, zatem metoda działa poprawnie.

2.2. Funkcję/metodę, która obliczy wyznacznik macierzy.

```
template <typename T>
T determinantOfMatrix(int n, std::vector<std::vector<T>> d) {
  std::vector<std::vector<T>> next(n, std::vector<T>(n, 0));
  T det = 0;

  if (n == 1) {
    return d[0][0];
```

```
  } else {
    for (int c = 0; c < n; c++) {
        int y = 0, x = 0;
        for (int i = 1; i < n; i++) {
          for (int j = 0; j < n; j++) {
            if (j != c) {
              next[y][x] = d[i][j];
              x++;
            }
          }
          y++;
          x = 0;
        }
        if (c % 2 == 0) {
          det += d[0][c] * determinantOfMatrix(n - 1, next);
        } else {
          det -= d[0][c] * determinantOfMatrix(n - 1, next);
        }
      }
  }
  return det;
}

template <typename T>
T AGHMatrix<T>::determinant() {
  if (this->get_rows() != this->get_cols()) {
    throw std::invalid_argument(
        "Matrix does not have equal number of columns and rows");
  } else {
    return determinantOfMatrix(this->get_rows(), this->matrix);
  }
}
```

Funkcja determinant() sprawdza czy macierz jest kwadratowa i przekazuje parametry do funkcji determinantOfMatrix(), która oblicza wyznacznik metodą LaPlace'a.
Dla macierzy:

$$\begin{pmatrix} 1 & 2 & 3 \\ 2 & 3 & 1 \\ 3 & 1 & 2 \end{pmatrix}$$

Metoda zwraca wartość: -18, co jest wartością poprawną.

2.3. Metodę transpose()

```
template <typename T>
AGHMatrix<T> AGHMatrix<T>::transpose() {
  if (this->isSymmetrical()) {
    return *this;
  } else {
    std::vector<std::vector<T>> matrix;
    matrix.resize(this->get_cols());
    for (unsigned i = 0; i < this->get_cols(); i++) {
      matrix[i].resize(this->get_rows(), 0);
    }

    for (int i = 0; i < this->get_cols(); i++) {
      for (int j = 0; j < this->get_rows(); j++) {
        matrix[i][j] = this->matrix[j][i];
      }
    }
    AGHMatrix<T> transposed(matrix);
    return transposed;
  }
}
```

W przypadku, gdy macierz jest symetryczna, metoda zwraca tę samą macierz. W innym przypadku zwraca macierz transponowaną.

Dla macierzy:

$$\begin{pmatrix} 2 & 1 \\ 1 & 1 \\ 5 & 1 \end{pmatrix}$$

Zwracana jest macierz:

```
2, 1, 5,
1, 1, 1,
```

Więc metoda działa poprawnie.

3. Proszę zaimplementować algorytm faktoryzacji LU macierzy.

```cpp
template <typename T>
std::vector<AGHMatrix<T>> AGHMatrix<T>::luDecomoposition() {
  if (this->get_cols() == this->get_rows()) {
    std::vector<AGHMatrix<T>> LU;

    std::vector<std::vector<T>> L;
    L.resize(this->get_cols());
    for (unsigned i = 0; i < this->get_cols(); i++) {
      L[i].resize(this->get_rows(), 0);
    }
    std::vector<std::vector<T>> U;
    U.resize(this->get_cols());
    for (unsigned i = 0; i < this->get_cols(); i++) {
      U[i].resize(this->get_rows(), 0);
    }
    int n = this->get_cols();
    T sum = 0;
    for (int j = 0; j < n; j++) {
      U[0][j] = this->matrix[0][j];
    }
    for (int i = 0; i < n; i++) {
      L[i][i] = 1;
    }
    for (int i = 1; i < n; i++) {
      L[i][0] = this->matrix[i][0] / U[0][0];
    }
    for (int j = 1; j < n; j++) {
      for (int i = 1; i <= j; i++) {
        for (int k = 0; k <= i - 1; k++) {
          sum = sum + (L[i][k] * U[k][j]);
        }
        U[i][j] = this->matrix[i][j] - sum;
        sum = 0;
      }
      for (int i = j + 1; i < n; i++) {
        for (int k = 0; k <= j - 1; k++) {
          sum = sum + (L[i][k] * U[k][j]);
        }
        L[i][j] = (this->matrix[i][j] - sum) / U[j][j];
        sum = 0;
      }
    }

    AGHMatrix<T> Lmatrix(L);
    AGHMatrix<T> Umatrix(U);
```

```
    LU.push_back(Lmatrix);
    LU.push_back(Umatrix);
    return LU;
  } else {
    throw std::invalid_argument(
        "Matrix does not have equal number of columns and rows");
  }
}
```

Metoda sprawdza, czy macierz jest macierzą kwadratową; jeśli nie - zwraca wyjątek. W przeciwnym wypadku zwraca dwuelementowy wektor macierzy, w którym pierwsza pozycja jest macierzą L, a druga macierzą U. Dla macierzy:

$$\begin{pmatrix} 5 & 3 & 2 \\ 1 & 2 & 0 \\ 3 & 0 & 4 \end{pmatrix}$$

Zwraca dwie macierze:

```
1, 0, 0,
0.2, 1, 0,
0.6, -1.28571, 1,

5, 3, 2,
0, 1.4, -0.4,
0, 0, 2.28571,
```

Co jest zgodne ze źródłami, więc metoda działa poprawnie.

4. Proszę zaimplementować algorytm faktoryzacji Cholesky'ego macierzy.

```cpp
template <typename T>
std::vector<AGHMatrix<T>> AGHMatrix<T>::choleskyDecomoposition() {
  if (this->isSymmetrical()) {
    std::vector<AGHMatrix<T>> result;

    std::vector<std::vector<T>> cholesky;
    cholesky.resize(this->get_cols());
    for (unsigned i = 0; i < this->get_cols(); i++) {
      cholesky[i].resize(this->get_rows(), 0);
    }

    int n = this->get_cols();
    for (int i = 0; i < n; i++) {
      for (int j = 0; j <= i; j++) {
        T sum = 0;

        if (j == i) {
          for (int k = 0; k < j; k++) {
            sum += pow(cholesky[j][k], 2);
          }
          cholesky[j][j] = sqrt(this->matrix[j][j] - sum);
        } else {
          for (int k = 0; k < j; k++) {
            sum += (cholesky[i][k] * cholesky[j][k]);
          }
          cholesky[i][j] = (this->matrix[i][j] - sum) / cholesky[j][j];
        }
      }
    }

    AGHMatrix<T> choleskymatrix(cholesky);
    result.push_back(choleskymatrix);
    result.push_back(choleskymatrix.transpose());
    return result;
  } else {
    throw std::invalid_argument(
        "Matrix does not have equal number of columns and rows or is not "
        "symmetrical");
  }
}
```

Metoda sprawdza, czy macierz jest macierzą symetryczną; jeśli nie - zwraca wyjątek. W przeciwnym wypadku zwraca dwuelementowy wektor macierzy, w którym pierwsza pozycja jest macierzą l, a druga macierzą L^T. Dla macierzy:

$$\begin{pmatrix} 4 & 12 & -16 \\ 12 & 37 & -43 \\ -16 & -43 & 98 \end{pmatrix}$$

Zwracane są macierze:

```
2, 0, 0,
6, 1, 0,
-8, 5, 3,

2, 6, -8,
0, 1, 5,
0, 0, 3,
```

Co jest zgodne ze źródłami. Więc metoda działa poprawnie.

Zarówno w przypadku faktoryzacji LU, jak i faktoryzacji Cholesky'ego macierz kwadratowa A jest dzielona na iloczyn macierzy trójkątnych. Faktoryzacja Cholesky’ego jest wygodniejsza pamięciowo, gdyż wystarczy wyliczyć i przechowywać w pamięci tylko jedną macierz - drugą można uzyskać za pomocą algorytmu na macierz transponowaną.
Jednak faktoryzację Cholesky’ego można przeprowadzić tylko dla macierzy symetrycznej. Jeśli nie jest symetryczna - faktoryzacja nie powiedzie się.

5. Proszę napisać funkcję (lub klasę wraz z metodami), która realizuje eliminacje Gaussa.

```
template <typename T>
AGHMatrix<T> AGHMatrix<T>::gaussianElimination() {
  if (this->get_rows() + 1 == this->get_cols()) {
    std::vector<std::vector<T>> a = this->matrix;
    std::vector<T> results;
    int n = this->get_rows();
    T x[n];
    for (int i = 0; i < n; i++)
      for (int k = i + 1; k < n; k++)
        if (abs(a[i][i]) < abs(a[k][i]))
          for (int j = 0; j <= n; j++) {
            T temp = a[i][j];
            a[i][j] = a[k][j];
```

```
            a[k][j] = temp;
          }

    for (int i = 0; i < n - 1; i++) {
      for (int k = i + 1; k < n; k++) {
        T t = a[k][i] / a[i][i];
        for (int j = 0; j <= n; j++) {
          a[k][j] = a[k][j] - t * a[i][j];
        }
      }
    }
    AGHMatrix<T> res(a);
    return res;
  } else {
    throw std::invalid_argument(
        "Matrix does not represent system of equations");
  }
}
```

Redukcja wierszy powoduje rozkład oryginalnej macierzy.
Algorytm dla każdej kolumny znajduje k-tą oś przestawiając rzędy, aby przenieść pozycję o największej wartości bezwzględnej do pozycji przestawnej. Następnie dla każdego rzędu poniżej osi obliczany jest współczynnik t, który powoduje, że k-ty rząd zostaje wyzerowany. Dla każdego elementu w rzędzie odejmuje odpowiednią wielokrotność odpowiedniego elementu w k-tym rzędzie.

6. Implementacja metody Jackobiego

```
template <typename T>
std::vector<T> AGHMatrix<T>::JacobiMethod() {
  int iterations = 10000;
  if (this->get_rows() + 1 == this->get_cols()) {
    int n = this->get_rows();
    std::vector<double> x, c, old;
    x.resize(n, 0);
    c.resize(n, 0);
    old.resize(n, 0);
    bool check = true;
    int m = 0;
    do {
      for (int i = 0; i < n; i++) {
        old[i] = x[i];
      }
      for (int i = 0; i < n; i++) {
        c[i] = this->matrix[i][n];
```

```
        for (int j = 0; j < n; j++) {
          if (i != j) {
            c[i] = c[i] - this->matrix[i][j] * x[j];
          }
        }
      }
      for (int i = 0; i < n; i++) {
        x[i] = c[i] / this->matrix[i][i];
      }
      m++;
      int calc = 0;
      for (int i = 0; i < n; i++) {
        if (old[i] == x[i]) {
          calc++;
        }
      }
      if (calc == n) {
        check = false;
      }
    } while (m < iterations && check);
    std::vector<T> results;
    for (int i = 0; i < n; i++) {
      results.push_back(x[i]);
    }
    return results;
  } else {
    throw std::invalid_argument(
        "Matrix does not represent system of equations");
  }
}
```

Metoda Jacobiego jest iteracyjnym algorytmem służącym do określania rozwiązań układu równań liniowych. Przybliżone metody rozwiązywania układu równań liniowych umożliwiają uzyskanie wartości pierwiastków układu z określoną dokładnością jako granicą sekwencji niektórych wektorów. Proces jest następnie iterowany, dopóki różnica wartości między iteracjami dla każdej niewiadomej układu będzie nieznaczna co do epsilonu maszynowego.