Grzegorz Gackowski
MowNiT
Układy równań liniowych
Sprawozdanie

Zadanie 1. Napisz funkcje realizujące dodawanie oraz mnożenie macierzy

Szablon metody realizujący dodawanie macierzy:

```
template<typename T>
AGHMatrix<T> AGHMatrix<T>::operator+(const AGHMatrix<T>& rhs)
{
   AGHMatrix<T> newMatrix(this->rows, rhs.cols, 0);
   std::cout << this->rows << " " << rhs.cols << std::endl;
    for (auto i = 0; i < this->matrix.size(); ++i)
        for (auto j = 0; j < this->matrix[i].size(); ++j)
            newMatrix.matrix[i][j] = this->matrix[i][j] + rhs.matrix[i][j];
    return newMatrix;
}
```

Szablon metody realizujący mnożenie macierzy:

```
template<typename T>
AGHMatrix<T> AGHMatrix<T>::operator*(const AGHMatrix<T>& rhs)
{
   AGHMatrix<T> newMatrix(this->rows, rhs.cols, 0);

    for (auto i = 0; i < this->matrix.size(); ++i)
        for (auto j = 0; j < this->matrix[i].size(); ++j)
          for (auto k = 0; k < this->cols; ++k)
            newMatrix.matrix[i][j] += this->matrix[i][k] * rhs.matrix[k][j];

    return newMatrix;
}
```

Zadanie 2. Zaimplementuj metodę, która sprawdzi czy macierz jest symetryczna, metodę, która obliczy wyznacznik macierzy oraz metodę transpose().

Metoda sprawdzająca czy macierz jest symetryczna:

```
template <typename T>
bool AGHMatrix<T>::isSymetric() {
    if (cols != rows) return false;
    for (auto i = 0; i < matrix.size(); ++i)
        for (auto j = 0; j < matrix[i].size(); ++j)
            if (matrix[i][j] != matrix[j][i])
                return false;
    return true;
}
```

Metoda obliczająca wyznacznik:

```
template <typename T>
T AGHMatrix<T>::det() {
  if (rows != cols) return -1;
  return _det(rows, cols, matrix);
}
```

uruchamiająca funkcję rekurencyjną:

```
template  <typename T>
T _det(int rows, int cols, std::vector<std::vector<T>> matrix) {
  T det = 0;
  if (rows == 1) return matrix[0][0];
  else {
    for (int i = 0; i < rows; ++i) {
        std::vector<std::vector<T>> tmp;
          for (int k = 0; k < rows; ++k) {
            if (k != i)
              tmp.push_back(std::vector<T>());
            for (int l = 0; l < cols; ++l) {
              if (k != i && l != 0) {
                tmp[tmp.size() - 1].push_back(matrix[k][l]);
              }
            }
          }
       if (i % 2 == 0)
        det +=  matrix[i][0] * _det(rows - 1, cols - 1, tmp);
      else
        det -=  matrix[i][0] * _det(rows - 1, cols - 1, tmp);
    }
    return det;
  }
}
```

Metoda transpose():

```
template <typename T>
AGHMatrix<T> AGHMatrix<T>::transpose() {
  AGHMatrix<T> newMatrix(this->cols, this->rows, 0);
  for (auto i = 0; i < this->rows; ++i)
    for (auto j = 0; j < this->cols; ++j)
      newMatrix.matrix[j][i] = this->matrix[i][j];

  return newMatrix;
}
```

Zadanie 3. Zaimplementuj algorytm faktoryzacji LU macierzy.

```
template <typename T>
std::pair<AGHMatrix<T>, AGHMatrix<T>> AGHMatrix<T>::LU() {
  AGHMatrix<T> L(rows, cols, 0);
  AGHMatrix<T> U(rows, cols, 0);
  for (int i = 0; i < cols; ++i)
    L.matrix[i][i] = 1;

  for (int i = 0; i < cols; ++i) {
    for (int j = i; j < rows; ++j) {
      T sum1 = 0;
      U.matrix[i][j] = matrix[i][j];
      for (int k = 0; k <= i - 1; ++k)
        sum1 += L.matrix[i][k] * U.matrix[k][j];
      U.matrix[i][j] -= sum1;
    }
    for (int j = i + 1; j < rows; ++j) {
        T sum2 = 0;
        L.matrix[j][i] = 1.0 / U.matrix[i][i];
        for (int k = 0; k <= i - 1; ++k)
            sum2 += L.matrix[j][k] * U.matrix[k][i];
        L.matrix[j][i] *= (matrix[j][i] - sum2);
    }
  }
  return std::make_pair(L, U);
}
```

Algorytm został przetestowany na przykładzie z Wikipedii:

```
    std::vector<std::vector<double>> init3 { { 5.0, 3.0, 2.0 },
                                             { 1.0, 2.0, 0.0 },
                                             { 3.0, 0.0, 4.0 }
                                             };
 auto lu = mat6.LU();
 std::cout << lu.first << std::endl << lu.second << std::endl;
```

Otrzymane wyniki przedstawiają kolejno macierz L oraz macierz U:

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


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

Zadanie 4. Zaimplementuj algorytm faktoryzacji Cholesky'ego macierzy. Opisz różnice w konstrukcji obu algorytmów faktoryzujących.

```
template <typename T>
AGHMatrix<T> AGHMatrix<T>::Cholesky() {
  AGHMatrix<T> L(rows, cols, 0);

for (int i = 0; i < rows; ++i) {
  for (int j = i; j < cols; ++j) {
    if (i != j) {
      T sum = 0;
      for (int k = 0; k <= i - 1; ++k)
        sum += L.matrix[j][k] * L.matrix[i][k];

    L.matrix[j][i] = (matrix[j][i] - sum) / L.matrix[i][i];
    }
    else {
      T sum = 0;

      for (int k = 0; k <= i - 1; k++)
        sum += L.matrix[i][k] * L.matrix[i][k];

      L.matrix[i][i] = std::sqrt(matrix[i][i] - sum);
    }
  }
}
return L;
}
```

Test na przykładzie z Wikipedii:

```
     std::vector<std::vector<double>> init4 { { 4.0, 12.0, -16.0 },
                                              { 12.0, 37.0, -43.0 },
                                              { -16.0, -43.0, 98}
                                             };
AGHMatrix<double> mat7(init4);

std::cout << mat7.Cholesky() << std::endl;
```

Wynik:

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

Porównanie faktoryzacji LU z faktoryzacją Cholesky’ego.

Faktoryzacja Cholesky’ego jest około dwukrotnie szybsza od faktoryzacji LU, jednak można ją stosować tylko dla szczególnych przypadków macierzy (symetrycznych dodatnio określonych). Z kolei faktoryzacja LU jest wolniejsza, ale możliwa do wykorzystania dla szerszej gamy problemów.

Zadanie 5. Napisz funkcję realizującą eliminacje Gaussa.

```
template <typename T>
AGHMatrix<T> AGHMatrix<T>::gauss() {
  AGHMatrix<T> newmatrix(*this); //nowa macierz kopią wejściowej macierzy.
  for (int k = 0; k < rows - 1; ++k) {
    T pivot = newmatrix.matrix[k][k]; // będziemy wykonywać operacje elementarne z
                                      // wykorzystaniem wartości pivota.
    for (int i = k + 1; i < rows; ++i) {   // przejście po kolumnie od miejsca poniżej
                                           // pivota do dołu
      T factor = newmatrix.matrix[i][k] / pivot; // obliczenie współczynnika potrzebnego
                                                 // do wyzerowania danej komórki macierzy
      for (int j = 0; j < cols; ++j) {
        newmatrix.matrix[i][j] -= factor * newmatrix.matrix[k][j];
        // dokonanie operacji elementarnej na wierszu. Dany element zerowany.
      }
    }
  }
  return newmatrix;
}
```

Algorytm został przetestowany na następującym przykładzie:

```
     std::vector<std::vector<double>> init5 { {1, 2, -3, -1, 0},
                                              {0, -3, 2, 6, -8},
                                              {-3, -1, 3, 1, 0},
                                              {2, 3, 2, -1, -8}
                                            };
AGHMatrix<double> mat9(init5);
```

Wynik:

```
1, 2, -3, -1, 0,
0, -3, 2, 6, -8,
0, 0, -2.66667, 8, -13.3333,
0, 0, 0, 21, -42,
```

Otrzymana macierz zgadza się z obliczeniami przeprowadzonymi ręcznie. Rozwiązanie równania to: (-1, 2, -1, -2).

Zadanie 6. Zaimplementuj metodę Jackobiego.

```
template <typename T>
std::vector<T> AGHMatrix<T>::Jacobi(int iterations) {
  std::vector<T> res(rows);
  std::vector<T> res_old(rows);
  for (int i = 0; i < rows; ++i) {
    res[i] = 0;
  }
  for (int it = 0; it < iterations; ++it) {
    for (int w = 0; w < rows; ++w) {
      T sum = 0;
      res_old[w] = res[w];
      for (int j = 0; j < rows; ++j)
        if (w != j) {
          sum += matrix[w][j] * res_old[j];
```

```
        }
      res[w] = (matrix[w][cols - 1] - sum) / matrix[w][w];
    }
  }
  return res;
}
```

Metoda została przetestowana na następującym przykładzie:

```
     std::vector<std::vector<double>> init6 { {4, -1, -0.2, 2, 30},
                                               {-1, 5, 0, -2, 0},
                                               {0.2, 1, 10, -1, -10},
                                               {0, -2, -1, 4, 5}
                                              };
AGHMatrix<double> mat8(init6);
auto res = mat8.Jacobi(300);
    for (int i = 0; i < res.size(); ++i)
      std::cout << "x[" << i << "] = " << res[i] << std::endl;
```

Wynik dla 3 iteracji:

(6.82, 2, -1.71, 1.71)

Wynik dla 300 iteracji:

(6.96, 2.22, -1.15, 2.07)

Wynik dokładny:

(6.96, 2.22, -1.15, 2.07)

Plusem tej metody jest możliwość zrównoleglenia obliczeń, ponieważ miejsca zerowe obliczane są niezależnie od siebie. W kolejnych krokach jesteśmy w stanie znaleźć kolejne przybliżenia rozwiązania układu równań, jednak nie zawsze odpowiednio dokładne przybliżenie uda się uzyskać w akceptowalnej liczbie kroków.