Maciej Kozub

Laboratorium 2 – układy równań liniowych

1. Dodawanie i mnożenie macierzy

```
for (int i = 0; i < this->get_rows(); i++) {
    for (int j = 0; j < this->get_cols(); j++) {
        mat3.matrix[i][j] = this->matrix[i][j]+rhs.matrix[i][j];
    }
}
return mat3;
```

```
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++){
            mat3.matrix[i][j] += this->matrix[i][k]*rhs.matrix[k][j];
        }
    }
}
return mat3;
```

2. Macierz symetryczna, transponowana i wyznacznik macierzy

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

```
AGHMatrix<T> AGHMatrix<T>::transpose() {
    if(this->is_symmetric())
        return *this;

    AGHMatrix<typeof(this->matrix[0][0])> mat3 (this->get_cols(), this->get_rows(), 0);
    for (int i = 0; i < this->get_rows(); i++) {
        for (int j = 0; j < this->get_cols(); j++) {
            mat3.matrix[j][i] = this->matrix[i][j];
        }
    }
    return mat3;
}
```

```
double tmp1, tmp2, result = 1, idx, counter = 1;
int temp[this->get_rows() + 1];
for(int i = 0; i < this->get_rows(); i++){
    idx = i;
    while(this->matrix[idx][i] == 0 && idx < this->get_rows()){
        idx++;
    }
    if(idx == this->get_rows()){
        continue;
    }
    if(idx != i){
        for(int j = 0; j < this->get_rows(); j++){
            std::swap(this->matrix[idx][j], this->matrix[i][j]);
        }
        result *= pow( _X: -1,  _Y: idx - i);
    }
    for(int j = 0; j < this->get_rows(); j++){
        temp[j] = this->matrix[i][j];
    }
    for(int j = i+1; j < this->get_rows(); j++){
        tmp1 = temp[i];
        tmp2 = this->matrix[j][i];
        for(int k = 0; k < this->get_rows(); k++){
            this->matrix[j][k] = (tmp1 * this->matrix[j][k]) - (tmp2 * temp[k]);
        }
        counter *= tmp1;
    }
}
for(int i = 0; i < this->get_rows(); i++){
    result *= this->matrix[i][i];
}
return (result / counter);
```

3. Faktoryzacja LU

```
for (int i = 0; i < this->get_cols(); i++) {
    for (int k = i; k < this->get_cols(); k++) {
        typeof(this->matrix[0][0]) sum = 0;
        for (int j = 0; j < i; j++)
            sum += (lower.matrix[i][j] * upper.matrix[j][k]);
        upper.matrix[i][k] = this->matrix[i][k] - sum;
    }
    for (int k = i; k < this->get_cols(); k++) {
        if (i == k)
            lower.matrix[i][i] = 1;
        else {
            typeof(this->matrix[0][0]) sum = 0;
            for (int j = 0; j < i; j++)
                sum += (lower.matrix[k][j] * upper.matrix[j][i]);
            lower.matrix[k][i] = (this->matrix[k][i] - sum) / upper.matrix[i][i];
        }
    }
}
std::pair<AGHMatrix<T>, AGHMatrix<T>> pair = std::make_pair(lower, upper);
return pair;
```

4. Faktoryzacja Cholesky'ego

```
for (int i = 0; i < this->get_cols(); i++) {
    for (int j = 0; j <= i; j++) {
        typeof(this->matrix[0][0]) sum = 0;
        if (j == i){
            for (int k = 0; k < j; k++)
                sum += pow(lower.matrix[j][k], 2);
            lower.matrix[j][j] = sqrt(this->matrix[j][j] - sum);
        } else {
            for (int k = 0; k < j; k++)
                sum += (lower.matrix[i][k] * lower.matrix[j][k]);
            lower.matrix[i][j] = (this->matrix[i][j] - sum) /
                          lower.matrix[j][j];
        }
    }
}
upper = lower.transpose();
std::pair<AGHMatrix<T>, AGHMatrix<T>> pair = std::make_pair(lower, upper);
return pair;
```

Algorytm faktoryzacji LU działa dla wszystkich kwadratowych macierzy (pod warunkiem że na przekątnej nie znajdują się wartości zerowe) i wyznacza on naprzemiennie wiersze z macierzy upper i kolumny z macierzy lower. Jego zasada działania opiera się na założeniu że w jednej z wynikowych macierzy na przekątnej znajdują się same jedynki, wtedy jesteśmy w stanie wyznaczać kolejne wartości obydwóch macierzy wynikowych.

Algorytm faktoryzacji Choleskiego działa jedynie dla macierzy symetrycznych dodatnio określonych. Taką macierz możemy rozłożyć na macierz dolną trójkątną i jej transpozycję. Wystarczy więc wyznaczyć macierz dolną trójkątną zgodnie ze wzorami:

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

A następnie dokonać transpozycji macierzy lower i uzyskać macierz trójkątną górną upper. Algorytm ten oblicz współczynniki macierzy lower wierszami.

5. Eliminacja Gaussa

```
for (int i = 0; i < this->get_rows() - 1; i++) {
    for (int j = i+1; j < this->get_rows(); j++) {
        if(this->matrix[i][i] == 0) {
            std::cout<<"error -> dividing by 0";
            return;
        }
        double factor = -this->matrix[j][i] / this->matrix[i][i];
        for (int k = i; k < this->get_cols(); k++)
            this->matrix[j][k] += factor * this->matrix[i][k];
    }
}
for (int i = this->get_rows()-1; i >= 0; i--) {
    for (int j =this->get_rows()-1; j > i; j--) {
        this->matrix[i][this->get_rows()] -= this->matrix[i][j] * this->matrix[j][this->get_rows()];
        this->matrix[i][j] = 0;
    }
    if(this->matrix[i][i] == 0) {
        std::cout<<"error -> dividing by 0";
        return;
    }
    this->matrix[i][this->get_rows()] /= this->matrix[i][i];
    this->matrix[i][i] = 1;
}
```

W metodzie eliminacji Gaussa wykonujemy operacje na wierszach macierzy tak aby otrzymać macierz trójkątną. Kolejno przechodząc po wierszach macierzy odejmujemy od nich pierwszy wiersz przemnożony przez taki czynnik aby kolejne pierwsze elementy w wierszy zerowały się. Czynność powtarzamy zaczynając od kolejnych wierszy i zerując współczynniki macierzy na kolejnych pozycjach w wierszu aż do otrzymania macierzy trójkątnej. Wtedy przechodzimy do wyznaczania do znajdowania niewiadomych, zaczynając od ostatniego wiersza macierzy (zawiera tylko wartość na przekątnej – ostatnią w wierszu) wyznaczamy pierwszą niewiadomą. Następnie przesuwając się w górę macierzy wyznaczamy kolejne niewiadome jeżeli to konieczne odejmując wartości wcześniej obliczonych (odpowiadające im kolumny macierzy) aż wyznaczymy wszystkie.

6. Metoda Jacobiego

```
for(int m = 0; m < iterations; m++) {
    for (int i = 0; i < this->get_rows(); i++) {
        temp.matrix[0][i] = this->matrix[i][this->get_rows()];
        for (int j = 0; j < this->get_rows(); j++) {
            if (i != j) {
                temp.matrix[0][i] -= this->matrix[i][j] * result.matrix[0][j];
            }
        }
    }
    for (int i = 0; i < this->get_rows(); i++) {
        result.matrix[0][i] = temp.matrix[0][i] / this->matrix[i][i];
    }
}
return result;
```

Metoda Jacobiego działa tylko dla macierzy silnie diagonalnie dominującej i jest algorytmem aproksymującym. Początkowo ustalony jest wektor rozwiązania i w kolejnych iteracjach jest on coraz dokładniej przybliżany. Wraz ze wzrostem iteracji osiągana jest większa zbieżność otrzymanych wyników z rzeczywistymi. Początkowo wektor wynikowy jest wektorem zerowym następnie w każdej iteracji składowym tego wektora przypisuje się wartość wyznaczoną z macierzy podzieloną przez współczynnik na przekątnej macierzy o odpowiednim indeksie.