Laboratorium 2: Równania liniowe

Metody Obliczeniowe w Nauce i Technice

Wiktor Tarsa

Zadanie 1
Zgodnie z instrukcją zaimplementowałem funkcje realizujące:

dodawanie macierzy

```
// Addition of two matrices
template<typename T>
AGHMatrix<T> AGHMatrix<T>::operator+(const AGHMatrix<T>& rhs)
{
   AGHMatrix<T> result(rhs);
   for(int r = 0; r < rows; r++){
       for(int c = 0; c < cols; c++){
           result.matrix[r][c] += matrix[r][c];
       }
   }
   return result;
}
```

mnożenie macierzy

```
// Left multiplication of this matrix and another
template<typename T>
AGHMatrix<T> AGHMatrix<T>::operator*(const AGHMatrix<T>& rhs)
{
   AGHMatrix<T> result(rows, rhs.cols, 0);
   for(int r = 0; r < rows; r++){
       for(int c = 0; c < rhs.cols; c++){
           for(int i = 0; i < rhs.rows; i++){
               result.matrix[r][c] += matrix[r][i]*rhs.matrix[i][c];
           }
       }
   }
   return result;
}
```

Zadanie 2
Zgodnie z instrukcją zaimplementowałem funkcje:

symmetric
Sprawdza, czy macierz jest symetryczna.

```
template<typename T>
bool AGHMatrix<T>::symmetric() {
    for(int r = 0; r < rows; r++){
        for(int c = 0; c < cols; c++){
            if(matrix[r][c] != matrix[c][r]) return false;
        }
    }
    return true;
}
```

complementary_cofactor
Funkcja pomocnicza - wyznacza i zwraca minor macierzy głównej.

```
template<typename T>
AGHMatrix<T> AGHMatrix<T>::complementary_cofactor(unsigned col, unsigned
row) {
    AGHMatrix<T> result(this->rows-1, this->cols-1, 0);
    int r_shift = 0;
    int c_shift = 0;
    for(int r = 0; r < this->rows-1; r++){
        if(r == row) r_shift = 1;
        for(int c = 0; c < this->cols-1; c++){
            if(c == col) c_shift = 1;
            result.matrix[r][c] = this->matrix[r+r_shift][c+c_shift];
        }
        c_shift = 0;
    }
    return result;
}
```

find_determinant
Funkcja zwracająca wyznacznik macierzy używając rozwinięcia Laplace'a.

```
template<typename T>
T AGHMatrix<T>::find_determinant(AGHMatrix<T> rhs, unsigned size) {
    T result = 0;
    if(size == 1) return rhs.matrix[0][0];
    if(size == 2) {
```

```
        return rhs.matrix[0][0]*rhs.matrix[1][1]-
               rhs.matrix[0][1]*rhs.matrix[1][0];
    }
    for(int col = 0; col < size; col++){
        result += pow(-1, col+2) * rhs.matrix[0][col] *
find_determinant(rhs.complementary_cofactor(col, 0), size-1);
    }
    return result;
}
```

transpose

Zwraca macierz transponowaną.

```
template<typename T>
AGHMatrix<T> AGHMatrix<T>::transpose() {
    AGHMatrix<T> result(cols, rows, 0);
    for(int r = 0; r < cols; r++){
        for(int c = 0; c < rows; c++){
            result.matrix[r][c] = matrix[c][r];
        }
    }
    return result;
}
```

Zadanie 3

Korzystając z kodu dostarczonego do laboratorium zaimplementowałem algorytm faktoryzacji macierzy LU. Algorytm został przetestowany na przykładzie z wikipedii (https://pl.wikipedia.org/wiki/Metoda_LU). Otrzymane wyniki są zgodne z przewidywanymi.

```
template<typename T>
void AGHMatrix<T>::LU_decomposition(AGHMatrix<T> &L, AGHMatrix<T> &U) {
    for(int i = 0; i < rows; i++){
        L.matrix[i][i] = 1;
    }
    for(int i = rows; i > 0; i--){ // for each row and column
        for(int c = cols-i; c < cols; c++){ // row in U matrix
            U.matrix[rows-i][c] = 1;
            double tmp = 0;
            for(int j = 0; j < cols; j++){ // multiplication
                if(j == rows-i) U.matrix[rows-i][c] *= U.matrix[j][c];
                else tmp += L.matrix[rows-i][j]*U.matrix[j][c];
```

```
            }
            U.matrix[rows-i][c]=(matrix[rows-i][c]-tmp)/U.matrix[rows-i][c];
        }

        for(int r = rows-i; r < rows; r++){ // col in L matrix
            L.matrix[r][cols-i] = 1;
            double tmp = 0;
            for(int j = 0; j < cols; j++){ // multiplication
                if(j == cols-i) L.matrix[r][cols-i] *= U.matrix[j][cols-i];
                else tmp += L.matrix[r][j]*U.matrix[j][cols-i];
            }
            L.matrix[r][cols-i]=(matrix[r][cols-i]-tmp)/L.matrix[r][cols-i];
        }
    }
}
```

Zadanie 4

Korzystając z kodu dostarczonego do laboratorium zaimplementowałem algorytm faktoryzacji Cholesky'ego macierzy. Algorytm przetestowałem na przykładzie z wikipedii. (https://en.wikipedia.org/wiki/Cholesky_decomposition). Otrzymane wyniki są zgodne z przewidywanymi.

```
template<typename T>
void AGHMatrix<T>::cholesky_decomposition(AGHMatrix<T> &L, AGHMatrix<T>
&LT) {
    for(int i = 0; i < rows; i++){
        for(int j = 0; j <= i; j++){
            double tmp = 0;
            if(i == j){
                for(int k = 0; k < j; k++){
                    tmp += pow(L.matrix[j][k], 2);
                }
                L.matrix[j][j] = sqrt(matrix[j][i] - tmp);
            }
            else{
                for(int k = 0; k < j; k++){
                    tmp += (L.matrix[i][k]*L.matrix[j][k]);
                }
                L.matrix[i][j] = (matrix[i][j] - tmp)/L.matrix[j][j];
            }
        }
    }
```

```
    LT = L.transpose();
}
```

Zarówno algorytm LU jak i algorytm Cholesky'ego służą do faktoryzacji macierzy. Jednak występują między nimi pewne różnice:

- algorytm Cholesky'ego można używać tylko do faktoryzacji macierzy symetrycznych
- algorytm faktoryzacji LU można używać do faktoryzacji każdej macierzy, jednak nieodpowiednia implementacja algorytmu może doprowadzić do sytuacji, w której algorytm dzieli przez zero.
- algorytm Cholesky'ego jest mniej więcej dwa razy bardziej wydajny od algorytmu faktoryzacji LU w rozwiązywaniu układów równań liniowych.

Główna idea metody LU to naprzemienne wyznaczanie kolejnych rzędów i kolumn odpowiednio w macierzach U i L. Macierz L jest macierzą trójkątną dolną, a macierz U - macierzą trójkątną górną, dlatego podczas wyznaczania kolejnych elementów zawsze otrzymujemy równanie z jedną niewiadomą. Porównując je z wartością w macierzy, w której dokonujemy faktoryzacji możemy wyznaczać kolejne wartości w macierzach L i U.

Główna idea metody Cholesky'ego to rozkład macierzy głównej ma macierz L oraz macierz LT(transponowaną macierzy L). Algorytm polega na wyznaczeniu macierzy L - wartości macierzy LT możemy uzyskać przestawiając numer rzędu i kolumny w macierzy L.
Wartości macierzy L wyznaczamy od rzędu o najmniejszym indeksie oraz od kolumny o najmniejszym indeksie. Sposób wyznaczenia konkretnej wartości to po prostu przemnożenie macierzy L przez macierz LT - kolejność wyznaczania wartości zapewnia otrzymanie równania z dokładnie jedną niewiadomą. Porównując je z wartością macierzy głównej wyznaczamy kolejne wartości w macierzy L.

Warto w tym miejscu przypomnieć, że macierz L jest macierzą trójkątną - niemal połowę jej wartości stanowią zera. Podczas rozkładu nie wyznaczamy wartości w miejscach, gdzie wartość macierzy jest równa 0. W związku z tym wykonujemy mniej operacji - mniej więcej o połowę. To tłumaczy, dlaczego algorytm faktoryzacji Cholesky'ego macierzy jest mniej więcej dwa razy bardziej wydajny od algorytmu faktoryzacji LU.

Zadanie 5
Zgodnie z instrukcją, zaimplementowałem funkcję, która realizuje eliminację Gaussa.
Funkcja zwraca macierz wyników rozwiązanego układu równań.

```
template<typename T>
AGHMatrix<T> AGHMatrix<T>::gauss_elimination() {
    AGHMatrix result(rows, 1, 0);
```

Cała funkcja składa się z trzech głównych części, dlatego opiszę każdą z nich osobno.

Układanie wierszy macierzy głównej w malejącej kolejności

```
for (int i = 0; i < rows; i++) {
   for (int k = i + 1; k < rows; k++) {
       if (abs(matrix[i][i]) < abs(matrix[k][i])) {
           for (int j = 0; j <= rows; j++) {
               double temp = matrix[i][j];
               matrix[i][j] = matrix[k][j];
               matrix[k][j] = temp;
           }
       }
   }
}
```

Wiersze w macierzy głównej układam w kolejności malejącej (porównuję wartości bezwzględne poszczególnych komórek macierzy). Takie ułożenie znacznie ułatwia sprowadzanie macierzy do postaci schodkowej.

Sprowadzanie macierzy do postaci schodkowej

```
for (int i = 0; i < rows - 1; i++) {
   for (int k = i + 1; k < rows; k++) {
       double t = matrix[k][i] / matrix[i][i];
       for (int j = 0; j <= rows; j++)
           matrix[k][j] = matrix[k][j] - t*matrix[i][j];
   }
}
```

Sprowadzam macierz do postaci schodkowej. W tym celu wyznaczam iloraz współczynnika i-tego rzędu oraz i-tej kolumny macierzy do wszystkich współczynników znajdujących się w niższych rzędach tej samej kolumny. Wyznaczony iloraz umożliwia wyzerowanie współczynników niższych rzędów poprzez odjęcie odpowiedniej wielokrotności i-tego rzędu od niższych rzędów.

Wyznaczenie rozwiązania

```
for (int i = rows - 1; i >= 0; i--) {
   result.matrix[i][0] = matrix[i][rows];
   for (int j = i + 1; j < rows; j++) {
       if (j != i) result.matrix[i][0] = result.matrix[i][0] - matrix[i][j]
* result.matrix[j][0];
   }
```

```
    result.matrix[i][0] = result.matrix[i][0] / matrix[i][i];
}
```

Wyznaczam rozwiązanie równania. W ostatnim rzędzie powinienem dostać równanie z jedną niewiadomą, co pozwala na natychmiastowe otrzymanie jej wartości.
Iterując od końca, w każdym kolejnym rzędzie powinienem otrzymać równanie z liczbą niewiadomych większą od 1 od liczby niewiadomych w poprzednim rzędzie. Jednak wartości niewiadomych w poprzednich rzędach zostały wyznaczone - dlatego za każdym razem mam do wyznaczenia wartość jednej niewiadomej.

Testowanie rozwiązania

Do przetestowania rozwiązania użyłem kodu w języku python dołączonego do materiałów tego laboratorium. Otrzymałem następujące wyniki:

```
wiktor@wiktor-ThinkPad-T470s:~/studia/mownit/laborki/lab2$ python3 test.py
True
[[ 0.21602477]
 [-0.00791511]
 [ 0.63524333]
 [ 0.74617428]]
```

Wyniki zwrócone przez mój program są dokładnie takie same:

```
0.21602477,
-0.00791511,
0.63524333,
0.74617428,
```

dlatego uważam, że funkcja została zaimplementowana poprawnie.

Zadanie 6

Zgodnie z instrukcją zaimplementowałem funkcję, która rozwiązuje układ równań korzystając z metody Jacobiego.

Funkcja zwraca macierz wyników rozwiązanego układu równań.

```
template<typename T>
AGHMatrix<T> AGHMatrix<T>::jacobi_method() {
    AGHMatrix result(rows, 1, 0);
    AGHMatrix tmp(rows, 1, 0);
    AGHMatrix N(rows, 1, 0);
    AGHMatrix LU(rows, rows, 0);

    for (int i = 0; i < rows; i++) {
        N.matrix[i][0] = 1 / matrix[i][i];
    }
```

```
    for (int i = 0; i < rows; i++) {
        for (int j = 0; j < rows; j++) {
            if (i == j) {
                LU.matrix[i][j] = 0;
            } else {
                LU.matrix[i][j] = -(matrix[i][j] * N.matrix[i][0]);
            }
        }
    }

    for (int k = 0; k < 25; k++) {
        for (int i = 0; i < rows; i++) {
            tmp.matrix[i][0] = N.matrix[i][0] * matrix[i][rows];
            for (int j = 0; j < rows; j++)
                tmp.matrix[i][0] += LU.matrix[i][j] * result.matrix[j][0];
        }
        for (int i = 0; i < rows; i++)
            result.matrix[i][0] = tmp.matrix[i][0];
    }

    return result;
}
```

Przetestowałem rozwiązanie korzystając z tych samych danych wejściowych, co w zadaniu 5. Otrzymane wyniki są dokładnie takie same:

```
eliminacja Gaussa:
0.21602477,
-0.00791511,
0.63524333,
0.74617428,


metoda Jacobiego:
0.21602477,
-0.00791511,
0.63524333,
0.74617428,
```

Niewątpliwym plusem metody Jacobiego jest możliwość otrzymania dowolnie dużej dokładności rozwiązania. Każda dodatkowa iteracja w końcowej pętli funkcji jacobi_method zwiększa dokładność rozwiązania.