MOwNiT2 Lab3

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

Dodawanie macierzy:

```
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()){
   std::string txt = "Different sizes of matrices.";
   throw std::invalid_argument(txt);
 }

 AGHMatrix<T> newMatrix(this->matrix);
 for(int i=0; i<this->get_cols(); i++){
   for(int j=0; j<this->get_rows(); j++){
     newMatrix(i,j) += rhs(i,j);
   }
 }
 return newMatrix;
}
```

$$\begin{bmatrix} 1.2 & 1.2 & 1.2 & 1.2 & 1.2 \\ 1.2 & 1.2 & 1.2 & 1.2 & 1.2 \\ 1.2 & 1.2 & 1.2 & 1.2 & 1.2 \\ 1.2 & 1.2 & 1.2 & 1.2 & 1.2 \\ 1.2 & 1.2 & 1.2 & 1.2 & 1.2 \end{bmatrix} + \begin{bmatrix} 2.8 & 2.8 & 2.8 & 2.8 & 2.8 \\ 2.8 & 2.8 & 2.8 & 2.8 & 2.8 \\ 2.8 & 2.8 & 2.8 & 2.8 & 2.8 \\ 2.8 & 2.8 & 2.8 & 2.8 & 2.8 \\ 2.8 & 2.8 & 2.8 & 2.8 & 2.8 \end{bmatrix} =$$

Wynik:

```
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,
```

- Dodawanie macierzy jest zdefiniowane dla macierzy tych samych rozmiarów.
- Wynikiem jest macierz tego samego rozmiaru.
- Elementy są sumą wyrazów o tych samych współrzędnych.

Mnożenie macierzy:

```
template<typename T>
AGHMatrix<T> AGHMatrix<T>::operator*(const AGHMatrix<T>& rhs)
{
 if(this->get_cols() != rhs.get_rows()){
   std::string txt = "Invalid size of matrices.";
   throw std::invalid_argument(txt);
 }
 AGHMatrix<T> newMatrix(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++){
       newMatrix(i,j) += this->matrix[i][k] * rhs.matrix[k][j];
     }
   }
 }
 return newMatrix;
}
```

$$\begin{bmatrix} 1 & 2 & 3 \\ 4 & 5 & 6 \end{bmatrix} * \begin{bmatrix} 7 & 8 \\ 9 & 10 \\ 11 & 12 \end{bmatrix} =$$

Wynik:

```
58, 64,
139, 154,
```

- Pierwsza macierz musi mieć tyle kolumn co druga wierszy.
- Wynikiem jest macierz, która ma tyle wierszy co pierwszy argument mnożenia i kolumn tyle co drugi argument.
- Elementy są iloczynem skalarnym odpowiednich wierszy pierwszej macierzy i kolumn drugiej.
- Mnożenie macierzy nie jest przemienne!

Zadanie 2 Proszę zaimplementować:

1. Funkcję/metodę, która sprawdzi czy macierz jest symetryczna.
2. Funkcję/metodę, która obliczy wyznacznik macierzy.
3. (*) Metodę transpose().

1. isSymmetric():

```
template<typename T>
bool AGHMatrix<T>::isSymmetric()
{
 if(this->get_cols() != this->get_rows()){
   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 detRecursively() korzysta ze wzoru (dla macierzy $A = [a_{ij}]$):

- ➔ jeśli $n = 1$: $detA \ = a_{11}$
- ➔ jeśli $n = 2$: $detA \ = a_{11} * a_{22} - a_{12} * a_{21}$
- ➔ jeśli $n > 2$: $detA \ = \sum_{i=1}^{n}(-1)^{i+j}a_{ij}A_{ij}$, gdzie $A_{ij}$to minor powstały poprzez skreślenie i-tego wiersza i j-tej kolumny.

$$\begin{bmatrix} 1 & 2 & 3 \\ 2 & 5 & 4 \\ 3 & 4 & 9 \end{bmatrix}$$

Symetryczna?

```
Is symmetric? 1
```

2. det():

```
template<typename T>
T AGHMatrix<T>::det()
{
 if(this->get_cols() != this->get_rows()){
   std::string txt = "Not a square matrix.";
   throw std::invalid_argument(txt);
 }

 return detRecursively(this->matrix,this->get_rows());
}
```

```
template<typename T>
T detRecursively(std::vector<std::vector<T>> matrix, int size)
{
 // Base cases
 if(size == 1) return matrix[0][0];
 else if(size == 2) return matrix[0][0] * matrix[1][1] - matrix[0][1] *
matrix[1][0];
 else{
   T result = 0;
   int sign = 1;
   for(int i=0; i<size; i++){

     // new matrix without 0-th row and i-th column
     std::vector<std::vector<double>> tmp;
     for(int j=0; j<size; j++){
       // exluding i-th row
       if(j != 0){
         tmp.push_back(matrix[j]);
         // excluding i-th column
         tmp.back().erase(tmp.back().begin() + i);
       }
     }
     result += sign * matrix[0][i] * detRecursively(tmp, size-1);
     sign = -sign;
   }
   return result;
 }
}
```

Wyznacznik powyższej macierzy:

```
Det = -4
```

3. transpose():

```
template<typename T>
AGHMatrix<T> AGHMatrix<T>::transpose()
{
 AGHMatrix<T> transposed(this->get_cols(), this->get_rows(), 0.0);
 for(int i=0; i<this->get_rows(); i++){
   for(int j=0; j<this->get_cols(); j++){
     transposed(j,i) = this->matrix[i][j];
   }
 }
 return transposed;
}
```

$$\begin{bmatrix} 1 & 2 & 3 \\ 4 & 5 & 6 \end{bmatrix}$$

Transpozycja:

```
1, 4,
2, 5,
3, 6,
```

Zadanie 3 Proszę zaimplementować algorytm faktoryzacji LU macierzy. Algorytm przetestować na przykładzie z wikipedii lub korzystając z poniższego kodu.
Algorytm:

```
template<typename T>
std::pair<AGHMatrix<T>, AGHMatrix<T>> AGHMatrix<T>::LU()
{
 if(this->get_cols() != this->get_rows()){
   std::string txt = "Not a square matrix.";
   throw std::invalid_argument(txt);
 }
  std::pair<AGHMatrix<T>, AGHMatrix<T>> result(
     AGHMatrix<T>(this->get_rows(),this->get_cols(),0),
     AGHMatrix<T>(this->get_rows(),this->get_cols(),0)
 );
 for(int i=0; i<result.first.get_cols(); i++){
   result.first(i,i) = 1;
 }
 T element;

 for(int i=0; i<result.first.get_cols(); i++){
   // i-th row in second matrix
   for(int j=0; j<result.first.get_rows(); j++){
     // only above or on diagonal
     if(j >= i){
       element = this->matrix[i][j];
       for(int k=0; k<i; k++){
         element -= result.first(i,k) * result.second(k,j);
       }
       result.second(i,j) = element;
     }
   }
   // i-th column in first matrix
   for(int j=0; j<result.first.get_rows(); j++){
     // only below diagonal
     if(i < j){
       element = this->matrix[j][i];
       for(int k=0; k<i; k++){
         element -= result.first(j,k) * result.second(k,i);
       }
       result.first(j,i) = element / result.second(i,i);
     }
   }
 }
 return result;
}
```

Zadanie 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.

Algorytm:

```
template<typename T>
std::pair<AGHMatrix<T>, AGHMatrix<T>> AGHMatrix<T>::cholesky()
{
 if(this->get_cols() != this->get_rows()){
   std::string txt = "Not a square matrix.";
   throw std::invalid_argument(txt);
 }
  AGHMatrix<T> mat(this->get_rows(),this->get_cols(),0);

 for(int i=0; i<this->get_rows(); i++){
   // only below or on diagonal
   for(int j=0; j<=i; j++){
     T sum = 0;

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

 return result;
}
```

LU:	Cholesky:
$\begin{bmatrix} 5 & 3 & 2 \\ 1 & 2 & 0 \\ 3 & 0 & 4 \end{bmatrix}$	$\begin{bmatrix} 4 & 12 & -16 \\ 12 & 37 & -43 \\ -16 & -43 & 98 \end{bmatrix}$
Wyniki:	

`1, 0, 0,`
`0.2, 1, 0,`
`0.6, -1.28571, 1,`
 	
`5, 3, 2,`
`0, 1.4, -0.4,`
`0, 0, 2.28571,`
	`2, 0, 0,`
	`6, 1, 0,`
	`-8, 5, 3,`
	
	`2, 6, -8,`
	`0, 1, 5,`
	`0, 0, 3,`

Przedstawiamy macierz jako dwie macierze: dolnotrójkątną, **która ma tylko 1 na przekątnej** oraz górnotrójkątną.	Przedstawiamy macierz jako dwie macierze: dolnotrójkątną oraz górnotrójkątną, **która jest macierzą transponowaną pierwszej**.
$A = L * U$ $A = [a_{ij}],\ L = [l_{ij}],\ U = [u_{ij}]$	$A = L * L^T$ $A = [a_{ij}],\ L = [l_{ij}]$
Metoda naprzemiennie liczy wiersze macierzy U i kolumny macierzy L.	Metoda liczy kolejno wiersze dla jednej z macierzy, a drugą otrzymuje poprzez transpozycję.
$u_{ij} = a_{ij} - \sum_{k=1}^{i-1} l_{ik}u_{ki}$ $l_{ji} = 1/u_{ii} * (a_{ji} - \sum_{k=1}^{i-1} l_{jk}u_{ki})$	$l_{jj} = \sqrt{a_{jj} - \sum_{k=1}^{j-1} {l^2}_{jk}}$ $l_{ij} = 1/l_{jj}(a_{ij} - \sum_{k-1}^{j-1} l_{ik}l_{jk}),$ dla $i > j$
Należy uważać na dzielenie przez zero (można zmodyfikować metodę, w celu uniknięcia tego).	Działa tylko dla macierzy symetrycznych, dodatnio określonych.

Zadanie 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).

Algorytm:

```
template<typename T>
AGHMatrix<T> gaussEl(AGHMatrix<T> augmented)
{
 if(augmented.get_rows() != augmented.get_cols() - 1){
   std::string txt = "Incorrect size.\nAugmented matrix has size N x N+1.";
   throw std::invalid_argument(txt);
 }

 int rows = augmented.get_rows();

 // zeroing coeffs below the leading one in the i-th column
 for(int i=0; i<rows-1; i++){
   // skipping the leading coefficient
   for(int j=i+1; j<rows; j++){
       // getting the factor zeroing in the given row
       double factor = -1*augmented(j,i)/augmented(i,i);
       for(int k=i; k<rows+1; k++){
         // operation must be row elementary so repeating
         // for each element in row
         augmented(j,k) += factor*augmented(i,k);
       }
   }
 }
 // matrix is in the correct form

 for(int i=0;i<rows;i++){
    if(augmented(i,i)==0){
     std::string txt;
     if(augmented(i,i+1)==0) txt = "Infinitely many solutions.";
     else txt = "No solution.";
     throw std::invalid_argument(txt);
   }
  }

 AGHMatrix<T> solution(rows, 1, 0);
```

```
// getting the answers (starting from the last row)
 for(int i=rows-1; i>=0; i--){
   // row is in form: augmented(i,i)*solution(i,1) = augmented(i,n)
   solution(i,0) = augmented(i,rows) / augmented(i,i);
   // zeroing coeffs connected with computed variable
   // (to achieve rows: a * x = b)
   for(int j=i-1; j>=0; j--){
     // its like getting the constants on the same side of equation
     // since variable has been computed
     augmented(j,rows) -= solution(i,0) * augmented(j,i);
     // not needed in the algorithm
     augmented(j,i) = 0;
   }
 }
  return solution;
}
```

- Metoda polega na doprowadzeniu macierzy do postaci schodkowej:

$$\begin{bmatrix} a_{11} & a_{12} & a_{13} & \dots & a_{1m} \\ 0 & a_{22} & a_{23} & \dots & a_{2m} \\ 0 & 0 & a_{33} & \dots & a_{3m} \\ \vdots & \vdots & \vdots & \ddots & \vdots \\ 0 & 0 & 0 & \dots & a_{nm} \end{bmatrix}$$

- Można korzystać wyłącznie z operacji elementarnych na wierszach.
- Po przekształceniu macierzy można idąc od dołu łatwo obliczać wartości kolejnych zmiennych i wstawiać je wyżej.

Macierz rozszerzona do testu:

$$\begin{bmatrix} 0.0001 & -5.03 & 5.809 & 7.832 & 9.574 \\ 2.266 & 1.995 & 1.212 & 8.008 & 7.219 \\ 8.85 & 5.681 & 4.552 & 1.302 & 5.73 \\ 6.775 & -2.253 & 2.908 & 3.97 & 6.291 \end{bmatrix}$$

Program testowy (Python):

```
import numpy as np

A = np.matrix([[0.0001, -5.0300, 5.8090, 7.8320],
              [2.2660, 1.9950,  1.2120, 8.0080],
              [8.8500, 5.6810,  4.5520, 1.3020],
              [6.7750, -2.253,  2.9080, 3.9700]])

b = np.matrix([9.5740, 7.2190, 5.7300, 6.2910]).transpose()

x = np.linalg.solve(A, b)

# Checking
print(np.allclose(np.dot(A, x), b))

print(x)
```

Wynik testu:

```
True
[[ 0.21602477]
 [-0.00791511]
 [ 0.63524333]
 [ 0.74617428]]
```

Wynik działania napisanego algorytmu dla tej samej macierzy:

```
0.216025,
-0.00791511,
0.635243,
0.746174,
```