C++ Matrix Library – Report

○ Introduction

The topic of this project is to write C++ class that allows to manipulating, doing computations and working with matrices. The project is made of the following files :

- Matrix.hpp (header file)
- Matrix.cpp (code of each function)
- main.cpp (test program)
- Makefile (to compile the whole project in a very simple way)

Now I'm going to describe the private and public parts of the class and finally, I will briefly explain the test program.

○ private part

The private part is very simple, I have only put two integers M and N for dimensions of the matrix, respectively the number of lines and the number of columns.

Finally, of course, we need to store the matrix itself. To do it, I have chosen to use a vector of vectors of floats. I use it because vectors are a very very smart thing in C++. Effectively, without using dynamic allocation, vectors allow to use memory, resizing it as we want, adding, deleting elements very simply. The limit of this technique is that all is stored in the stack, and not in the heap contrary to dynamic allocation. So if we want to work with really big matrices, we should update this program to dynamically allocate the vectors. But in this case, the complexity of the code is increased because it will not be so simply to resize the vectors for example.

○ public part

▪ Constructors

- **Matrix();**

This constructor is the default one, with no parameters. In this case, I chose to initialize the matrix with dimensions (1, 1) and a 0 for the value. So the only thing to do is to call the « main » constructor that is the next one, with the m and n values (1 and 1), and the Zero constant (that I will explain in the next constructor) to put the 0 value.

- **Matrix(int m, int n, Type t);**

This constructor is the « main » one because it initializes the vectors array with the dimensions passed by parameters, and it also initializes with values, depending on the Type parameters. I created this type thanks to enum with three values : Identity, Zero and Random.

In the first case, the matrix is an identity matrix ($a_{i,i}$ = 1 and 0 anywhere else), so of course it works only for square matrices. If m != n, we just put 0 value anywhere.

In the second case, we just put 0 value in all the matrix.

Finally, in the last case, we put random values in all the matrix.

To conclude, this constructor will be called systematically by all the other ones, to initialize the matrix with 0 values in all the cases at the beginning.

- **Matrix(int m, int n, list<float> l);**

This constructor allows us to create a matrix in wich the values are contained in a list passed by parameter. I chose to use only 1D lists and not 2D for simplicity, but it works also fine. For example, for a 4x5 matrix, the list will have a length equal to 20.

If the size of the list is different of the m*n product, we just initialize the matrix with 0 values.

- **Matrix(int m, int n, string s);**

Finally, this constructor is the last one and allows us to create a matrix in wich the values are contained in a file whose name is passed by parameter. In the file, the values have to be separated by a space or tab characters. If the number of lines and/or columns in the file is different of the m and n parameters, 0 values will be inserted for padding the missing values.

If the file’s name passed by parameter doesn’t exist, the matrix will be initialized with 0 values.

Functions

- **Matrix plus(Matrix m);**

This function computes the sum of two matrices. The first one is the object itself, and the second one is passed by parameter. To do it, we start by making sure that the two matrices have same dimensions, and we create a new matrix with the same dimensions. After that, we go through the two matrices and we put in the resulting matrix the sum of each element. Finally, we return the resulting matrix.

- **Matrix minus(Matrix m);**

This function computes the substraction of two matrices. The principle is exactly the same as for the sum.

- **Matrix times(Matrix m);**

This function computes the dot product of two matrices. The principle is approximately the same as for the sum and the substraction, but the computation is a little bit more complex. First of all, we have to make sure that the number of columns of the first matrix is the same than the number of lines of the second one, due to mathematical reasons. After that, we create the resulting matrix whose the number of lines is the same as for the first matrix, and the number of columns is the same as for the second matrix, one more time due to mathematical reasons. Once these things are done, we have to go through the two matrices and applying the matrices dot product formula, and putting the result of each element in the resulting matrix. Finally, we return this matrix.

- **Matrix operator+(const Matrix& m);**

It is the + operator overloading. It allows us to compute the sum of two matrices by simply using the + operator, without using directly the sum() function. To make it working, we only have to return the result provided by the sum() function.

- **Matrix operator-(const Matrix& m);**

It is the – operator overloading. It is exactly the same principle as for the + operator.

- **Matrix operator*(const Matrix& m);**

It is the * operator overloading. It is exactly the same principle as for the + and – operators.

- **string toString() const;**

This function returns the matrix in a characters string format, with the string type that doesn’t exist in C and wich is a very smart way to manipulate strings in C++. With this type we can use strings, modify, truncate them without paying attention to memory allocation. It is going to be very useful for using it in the next functions.

- **void print();**

This method just prints the matrix on the screen, by using the previous toString() function.

- **friend ostream& operator<<(ostream& os, Matrix& m);**

It is the << operator overloading. It allows to print a matrix on the screen by simply using the << operator, as for printing a string or a number, without using the previous print() method. To do it, we use the toString() function once again.

- **void exportToFile(string s);**

This method prints a matrix in a file whose name is passed by parameter. Also here we use the toString() function to get the string and we write it in the file.

- **Matrix getColumn(int n);**

This function returns a Matrix that contains the n-th column of the current matrix. It is called a column-matrix. Its number of lines is equal to the number of lines of the current matrix and the number of columns is one, since it is a column. It is better to return it in the Matrix type rather than the vector type for example, because it will allows us to manipulate it as a Matrix with all the other functions of the class.

- **Matrix getRow(int m);**

This function returns a Matrix that contains the m-th line of the current matrix. The principle is the same as for the getColumn() function but here the the number of lines is one and the number of columns is equal to the number of columns of the current matrix. It is a row-matrix.

- **void removeColumn(int n);**

This method removes the n-th column of the matrix. To do it, we need to remove the n-th element of each vector contained in the main vector. Thanks to the C++ vector type, we can do it in a very easy way by using the erase() method.

- **void removeRow(int m);**

This method removes the m-th line of the matrix. Since each line is represented by a vector, we only need to use the erase() method one time on the main vector to remove the m-th vector, contrary to the removeColumn() method.

- **float max();**

This function returns the biggest value of the matrix. To do it, we go through the whole matrix and if the current value is biggest than the previous one, we save it. Finally, we return the last value saved.

- **float min();**

This function returns the smallest value of the matrix. It is exactly the same principle as for the max() function but here we save the current value if she is smaller than the previous one.

- **bool invertible();**

This function returns True if the matrix is invertible or ‘0’ if not. The invertibility criterion is the value of the determinant. If it is equal to 0, the matrix isn’t invertible, otherwise it is. So do it, we simply use the det() function that returns the determinant of the matrix.

- **bool orthogonal();**

This function returns ‘1’ if the matrix is orthogonal, ‘0’ otherwise. An orthogonal matrix means that the matrix inverse is equal to the matrix transpose. In mathematical terms we have :

$$A^{-1} = A^{T}$$

And we know that

$$A^{-1} = \frac{com(A)^T}{det(A)}$$

so to be orthogonal, the matrix must verify the following condition :

$$\frac{com(A)^T}{det(A)} = A^T$$

To let this equality being true, we deduce the following conditions :

$$com(A) = A$$

and

$$det(A) = 1$$

At this point, we have already two ways to verify the orthogonality of the matrix :

1. Computing the matrix inverse and the matrix transpose and verify that these two ones are equal
2. Computing the comatrix and the determinant and verify that the comatrix is equal to the matrix and verify that the determinant is equal to 1.

The problem of these two methods is that they force us to compute determinant, comatrix, inverse… so the computations can be quite complex, in particular for big matrices. So the smartest way to verify the orthogonality of a matrix is the following. We know that :

$$A\,A^{-1} = I$$

And if the matrix is orthogonal :

$$A^{-1} = A^T$$

So if the matrix is orthogonal :

$$A\,A^T = I$$

In this way, we only have to compute the matrix transpose and multiplying the matrix by its transpose. Finally, we compare this result with the identity matrix. If they are equal, the matrix is orthogonal. This method is clearly the best way to do it because computing a matrix transpose and comparing two matrices are very easy. In our case, to implementing this algorithm, we just have to use the transpose() function and the * and == operators to verify the orthogonality of a matrix.

- **bool diagDominant();**

This function returns ‘1’ if the matrix is diagonally dominant, ‘0’ otherwise. A matrix is diagonally dominant if on each line, the absolute value of the diagonal element ($a_{i,i}$) is greater or equal to the

sum of the absolute values of all the others elements of the line. Of course we must not include the diagonal element in the sum because in this case, the ≥ condition will never be verified.

If this condition is verified in all the lines, the matrix is diagonally dominant. To do it, we just have to go through all the lines of the matrix and each time computing the sum as explained previously, and compare the result with the diagonal element.

- **bool upperTri();**

This function returns ‘1’ if the matrix is upper-triangular, ‘0’ otherwise. A upper-triangular matrix means that all the values under the diagonal are 0. To verify it, we go through all the lines and for each of them, we go through the line from the first element until the diagonal element. If an element has a value different of 0, we return False. If all the values are 0, we return True.

- **bool lowerTri();**

This function returns ‘1’ if the matrix is lower-triangular, ‘0’ otherwise. It is the same principle as for the upperTri() function but in this case, a lower-triangular matrix means that all the values above the diagonal are 0. To verify it, we go through all the lines and for each of them, instead of starting at the first element, we go through the line starting at the element just after the diagonal one, until the end of the line. After that, we apply the same algorithm than previously.

- **Matrix toColumn();**

This function returns a column-matrix that contains all the elements of the matrix, but in a column format. So the number of lines of the vector is the product of the dimensions of the matrix M*N. After creating the result matrix with dimensions (M*N, 1), we go through the whole matrix line by line and we put each element one under the other in the vector. It is an equivalent to transpose each line and put them one under the other. Finally, we return the column-matrix.

- **Matrix toRow();**

This function returns a row-matrix that contains all the elements of the matrix, but in a row-format. The principle is the same as for the toColumn() function but here, the dimensions of the resultat matrix are (1, M*N) since we have one line, and the number of columns is the number of elements of the matrix. In this case, it is an equivalent to put each line next to the previous one, without transposing. Finally, we return the row-matrix.

- **Matrix transpose();**

This function returns the matrix transpose. It means that we invert each element symetrically to the diagonal. For example, the element (3, 1) becomes the element (1, 3). Due to that, the dimensions of the result matrix are the inverse of the original one. For example, if we transpose a 4x7 matrix, we obtain a 7x4 matrix. To do it, we create a matrix with dimensions (N, M) and we go through the two matrices. The elements at position i, j in the first one are put at position j, i in the second one.

- **float det();**

This function computes the determinant of the matrix. To do it, we apply the well-known algorithm of going through a random line or column of the matrix and each time, we add the minor of this position, multiplying by the current element and by -1 at power i+j. Of course, for each element if its value is not 0 because in this case, the result will be 0 so it useless to compute the minor. We can go directly to the next one.

The determinant function is recursive because when the matrix dimension is at least 3, we need to auto-call the function in itself to compute all the minors. So when the dimension is big, the function is going to be called a lot and the computation is more and more long and complex.

- **float Minor(int i, int j);**

This function returns the minor of the matrix at position i, j. The minor is the determinant of the submatrix extracted from the position i, j. So in this function we only have to use the subMat() and det() functions.

- **Matrix operator*(float n);**

It is the * operator overloading. Contrary to the previous * operator overloading, this one is used to multiply the matrix by a constant value. The only difference with the previous one is that here, the parameter is a float. So we just have to go through the whole matrix and multiplying each element by this value.

- **Matrix operator/(float n);**

It is the / operator overloading. It is exactly the same principle as for the previous and here, instead of going through the whole matrix for dividing each value, we directly use the previous * operator overloading by multiplying by 1/n.

- **Matrix inv();**

This function returns the matrix inverse. The inverse is given by the following formula, as we have seen previously :

$$A^{-1} = \frac{com(A)^T}{det(A)}$$

Since we already have all the necessary functions, the code is very simple. In a first time, we verify that the matrix is invertible with invertible() function and then, when return the comatrix transpose, divided by the determinant.

- **Matrix invQR();**

This function returns the matrix inverse but here, we use the QR decomposition method. The goal of the QR decomposition is to express a matrix A by the product of an orthogonal matrix (Q) with a upper-triangular matrix (R) :

$$A = QR$$

The properties of these two matrices Q and R are very interesting because an orthogonal matrix is very easy to invert since its inverse is its transpose, and a upper-triangular matrix is easier to invert than a classic square matrix, we are going to explain why.

When we have :

$$A = BC$$

where A, B and C are three arbitrary matrices, we know that :

$$A^{-1} = C^{-1}B^{-1}$$

So in our case, we have :

$$A^{-1} = R^{-1}Q^{-1}$$

and thanks to the orthogonality of Q, we obtain :

$$A^{-1} = R^{-1}Q^{T}$$

So now, the only thing that we have to do is to find a way to invert a upper-triangular matrix (R) without using the classical way to invert a matrix used by the inv() function, precisely because the goal of using the QR decomposition method is to not compute comatrix, determinant, etc...

To do that, I started by the following :

$$RR^{-1} = I$$

Let's call the coefficients of R $a_{i,i}$ and the coefficients of R^{-1} $b_{i,i}$, and we know that the inverse of a upper-triangular matrix is also upper-triangular, so that we obtain :

$$\begin{pmatrix} a_{1,1} & a_{1,2} & \cdots & a_{1,M} \\ 0 & a_{2,2} & \ddots & \vdots \\ \vdots & 0 & \ddots & a_{M-1,M} \\ 0 & \cdots & 0 & a_{M,M} \end{pmatrix} * \begin{pmatrix} b_{1,1} & b_{1,2} & \cdots & b_{1,M} \\ 0 & b_{2,2} & \ddots & \vdots \\ \vdots & 0 & \ddots & b_{M-1,M} \\ 0 & \cdots & 0 & b_{M,M} \end{pmatrix} = \begin{pmatrix} 1 & 0 & \cdots & 0 \\ 0 & 1 & 0 & \vdots \\ \vdots & 0 & \ddots & 0 \\ 0 & \cdots & 0 & 1 \end{pmatrix}$$

Let's start with the diagonal. For the n-th diagonally element, we have to multiply the n-th line of the R matrix by the n-th column of the R^{-1} matrix. So finally in a generally way, we have :

$$a_{i,i}b_{i,i} = 1$$

So :

$$b_{i,i} = \frac{1}{a_{i,i}}$$

And then, we apply the same method for all the diagonally lines above the main diagonal. Finally, in a recursive way, we obtain the following formula :

$$b_{i,i+n} = -\frac{\frac{a_{i,i+n}}{a_{i+n,i+n}} + \sum_{k=1}^{n-1} a_{i,i+k} b_{i+k,i+n}}{a_{i,i}}, n \in [1, M-1]$$

Once we have this formula, we can compute each diagonally line, one after the other, in the numerical order (n=1, n=2, …) because the coefficients of each diagonally line are used in the computation of the next one.

When we have computed R^{-1} , the last thing to do is to multiply this one by Q^T in order to obtain A^{-1} .

With this method of inverting, the criterion of invertibility is that there is no coefficient with 0 value in the diagonal of R. Effectively, the determinant of a triangular matrix is simply the product of each coefficient of the diagonal.

During the tests, to see if the A^{-1} is correct, we multiply A by A^{-1} and normally, we must obtain I . In fact, we don't obtain exactly I , and I think that it is due to the computation of R^{-1} , more exactly the precision and the recursive algorithm.

- **float norm();**

This function returns the norm of a column vector thanks to this formula :

$$\|v\| = \sqrt{\sum_{i=1}^{M} v_i^2}$$

- **Matrix subMat(int l, int c);**

This function returns the submatrix extracted from position (l, c). It means that we remove the l line and the c column in the current matrix, and we return the resulting martrix whose dimensions are (M-1, N-1). The submatrices are very used is the determinant computations for example.

- **Matrix com();**

This function returns the comatrix of the matrix. It is also called the matrix of cofactors because for each element (i, j), we replace it by the minor of the same position (we can also say the determinant of the submatrix extracted from position i, j) multiplied by -1 at power i+j. So of course, the comatrix is only defined for the square matrices.

- **bool operator==(Matrix m);**

It is the == operator overloading. We use it to test the equality of two matrices. The first one is the current matrix (this), and second one is passed by parameter. To do it, we just have to go through the whole matrices and verifying the equality of each element. If the two matrices are equal we return '1', '0' otherwise.

- **float& operator()(int i, int j);**

This function returns the reference to the element (i, j) of the matrix. It is really important to return the reference (thanks to the &) and not just the value, because it allows us to not only read the value, but also to modify it. Thanks to that, we can read and write all the matrix values without accessing directly the mat member.

- **int getM() const;**

This function returns the numbers of lines of the matrix M

- **int getN() const;**

This function returns the number of columns of the matrix N

○ Test program

To test the Matrix class, we finally write a test program in the main.cpp file. In this program, we test at least one time each function, method and operator previously defined to make sure that they work correctly. To start, we create random matrices. After that, the program is quite easy to understand.

Just for orthogonality part, I have created a special 2x2 matrix wich will always be orthogonal, to test the orthogonal() function. This one is the following :

We want a matrix where

$$com(A) = A$$

So a very simple and good example is the following :

$$A = \begin{pmatrix} a & b \\ -b & a \end{pmatrix}$$

Now we compute the comatrix and we obtain :

$$com(A) = \begin{pmatrix} a & b \\ -b & a \end{pmatrix} = A$$

Finally, the last condition to be orthogonal is that the determinant is equal to 1 :

$$a^2 - (-b)\,b = a^2 + b^2 = 1$$

$$b^2 = 1 - a^2$$

$$b = \pm\sqrt{1 - a^2}$$

From that we obtain :

$$A = \begin{pmatrix} a & \sqrt{1-a^2} \\ -\sqrt{1-a^2} & a \end{pmatrix}$$

And to let the square root be always defined, the final condition is :

$$|a| \leqslant 1$$

So we know that this matrix will always be orthogonal. To implement it, I used a list because it allows me to test at the same time the constructor with a list parameter.