First assignment

Baldan Nikita, Emanuele Motto, Mauro Noris

November 27, 2018

Contents

1 Introduction

The project consists in designing and implementing a libray in c++ that handles matrices, vectors and some of their operations.

The vectors are defined as matrices with only a row or a column (or both).
It was requested to implement some specific operations such as:

- submatrix: return a matrix that contains a contiguous subrange of rows and columns of the original matrix
- transpose: return a matrix with rows and columns inverted
- diagonal: return a vector containing the diagonal elements of a given matrix
- diagonalmatrix: return an unmodifiable diagonal matrix whose diagonal elements are those of a given vector

The matrices returned by such operators must share the data with the original matrix, but direct copy of the matrices and vectors must result in a deep copy (sharing a matrix can be obtained by taking a submatrix extending the whole row and column ranges).

Was requested also a set of iterators for the matrices traversing it in both column and row order. Every matrix need to be traversable in both directions regardeless of its natural representation order. Any given iterator will traverse only in a given order.

2 Analysis and implementation

We choose to implement our library with an imperative design, organizing the project three files, the matrix.h file (the template libray), the iterators.h file (the file including all the iterators) and the main.cpp (the test file where we try the functionality of the library).
We also included a Makefile to allow an easier complilation process and a documentation generated with Doxygen.
Given that we decided to define iterators and matrix library in two different files, we used forward declaration (alias matrix_forward.h) the permit the use of matrix type in the constructor of the iterators without having to include the library(which would result in an error, because then matrix.h file would include iterator.h that include matrix.h, creating a cicle).

Firstly we implemented the must-have methods of container classes, such as default,copy and constructor, the destructor(which is only a facade because the vector one will be called by shared_ptr when its last matrix object will go out of scope) and the move/default assignment(the move one will not be called thanks to RVO). On the default constructor, we initialize the elements with the deafult constructor of type T, which constructs an empty element of that particular type.
When we create a matrix we have four different variables to indicate its dimensions, the rows (number of rows), columns (nomber of columns), effective rows and effective columns. We added effective rows and effective columns to indicate the effective dimension of the matrix after an operation that modify its dimension (ex. submatrix) and keep rows and columns untouched because they indicate how many elements there are in the effective vector linked to the matrix to be able to

find correctly its elements.

We represented the matrices as vectors putting every row of the matrix one after the other in row-order. since the matrix created with the various operations needed to share memory with the original matrix it points to its vector with a shared pointer.

This way we can easily have different matrices that shares data in the project. In our project the various operations (transpose, submatrix, diagonal and diagonalMatrix) return a matrix object with the appropriate charatertistics, in particular we use various flags to identify if a matrix is transposed, diagonal a submatrix or , in the special case of diagonal matrix from which matrix it is derivated.

We need these flags because if the physical memory is shared we need such information to be able to display the correct information with the various operators (for example the operator() checks which matrix it is working on before retriving the elements).

The practical effect of these methods is the creation of a different view of a given matrix, because when working with large matrixes it's pretty useful being able to simplify and regroup certain parts of the data, possibly without a great time/resource consumption. Thanks to shared_ptr and RVO we are able to perform these tasks pretty easily(that's why our operations use an optional constructor, to use RVO) and without any complicated memory management. The constuctor we use to implement these operation is obviously private, because it's not meant for the user and one of its parameter is a pointer, so it is not safe put this kind of constructor in the public interface.

2.1 Operations

2.1.1 transpose

In our transpose method we just return call to the contructor with the correct rows, column, effective rows and effective columns. we also change the flag ”transp” to indicate how the operators need to retrive the informations form the ”physical” memory.

2.1.2 submatrix

in the submatrix method we calculate the correct number of effective rows/columns and then we effetuate a call to the constructor that returns the matrix. the various operator (and the iterators) will use the informations about effective rows/columns to iterate the correct matrix even though in the memory we still have every starting elements.

2.1.3 diagonal

The diagonal operation effetuate a call to the constructor after calculating the correct parameters for the new matrix, changing the correct flag.

2.1.4 diagonalMatrix

diagonal matrix is different from the other operation, it has its own constructor since it needs to check from which matrix it is derivated to display correctly its informations. With this one we had to use two additional flags in respect to the other operations, because the vector which calls this method(it has ton be a vector in order to call this method) can be saved differently in memory. Basically we have 3 cases:

- Standard covector,submatrix covector and standard vector
- Submatrix vector
- Diagonal cases

In the first case, the elements of the diagonal are contiguosly saved in the memory(so they are all next to each other) so taking the correct elements from the original matrix is simple.
In the second case, however, the elements are saved at a certain distance between one another, so when using the operator() a flag that tells you that that diagonal matrix was built from a submatrix vector is needed(I can't really tell this from the diagonal matrix itself unfortunately).
Same for the last case, in which the elements are saved in a different way, and the operator() needs to know this otherwise it will pick the wrong elements.
A note on the deep copy of a matrix, we allow deep copy through copy constructor or assignment only on full fledged matrix (so lhs lvalue reference) and not on temporary object.
For example matrix<int> A(B.transpose()) will not deep copy the matrix result of tranpose operation, but it will move that object into A.
If a deep copy of an object has to be done, it must be from a matrix variable.
Ex matrix<int> A(B); or matrix<int> A; A = B;.
As for the iterators, we implemented 2 kind of iterators(const and non-const), one that visit a subset of the matrix in row-order, the other in column order instead; for the plain matrix, given that we use a shared_ptr to a vector, we use the standard vector iterators.
The iterators extract the element of the matrix using the operator(), so the type of view of the matrix we want to iterate on does not concern the iterator itself.
In the main.cpp we included some test performed on our library, divided in methods to keep the code as readable as possible.
As for the const matrixes, we decided that every view of the matrix should be constant as well, so the elements of it might not be modified. The only way of modyfing those element is to use a previous view variable which was not constant or perform a deep copy.

3 Problems about diagonalMatrix and const methods

Now we explain the main limitation of our imperative design solution, we decided to give it a separated section to be as much clear as possible.
The const methods and diagonal matrix work just fine when dealing with temporary object operations like:
A.diagonalMatrix()(0,0) = 1 !!ERROR!! read-only location
A.diagonalMatrix().transpose()(0,0) = 1 !!ERROR!! read-only location

When we use RVO to optimize the time/space consumption of our operations, then there is a BIG problem assigning the result of that operation to another variable.
What if I want to return a const object through RVO?
We thought that the compiler would understant that making the return object const would imply that if the other object is non-const that is:
matrix<T> A = B.diagonalMatrix(); would return an error of some sort, because practically you

can’t move a const object. Problem is that the rhs of this is a temporary value, and is ”forced” into a non-const object, so if it really shares a memory I can modify its elements so
matrix<T> A = B.diagonalMatrix();
A(1,1) = 1;
WOULD WORK AND WILL OVERWRITE EVERY MATRIX THAT SHARES THAT PARTICULAR POSITION OF THE MATRIX
In this case there are several options:

- Force everyone(maybe with assert) using the library to use const matrix<int> declaration instead of auto(which will guess the type ad matrix<int>) when working with diagonal matrixes and every view that comes after that, and this will make everything work.

- Using a additional class that is a kind of matrix, with the const member representing the vector, but of course it will be used as a return of diagonal matrix method and maybe as const operation methods.

4 Conclusions

The route we took to implement this matrix library was a fully imperative one, which was easy to design, the implementation was not so simple as we initially thought but it was manageable, and of course the memory management and the various operation had to be implemented in the best way possible.
There are some cons to our solution thought, first of all it’s not so easily improved and the more operation you add to this kind of library, the more your code will exponentialy growth and become pretty unreadable very soon.
For example the operator(), the main point which basically does all the logic ”division” of the views of the matrix, in case you might add some other operation, will get pretty complicated. Moreover, as we stated before, we use flags(boolean variables) to determine the various condition and differentiate the various views we create with the operations, but that will create an overhead of the information(Plain matrixes of course have all that flags set to zero, and that is a waste); in that way an OO design is the better choice.
On the other hand, the memory manegement and the object ownership in our solution is very simple, every view is still a matrix object after all, and the implementation of the various operation was not really difficult to think.