Advanced Algorithms and Programming Methods

Report

Assignment 1: Matrix Library

12^{th} *November 2017*

Group Members:
Fabio Rosada 851772
Feliks Hibraj 854342

Index

1 Introduction

1.1 Assignment

Implement a templated library handling matrices, vectors, and covectors; where vectors are nx1 matrices and covectors are 1xn matrices.
the library must offer operation like:

- ◇ 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).
Create iterators for the matrix traversing it either in row-major or column-major order. Any matrix should be traversable in either direction regardless of its natural representation order. Any given iterator will traverse only in a given order.

2 API (Methods)

These libraries help the user to create and manage matrices that share data. With the methods provided, users are able to create a tree of matrices with a hierarchy, for example they can create a matrix, then a sub matrix of the matrix and finally a transpose matrix of the sub matrix creating the following structure:

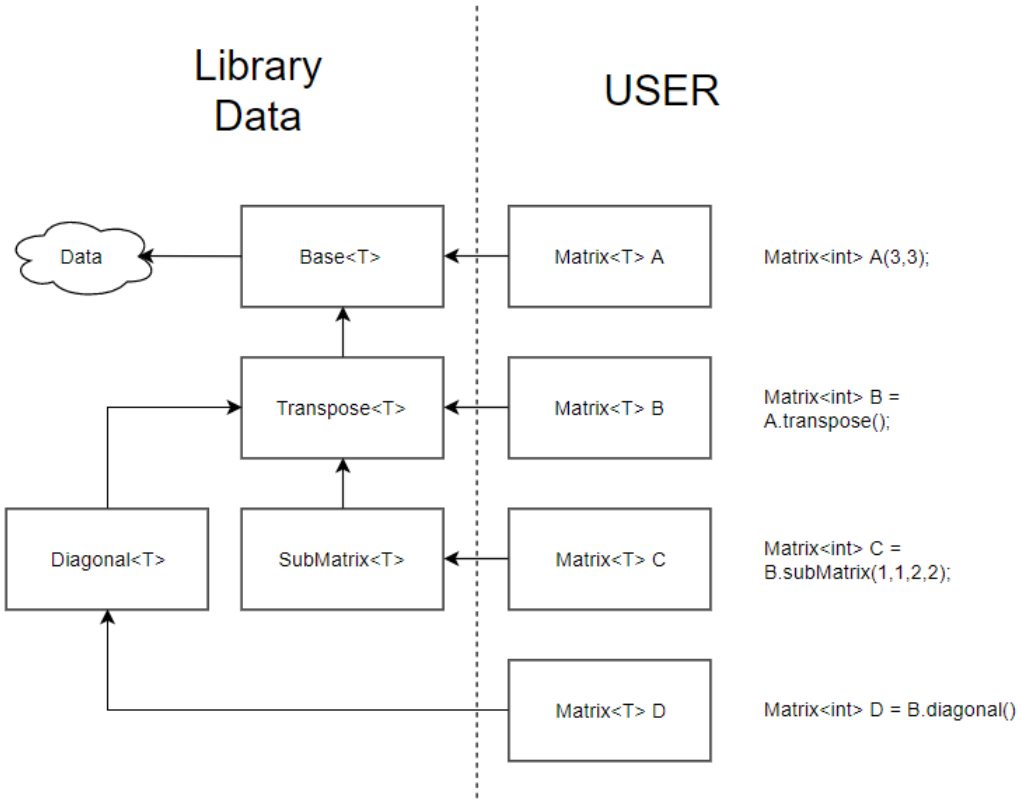


Figura 1: Structure of the hierarchy.

2.1 Matrix.h

This is a templated class that provides all the methods needed for end users.

2.1.1 Constructor

Public constructor: takes row and col in order to create a new matrix from scratch. The matrix will be initialized with default values for the specific type of data.

```
Matrix(unsigned int rows, unsigned int cols);
```

Private constructor: takes a pointer to Imple<T> and creates a new matrix. This constructor is used only by the library code to build and return a new *"Son"* matrix.

```
Matrix(Imple<T>* pimpl);
```

2.1.2 Destructor

The destructor have to manage the tree hierarchy of matrices. Each implementation node can have a lot of sons, so each implementation node have to keep a counter of the number of sons that are pointing to it, and finally, if the matrix is still pointing it.
When we destroy a matrix, it have to notify its implementation node that it is not pointing it anymore and next it asks its implementation node to check if someone is still pointing it, if not, the node will be destroyed (as we will see in section 2.2)

2.1.3 Manupulation Methods

There are 4 matrix manipulation methods that coincide with the 4 equivalent implementations (2.2):

- ◇ **transpose()**: returns a Matrix representing a transpose of the current matrix.
- ◇ **subMatrix(int start_row, int start_col, int end_row, int end_col)**: returns a Matrix representing a submatrix of the current one with
  rows = *end_row - start_row + 1* and
  columns = *end_col - start_col + 1*
- ◇ **diagonal()**: return a Matrix $N \times 1$ (vector) representing diagonal elements of the current matrix.
- ◇ **diagonalMatrix()**: this function works only for a vector. Returns a matrix with all null value except for the diagonal elements, which are the elements of the starting vector. Works both with vectors and covectors.

and some data manipulation methods:

- **void set_value(int row, int col, T value)**: sets the value of the element in position *(row, col)* to the provided one.
- **T operator()(int row, int col)**: returns element in position *(row,col)*.
- **T* get_ptr(int row, int col)**: returns the pointer to the element in position *(row, col)*.
- **int getRows()**: returns number of rows of the matrix.
- **int getCols()**: returns number of columns of the matrix.
- **void print_matrix()**: prints the matrix using iterators.
- **IterRow get_row_iterator()**: return an iterator that scans the matrix in row wise order.
- **IterCol get_col_iterator()**: return an iterator that scans the matrix in column wise order.
- **IterCol get_reverse_row_iterator()**: return an iterator that scans the matrix in reverse row wise order.
- **IterCol get_reverse_col_iterator()**: return an iterator that scans the matrix in reverse column wise order.

2.1.4 Iterators

There are four types of iterators:

- IterRow
- IterCol
- IterReverseRow
- IterReverseCol

These iterators define a way to scan a matrix. In order to reach the goal the library defines a set of iter classes that iterate through the matrix, keeping a counter to the position and defining operators, in this way the user can use an iterator in a for-cycle simply using (same for other iterators):

```
Matrix<T>::IterRow iter = myMat.get_row_iterator();
for(iter; iter != iter.end(); ++iter){
        T value = *iter;         //dereferencing value
        do_something_with_value(value)
}
```

The redefined operators are:

- ◇ **operator ***: in in this case, when you try to dereference an iterator, it returns a pointer to the current element.
- ◇ **operator ==**: this helps you to check if two iterators are iterators of the same matrix and if they are at the same position.
- ◇ **operator !=**: same as the previous one but checks if they are different.
- ◇ **operator ++**: by calling ++iter, the iterator will return an iterator with the next element of the matrix.

2.2 Implementation.h

Every implementation class keeps a father reference, its rows and its cols. The behavior of these classes is that when you call a method, they know how to manupulate given indices to get the right element.

2.2.1 Imple<T>

This class represents an interface for the common methods of all the following classes and define type T of the matrix.

2.2.2 Base<T>

Extends Imple<T> and is the only class that points directly to the data.

Constructor Base<T>(unsigned int rows, unsigned int cols) : initialize a matrix of $Rows \times Cols$ dimensions with type T.

Destructor destroy the data.

2.2.3 Decorator<T>

Extends Imple<T> and defines a common interface for the final implementation classes.

Constructor Decorator<T>(Imple<T> *i) : save the Imple<T> pointer as its father, and set its rows and its columns at the same size of the dimension of the father.

Destructor destroy the object. In addition to the standard destructor, there is a method that checks the counter *pointedBy* (the number of elements that points to the Decorator) in order to decide if it can be destroyed safely.

Transpose: extends Decorator<T>. For setter and getter methods this class inverts the row and col indices, in this way we get the right transposed element. Also for the constructor it inverts row and col dimensions.

SubMatrix: extends Decorator<T>, the constructor needs starting and ending coordinates, both these coordinates are inclusive (i.e: a submatrix between (1,1) and (2,2) contains two rows and two columns) in the new sub matrix.
The dimensions of this submatrix are: rows = end_row - start_row + 1 (because both are included), columns = end_cols - start_cols + 1.
For getter and setter it adds starting coordinates values to the given index, in order to shift the index to the right position, and finally checks if the shifted value is still lower than ending coordinates.

Diagonal: extends Decorator<T> and creates a matrix $N \times 1$ (vector). In the constructor it sets columns to 1 (because we want a vector) and rows (N) to the minimum value between fathers rows and columns.
For getter and setter functions, it simply checks if col value is equal to 1, and then asks its father the element in (row, row) position, that is actually the diagonal of the matrix.

DiagonalMatrix: extends Decorator <T>, given a vector ($N \times 1$) we can obtain a matrix $N \times N$ with each element to 0 (or null, it depends on the type T of the matrix) except for the diagonal that contains the vector elements.
The getter and setter functions return value at father's (row, 0) when you ask for (row, col) value with row = col (that means you asked for elements in the diagonal), else they return null value.

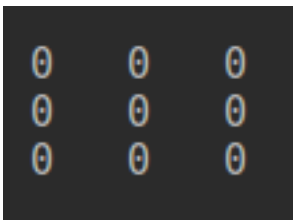
3 Usage Examples

3.1 Print_matrix

Let's start from the very basic instruction we can use to print a Matrix after creating it.

```
Matrix<int> matrix1(3, 3);
matrix1.print_matrix();
```

We can choose the type of datas we want the matrix to have, in this case *int*, we also specify number of rows and columns of the matrix and a NULL-value filled matrix will be returned.

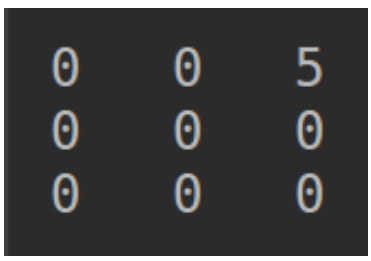


The meaning of **NULL**-value is the basic value associated to any type of datas. In this case the null-value of int type is 0. For strings it's the empty string "" and so on.

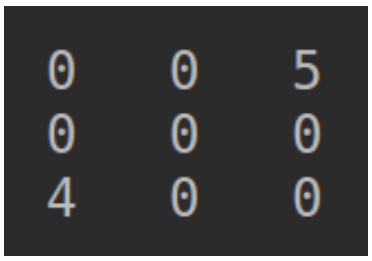
3.2 Set_value

The method *set_value* is used to change a value of the matrix. An example is the following:

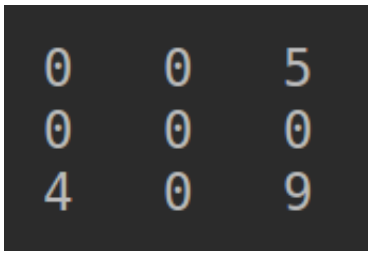
```
matrix1.set_value(0, 2, 5)
matrix1.set_value(2, 0, 4)
matrix1.set_value(2, 2, 9)
```



(a) set_value(0, 2, 5)



(b) set_value(2, 0, 4)



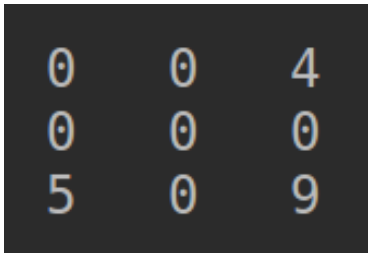
(c) set_value(2, 2, 9)

3.3 Operations

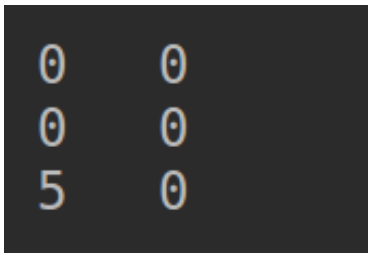
The fundamental operations are the core of this library.

3.3.1 Tranpose & SubMatrix

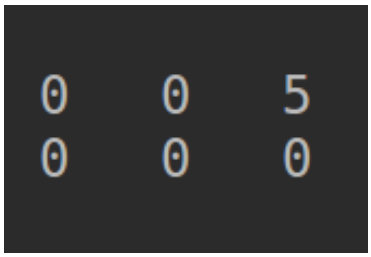
```
matrix1 = matrix1.transpose();
matrix1 = matrix1.subMatrix(0, 0, 2, 1);
matrix1 = matrix1.transpose();
```



(a) transpose()



(b) subMatrix(0, 0, 2, 1)



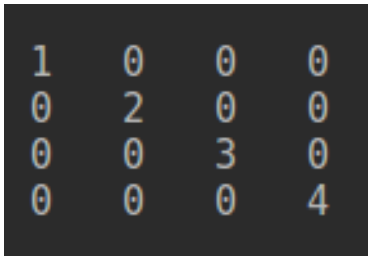
(c) transpose()

As you can see in *(a)* the matrix elements have been transposed, in fact elements 4 and 5 are switched.
Than we created a *SubMatrix* that strarts from the origin(row=0, col=0) and ends in (row=2, col=1), which basically excludes the last column as you can see in *(b)*.
The resulting matrix is a 3×2 matrix, which transposed returns a 2×3 matrix, in *(c)*.

3.3.2 Diagonal & DiagonalMatrix

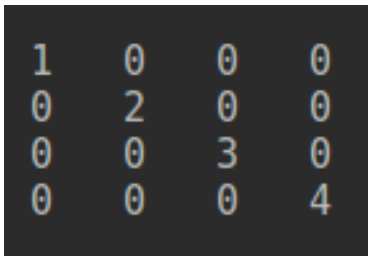
```
Matrix<int> matrix2(4, 4);

matrix2.set_value(0, 0, 1);
matrix2.set_value(1, 1, 2);
matrix2.set_value(2, 2, 3);
matrix2.set_value(3, 3, 4);

matrix2 = matrix2.diagonal();
matrix2 = matrix2.diagonalMatrix();
```

Consider a new matrix 4×4 with diagonal values set in incremental way *(a)*.
When we call transpose method on it, we get a vector 1×4 which is built from the diagonal elements of the new matrix *(b)*.
If we want to get a covector we can simply tranpose that vector getting a



(a) starting matrix

(b) diagonal()



(c) diagonalMatrix()

4×1 covector (not shown in figure).
On last subfigure *(c)* we can see the resulting matrix from the operation *DiagonalMatrix* on previous vector.

3.4 Get element

In order to facilitate the usage of matrices we provide a fast method to access elements by parenthesis as shown below:

```
matrix2 = matrix2.diagonal();
cout << matrix2(2, 0);
cout << matrix2(2, 2);
cout << matrix2(1, 0);

3
#Exception: [Diagonal] Wrong index#
2
```

Starting from the following vector (matrix 1×4): by calling *matrix2(2, 0)*

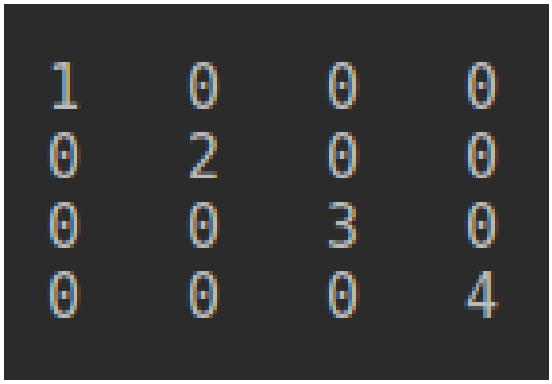
we retrieve the element in position *(row=2, col=0)* which is number 3.
If we request an element out of bounds *(row=2, col=2)* we get an exception which is handled in such a way to not interrupt the execution of code, in fact the next call is executed anyway (*matrix2(1, 0) -> 2*).

3.5 Iterators

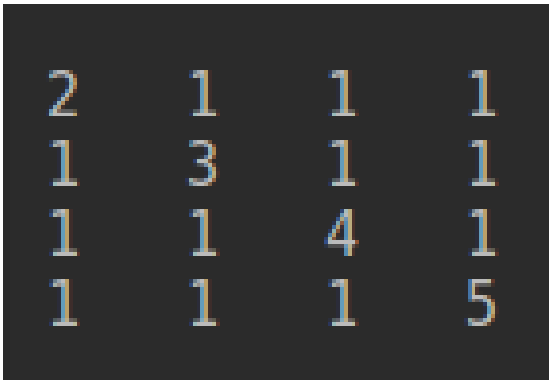
There are 4 different iterators we can use on Matrices and they are really helpful, for example by executing the following piece of code:

```
Matrix<int>::IterRow iter = matrix.get_row_iterator();
    for(iter; iter != iter.end(); ++iter){
        *iter = *iter + 1;
    }
```

we can easily add 1 to all elements of our matrix producing the following effect (from *(a)* to *(b)*):



(a) starting matrix



(b) resulting matrix

4 Further Details

4.1 Copy Constructor & Copy Assignment

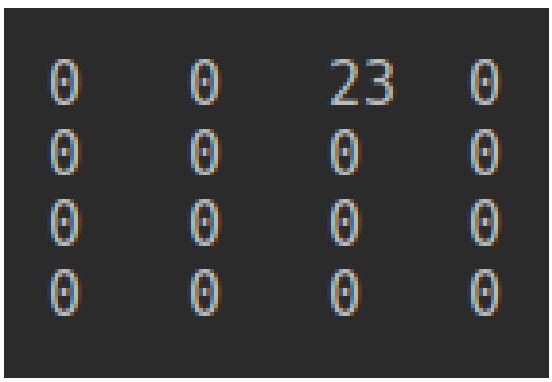
These fundamental operators have been overriden to fulfill our desired behaviour.
The behaviour we want is that of avoiding deep-copy of data as much as possible. In fact the structure that lies behind our implementation creates a hierarchy composed of all the transformations the user requires (you can see it in figure 1). That means that calls like the following will not cause a deep-copy of data:

```
Matrix<int> matrix1(4, 4);
Matrix<int> matrix2 = matrix1.transpose()
                            .subMatrix(0, 0, 2, 2)
                            .transpose()
                            .diagonal()
                            .diagonalMatrix();
```

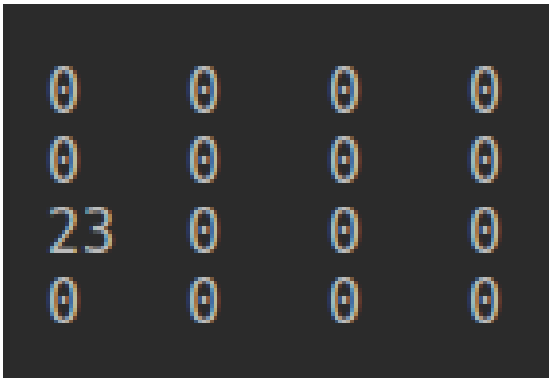
because we assume that the user wants to create a sequence of transformations and wants the data to be shared among all these transformations. In fact if we apply changes to one of these elements in the hierarchy, changes will be shared. For example:

```
Matrix<int> matrix1(4, 4);
Matrix<int> matrix2 = matrix.transpose();

matrix1.set_value(0, 2, 23);

matrix1.print_matrix();
matrix2.print_matrix();
```

Is going to produce the following result.



(a) matrix1



(b) matrix2

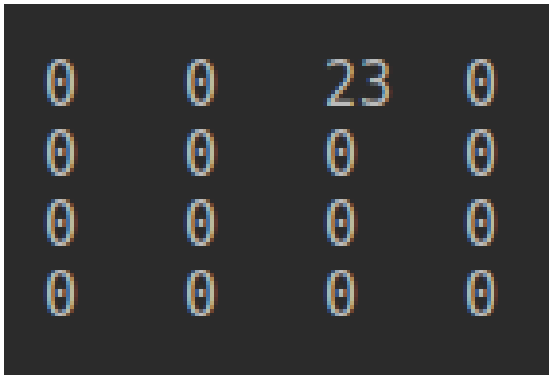
Even if we didn't modify matrix2, its values have changed because their datas

are shared, so a modification on matrix1 is reflected on him.
In this case, the call is managed by the Copy-Assignment, which is handles using the **Copy-and-Swap** idiom for safer code.

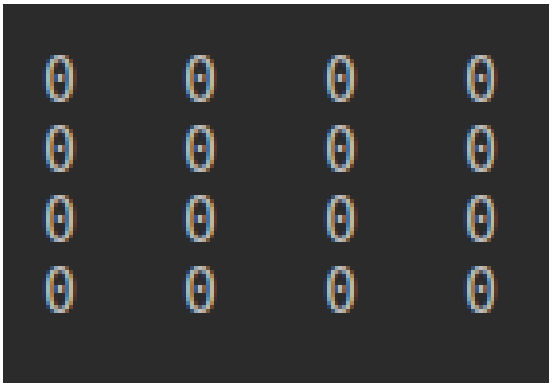
On the other hand, if the user wants 2 independent matrices then he is going to explicit it, for example in the following code a deep-copy occurs (managed by the Copy Constructor):

```
Matrix<int> matrix1(4, 4);
Matrix<int> matrix2 = matrix1;

matrix1.set_value(0, 2, 23);

matrix1.print_matrix();
matrix2.print_matrix();
```

In fact, by modifying the value in matrix1, matrix2 is not affected and you can see the result in the following figure:



(a) matrix1



(b) matrix2

4.2 Destructors & Shared Pointers

We have used the C++ provided shared pointers but we didn't like their behaviour and we didn't trust something not coded from us so we decided to avoid them and to manage by ourselves the execution flow, so we emulated that part of shared_ptr that interested us.
We know very well that shared_ptr never fails, but we took the risk anyway.

So we created a behaviour for destructors such that when a Matrix is destroyed, it causes the destruction of all the implementations that he points to. Unless they are pointed from any other Matrix, in that case the destruction is avoided and it's going to occur only when the last Matrix which points

to that implementation is destroyed. So they have an internal counter which serves for that purpose.

5 Group Management

The first part of the project was designing.
For that part we have discussed for long and a lot of meetings have been organised.
After the design part we started coding together, on the same computer, to decide together the structure of the code.
The remaining part of the work has been done individually by setting up a *BitBucket Repository*. We tryied to work on different parts of the program, but many times we had to work on the same problem and eventualy think again about the design of the project to overpass the new problems we got to know.
Really important is the fact that we constantly comunicated with each other, whether by messages or by meeting in library, to update on every single change, achievement or commit. That fact was really important because for every update on the code we discussed about it with a critical point of view to facilitate the discovering of bugs. Many bugs were discovered!
Even though our idea was to split the work, in the end we both worked at some point on the same part of code.