Ca Foscari University of Venice

Advance Algorithm and Programming Methods (C++)

Project on ”Matrix Template Library”

Group Memebers:

Lamin Jatta
Mat#:- 882561

Merjem Stulanovic
Mat#:- 882311

January 24, 2021

Abstract

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

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.

Provide matrix addition and (row-by-column) multiplication. When dealing with multiple multiplications, the library must perform the sequence of multiplications in the most computationally efficient order, i.e., by multiplying matrices contracting the largest dimension first.

For example, let A, B, and C be 2x3, 3x5, and 5x2 matrices respectively, then the expression A*B*C should be computed as if it where A*(B*C), performing first the multiplication B*C since it is contracting the dimension 5, and then multiplying A by the result, contracting the smaller dimension 3.

Contents

1 Introduction

The aim of this project is the realization of Implementing a templated library handling matrices, vectors, and covectors; where vectors are nx1 matrices and covectors are 1xn matrices. modifying its different components and making operation from it.

This project, which is part of the Advance Algorithm Programming and Programming Method Masters course, serves to expose students to understanding the practice behind implementing a Library for a system that can me used in solving problem at an optimal time. By modifying existing Source code components, We (students) learn concepts by seeing how such libraries are design and built and mostly importantly have fun to understand the programming language and learn new ideas.

For this project, students realized the implementation of a standard template library, MATRIX, for dynamically changing the vectors to acquire the sub-matrix from a given matrix and performing transposing of matrix i.e rows and columns inverted. Such a dynamic of Matrix allow us to perform some basic operation for sum and multiplication of two or more matrix. This project can thus be seen as an extension of the Matrix function add support for operation of high dimension Matrix.

This project was realized using the usage of library module in C++ for making it efficient. The code writing was mostly done by other through google search using the terminal editor.

1.1 Organization

This report mainly serves as a documentation for the understanding of the implementation of a Template Library for Handling Matrix. As the purpose of any documentation is to mainly serve as a reference, this report therefore serves to describe the implementation of the project and the testing scenarios that were made. As such, the report is organize as follows:

1. The `introduction` serves to give a description of the project, for the purpose of introducing readers to the content of the report.
2. The `design` part presents the different components (Methods) that were used and modified and explains the design approach of template library.
3. We discuss how the design was transform to code in the implementation section of the report.
4. As testing is an integral component of software design, we test the implementation of the Matrix Template Library and respectively discuss the results of the tests and usecase.

2 Design

Matrix, in all its versions, The relationships between the various kinds of matrices has to be analyzed. In particular, some possible choices might be: Firstly, Using the Strategy design pattern to choose the correct algorithm depending on the type of matrix we are using; Secondly, using the Decorator design pattern to decorate the basic matrix into more and more "decorated" types (e.g. transpose of a submatrix of a matrix); However, avoid using any design pattern and heavily rely on a "standard" hierarchy of subclasses.

As said, the goal of the Matrix template library design is to make the portion of the operation as simple as possible by reducing its number of code lines and performance. This means that significant components of the system should be moved out of memory space to user space, a space that typical user-level applications runs in the Monolithic design.

The goal of any piece of software design is reducing complexity and libraries are not an exception to this design rule. A typical computer science design approach for solving the complexity of software is by breaking it down. It is known that breaking a problem down into manageable pieces helps reduce the complexity of the problem and consequently makes it easier to tackle and solve. This idea is highly reflected in the design of most software systems.

For instance, the TCP/IP protocol stack organizes its different components into seven(7) different layers, each responsible and handling a separate task, and communicating with the other different layers if need be. Such a layered design approach is highly embodied in the Template Library design.

The Matrix Template Library has a total of 9 header files and one main file. All headers but **Matrix.h and MultipleMethod.h**, which holds the general method implementation, runs in Library space. Therefore to realize this project, various modifications of the method components were made.

In this Matrix Library, the idea for this project is to share data between subsequent calls. For example, taking the transposed matrix doesn't require a copy of the data. The behaviour of the copy constructor deep copies the data into a new matrix.

The library has been implemented using the **decorator pattern**. Decorator pattern because the concept of decoration provides an intuitive analogy to be used in the program's logic and eases the developer's life when it comes to "aggregate" decorations (e.g. make a diagonal matrix from the transpose of a submatrix).

The full type information is added in the template of **Matrix.h and StaticMatrixSize.h**, in order to increase performances. The library keeps full type information about the operation performed. This means that accessing the data doesn't require any call to virtual methods. This allows the compiler to fully inline the code, even after multiple call chains.

The library is templated to the type of the data contained in the matrix.

3 Implementation

This project involves implementing a dynamic templated library handling matrices, vectors, and co-vectors; where vectors are $nx1$ matrices and co-vectors are $1xn$ matrices. modifying its different components and making operation from it.

3.1 The Basic Matrix

The basic matrix consists in a simple matrix with no decoration whatsoever. This matrix is described by the data-type contained in its cells (i.e. T) and by its "special" type. The basic idea for this project is to share data between subsequent calls. For example, taking the transposed matrix doesn't require a copy of the data. The behaviour of the copy constructor deep copies the data into a new matrix.

To instantiate a new matrix and to read and write data, with the () operator you can use the following code:

```
 Matrix<int> m(10, 20);
m(1,3) = 10;
 std::cout << m(1,3); //Prints 10
```

Listing 1: Matrix.h

Deep copy for which is a direct copy of the original Matrx:

```
Matrix<int> m2 = m; //Triggers copy constructor that does a deep copy
m2(1,3) = 20;
std::cout << m(1,3); //Prints 10
std::cout << m2(1,3); //Prints 20
```

Listing 2: Matrix.h

Supports `const` version:

```
const Matrix<int> m3 = m;
std::cout << m3(1,3); //Prints 10
m3(1,3) = 30; //Compile error
```

Listing 3: Matrix.h

3.2 The Static Matrix Sizer

If you know the size of the matrix at static time, you could use the class `StaticMatrixSize.h`, which extends `Matrix.h`.

```
StaticSizeMatrix<10, 20, int> m; //Creates a 10x20 matrix of integers
```

Listing 4: StaticMatrixSize.h

You can use all the methods of `Matrix` on a `StaticSizeMatrix`, however all the methods add additional static information about the matrix dimensions.

For example:

```
StaticSizeMatrix<10, 20, int> m; //Creates a 10x20 matrix of integers
//As in Matrix<int>
m(1,3) = 10;
std::cout << m(1,3); //Prints 10
//Unique for StaticSizeMatrix
m.get<1, 3>() = 10;
std::cout << m.get<1, 3>(); //Prints 10
m.get<40, 60>() = 10; //Compiler error! Index out of bounds
```

Listing 5: StaticMatrixSize.h

On the other hand, the C library function to make (and tried to enforce using assert) is that users will try to do legit operations, i.e. no out of range operations, wrongly called constructors and similar. When using a StaticSizeMatrix, some methods will be enabled only for some specific matrices. For example, the diagonal can be obtained only for square matrices.

3.3 The Iterator

Iterators and the relative const variants are provided. In particular, they are defined in such a way that a matrix can be traversed either row-wise or column-wise, only going forward. The row iterator is the standard one (i.e. a [row] iterator is returned when the methods begin() and end() are called on a matrix).

Heavy use of function overloading lets the user get different iterators using the same methods: it is always possible to iterate the single row/column and it is also possible to iterate the whole matrix by row using the standard begin() and end() methods. Iterating over columns can be done only by single column; this means that iterating the whole matrix by column requires a for loop.

There are two iterators, one for row-major order, one for column-major order:

```
//For each row, for each column
for (auto it = m.beginRowMajor(); it != m.endRowMajor(); ++it) {
    std::cout << *it << std::endl;
}

//For each column, for each row
for (auto it = m.beginColumnMajor(); it != m.endColumnMajor(); ++it) {
    std::cout << *it << std::endl;
}
```

Listing 6: StaticMatrixSize.h

3.4 The Diagonal

Given a nxm matrix: then mxn matrix is diagonal, if $n <= m$ and the mxm matrix is diagonal, otherwise.

Calling the diagonalMatrix() method returnd a vector (i.e. a n x 1 matrix) whose elements are the ones of the diagonal of the matrix the method is called on. Iterating this vector by column, we obtain a single column representing the vector (the Matrix) iterating by row, we obtain min(n, m).

3.5 The Transpose Matrix

Calling the transpose() method on a matrix returns the transpose of that matrix. The transpose is simply obtained by taking the matrix and logically swapping rows and columns: row iterators and column iterators are exchanged, the direct access operator automatically swaps the column number and the row number and so on. This saves us much time since we can rely on an already written logic

3.6 The SubMatrix

Calling the submatrix() method on a matrix requires 4 arguments: `first row, last row, first column, last column`. Note that the last row/column are excluded from the submatrix, i.e. the intervals taken are closed on the "begin" side and open on the "end" side.

Following the logic of the transpose, the submatrix relies on the matrix from whom it is built for almost all operations: for instance, the operator() relies on (and "shifts") the decorated matrix's operator().

3.7 The Memory Shared Data

Sharing the memory is done through a shared_ptr. This provides a very powerful tool to apply changes even to temporary variables.

Since the decorated matrices share the memory with the original ones calling the transpose changes matrix even though the temporary transpose matrix is deleted. Heavy use or return value optimization is made, especially when decorating matrixes

The operator is not redefined for iterator to enforce the creation on new ones every time one needed, in order to try and avoid the use of not valid anymore iterator.

```
/m.transpose()(3,1) = 40;
std::cout << m(1,3); //Prints 40


auto sm = m.submatrix(2, 2, 3, 3); //Does NOT trigger copy constructor

sm(0,0) = 50;
std::cout << m(2,2); //Prints 50

m(3,3) = 60;
std::cout << sm(1,1); //Prints 60

//Chained calls also share data
m.traspose().diagonal()(1,0) = 70;
std::cout << m(1,1); //Prints 70
```

Listing 7: StaticMatrixSize.h

3.8 The Matrix Method Data

The MatrixData class is an abstract class whose purpose is to expose the getter and setter for the data. In our implementation, the set can throw an exception if the operation is not supported.

The base implementation is `VectorMatrixData<T>`: it holds the data in a linearised `std::vector<T>`. Other implementations such as `SubmatrixMD<T>` or `TransposedMD<T>` wrap another `MatrixData<T>` and change the behaviour of the getter and the setter.

The base (int, int) constructor of `Matrix<T>` creates a `VectorMatrixData<T>` by default.

3.9 The MatrixCell Method

The (int, int) operator of Matrix, used to access and set the cells, returns a `MatrixCell<T>`. This class exposes the operations required to use it as a T, and the = operator in order to change the value of the cell. This is done because for some operations (such as returning the zeroes in diagonalMatrix) it's not possible to return a reference to the value.

A const `Matrix<T>` cannot be modified, so it needs to return a const `MatrixCell<T>`. If we allowed the copy constructor it would be possible to assign a const `MatrixCell<T>` to a `MatrixCell<T>`, thus allowing to modify a const `Matrix<T>`. For this reason we decided to delete the copy constructor. We could have triggered a deep copy, but this behaviour would have been unexpected by the end user and has no practical use.

3.10 Vectors and Co-Vectors

When using the class Matrix, vectors and covectors are not specially handled. They are simply a $n \times 1$ and $1 \times n$ matrices. There are the methods isVector() and isCovector(). We chose to do this because they are simply a property of a matrix, and are not a characterization (e.g. a 1×1 matrix is both a vector and a covector).

3.11 Multiplication optimizations

When performing a multiplication between three or more matrices, like $m1 \times m2 \times m3$, the order of operations will be rearranged in order to reduce the total number of calculations needed.

For example, then multiplying a 2×3, 3×5 and 5×2 matrices, the $(3 \times 5) \times (5 \times 2)$ multiplication will be executed first, since it will reduce by 5 the number of dimensions.

This is done inside the decorator class MultiplyMD. At the first access to the data, the following operations are performed:

- The chain of multiplications is saved inside a vector
- If the chain is only two matrices long, I can stop the optimization in order to keep accessing the data without using virtual operators
- While the chain has more than one element: Find the pair of matrices in the chain that will be the most efficient to multiply Replacing the two matrices with a MatrixData that represents their multiplication
- The last matrix in the chain is the result of the multiplication

In the end, there will be an optimized operation tree, which can be accessed in an optimal order.

3.12 Additions and multiplications

Both `Matrix.h` and `StaticMatrixSize.h` support addition and multiplication by another Matrix or StaticSizeMatrix.

When the size of the resulting matrix can be inferred at compile time, a StaticSizeMatrix is returned, otherwise a Matrix is returned.

The addition is allowed only for matrices of the same size and the multiplication is allowed only when the number of columns of the first matrix is the same as the number of rows of the second. When using StaticSizeMatrix, this check is performed at compile time, otherwise it is performed at runtime.

```
StaticSizeMatrix<2, 3, int> mA;
StaticSizeMatrix<3, 5, int> mB;
StaticSizeMatrix<5, 2, int> mC;
StaticSizeMatrix<2, 3, int> mD;

//...

(mA * mB * mC).print("\%02d"); //Prints the product of mA, mB and mC
(2 * mA + mD + mD).print("\%02d"); //Prints 2mA + 2mD
auto err = mA + mB;//Compiler error: the matrix must be of the same size
auto err2 = mA * mC;//Compiler error: incompatible sizes
```

Listing 8: StaticMatrixSize.h

When performing a chain of multiplications, like $mA*mB*mC$ in the example above, some optimization on the order of operations is made, in order to minimize the total number of calculations to perform.

3.13 Sum and multiplication between matrices of different types

To sum or multiply matrices of different types, you first have to cast one of them, so they are of the same type.

For example:

```
StaticSizeMatrix<4, 9, double> A;
StaticSizeMatrix<4, 9, int> B;

...

auto sum = A + B.cast<double>();
```

Listing 9: StaticMatrixSize.h

4 Conclusion

Matrix Template Library project is a very interesting as we have seen so much we have learn from this project implementation. It however doesn't implement all standard which consequently makes it a rather restricted environment to work with. The aim of this project was to realize the implementation of a templated library handling matrices, vectors, and co-vectors, where vectors are $n \times 1$ matrices and co-vectors are $1 \times n$ matrices.

To this end, various modifications were made to the different Modules. We've also seen that such high performance vector and matrix computations. It can be used only in problems when the size of the matrices is known at compile time,

But the advantages of such designs is; it is based on a C++ technique, called expression templates, to achieve an high level optimization. The C++ templates can be used to implement vector and matrix expressions such that these expressions can be transformed at compile time to code which is equivalent to hand optimized code, improved scalability thus new features can be added to the implementation without making much modifications in the design.

From the design of Matrix to the Template implementation and to finally carry out some sample test cases, we manage to present in this report the very basic essential of how Template can provide support for handling Matrices of Vector and Co-Vector

A Future Work

Hopefully, as a student playing with source code make it more interesting to learn new ideas and knowledge from which more correction are adopted. There is a lot that can be learn from this Project and some extension can be added. Me and my partner have learn a lot from this project and yet wish to improve on it for some extension of operators that can be implement.

B User Manual

In the main.cpp file there is a main function that can be called to ensure that all tests are successful. Every major method is tested.

The library has been complied and tested with CMake under the IDE "CLion".

However the compilation on terminal use:

$clang++ - std = c++14 - stdlib = libc++ - Weverything$

Followed by *main.cpp*

You can make the name `-o text.exe` at the end to have the object file translated to an exe file. The std=c++14 otherwise in version 11 is so full of errors I could not figure out how possible it happen. The compilation process was tested both in clang++ 12.0.0 (clang-1200.0.32.28)

C Classes

Class Name	**Description**
Matrix.h	Briefly state the purpose of this class
MultipleMethod.h	Implementation of matrix operations class is an abstract class whose purpose is to expose the getter and setter for the data
StaticMatrixSize.h	size of a matrix & enable additional checks at compile time reduce as much as possible the number of errors at runtime
Sum.h	exposes the sum of the two given matrices
Multiplication.h	exposes the multiplication of the two given matrices
Iterator.h	Iterator that iterates in row-major order & in column-major order
MatrixCell.h	The operator that used to access and set the cells of Matrix data
MatrixUtils.h	Implement the Optimizer fort he Matrix Data
Utils.h	Contains helper function
main.cpp	Test of method implemented

Table 1: List of Added classes