Advanced algorithms and programming methods

Year: 2017/2018

Assignment 2

Matrix Library

17th December 2017

Team: **CppTeam**
Chiarot Giacomo 854893
Hibraj Feliks 854342
Marcuzzi Federico 853770

Index

1 Introduction

1.1 Project overview

Extend the matrix library (either the one presented or, if you feel confident and lucky, your own) to include the following features:

1. **Compile-time matrix dimensions**: provide a variant of the matrix that maintains compile-time information about height and width. allow accessors to be bounds-checked at compile time and maintain compile-time size information as much as possible through matrix operators.
2. **Matrix operations**: provide matrix addition and (row-by-column) multiplication. Operations should interact with compile-time dimensions correctly (dimensions checked at compile-time).

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.

2 Implementation

2.1 Design

The library design is based on templates. Inheritance is solved in compile time: when the user does a transformation of an original matrix, the compiler creates a new type which inherits values and methods from the original one.
The distinction between static and dynamic matrices is based on templates too, as shown in the diagram below.

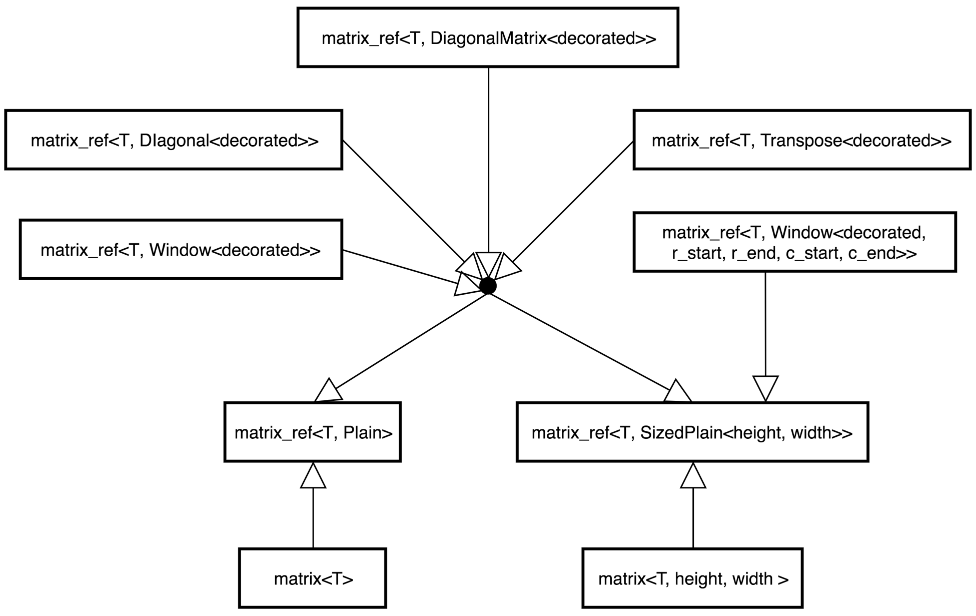
matrix_ref<T, DiagonalMatrix<decorated>>
matrix_ref<T, DIagonal<decorated>>
matrix_ref<T, Transpose<decorated>>
matrix_ref<T, Window<decorated>>
matrix_ref<T, Window<decorated, r_start, r_end, c_start, c_end>>
matrix_ref<T, Plain>
matrix_ref<T, SizedPlain<height, width>>
matrix<T>
matrix<T, height, width >

Figure 1: Matrix Design

As you can see from (Fig. 1), the distinction between static and dynamic matrices (`matrix` or `matrix_ref`) is based on Plain and SizedPlain. For the other types of matrices (transpose, window, ...) there is no distinction from the graph shown above.
Plain represents a dynamic matrix where informations (content) are acquired at run-time. While SizedPlain keeps static information from compile-time (Height, Width). These informations are kept only on the head of the template chain created on further transformations.
Given this choice, we decided to use 2 methods to help us retrieving these variables in a `matrix`, which template type is nested as we just said. Methods must give us the result at compile-time in order to operate dimension checks at compile-time.

We have defined methods `get_height_static()` and `get_width_static()` in any type of matrices, either static or dynamic.
In `matrix_ref<height,width>`, which is static, above mentioned methods staticly returns parameters height and width included as template parameters for the matrix.

```
...
 /* return static parameters height e width included as
    template parameters of matrix */
 static constexpr unsigned get_height_static() {return height;}
 static constexpr unsigned get_width_static() {return width;}
...
```

Dynamic matrices instead, given the fact that their dimensions are not template parameters, return 0 on such methids:

```
...
 // ficticious method returning 0
 static constexpr unsigned get_height_static() { return 0; }
 static constexpr unsigned get_width_static() { return 0; }
...
```

All other matrix types call methods of base class as follows:

```
...
// calls static method of father
 constexpr static unsigned get_height_static() {
   return base::get_width_static();
 }
 constexpr static unsigned get_width_static() {
   return base::get_height_static();
 }
...
```

When static controls are done, we have to pay attention to matrices received in input from functions, operators and constructors.
If matrices are dynamic, dimension check can't be done staticaly, even though, as we said, they have the same static methods as static matrices. But we took care of distinguishing such matrices by adding a constexpr static boolean attribute `matrixStatic` on classes `matrix_ref` and `matric_ref_static`, set to:

- **true:** for static matrices
- **false:** for synamic matrices

By doing so, we first check if the `matrix` is static, if so then we check its dimensions.

2.2 Access control

Access control is made on run time or compile time depending on the situation. Controls are made before accessing to a memory location or performing any operation.
To access an element in a matrix, two methods are provided: one is static, the other dynamic. Dynamic matrices have only the dynamic one. This gives developers the choice to use the one they need.
Operations make dimension checks in compile-time only if the two matrices provided are both static, otherwise the control is made at run-time.
In case a control fails, we have two different behaviours:

- in case the control is made by the compiler then a compilation message error will come out.
- in case the control is made on run-time the library will throw an exception.

This way, developers are able to handle the error without interrupting the application execution.

In case of expressions involving mixed types of matrices (static and dynamic), at compile-time a control will be done on pairs of contiguous static matrices in that expression for same operator (+ or $*$), thus providing a partial control as far as it's possible, giving the ability to make controls even when dynamic matrices are involved.

```
int main() {
   matrix<int, 4, 5> static_A; //A is a static matrix
   matrix<int, 7, 5> static_B; //B is a static matrix
   matrix<int> dynamic_C(4,5); //B is a dynamic matrix

   // the compiler rises an error because static_A and static_B
   // have different shapes and are next to each other
   auto C = static_A + static_B + dynamic_C;

   // the compiler doesn't rise an error because
   // static_A and static_B are not next to each other but at
   // run-time the program rises an exception anyway
   auto C = static_A + dynamic_C + static_B;
}
```

It's also possible to declare a static matrix from a dynamic one. This way the developer is able to do static controls on the accesses or operations over that matrix. In case dimensions does not match, a run-time exception will be thrown.

```
int main() {

   int i;
   int j;

   std::cin >> i;
   std::cin >> j;

   matrix<int> dynamic_A(i,j);  //A is a dynamic matrix
   matrix<int, 7, 5> static_B;  //B is a static matrix
```

```
    // if at run-time the user doesn't insert the
    // pair (5,7), the program will rise an exception
    matrix<int,5,5> static_C = dynamic_A * static_B;

    // the user can in any case check at compile-time
    // indices of static_C
    static_C.get<3,3>(); // correct
    static_C.get<3,7>(); // correct but compiler rises an error
                         // at compile-time
}
```

2.3 Operation

In this library sum and multiplication between matrices are implemented. Multiplications are optimized in order to minimize the complexity of computations. To do that, the multiplication operator returns an object, `lazy_product`, which contains a list of all matrices contained in a chain of multiplications. A `solve()` method is called to compute the result of the matrix chain. The library has two types of `lazy_product`, one is dynamic, the other one is static. The static one contains in its template the dimensions of the last matrix inserted in the list. This way, static control between pairs of static matrices can be done by the compiler. Below, the graph showing relations between the two classes.

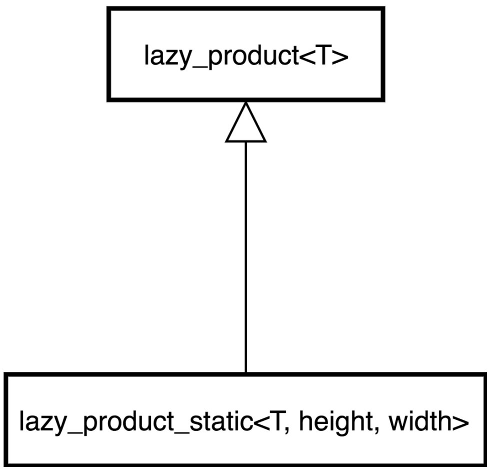

Figure 2: lazy_product design

Operators are defined outside matrix classes to avoid redundant definitions. Here an example of * operator: it takes in input two matrices of arbitrary type and returns a `lazy_product`.

```
template <typename T, class leftType, class rightType>
lazy_product<T>operator*(const matrix_ref<T, leftType>&left_m,
                         const matrix_ref<T, rightType>&right_m){
    if (left_m.get_width() == right_m.get_height())
        return lazy_product<T>(left_m, right_m);
    else
        throw std::runtime_error("Left matrix's column number
                   must be equal to right matrix's row number");
}
```

Since multiplications bewteen two matrices return a `lazy_product`, the product between a `lazy_product` and a matrix is defined too.

```
template<class type>
lazy_product<T> operator*(matrix_ref<T, type> &right) {
    list.push_back(matrix_wrap<T>(right));
    return *this;
}
```

Since the `solve()` method is called on a `lazy_product` only in presence of a sum at the beginning or at the end of a multiplication chain or in case it is assigned to a matrix variable, if the programmer specify the type `auto` before a multiplication chain, the result matrix is always a `lazy_product`.

```
...
   matrix<int> A(4, 5);   //A is a dynamic matrix
   matrix<int, 5, 3> B;   //B is a static matrix
   matrix<int, 3, 8> C;   //C is a static matrix

   // D is a lazy_product
   auto D = A * B * C;

   // E is a dynamic matrix
    matrix<int> E = A * B * C;
...
```

To create a static matrix from an expression the programmer must explicitly define the size.

```
...
   matrix<int> A(4, 5);   //A is a dynamic matrix
   matrix<int, 5, 3> B;   //B is a static matrix
   matrix<int, 3, 8> C;   //C is a static matrix

   // now D is a static matrix
   matrix<int, 4, 8> D = A * B * C;
...
```

2.3.1 Multiplication algorithm

To improve multiplication performance, our algorithm is composed of two different techniques combined. The first starts from right to left and gives priority to those couple of matrices in which the right matrix column is less or equal than the left one. The second technique starts from left and gives priority to those couples in which the left matrix column number is greater or equal than the right one. The two methods are applied cyclically until the list containing matrices to be evaluated becomes a singleton. Following an example of how it works:

```
 4x5    *    5x4    *    4x1    *    1x3    *    3x2

 4x5    *    5x4    *    4x1    *   (1x3    *    3x2)
 4x5    *    5x4    *    4x1    *   (1x2)

 4x5    *   (5x4    *    4x1)   *    1x2
 4x5    *   (5x4)   *    1x2

(4x5    *    5x4)   *    1x2
(4x1)   *    1x2

(4x1    *    1x2)
(4x2)
```

3 Semantic

In this section, semantic of our library is shown. Due to the fact that matrix types can become very complex, if the user does many trasformations over them, it's suggested to use `auto` as trasformation return type.

3.1 Matrices

Matrices can be instantiated in two different ways. In both cases matrix elements type must be specified. The first parameter sets the number of rows, the second the number of columns, they are `unsigned` (int) variables, so negative values will be converted into positive ones. In this example, matrix A is a dynamic matrix, while B is a static one.

```cpp
int main() {
    matrix<int> A(4, 5);   //A is a dynamic matrix
    matrix<int, 4, 5> B;   //B is a static matrix
}
```

The library provides matrix transformations too. In this update, since static matrices are provided, any operation over a static matrix will return a static matrix. The only transormation which can return either a static or a dynamic matrix is `window`, which has two different call functions. A way to obtain a dynamic matrix from a static one is to apply dynamic `window` on the entire matrix before doing any other transormation.

Following are some examples:

```
int main() {
   //the following instructions create dynamic matrix_refs
   // A is a dynamic matrix
   matrix<int> A(4, 5);
   auto A_transpose = A.transpose();
   auto A_window = A.window({1, 3, 1, 3});
   auto A_diagonal = A.diagonal();
   auto A_diagonal_matrix = A.diagonal_matrix();

   //the following instructions create static matrix_refs
   // B is a static matrix
   matrix<int, 4, 5> B;
   auto B_transpose = B.transpose();
   auto B_window = B.window<1, 3, 1, 3>();
   auto B_diagonal = B.diagonal();
   auto B_diagonal_matrix = B.diagonal_matrix();

   // dynamic matrix transformation from a static matrix
   auto C = B.window({0, 3, 0, 4}).transpose();
}
```

To access a matrix cell location the library provides two different syntaxes. The first one is called dynamic because it does the control of indices on run-time and, in case indices are not allowed it throws an exception. The second one is called static because the control is made on compile-time and, in this case, if the indices are wrong the compiler will provide an error. Static access can be called only on a static matrix, otherwise the compiler will complain.
Here an example of how to access a matrix:

```
...
  matrix<int> A(4, 5);       //A is a dynamic matrix
  int a0 = A(0, 0);          //correct
  int a1 = A(10, 10);        //correct, but will throw an exception
  int a2 = A.get<0, 0>();    //wrong, the compiler will complain
                             //(A is dynamic)

  matrix<int, 4, 5> B;       //B is a static matrix
  int b0 = B(0, 0);          //correct
  int b1 = B(10, 10);        //correct, but will throw an exception
  int b2 = B.get<0, 0>();    //correct
  int b3 = B.get<10, 10>();  //correct, but the compiler will
                             //complain because of wrong indices
...
```

4 Limits

This library has some limits, related to operations between matrices:
Since the list created by products is implemented through an `std::vector`, we can put only elements of the same type.
Our implementation uses `matrix_wrap` as a super type which can be used in this context. Due

to const correctness problems we cannot create a `matrix_wrap` from a diagonal matrix and from a static window. We cannot include a `diagonal_matrix` in an expression. This is caused by the incompatibility from the `wrap` implementation and the const characteristic of `matrix_wrap`. The class `concrete_matrix_wrap_impl` defines the following function:

```
...
T& get(unsigned i, unsigned j) override { return mat(i,j); }
...
```

This fuction tries to return a `T&` and this breaks the const-correctness.
We have a similar problem when we try to include a matrix `window` defined staticaly in an expression:

```
...
auto YY = WW * KK.window<0,5,0,4>();
...
```

This kind of problem too is related to `matrix_wrap`.

Another problem related to `std::vector` is that matrices contained on it must have same value types. So operations between diagonal matrices, static windows and matrices with different values types are not defined.
To give homogeneity to this solution, sum is defined between matrices with same value types too, although decltype could be easily used to derive the resulting type.
Another limit is that an expression cannot be passed as a value of a function since the expression contains multiplications, and `solve()` function may not be called.

The last limit of our library is that the compile-time check cannot be done between different operator types.