LAL Tutorial

The Basics

LAL library has three major classes, matrix, vector and dynamic_matrix. The namespace of LAL is lal. Note, the vector class in this tutorial only means lal::vector and is different from std::vector in standard library. The tutorial will introduce the basic usage of these three classes.

Matrix

Matrix is a fixed-size table of elements (usually numbers) with the same type. The number of rows and columns could not be changed after creation. Matrix size means the number of elements in the matrix.

The below example matrix has 2 rows and 3 columns. Its size is 6 and the type of elements is int.

```
1 2 3
4 5 6
```

Template Parameters

Matrix has four template parameter.

`NumericType` is the type of element in matrix. It could be int, double or any type. As matrix supports mathematical operations, we suggest to use numerical type.

`Rows` is the number of rows in matrix. It could be any non-negative integer.

`Cols` is the number of columns in matrix. It could be any non-negative integer.

`On_stack` is an optional boolean value indicating whether the matrix is stored on stack or on heap. The default setting is matrices with size smaller that 256 will be stored on stack and matrices with size bigger or equal than 256 will be stored on heap.

Matrix Creation

There are several ways to construct a matrix. You could create a matrix by listing elements. The below example constructs a int matrix with 6 elements and stored on stack.

```
#include <iostream>
#include <"matrix.hpp">

int main(){
    constexpr lal::matrix<int, 2, 3, true> a {{1, 2, 3, 4, 5, 6}};
    std::cout << a << "\n";
    return 0;
}
```

Output

```
1 2 3
4 5 6
```

You could also create a zero matrix by giving no element or create a matrix with same-value elements by giving only one element. The below example constructs a matrix `a` with 6 elements and stored on stack. It constructs a matrix `b` with 3 elements and stored on heap.

```
#include <iostream>
#include "matrix.hpp"

int main(){
    constexpr lal::matrix<int, 2, 3> a;
    std::cout << "matrix a" << "\n";
    std::cout << a << "\n";
    constexpr lal::matrix<int, 1, 3, false> b {1};
    std::cout << "matrix b" << "\n";
    std::cout << b << "\n";
    return 0;
}
```

Output

```
matrix a
0 0 0
0 0 0

matrix b
1 1 1
```

Basic Operation

The below examples shows how you could get the size, number of rows and number of columns of a matrix.

```
#include <iostream>
#include "matrix.hpp"

int main(){
    constexpr lal::matrix<int, 2, 3, true> a {{1, 2, 3, 4, 5, 6}};
    std::cout << "size of a: " << a.size() << "\n";
    std::cout << "number of rows: " << a.rows() << "\n";
    std::cout << "number of columns: " << a.columns() << "\n";
    return 0;
}
```

Output

```
size of a: 6
number of rows: 2
number of columns: 3
```

You could access a member by using subcript as how you access a member of a real-world matrix. You could also use at(). Member access allows you to change value of elements in a matrix.

```
#include <iostream>
#include "matrix.hpp"

int main(){
    lal::matrix<int, 2, 3, true> a {{1, 2, 3, 4, 5, 6}};
    std::cout << a;
    std::cout << "a[0][2]: " << a[0][2] << "\n";
    std::cout << "a[1][2]: " << a.at(1, 2) << "\n";
    a[0][2] = 30;
    a[1][2] *= 5;
    std::cout << a;
    return 0;
}
```

Output

```
1 2 3
4 5 6
a[0][2]: 3
a[1][2] 5
1 2 30
4 25 6
```

Mathematical Operations

You could do element-wise addition or substraction between two matrices. The elements type must be the same and the two matrices must have the same number of rows and columns.

```
#include <iostream>
#include "matrix.hpp"

int main(){
    lal::matrix<int, 2, 3> a {{2, 3, 4, 5, 6, 7}};
    lal::matrix<int, 2, 3> b {{1, 2, 5, 2, 3, 1}};
    auto c = a + b;
    std::cout << c << "\n";
    c = a - b;
    std::cout << c << "\n";

    return 0;
}
```

Output

```
3 5 9
7 9 8

1 1 -1
3 3 6
```

Multiplications could be used on matrices with same element type or between number and matrix. You could also divide a matrix by a number.

```
#include <iostream>
#include "matrix.hpp"

int main(){
    lal::matrix<double, 2, 3> a {{2, 3, 4, 5, 6, 7}};
    lal::matrix<double, 2, 3> b {{1, 2, 5, 2, 3, 1}};
    //matrix multiplication
    auto c = a * b;
    std::cout << c << "\n";
    //multiplication between a number and a matrix
    auto d = 2 * b;
    std::cout << d << "\n";
    d = b / 2;
    std::cout << d << "\n";
    return 0;
}
```

Output

```
 29 14
 56 29

 2 4
 10 4
 6 2

 0.5 1
 2.5 1
 1.5 0.5
```

You could also transpose a matrix and the operation is the same as how transpose is defined in linear algebra.

```
#include <iostream>
#include "matrix.hpp"

int main(){
    lal::matrix<double, 2, 3> a {{2, 3, 4, 5, 6, 7}};
    std::cout << a.transpose() << "\n";
    return 0;
}
```

Output

```
 2 5
 3 6
 4 7
```

Iterations

You could iterate over all elements in a matrix by row or by col. We would suggest to iterate by row if possible as it is more efficient.

```
#include <iostream>
#include "matrix.hpp"

int main(){
    lal::matrix<int, 2, 3> a {{1, 2, 3, 4, 5, 6}};
    //iterate by row
    for (auto it = a.begin(); it < a.end(); it++){
        std::cout << *it << " ";
    }
    std::cout << "\n";
    //iterate by column
    for (auto it = a.col_begin(); it < a.col_end(); it++){
        std::cout << *it << " ";
    }
    std::cout << "\n";
    return 0;
}
```

Output

```
1 2 3 4 5 6
1 4 2 5 3 6
```

Vector

Vector is a fixed-size array of elements (usually numbers) with the same type. Note, lal::vector means vector in linear algebra and is different from std::vector in standard library. Vector size means the number of elements in the vector.

The below example vector has 5 elements and the type of elements is int.

```
1 2 3 4 5
```

Template Parameters

Matrix has two template parameter.

`NumericType` is the type of elements in vector. It could be int, double or any type. As vector supports mathematical operations, we suggest to use numerical type.

`Size` is the number of elements in vector. It could be any non-negative integer.

Vector Creation

There are several ways to construct a vector. You could create a vector by listing elements. The below example constructs a int vector with 5 elements.

```
#include <iostream>
#include "vector.hpp"

int main(){
    constexpr lal::vector<int, 5> a {1, 2, 3, 4, 5};
    std::cout << a << "\n";
    return 0;
}
```

Output

```
1 2 3 4 5
```

You could also create a zero vector by giving no element or create a vector with same-value elements by giving only one element. The below example constructs a vector `a` with 5 elements and a matrix `b` with 3 elements.

```
#include <iostream>
#include "vector.hpp"

int main(){
    constexpr lal::vector<int, 5> a;
    std::cout << "vector a" << "\n";
    std::cout << a << "\n";
    constexpr lal::vector<double, 3> b {1.0} ;
    std::cout << "vector b" << "\n";
    std::cout << b << "\n";
    return 0;
}
```

Output

```
vector a
0 0 0 0 0
vector b
1 1 1
```

Basic Operation

The below example shows how you could get the size of a vector.

```
#include <iostream>
#include "vector.hpp"

int main(){
    constexpr lal::vector<int, 5> a {1, 2, 3, 4, 5};
    std::cout << "size of a: " << a.size() << "\n";
    return 0;
}
```

Output

```
size of a: 6
```

You could access a member by using subcript as how you access a member of a real-world vector. You could also use at(). Member access allows you to change value of elements in a vector.

```
#include <iostream>
#include "vector.hpp"

int main(){
    lal::vector<int, 5> a {1, 2, 3, 4, 5};
    std::cout << "a[0]: " << a[0] << "\n";
    std::cout << "a[2]: " << a.at(2) << "\n";
    a[2] = 30;
    a[3] *= 5;
    std::cout << a << "\n";
    return 0;
}
```

Output

```
a[0]: 1
a[2]: 3
1 2 30 20 5
```

Mathematical Operations

You could do element-wise addition or substraction between two vectors. The vector size and element type must be the same.

```
#include <iostream>
#include "vector.hpp"

int main(){
    constexpr lal::vector<int, 5> a {1, 2, 3, 4, 5};
    lal::vector<int, 5> b {2, 2, 7, 4, 2};
    auto c = a + b;
    std::cout << c << "\n";
    c = a - b;
    std::cout << c << "\n";
    return 0;
}
```

Output

```
3 4 10 8 7
-1 0 -4 0 3
```

We support both dot product and cross product between two vectors with same element type. `*` is for dot product and will return a number while `vector_cross_product()` is used on cross product and returns a matrix.

```
#include <iostream>
#include "vector.hpp"

int main(){
    constexpr lal::vector<int, 5> a {1, 2, 3, 4, 5};
    lal::vector<int, 5> b {2, 3, 4, 5, 6};
    std::cout << "dot product result:\n";
    std::cout << a * b << "\n";
    std::cout << "cross product result:\n";
    std::cout << vector_cross_product(a, b) << "\n";
    return 0;
}
```

Output

```
dot product result:
70
cross product result:
 2 3 4 5 6
 4 6 8 10 12
 6 9 12 15 18
 8 12 16 20 24
 10 15 20 25 30
```

Multiplications could also be used between a number and a vector. You could also divide a vector by a number.

```
#include <iostream>
#include "vector.hpp"

int main(){
    constexpr lal::vector<int, 5> a {1, 2, 3, 4, 5};
    std::cout << 2 * a << "\n";
    std::cout << a / 2 << "\n";
    return 0;
}
```

Output

```
2 4 6 8 10
0.5 1 1.5 2 2.5
```

Iterations

You could iterate over all elements in a vector by an iterator.

```
#include <iostream>
#include "vector.hpp"

int main(){
    constexpr lal::vector<int, 5> a {1, 2, 3, 4, 5};
    for (auto it = a.begin(); it < a.end(); it++){
        std::cout << *it << " ";
    return 0;
}
```

Output

```
1 2 3 4 5
```

Dynamic Matrix

Dynamic matrix is a table of elements (usually numbers) with the same type. The number of rows and columns could be changed by inserting or erasing rows/columns. Dynamic matrix size means the number of elements in the dynamic matrix.

The below example dynamic matrix currently has 2 rows and 3 columns. Its size is 6 and the type of elements is int.

```
1 2 3
4 5 6
```

Template Parameters

Matrix has one template parameter.

`NumericType` is the type of element in matrix. It could be int, double or any type. As dynamic matrix supports mathematical operations, we suggest to use numerical type.

Dynamic Matrix Creation

There are several ways to construct a dynamic matrix. You could create a dynamic matrix by assigning number of rows, number of columns and element value.

```
#include <iostream>
#include "dmatrix.hpp"

int main(){
    lal::dynamic_matrix<double> a(2, 3, 1);
    std::cout << a << "\n";
    return 0;
}
```

Output

```
1 1 1
1 1 1
```

You could also create a dynamic matrix with elements same as those in a matrix.

```
#include <iostream>
#include "matrix.hpp"
#include "dmatrix.hpp"

int main(){
    lal::matrix<int, 2, 3> mat {{1, 2, 3, 4, 5, 6}};
    lal::dynamic_matrix<int> dmat(mat);
    std::cout << dmat << "\n";
    return 0;
}
```

Output

```
1 2 3
4 5 6
```

Basic Operation

The below examples shows how you could get the size, number of rows and number of columns of a matrix.

```
#include <iostream>
#include "matrix.hpp"
#include "dmatrix.hpp"

int main(){
    lal::matrix<int, 2, 3> mat {{1, 2, 3, 4, 5, 6}};
    lal::dynamic_matrix<int> dmat(mat);
    std::cout << "size of dmat: " << dmat.size() << "\n";
    std::cout << "number of rows: " << dmat.rows() << "\n";
    std::cout << "number of columns: " << dmat.columns() << "\n";
    return 0;
}
```

Output

```
size of dmat: 6
number of rows: 2
number of columns: 3
```

You could access a member in a dynamic matrix as how you acess a member in matrix. Member access allows you to change elements.

```
#include <iostream>
#include "matrix.hpp"
#include "dmatrix.hpp"

int main(){
    lal::matrix<int, 2, 3> mat {{1, 2, 3, 4, 5, 6}};
    lal::dynamic_matrix<int> dmat(mat);
    std::cout << dmat;
    std::cout << "dmat[0][2]: " << dmat[0][2] << "\n";
    std::cout << "dmat[1][2]: " << dmat.at(1, 2) << "\n";
    dmat[0][2] = 30;
    dmat[1][2] *= 5;
    std::cout << dmat;
    return 0;
}
```

Output

```
1 2 3
4 5 6
dmat[0][2]: 3
dmat[1][2] 5
1 2 30
4 25 6
```

You could also make changes to a dynamic matrix by inserting or erasing rows/columns. You should give a non-negative integer to indicate the position of the inserted or erased row/column in the dynamic matrix.

```
#include <iostream>
#include <vector>
#include "matrix.hpp"
#include "dmatrix.hpp"

int main(){
    lal::matrix<int, 2, 3> mat {{1, 2, 3, 4, 5, 6, 7, 8, 9}};
    lal::dynamic_matrix<int> dmat(mat);
    std::cout << dmat;
    //insert a row into dmat as first row
    dmat.insert_row(0, std::vector<int>(3,0));
    std::cout << "after inserting a row\n";
    std::cout << dmat;
    //insert a column into dmat as second row
    dmat.insert_column(1, std::vector<int>(4,1));
    std::cout << "after inserting a column\n";
    std::cout << dmat;
    //erase the third column from dmat
    dmat.erase_column(2);
    std::cout << "after erasing a column\n";
    std::cout << dmat;
    //erase the third row from dmat
    dmat.erase_row(2);
    std::cout << "after erasing a row\n";
    std::cout << dmat;
    return 0;
}
```

Output

```
 1 2 3
 4 5 6
 7 8 9

after inserting a row
 0 0 0
 1 2 3
 4 5 6
 7 8 9

after inserting a column
 0 1 0 0
 1 1 2 3
 4 1 5 6
 7 1 8 9

after erasing a column
 0 1 0
 1 1 3
 4 1 6
 7 1 9

after erasing a row
 0 1 0
 1 1 3
 7 1 9
```

Mathematical Operations

You could do element-wise addition or substraction between two dynamic matrices. The elements type must be the same and the two dynamic matrices must have the same number of rows and columns.

```
#include <iostream>
#include "matrix.hpp"
#include "dmatrix.hpp"

int main(){
    lal::matrix<int, 2, 3> a {{2, 3, 4, 5, 6, 7}};
    lal::matrix<int, 2, 3> b {{1, 2, 5, 2, 3, 1}};
    lal::dynamic_matrix<int> dmat1(a);
    lal::dynamic_matrix<int> dmat2(b);
    std::cout << dmat1 + dmat2 << "\n";
    std::cout << dmat1 - dmat2 << "\n";
    return 0;
}
```

Output

```
3 5 9 7 9 8
1 1 -1 3 3 6
```

Multiplications could be used on dynamic matrices with same element type or between number and dynamic matrix. You could also divide a dynamic matrix by a number.

```
#include <iostream>
#include "matrix.hpp"
#include "dmatrix.hpp"
int main(){
    lal::matrix<double, 2, 3> a {{2, 3, 4, 5, 6, 7}};
    lal::matrix<double, 2, 3> b {{1, 2, 5, 2, 3, 1}};
    lal::dynamic_matrix<int> dmat1(a);
    lal::dynamic_matrix<int> dmat2(b);
    //matrix multiplication
    auto c = dmat1 * dmat2;
    std::cout << c << "\n";
    //multiplication between a number and a matrix
    auto d = 2 * b;
    std::cout << d << "\n";
    d = b / 2;
    std::cout << d << "\n";
    return 0;
}
```

Output

```
 29 14
 56 29

 2 4
 10 4
 6 2

 0.5 1
 2.5 1
 1.5 0.5
```

You could also transpose a dynamic matrix as how you transpose a matrix.

```
#include <iostream>
#include "matrix.hpp"
#include "dmatrix.hpp"

int main(){
    lal::matrix<double, 2, 3> a {{2, 3, 4, 5, 6, 7}};
    lal::dynamic_matrix<int> dmat(a);
    std::cout << dmat.transpose() << "\n";
    return 0;
}
```

Output

```
 2 5
 3 6
 4 7
```

Iterations

You could iterate over all elements in a dynamic matrix by row or by col. We would suggest to iterate by row if possible as it is more efficient.

```
#include <iostream>
#include "matrix.hpp"
#include "dmatrix.hpp"

int main(){
    lal::matrix<int, 2, 3> mat {{1, 2, 3, 4, 5, 6}};
    lal::dynamic_matrix<int> dmat(mat);
    //iterate by row
    for (auto it = dmat.begin(); it < dmat.end(); it++){
        std::cout << *it << " ";
    }
    std::cout << "\n";
    //iterate by column
    for (auto it = dmat.col_begin(); it < dmat.col_end(); it++){
        std::cout << *it << " ";
    }
    std::cout << "\n";
    return 0;
}
```

Output

```
1 2 3 4 5 6
1 4 2 5 3 6
```