Report

member: 李佳纯 黄昊南 邓植仁

Designing

We design that the data structure is completely divided with the algorithm invocation. The data part is mainly described as OwnedMatrix. Actually, if it's allowed, the better realization would be the automatic matrix types. But the ownership controlling system must base on the OwnedMatrix series.

The second core part is the algorithm part. In this part, we just need to describe the algorithm itself but do not care about the concrete algorithm type. Then we can use the template type information to describe it then realize the generic algorithm like get inverse or get the sum.

To make full usage of the matrix access, we define the new type as proxy to temporary cache the index, and after that we can easily visit our element value by this.

Highlights

- Test Framework
  We build a private test framework system which can efficiently test our lib codes in a synchronously and asynchronously way.
- The advanced integral type
  We make a more wide integral type `int128_t` to use. Some more convenience techniques including the literals are used. Also the related `uint128_t` type is supported.
- Stringizing operator: Intelligently stringize an object. Make full use of the compiling rules to transfer an object as a string.
- Type Traits: Define quite a lot of type traits methods and operators for different types and for the future and unknown types.
- The exception inheriting system. Different exception are defined as an inheriting tree, and the core abstract exception classes cannot be instantiated.
- Assert Rule also is designed well in the inheriting system. Some more specific information would be given altogether with the context information.
- Format Library Support. It's convenient to construct a string in the format basic rule.
- Proxy Class for the visit control.
- Meta-Programming in the `uint128_t` and `int128_t` design.
- Type Upgrade automatically and intelligently.
- The algorithm design is divided completely with the matrix structure. It's open to design a better matrix based on the `OwnedMatrix` designs. For example the auto-owning matrixes and the inertia matrixes.

Difficulties

- Eigenvalue is hard to compute
- Hard to cope with complex in Eigenvalue computation
- OpenCV
- It needs hard thoughts to comparing different basic data types.

- Complex is quite difficult to deal with, because it seems a normal data type but completely different from other normal data types.

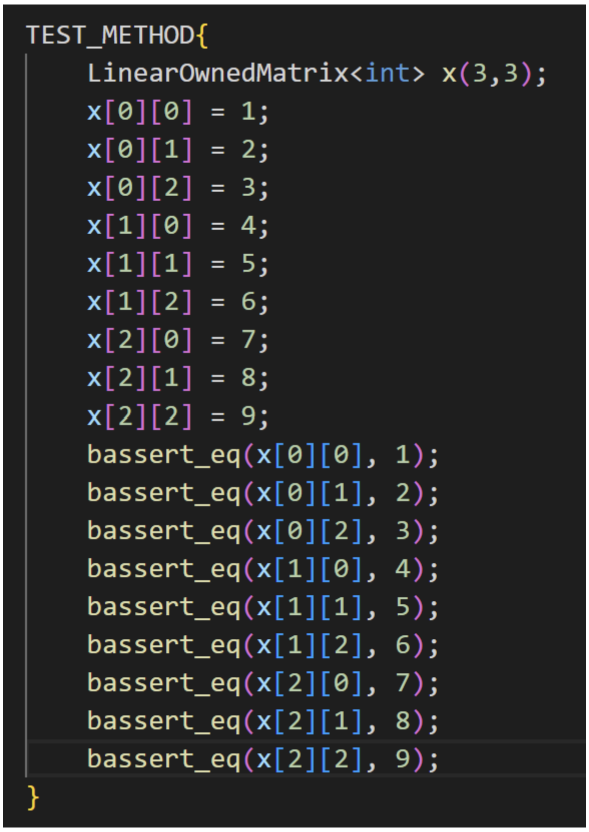
Core Codes & Tests

Class Matrix:

We use template class to implement the matrix. It supports numeric types and arbitrary size. And it is easy to extend.

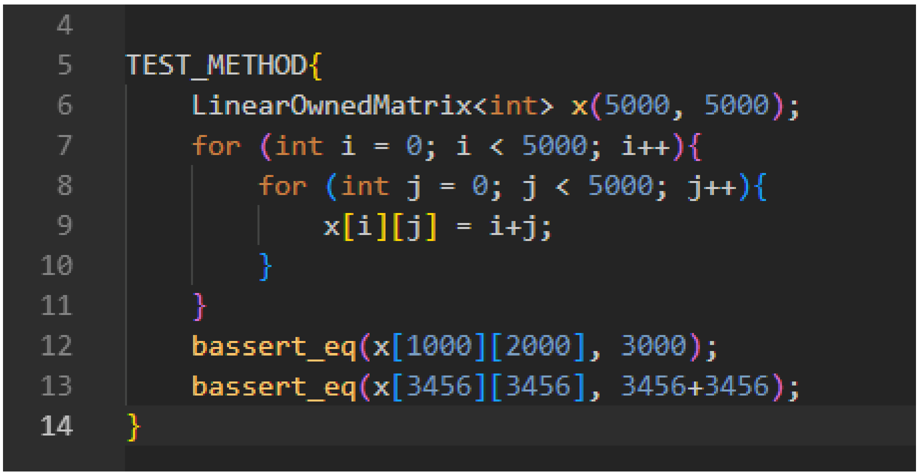
```
namespace matrix {
    template <typename ValueType_, template <typename ...> typename
ValueContainer_ = std::valarray>
    class LinearOwnedMatrix : ValueContainer_<ValueType_> {
        private:
            size_t m_row;
        public:
            using ValueType = ValueType_;
            template <typename... Args>
            using ContainerType = ValueContainer_<Args...>;
            using Super = ContainerType<ValueType>;
            using This = LinearOwnedMatrix;
            template <typename OtherDataType>
            using MatrixOfType = LinearOwnedMatrix<OtherDataType>;

            constexpr static bool is_fixed = false;

        private:
            LinearOwnedMatrix(Super &&self, size_t row): Super(std::move(self)),
m_row(row) {
                lassert (size());
            }
    }
}
```

Small Fixed-Size Matrix

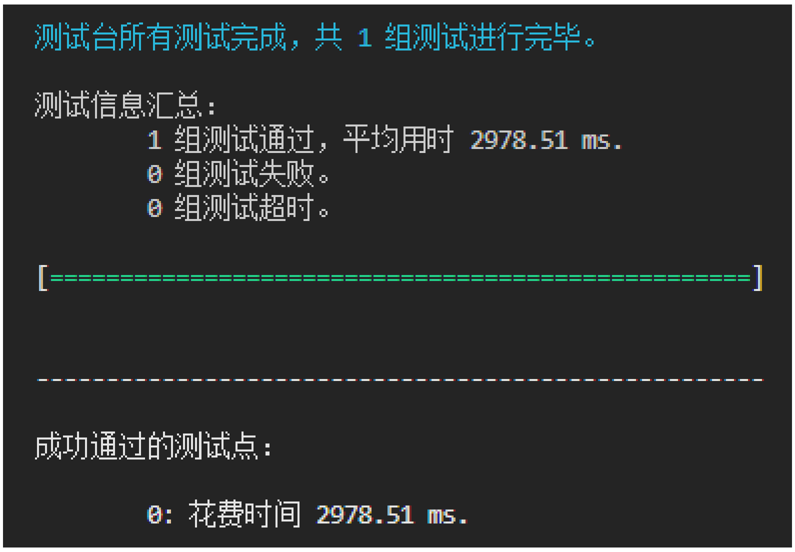


```
TEST_METHOD{
    LinearOwnedMatrix<int> x(3,3);
    x[0][0] = 1;
    x[0][1] = 2;
    x[0][2] = 3;
    x[1][0] = 4;
    x[1][1] = 5;
    x[1][2] = 6;
    x[2][0] = 7;
    x[2][1] = 8;
    x[2][2] = 9;
    bassert_eq(x[0][0], 1);
    bassert_eq(x[0][1], 2);
    bassert_eq(x[0][2], 3);
    bassert_eq(x[1][0], 4);
    bassert_eq(x[1][1], 5);
    bassert_eq(x[1][2], 6);
    bassert_eq(x[2][0], 7);
    bassert_eq(x[2][1], 8);
    bassert_eq(x[2][2], 9);
}
```

Arbitrarily Large Dense Matrix



```

TEST_METHOD{
    LinearOwnedMatrix<int> x(5000, 5000);
    for (int i = 0; i < 5000; i++){
        for (int j = 0; j < 5000; j++){
            x[i][j] = i+j;
        }
    }
    bassert_eq(x[1000][2000], 3000);
    bassert_eq(x[3456][3456], 3456+3456);
}
```

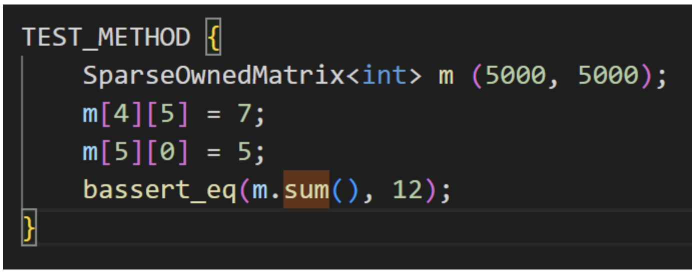


```
测试台所有测试完成，共 1 组测试进行完毕。

测试信息汇总：
        1 组测试通过，平均用时 2978.51 ms.
        0 组测试失败。
        0 组测试超时。

[===================================================]


-----------------------------------------------------

成功通过的测试点：

        0: 花费时间 2978.51 ms.
```

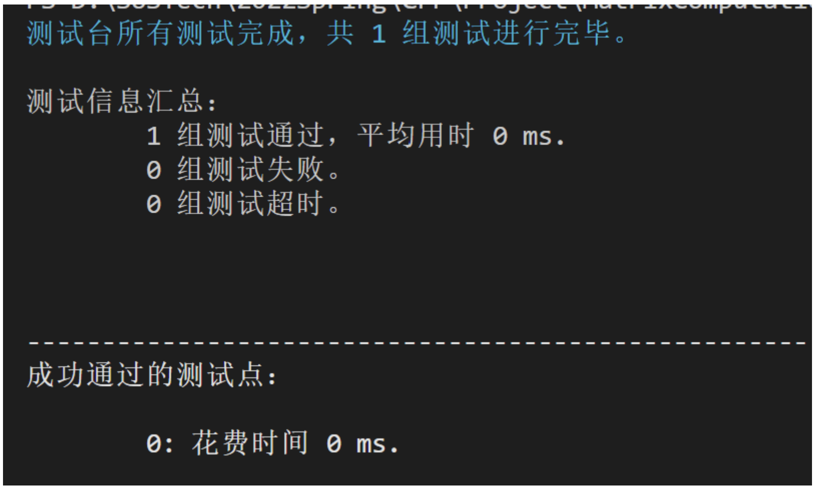
Sparse Matrix

```
            ValueType sum() const {
                ValueType result = 0;
                for (auto &&v: *this) {
                    result += v.second;
                }
                return result;
            }
```

```
template <typename T, template <typename ...> typename container = std::map>
    struct SparseOwnedMatrix : private container<std::size_t, T> {
        private:
            size_t row, col;
        public:
            using ValueType = T;
            template <typename O>
            using MatrixOfType = SparseOwnedMatrix<O, container>;
            using This = SparseOwnedMatrix;
            using Super = container<std::size_t, T>;

            SparseOwnedMatrix(size_t row, size_t col): row(row), col(col) {}
    }
```

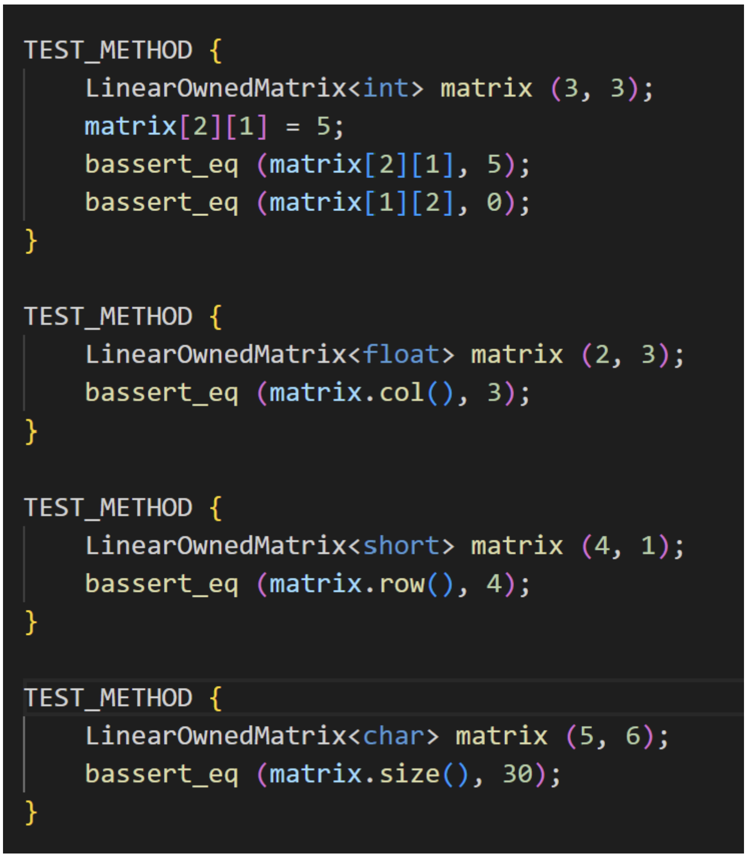


```
TEST_METHOD {
    SparseOwnedMatrix<int> m (5000, 5000);
    m[4][5] = 7;
    m[5][0] = 5;
    bassert_eq(m.sum(), 12);
}
```

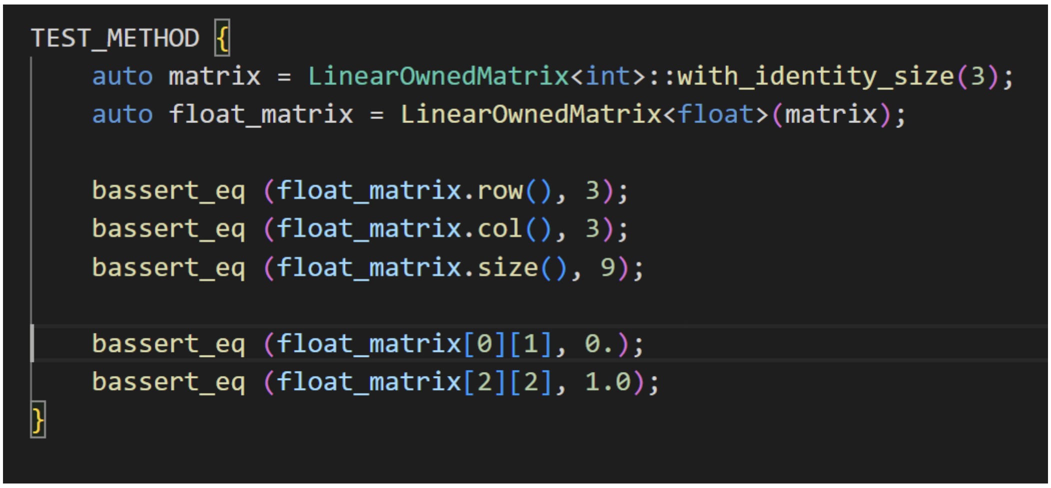


```
测试台所有测试完成，共 1 组测试进行完毕。

测试信息汇总:
        1 组测试通过, 平均用时 0 ms.
        0 组测试失败。
        0 组测试超时。



----------------------------------------------------
成功通过的测试点:

        0: 花费时间 0 ms.
```

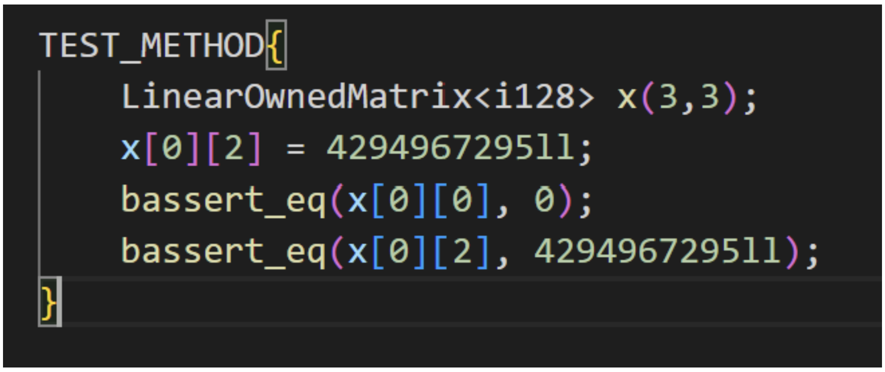
Numeric Types Support



```
TEST_METHOD {
    LinearOwnedMatrix<int> matrix (3, 3);
    matrix[2][1] = 5;
    bassert_eq (matrix[2][1], 5);
    bassert_eq (matrix[1][2], 0);
}

TEST_METHOD {
    LinearOwnedMatrix<float> matrix (2, 3);
    bassert_eq (matrix.col(), 3);
}

TEST_METHOD {
    LinearOwnedMatrix<short> matrix (4, 1);
    bassert_eq (matrix.row(), 4);
}

TEST_METHOD {
    LinearOwnedMatrix<char> matrix (5, 6);
    bassert_eq (matrix.size(), 30);
}
```



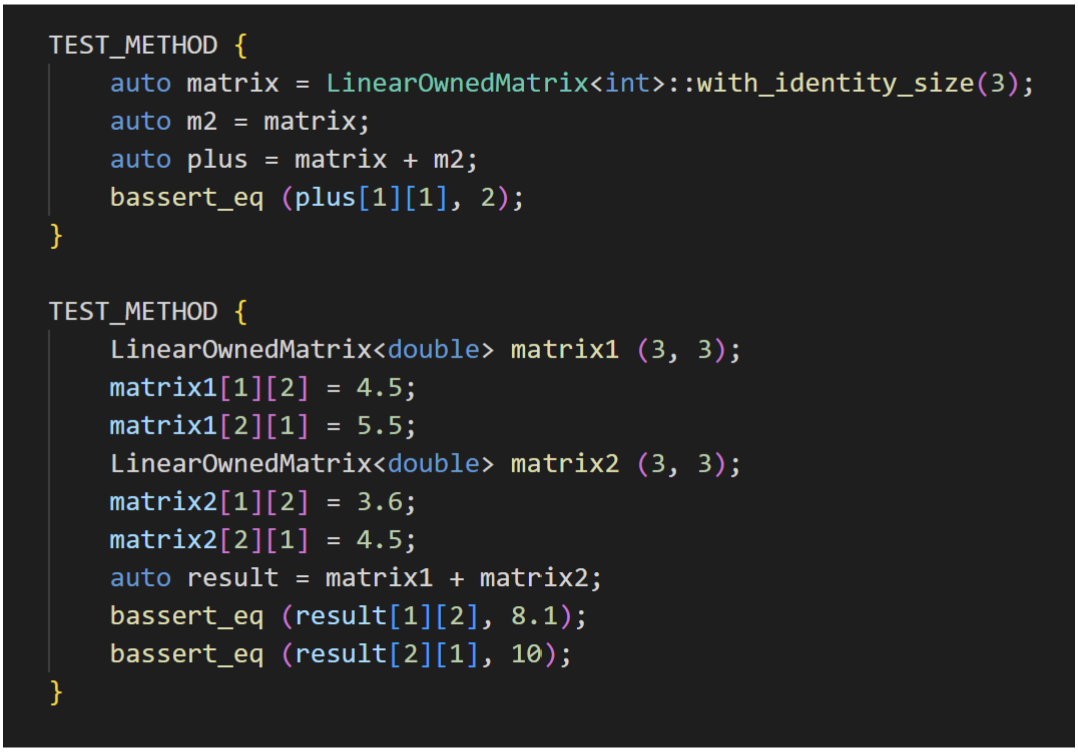
```
TEST_METHOD {
    auto matrix = LinearOwnedMatrix<int>::with_identity_size(3);
    auto float_matrix = LinearOwnedMatrix<float>(matrix);

    bassert_eq (float_matrix.row(), 3);
    bassert_eq (float_matrix.col(), 3);
    bassert_eq (float_matrix.size(), 9);

    bassert_eq (float_matrix[0][1], 0.);
    bassert_eq (float_matrix[2][2], 1.0);
}
```



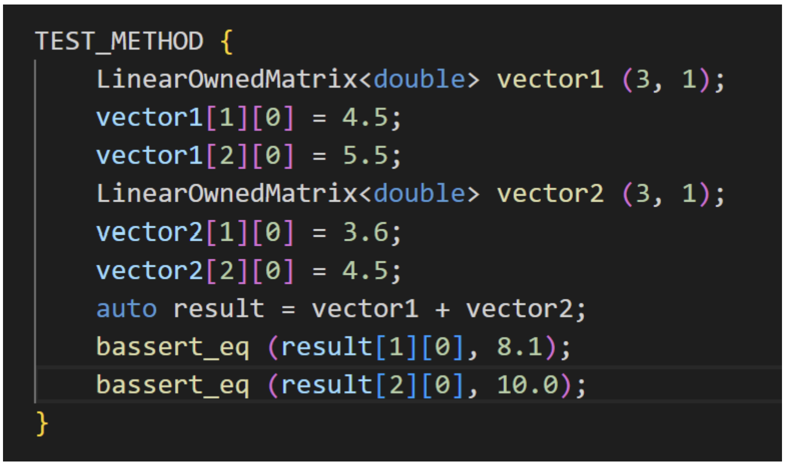
```
TEST_METHOD{
    LinearOwnedMatrix<i128> x(3,3);
    x[0][2] = 4294967295ll;
    bassert_eq(x[0][0], 0);
    bassert_eq(x[0][2], 4294967295ll);
}
```

Matrix and Vector Addition

```
LinearOwnedMatrix operator+(LinearOwnedMatrix const &rhs) const {
                auto row = m_row;
                // Throw an exception describes the mismatching of the matrix.
                Use matrix::exception;
                if (rhs.row() != row) {
                    throw MatrixStructureMismatchingException("LinearOwnedMatrix
addition with the mismatched row value. ");
                } else if (size() != rhs.size()) {
                    throw MatrixStructureMismatchingException("LinearOwnedMatrix
addition with the mismatched col value. ");
                }
                return LinearOwnedMatrix ((Super&)*this + (Super&)rhs, row);
            }
```



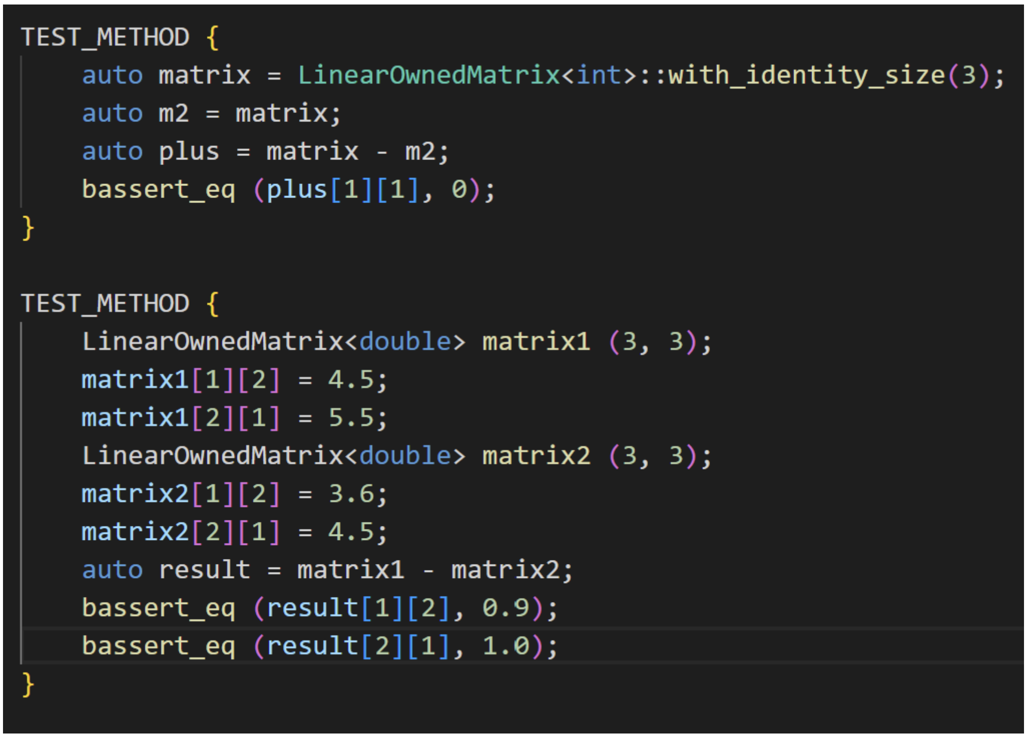
```
TEST_METHOD {
    auto matrix = LinearOwnedMatrix<int>::with_identity_size(3);
    auto m2 = matrix;
    auto plus = matrix + m2;
    bassert_eq (plus[1][1], 2);
}

TEST_METHOD {
    LinearOwnedMatrix<double> matrix1 (3, 3);
    matrix1[1][2] = 4.5;
    matrix1[2][1] = 5.5;
    LinearOwnedMatrix<double> matrix2 (3, 3);
    matrix2[1][2] = 3.6;
    matrix2[2][1] = 4.5;
    auto result = matrix1 + matrix2;
    bassert_eq (result[1][2], 8.1);
    bassert_eq (result[2][1], 10);
}
```



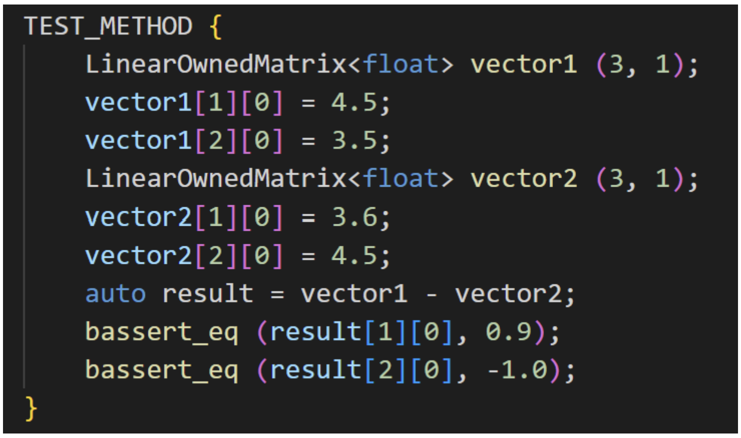
```
TEST_METHOD {
    LinearOwnedMatrix<double> vector1 (3, 1);
    vector1[1][0] = 4.5;
    vector1[2][0] = 5.5;
    LinearOwnedMatrix<double> vector2 (3, 1);
    vector2[1][0] = 3.6;
    vector2[2][0] = 4.5;
    auto result = vector1 + vector2;
    bassert_eq (result[1][0], 8.1);
    bassert_eq (result[2][0], 10.0);
}
```

Matrix and Vector Subtraction

```
            LinearOwnedMatrix operator-(LinearOwnedMatrix const &rhs) const {
                Use matrix::exception;
                auto row = m_row;
                if (rhs.row() != row) {
                    throw MatrixStructureMismatchingException("LinearOwnedMatrix
addition with the mismatched row value. ");
                } else if (size() != rhs.size()) {
                    throw MatrixStructureMismatchingException("LinearOwnedMatrix
addition with the mismatched col value. ");
                }
                return LinearOwnedMatrix ((Super&)*this - (Super&)rhs, row);
            }
```



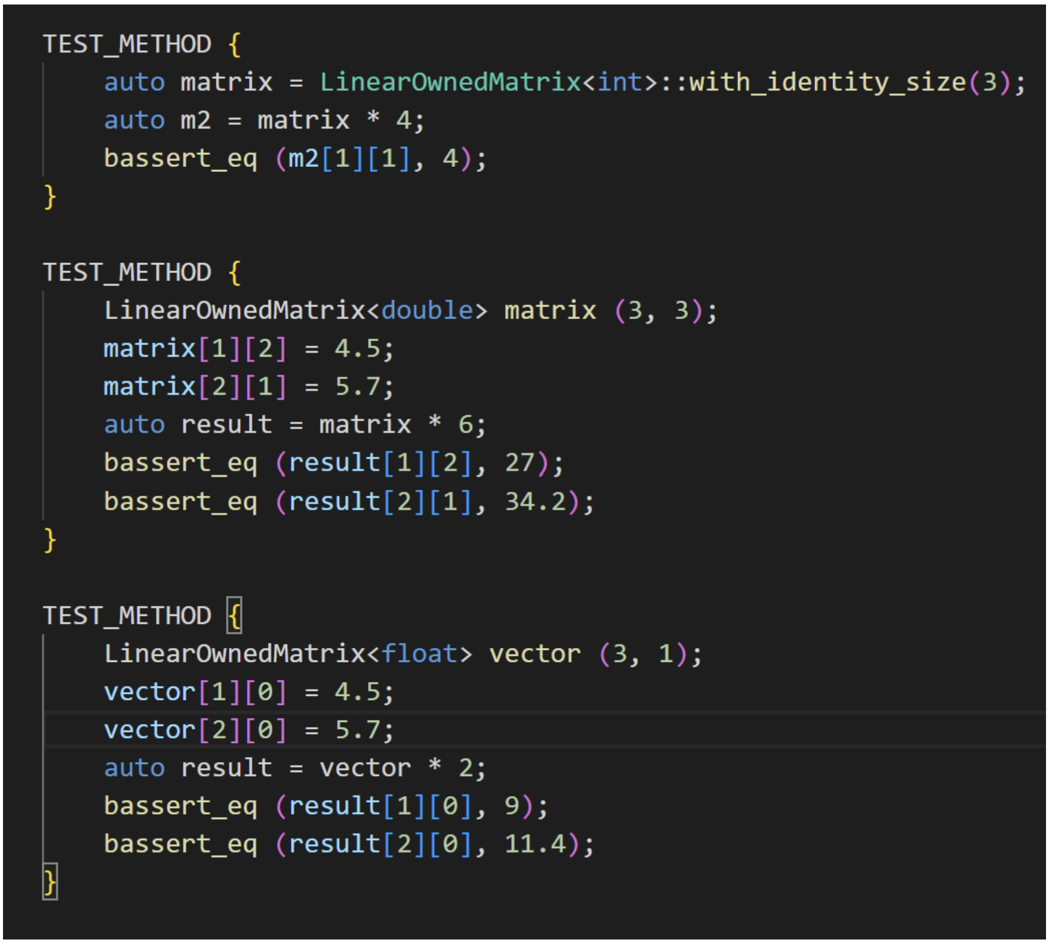
```
TEST_METHOD {
    auto matrix = LinearOwnedMatrix<int>::with_identity_size(3);
    auto m2 = matrix;
    auto plus = matrix - m2;
    bassert_eq (plus[1][1], 0);
}

TEST_METHOD {
    LinearOwnedMatrix<double> matrix1 (3, 3);
    matrix1[1][2] = 4.5;
    matrix1[2][1] = 5.5;
    LinearOwnedMatrix<double> matrix2 (3, 3);
    matrix2[1][2] = 3.6;
    matrix2[2][1] = 4.5;
    auto result = matrix1 - matrix2;
    bassert_eq (result[1][2], 0.9);
    bassert_eq (result[2][1], 1.0);
}
```



```
TEST_METHOD {
    LinearOwnedMatrix<float> vector1 (3, 1);
    vector1[1][0] = 4.5;
    vector1[2][0] = 3.5;
    LinearOwnedMatrix<float> vector2 (3, 1);
    vector2[1][0] = 3.6;
    vector2[2][0] = 4.5;
    auto result = vector1 - vector2;
    bassert_eq (result[1][0], 0.9);
    bassert_eq (result[2][0], -1.0);
}
```

Matrix and Vector Scalar Multiplication

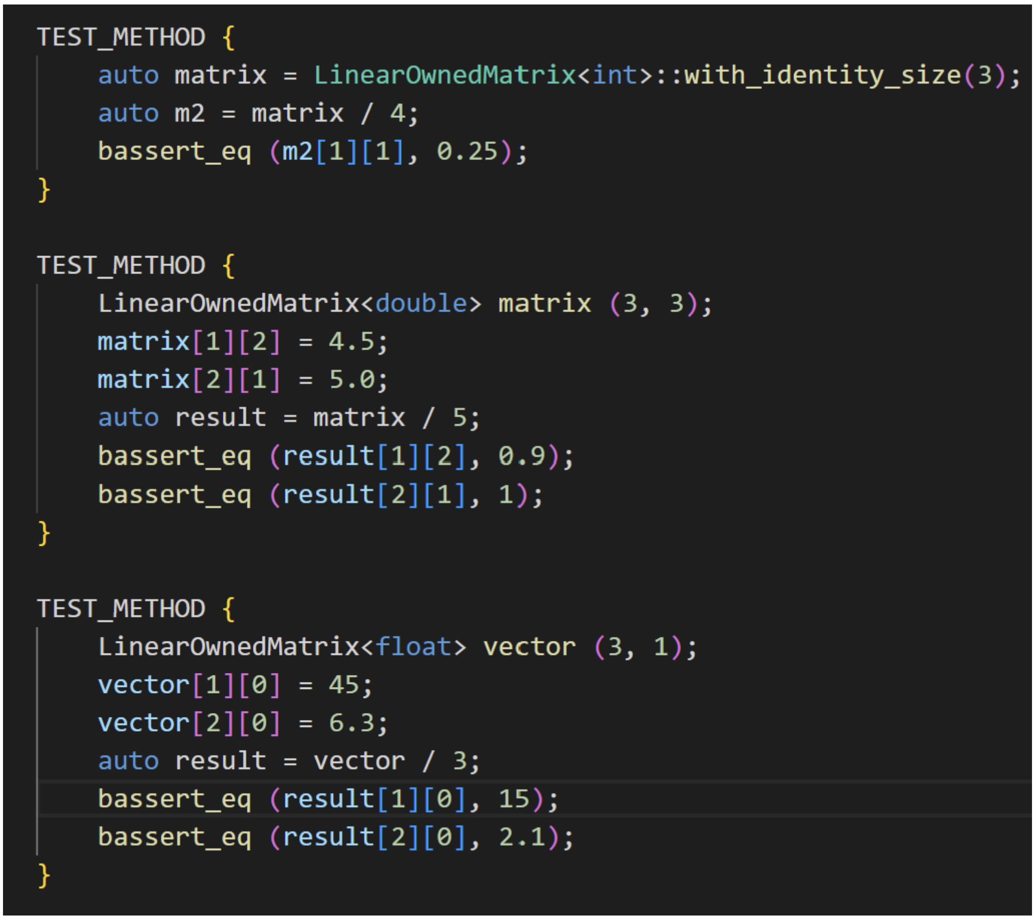
```
        LinearOwnedMatrix operator*(ValueType const &rhs) const {
            return LinearOwnedMatrix ((Super&)*this * rhs, row());
        }
```



```
TEST_METHOD {
    auto matrix = LinearOwnedMatrix<int>::with_identity_size(3);
    auto m2 = matrix * 4;
    bassert_eq (m2[1][1], 4);
}

TEST_METHOD {
    LinearOwnedMatrix<double> matrix (3, 3);
    matrix[1][2] = 4.5;
    matrix[2][1] = 5.7;
    auto result = matrix * 6;
    bassert_eq (result[1][2], 27);
    bassert_eq (result[2][1], 34.2);
}

TEST_METHOD {
    LinearOwnedMatrix<float> vector (3, 1);
    vector[1][0] = 4.5;
    vector[2][0] = 5.7;
    auto result = vector * 2;
    bassert_eq (result[1][0], 9);
    bassert_eq (result[2][0], 11.4);
}
```

Matrix and Vector Scalar Division

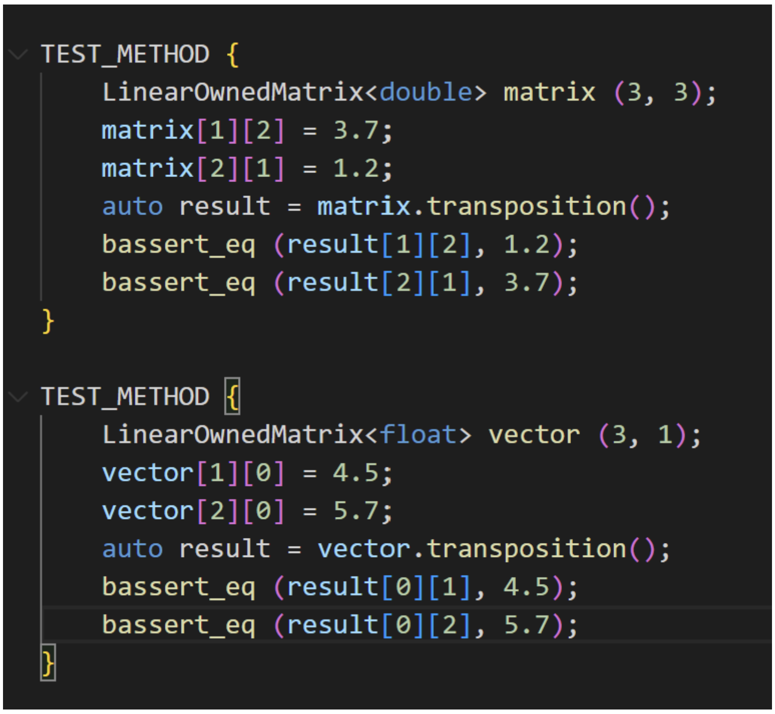
```
LinearOwnedMatrix operator/(ValueType const &rhs) const {
    return LinearOwnedMatrix ((Super&)*this / rhs, row());
}
```



```
TEST_METHOD {
    auto matrix = LinearOwnedMatrix<int>::with_identity_size(3);
    auto m2 = matrix / 4;
    bassert_eq (m2[1][1], 0.25);
}

TEST_METHOD {
    LinearOwnedMatrix<double> matrix (3, 3);
    matrix[1][2] = 4.5;
    matrix[2][1] = 5.0;
    auto result = matrix / 5;
    bassert_eq (result[1][2], 0.9);
    bassert_eq (result[2][1], 1);
}

TEST_METHOD {
    LinearOwnedMatrix<float> vector (3, 1);
    vector[1][0] = 45;
    vector[2][0] = 6.3;
    auto result = vector / 3;
    bassert_eq (result[1][0], 15);
    bassert_eq (result[2][0], 2.1);
}
```

Matrix and Vector Transposition

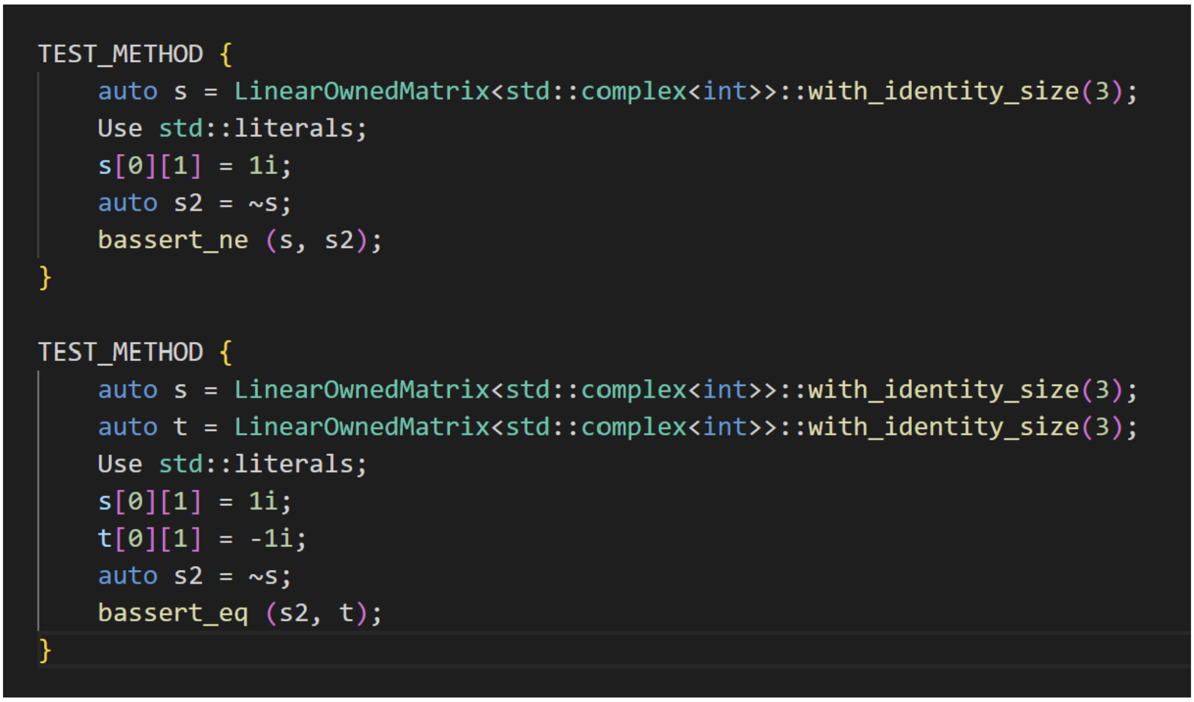
```
LinearOwnedMatrix transposition() const {
                This ans (size() / row(), row());
                for (size_t r = 0; r < size() / row(); ++r) {
                    for (size_t c = 0; c < row(); ++c) {
                        ans[r][c] = (*this)[c][r];
                    }
                }
                return ans;
            }
```



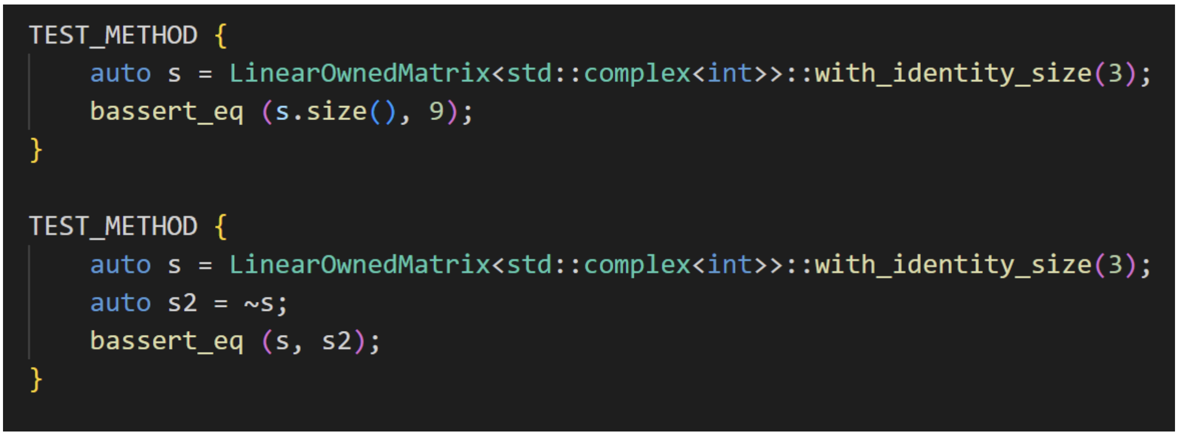
```
TEST_METHOD {
    LinearOwnedMatrix<double> matrix (3, 3);
    matrix[1][2] = 3.7;
    matrix[2][1] = 1.2;
    auto result = matrix.transposition();
    bassert_eq (result[1][2], 1.2);
    bassert_eq (result[2][1], 3.7);
}

TEST_METHOD {
    LinearOwnedMatrix<float> vector (3, 1);
    vector[1][0] = 4.5;
    vector[2][0] = 5.7;
    auto result = vector.transposition();
    bassert_eq (result[0][1], 4.5);
    bassert_eq (result[0][2], 5.7);
}
```

Matrix and Vector Conjugation

```
        This operator~ () const {
            return This(this->apply(std::conj), row());
        }
```



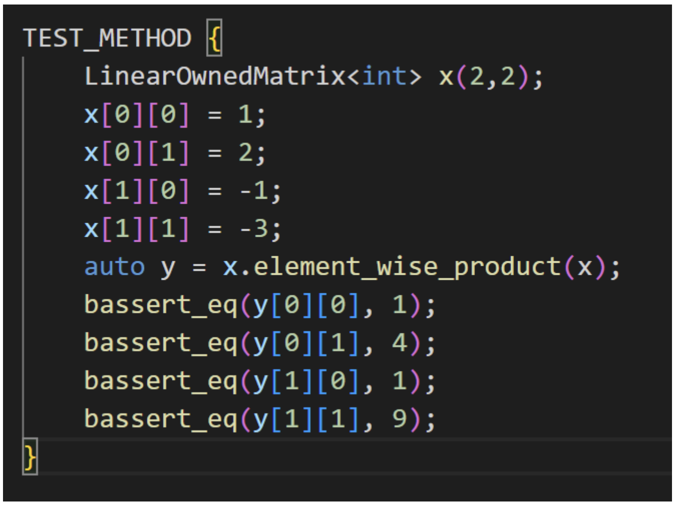
```
TEST_METHOD {
    auto s = LinearOwnedMatrix<std::complex<int>>::with_identity_size(3);
    Use std::literals;
    s[0][1] = 1i;
    auto s2 = ~s;
    bassert_ne (s, s2);
}

TEST_METHOD {
    auto s = LinearOwnedMatrix<std::complex<int>>::with_identity_size(3);
    auto t = LinearOwnedMatrix<std::complex<int>>::with_identity_size(3);
    Use std::literals;
    s[0][1] = 1i;
    t[0][1] = -1i;
    auto s2 = ~s;
    bassert_eq (s2, t);
}
```



```
TEST_METHOD {
    auto s = LinearOwnedMatrix<std::complex<int>>::with_identity_size(3);
    bassert_eq (s.size(), 9);
}

TEST_METHOD {
    auto s = LinearOwnedMatrix<std::complex<int>>::with_identity_size(3);
    auto s2 = ~s;
    bassert_eq (s, s2);
}
```

Matrix and Vector Element-Wise Multiplication

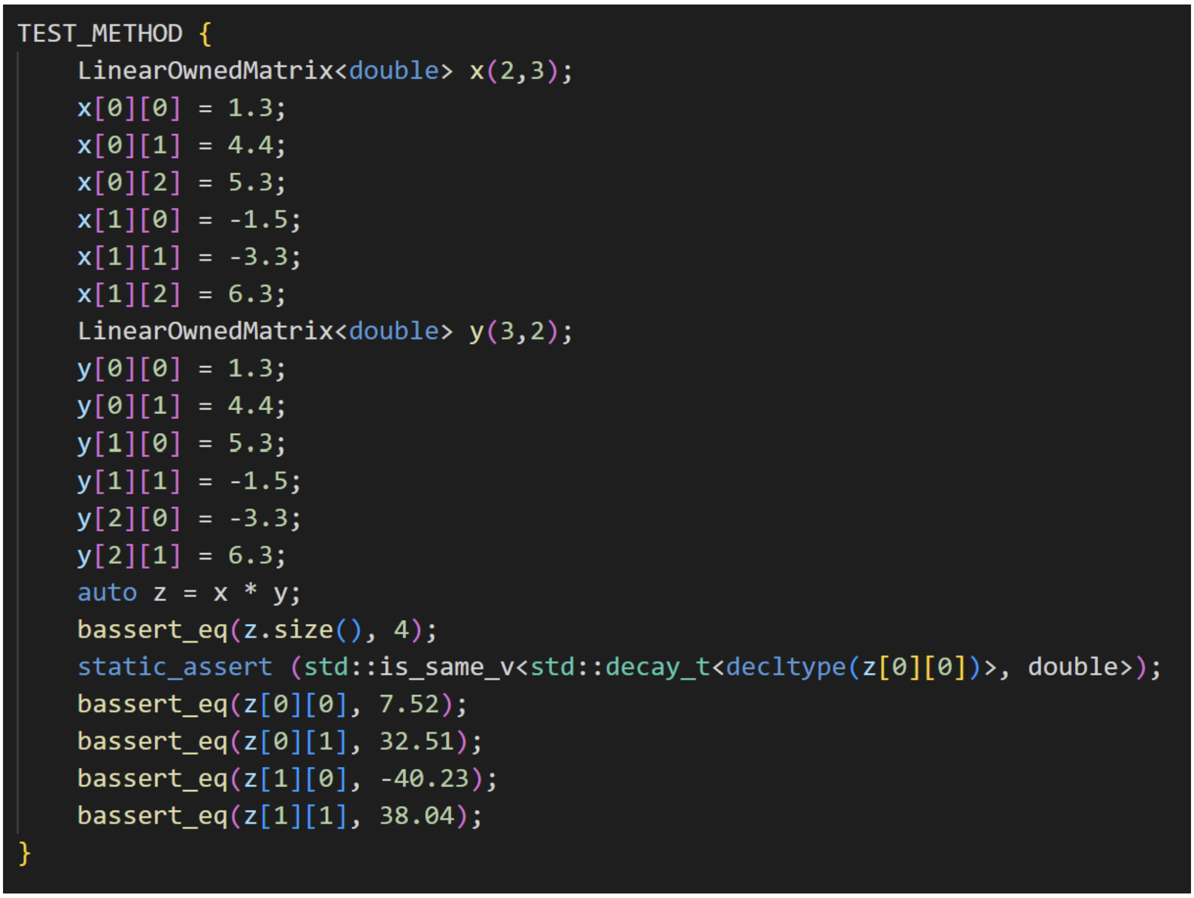
```
        LinearOwnedMatrix element_wise_product(LinearOwnedMatrix const &rhs)
const {
            Use matrix::exception;
            auto row = m_row;
            if (rhs.row() != row) {
                throw MatrixStructureMismatchingException("LinearOwnedMatrix
element_wise_product with the mismatched row value. ");
            } else if (size() != rhs.size()) {
                throw MatrixStructureMismatchingException("LinearOwnedMatrix
element_wise_product with the mismatched col value. ");
            }
            return LinearOwnedMatrix ((Super&)*this * (Super&)rhs, row);
        }
```



```
TEST_METHOD {
    LinearOwnedMatrix<int> x(2,2);
    x[0][0] = 1;
    x[0][1] = 2;
    x[1][0] = -1;
    x[1][1] = -3;
    auto y = x.element_wise_product(x);
    bassert_eq(y[0][0], 1);
    bassert_eq(y[0][1], 4);
    bassert_eq(y[1][0], 1);
    bassert_eq(y[1][1], 9);
}
```

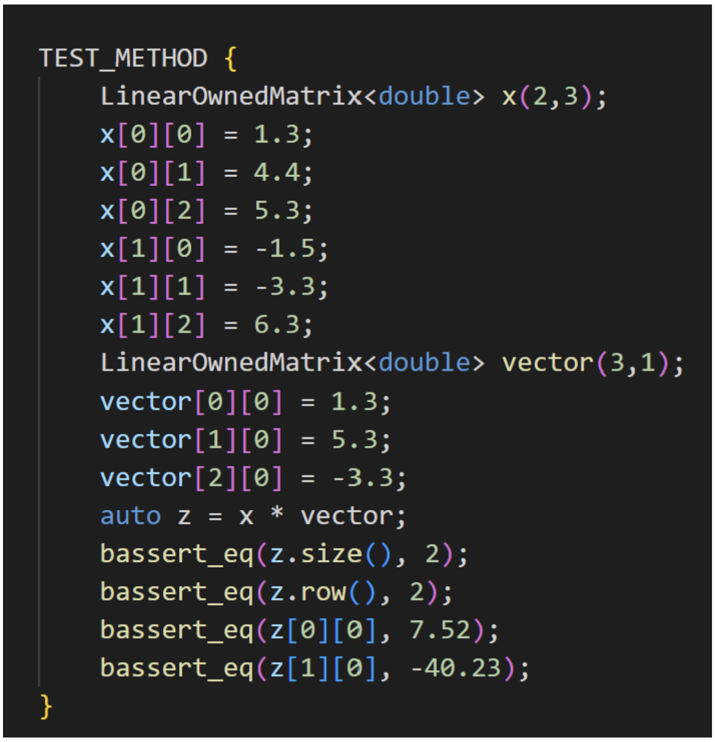
Matrix-Matrix Multiplication

```
            This operator * (This const &rhs) const {
                return cross_product(rhs);
            }
```



```
TEST_METHOD {
    LinearOwnedMatrix<double> x(2,3);
    x[0][0] = 1.3;
    x[0][1] = 4.4;
    x[0][2] = 5.3;
    x[1][0] = -1.5;
    x[1][1] = -3.3;
    x[1][2] = 6.3;
    LinearOwnedMatrix<double> y(3,2);
    y[0][0] = 1.3;
    y[0][1] = 4.4;
    y[1][0] = 5.3;
    y[1][1] = -1.5;
    y[2][0] = -3.3;
    y[2][1] = 6.3;
    auto z = x * y;
    bassert_eq(z.size(), 4);
    static_assert (std::is_same_v<std::decay_t<decltype(z[0][0])>, double>);
    bassert_eq(z[0][0], 7.52);
    bassert_eq(z[0][1], 32.51);
    bassert_eq(z[1][0], -40.23);
    bassert_eq(z[1][1], 38.04);
}
```

Matrix-Vector Multiplication

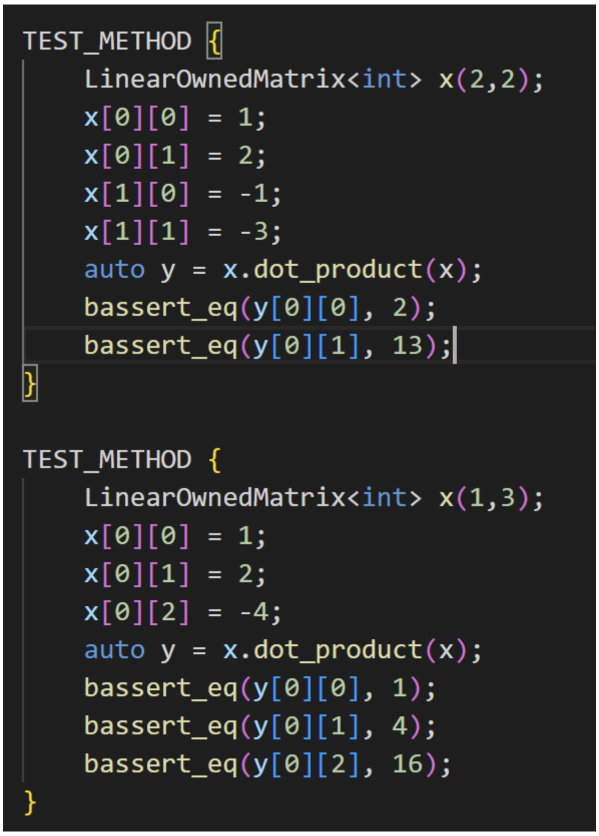


```
TEST_METHOD {
    LinearOwnedMatrix<double> x(2,3);
    x[0][0] = 1.3;
    x[0][1] = 4.4;
    x[0][2] = 5.3;
    x[1][0] = -1.5;
    x[1][1] = -3.3;
    x[1][2] = 6.3;
    LinearOwnedMatrix<double> vector(3,1);
    vector[0][0] = 1.3;
    vector[1][0] = 5.3;
    vector[2][0] = -3.3;
    auto z = x * vector;
    bassert_eq(z.size(), 2);
    bassert_eq(z.row(), 2);
    bassert_eq(z[0][0], 7.52);
    bassert_eq(z[1][0], -40.23);
}
```

Matrix and Vector Dot Product

```
         LinearOwnedMatrix dot_product(LinearOwnedMatrix const &rhs) const {
             Use matrix::exception;
             auto row = m_row;
             if (rhs.row() != row) {
                 throw MatrixStructureMismatchingException("LinearOwnedMatrix
 dot_product with the mismatched row value. ");
```

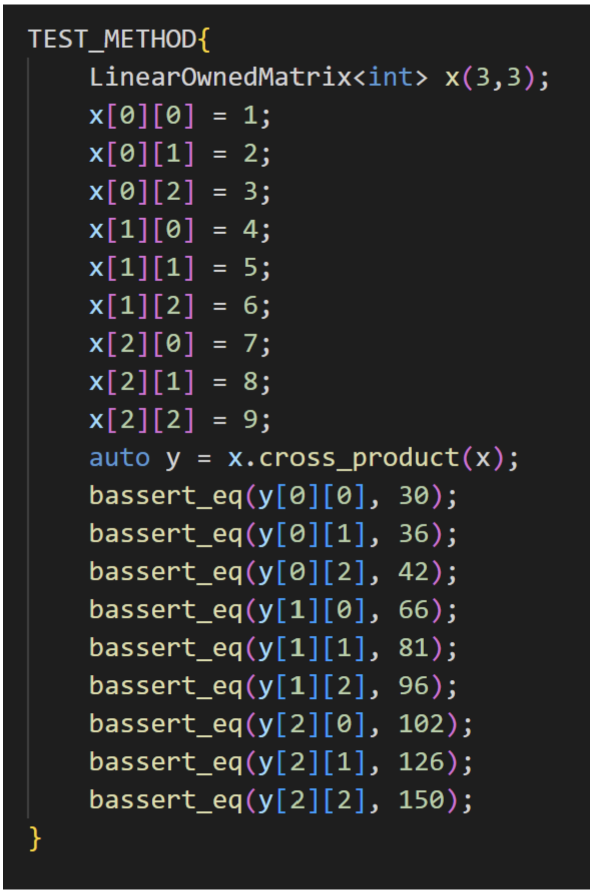
```
      } else if (size() != rhs.size()) {
        throw MatrixStructureMismatchingException("LinearOwnedMatrix
dot_product with the mismatched col value. ");
      }
      This temp = element_wise_product(rhs);
      This ans(1, size()/row);
      for (size_t r = 0; r < row; ++r){
        for (size_t c = 0; c < size()/row; ++c) {
          ans[0][c] += temp[r][c];
        }
      }
      return ans;
    }
```



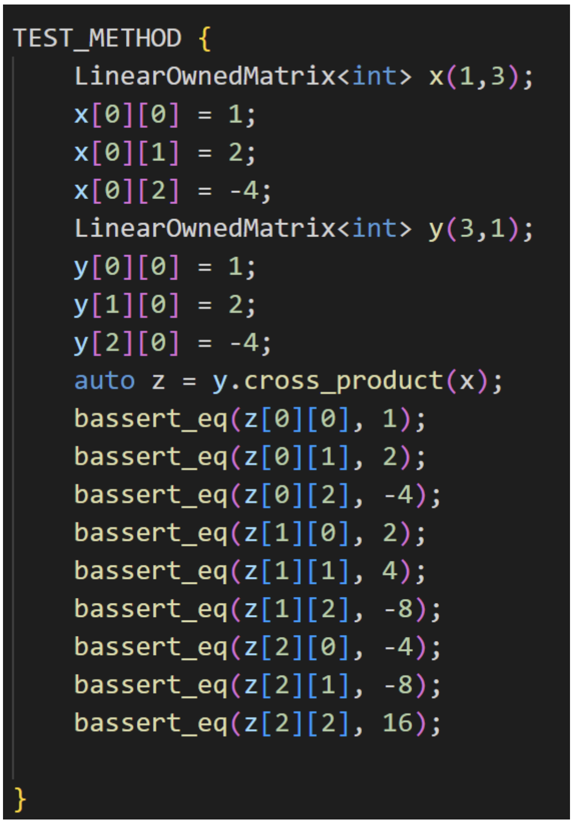
```
TEST_METHOD {
    LinearOwnedMatrix<int> x(2,2);
    x[0][0] = 1;
    x[0][1] = 2;
    x[1][0] = -1;
    x[1][1] = -3;
    auto y = x.dot_product(x);
    bassert_eq(y[0][0], 2);
    bassert_eq(y[0][1], 13);
}

TEST_METHOD {
    LinearOwnedMatrix<int> x(1,3);
    x[0][0] = 1;
    x[0][1] = 2;
    x[0][2] = -4;
    auto y = x.dot_product(x);
    bassert_eq(y[0][0], 1);
    bassert_eq(y[0][1], 4);
    bassert_eq(y[0][2], 16);
}
```

Matrix and Vector Cross Product

```
    LinearOwnedMatrix cross_product(LinearOwnedMatrix const &rhs) const {
      Use matrix::exception;
      auto row = m_row;
      if (size() / row != rhs.row()) {
        throw MatrixStructureMismatchingException("LinearOwnedMatrix
cross product with the mismatched matrix size. ");
      }
      This ans(row, rhs.size() / rhs.row());
      for (size_t r = 0; r < row; ++r){
        for (size_t c = 0; c < rhs.size() / rhs.row(); ++c) {
          for (size_t k = 0; k < size() / row; ++k){
            ans[r][c] += ((*this)[r][k] * rhs[k][c]);
          }
        }
      }
      return ans;
    }
```



```
TEST_METHOD{
    LinearOwnedMatrix<int> x(3,3);
    x[0][0] = 1;
    x[0][1] = 2;
    x[0][2] = 3;
    x[1][0] = 4;
    x[1][1] = 5;
    x[1][2] = 6;
    x[2][0] = 7;
    x[2][1] = 8;
    x[2][2] = 9;
    auto y = x.cross_product(x);
    bassert_eq(y[0][0], 30);
    bassert_eq(y[0][1], 36);
    bassert_eq(y[0][2], 42);
    bassert_eq(y[1][0], 66);
    bassert_eq(y[1][1], 81);
    bassert_eq(y[1][2], 96);
    bassert_eq(y[2][0], 102);
    bassert_eq(y[2][1], 126);
    bassert_eq(y[2][2], 150);
}
```

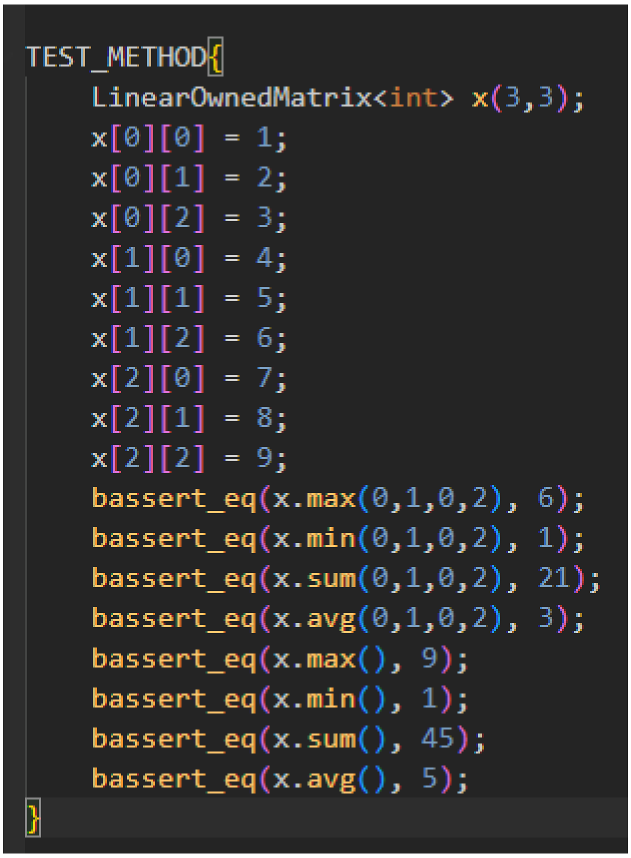


```
TEST_METHOD {
    LinearOwnedMatrix<int> x(1,3);
    x[0][0] = 1;
    x[0][1] = 2;
    x[0][2] = -4;
    LinearOwnedMatrix<int> y(3,1);
    y[0][0] = 1;
    y[1][0] = 2;
    y[2][0] = -4;
    auto z = y.cross_product(x);
    bassert_eq(z[0][0], 1);
    bassert_eq(z[0][1], 2);
    bassert_eq(z[0][2], -4);
    bassert_eq(z[1][0], 2);
    bassert_eq(z[1][1], 4);
    bassert_eq(z[1][2], -8);
    bassert_eq(z[2][0], -4);
    bassert_eq(z[2][1], -8);
    bassert_eq(z[2][2], 16);

}
```

Max Min Sum Avg

```
            ValueType max(size_t r0, size_t r1, size_t c0, size_t c1) const {
                Use matrix::exception;
                if (r1 < r0 || c1 < c0 || r0 > row() || r1 > row() || c0 > col()
|| c1 > col()){
                    throw MatrixStructureInvalidSizeException("LinearOwnedMatrix
max with invalid size");
```

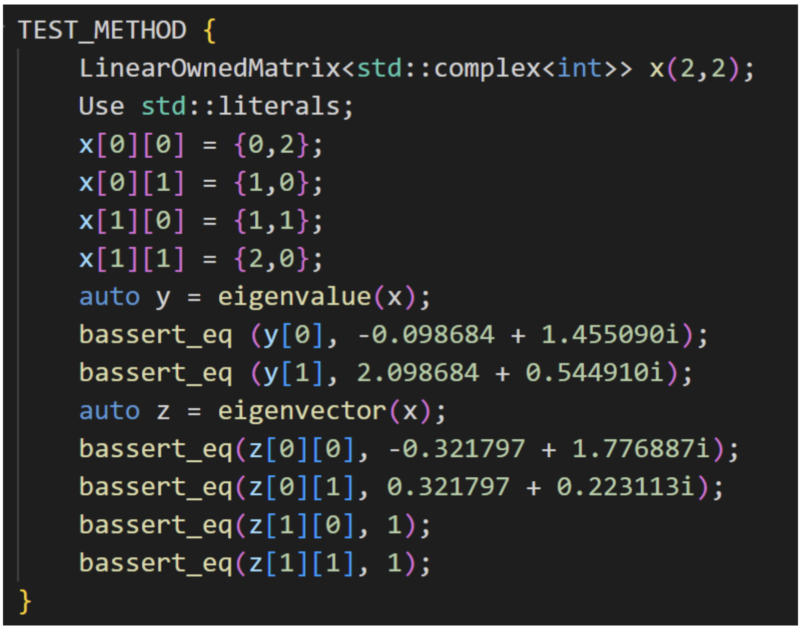
```
                }
                return slice(r0, r1, c0, c1).max();
            }

            ValueType min(size_t r0, size_t r1, size_t c0, size_t c1) const {
                Use matrix::exception;
                if (r1 < r0 || c1 < c0 || r0 > row() || r1 > row() || c0 > col() || c1 > col()){
                    throw MatrixStructureInvalidSizeException("LinearOwnedMatrix min with invalid size");
                }
                return slice(r0, r1, c0, c1).min();
            }

            ValueType sum(size_t r0, size_t r1, size_t c0, size_t c1) const {
                Use matrix::exception;
                if (r1 < r0 || c1 < c0 || r0 > row() || r1 > row() || c0 > col() || c1 > col()){
                    throw MatrixStructureInvalidSizeException("LinearOwnedMatrix sum with invalid size");
                }
                return slice(r0, r1, c0, c1).sum();
            }

            ValueType avg(size_t r0, size_t r1, size_t c0, size_t c1) const {
                Use matrix::exception;
                if (r1 < r0 || c1 < c0 || r0 > row() || r1 > row() || c0 > col() || c1 > col()){
                    throw MatrixStructureInvalidSizeException("LinearOwnedMatrix avg with invalid size");
                }
                return slice(r0, r1, c0, c1).avg();
            }

            using Super::max;
            using Super::min;
            using Super::sum;

            ValueType avg() const {
                return sum() / size();
            }
```



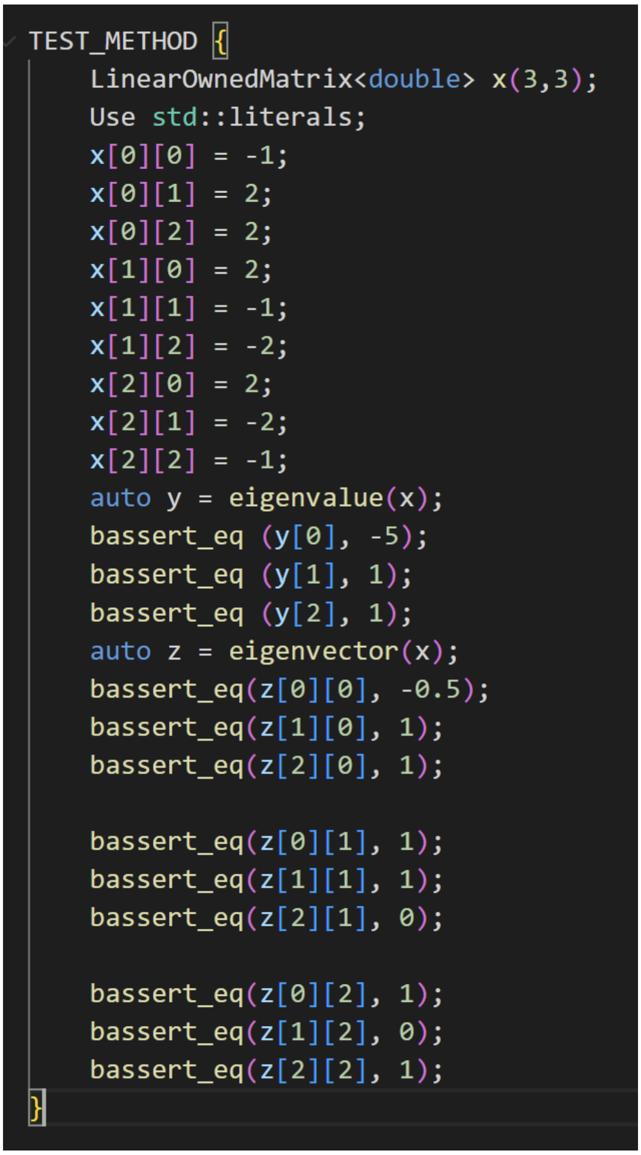
```
TEST_METHOD{
    LinearOwnedMatrix<int> x(3,3);
    x[0][0] = 1;
    x[0][1] = 2;
    x[0][2] = 3;
    x[1][0] = 4;
    x[1][1] = 5;
    x[1][2] = 6;
    x[2][0] = 7;
    x[2][1] = 8;
    x[2][2] = 9;
    bassert_eq(x.max(0,1,0,2), 6);
    bassert_eq(x.min(0,1,0,2), 1);
    bassert_eq(x.sum(0,1,0,2), 21);
    bassert_eq(x.avg(0,1,0,2), 3);
    bassert_eq(x.max(), 9);
    bassert_eq(x.min(), 1);
    bassert_eq(x.sum(), 45);
    bassert_eq(x.avg(), 5);
}
```

Eigenvalues and Eigenvectors

```
template <typename Matrix>
    auto eigenvalue(Matrix const &self) {

        constexpr int attempt_cnt = 100;

        using DataType = typename type_traits::template TypeUpgrade<typename Matrix::ValueType>::type;
        using ResultType = std::vector<DataType>;

        typename Matrix::template MatrixOfType<DataType> temp = self;
        for (int i = 0; i < attempt_cnt; ++i) {
            auto [q, r] = qr_factorization(temp);
            temp = r * q;
        }

        ResultType result;
        result.reserve(temp.row());

        for (int i = 0; i < temp.row(); ++i) {
            result.push_back(temp[i][i]);
        }

        sort(result.begin(), result.end(), helper::ComplexComp{});

        return result;
    }

    template <typename Matrix>
    auto gaussian_elimination_as_mut(Matrix &self) {
        size_t row = self.row();
        for (size_t i = 0; i < row; ++i) {
            auto col = self.col();
            if (i >= col)
                break;

            auto pivot = self[i][i];

            if (is_nearly_zero(pivot)) {
                size_t j;
                for (j = i + 1; j < row; ++j) {
                    if (!is_nearly_zero(self[j][i])) {
                        for (size_t k {}; k < col; ++k) {
                            std::swap(self[i][k], self[j][k]);
                        }
                        break;
                    }
                }
                if (j == row) {
                    return i;
                }
            }

            for (size_t j = i + 1; j < row; ++j) {
                auto divisor = self[j][i] / pivot;
                self[j][i] = {};
                for (size_t k = i + 1; k < col; ++k) {
                    self[j][k] -= self[i][k] * divisor;
                    if (algorithm::is_nearly_zero(self[j][k])) {
                        self[j][k] = 0;
                    }
                }

            }
        }

        return row;
    }

    template <typename Matrix, typename ResultDataType = typename
type_traits::template TypeUpgrade<typename Matrix::ValueType>::type>
    auto eigenvector (Matrix const &self) {
        auto len = self.row();
        if (len != self.col()) {
            throw matrix::exception::MatrixNonSquareException( __FILE__ ":"
STRING(__LINE__) " " STRING(__FUNCTION__) ": the matrix is not a square matrix.
");
        }

        typename Matrix::template MatrixOfType<ResultDataType> ans(len, len);

        size_t cnt = 0;
        auto eigenvalues = eigenvalue(self);

        auto last = eigenvalues[0];
        for (size_t i = 0; i < len; ++i) {
            lassert (i < eigenvalues.size());
            auto value = eigenvalues[i];

            if (i != 0 && type_traits::is_nearly_same(last, value)) {
                continue;
            }

            typename Matrix::template MatrixOfType<ResultDataType> temp = self;
            for (size_t j = 0; j < len; ++j) {
                temp[j][j] -= value;
            }

            gaussian_elimination_as_mut(temp);

            for (size_t j = 0; j < len; ++j) {

                if (!is_nearly_zero(temp[j][j])) {
                    auto pivot = temp[j][j];
                    for (size_t k = j; k < len; ++k) {
                        temp[j][k] /= pivot;
                    }
                } else {
                    for (size_t k = 0; k < len; ++k){
                        ans[k][cnt] = -temp[k][j];
                    }
                    ans[j][cnt] = 1;
                    cnt++;
                }

            }
            last = value;
        }
        return ans;
    }

    template <typename Matrix, typename ResultDataType = typename
type_traits::template TypeUpgrade<typename Matrix::ValueType>::type>
    auto eigenvector_depreacated (Matrix const &self) {
        auto len = self.row();
        if (len != self.col()) {
            throw matrix::exception::MatrixNonSquareException( __FILE__ ":"
STRING(__LINE__) " " STRING(__FUNCTION__) ": the matrix is not a square matrix.
");
        }

        using ResultType = std::vector<std::pair<ResultDataType,
std::vector<ResultDataType>>>;

        using SuggestedMatrix = typename Matrix::template
MatrixOfType<ResultDataType>;

        ResultType ans;
        ans.reserve(len);

        size_t cnt = 0;
        auto eigenvalues = eigenvalue(self);

        auto last = eigenvalues[0];
        for (size_t i = 0; i < len; ++i) {
            lassert (i < eigenvalues.size());
            auto value = eigenvalues[i];
            if (i != 0 && last == value)
                continue;

            Matrix temp = self;
            for (size_t j = 0; j < len; ++j) {
                temp[j][j] -= value;
            }

            gaussian_elimination_as_mut(temp);

            for (size_t j = 0; j < len; ++j) {
                if (!is_nearly_zero(temp[j][j])) {
                    auto pivot = temp[j][j];
                    for (size_t k = j; k < len; ++k) {
                        temp[j][k] /= pivot;
                    }
                } else {
                    std::vector<ResultDataType> t;
                    t.resize(len);
                    for (size_t k = 0; k < len; ++k)
                        t[k] = temp[k][j];
                    ans.push_back({value, std::move(t)});
                }
            }
        }
        return ans;
    }
```



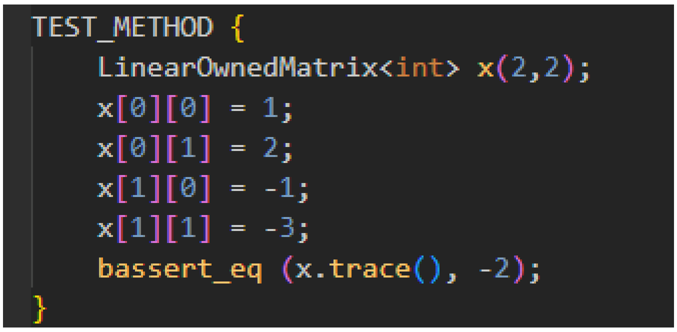
```
TEST_METHOD {
    LinearOwnedMatrix<std::complex<int>> x(2,2);
    Use std::literals;
    x[0][0] = {0,2};
    x[0][1] = {1,0};
    x[1][0] = {1,1};
    x[1][1] = {2,0};
    auto y = eigenvalue(x);
    bassert_eq (y[0], -0.098684 + 1.455090i);
    bassert_eq (y[1], 2.098684 + 0.544910i);
    auto z = eigenvector(x);
    bassert_eq(z[0][0], -0.321797 + 1.776887i);
    bassert_eq(z[0][1], 0.321797 + 0.223113i);
    bassert_eq(z[1][0], 1);
    bassert_eq(z[1][1], 1);
}
```



```
TEST_METHOD {
    LinearOwnedMatrix<double> x(3,3);
    Use std::literals;
    x[0][0] = -1;
    x[0][1] = 2;
    x[0][2] = 2;
    x[1][0] = 2;
    x[1][1] = -1;
    x[1][2] = -2;
    x[2][0] = 2;
    x[2][1] = -2;
    x[2][2] = -1;
    auto y = eigenvalue(x);
    bassert_eq (y[0], -5);
    bassert_eq (y[1], 1);
    bassert_eq (y[2], 1);
    auto z = eigenvector(x);
    bassert_eq(z[0][0], -0.5);
    bassert_eq(z[1][0], 1);
    bassert_eq(z[2][0], 1);

    bassert_eq(z[0][1], 1);
    bassert_eq(z[1][1], 1);
    bassert_eq(z[2][1], 0);

    bassert_eq(z[0][2], 1);
    bassert_eq(z[1][2], 0);
    bassert_eq(z[2][2], 1);
}
```

Traces

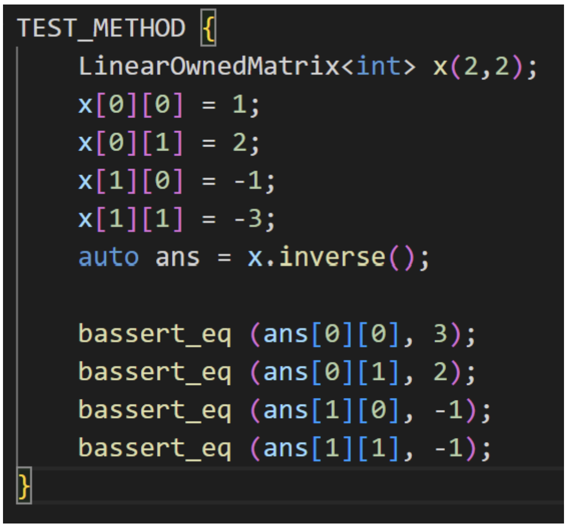
```
        ValueType trace() const {
            Use matrix::exception;
            auto row = m_row;
            if (row != (size() / row)){
                throw MatrixNonSquareException("LinearOwnedMatrix trace with non-square matrix. ");
            }
            ValueType trace{};
            for (size_t i = 0; i < row; i++){
                trace = trace + (*this)[i][i];
            }
            return trace;
        }
```



```
TEST_METHOD {
    LinearOwnedMatrix<int> x(2,2);
    x[0][0] = 1;
    x[0][1] = 2;
    x[1][0] = -1;
    x[1][1] = -3;
    bassert_eq (x.trace(), -2);
}
```

Inverse

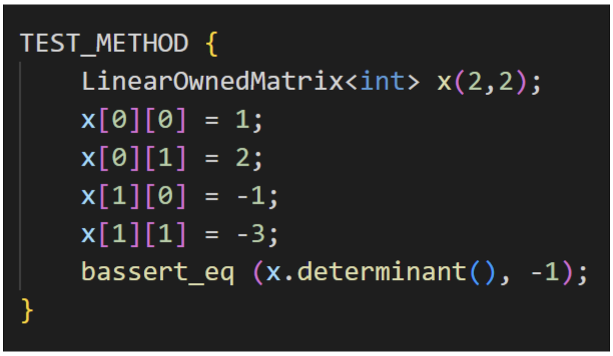
```
        LinearOwnedMatrix adjacent() const {
            Use matrix::exception;
            size_t row = m_row;
            if (size() / row != row){
                throw MatrixStructureMismatchingException("LinearOwnedMatrix
adjacent with non-square matrix. ");
            }
            return adjacent_cal(row);
        }

        LinearOwnedMatrix inverse() const {
            Use matrix::exception;
            auto row = m_row;
            if (size() / row != row){
                throw MatrixStructureMismatchingException("LinearOwnedMatrix
inverse with non-square matrix. ");
            }
            ValueType det = determinant();
            // if (is_nearly_zero(det)){
                // throw
MatrixStructureDeterminentEqualsZeroException("LinearOwnedMatrix inverse with
determinent-equals-zero matrix.");
            // }
            This ans(row, row);
            This adj = adjacent();
            for (size_t i = 0; i < row; ++i){
                for (size_t j = 0; j < row; ++j){
                    ans[i][j] = adj[i][j] / det;
                }
            }
            return ans;
        }
```



```
TEST_METHOD {
    LinearOwnedMatrix<int> x(2,2);
    x[0][0] = 1;
    x[0][1] = 2;
    x[1][0] = -1;
    x[1][1] = -3;
    auto ans = x.inverse();

    bassert_eq (ans[0][0], 3);
    bassert_eq (ans[0][1], 2);
    bassert_eq (ans[1][0], -1);
    bassert_eq (ans[1][1], -1);
}
```

Determinant

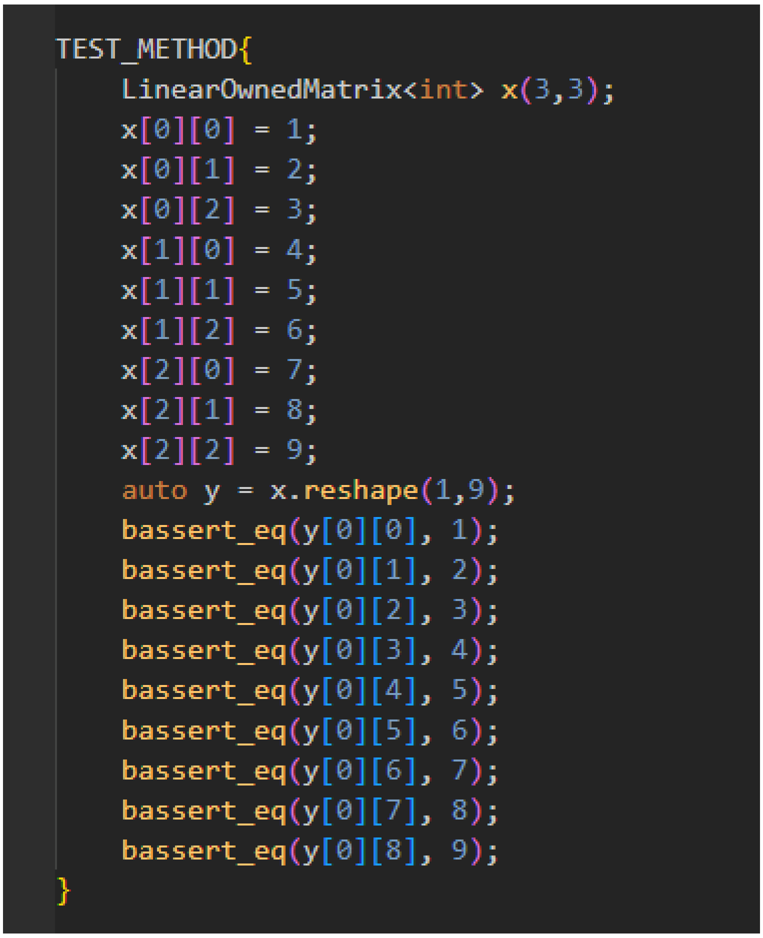
```
    ValueType determinant_calculate(LinearOwnedMatrix const &matrix, size_t
size) const {
      if (size == 1){
        return matrix[0][0];
      }
      ValueType ans{};
      for (size_t i = 0; i < size; ++i){
        ans = ans + matrix[0][i] *
determinant_calculate(determinant_cut(matrix, size - 1, i),size - 1) * (i % 2 ?
-1 : 1);
      }
      return ans;
    }

    LinearOwnedMatrix determinant_cut(LinearOwnedMatrix const &matrix,
size_t size, size_t i) const {
      This temp (size, size);
      for (size_t r = 0; r < size; ++r) {
        for (size_t c = 0; c < size; ++c) {
          temp[r][c] = matrix[r + 1][c + (c >= i)];
        }
      }
      return temp;
    }
```



```
TEST_METHOD {
    LinearOwnedMatrix<int> x(2,2);
    x[0][0] = 1;
    x[0][1] = 2;
    x[1][0] = -1;
    x[1][1] = -3;
    bassert_eq (x.determinant(), -1);
}
```

Reshape

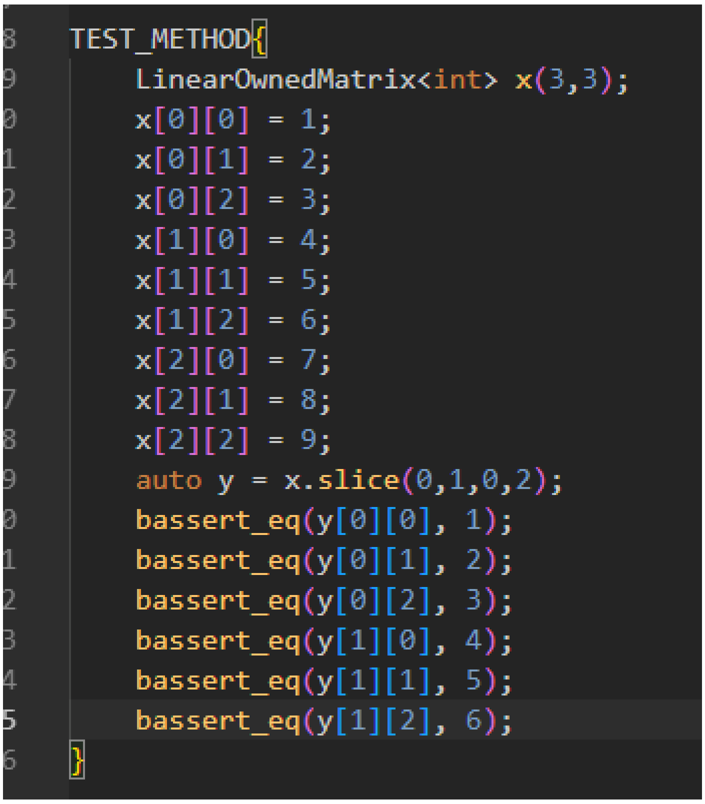
```
        LinearOwnedMatrix reshape(size_t r, size_t c) const {
            Use matrix::exception;
            if (r * c != size()){
                throw MatrixStructureMismatchingException("LinearOwnedMatrix reshape with not matching size");
            }

            Super base = *this;
            return (std::move(base), r);
        }
```



```
TEST_METHOD{
    LinearOwnedMatrix<int> x(3,3);
    x[0][0] = 1;
    x[0][1] = 2;
    x[0][2] = 3;
    x[1][0] = 4;
    x[1][1] = 5;
    x[1][2] = 6;
    x[2][0] = 7;
    x[2][1] = 8;
    x[2][2] = 9;
    auto y = x.reshape(1,9);
    bassert_eq(y[0][0], 1);
    bassert_eq(y[0][1], 2);
    bassert_eq(y[0][2], 3);
    bassert_eq(y[0][3], 4);
    bassert_eq(y[0][4], 5);
    bassert_eq(y[0][5], 6);
    bassert_eq(y[0][6], 7);
    bassert_eq(y[0][7], 8);
    bassert_eq(y[0][8], 9);
}
```

Slicing

```
        LinearOwnedMatrix slice(size_t r0, size_t r1, size_t c0, size_t c1) const {
            Use matrix::exception;
            if (r1 < r0 || c1 < c0 || r0 > row() || r1 > row() || c0 > col() || c1 > col()){
                throw MatrixStructureInvalidSizeException("LinearOwnedMatrix slice with invalid size");
            }
            This ans(r1-r0+1, c1-c0+1);
            for (size_t i = 0; i < r1-r0+1; ++i){
                for (size_t j = 0; j < c1-c0+1; ++j){
                    ans[i][j] = (*this)[r0+i][c0+j];
                }
            }
            return ans;
        }
```

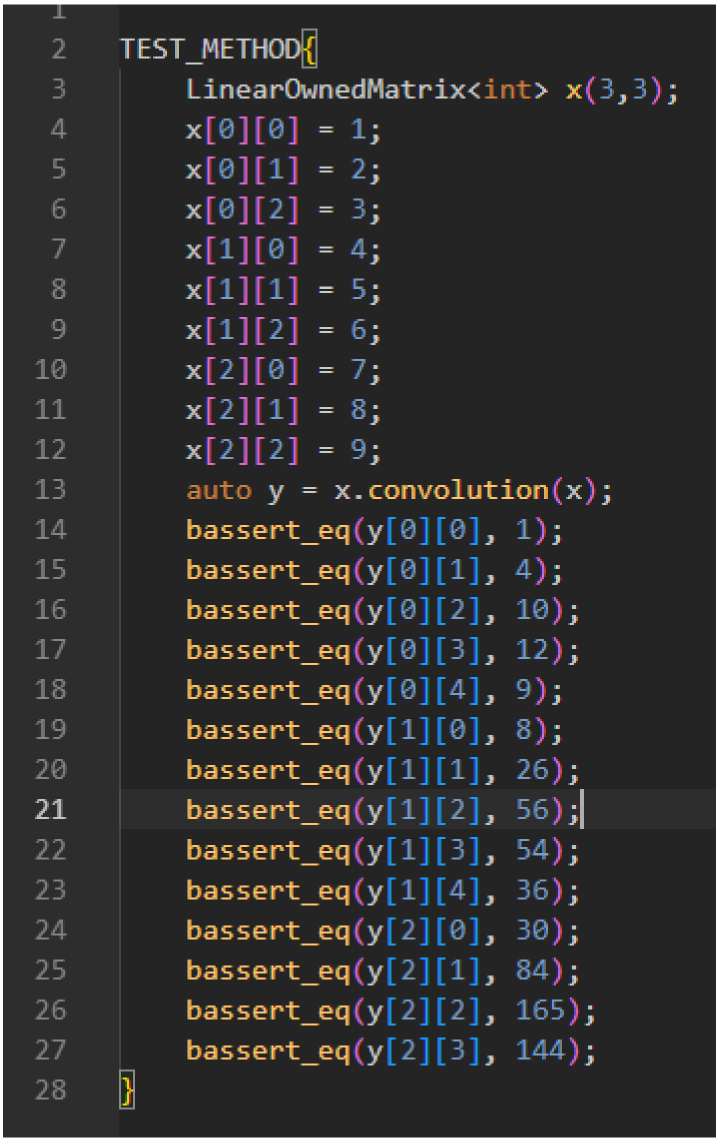


```
TEST_METHOD{
    LinearOwnedMatrix<int> x(3,3);
    x[0][0] = 1;
    x[0][1] = 2;
    x[0][2] = 3;
    x[1][0] = 4;
    x[1][1] = 5;
    x[1][2] = 6;
    x[2][0] = 7;
    x[2][1] = 8;
    x[2][2] = 9;
    auto y = x.slice(0,1,0,2);
    bassert_eq(y[0][0], 1);
    bassert_eq(y[0][1], 2);
    bassert_eq(y[0][2], 3);
    bassert_eq(y[1][0], 4);
    bassert_eq(y[1][1], 5);
    bassert_eq(y[1][2], 6);
}
```

Convolution

```
        LinearOwnedMatrix convolution(LinearOwnedMatrix & rhs) const {
            Use matrix::exception;
            if (row() < rhs.row() || col() < rhs.col()){
                throw MatrixStructureMismatchingException("LinearOwnedMatrix convolution with bigger core");
            }
            auto ans_row = row() + rhs.row() - 1, ans_col = col() + rhs.col() - 1;
            This ans(ans_row, ans_col);
            for (size_t i = 0; i < ans_row; ++i) {
                for (size_t j = 0; j < ans_col; ++j) {
                    for (size_t m = 0; m < rhs.row(); ++m) {
                        for (size_t n = 0; n < rhs.col(); ++n) {
                            if (i - m < row() && j - n < col())
```

```
                                ans[i][j] += (*this)[i - m][j - n] * rhs[m][n];
                        }
                    }
                }
            }
            return ans;
        }
```

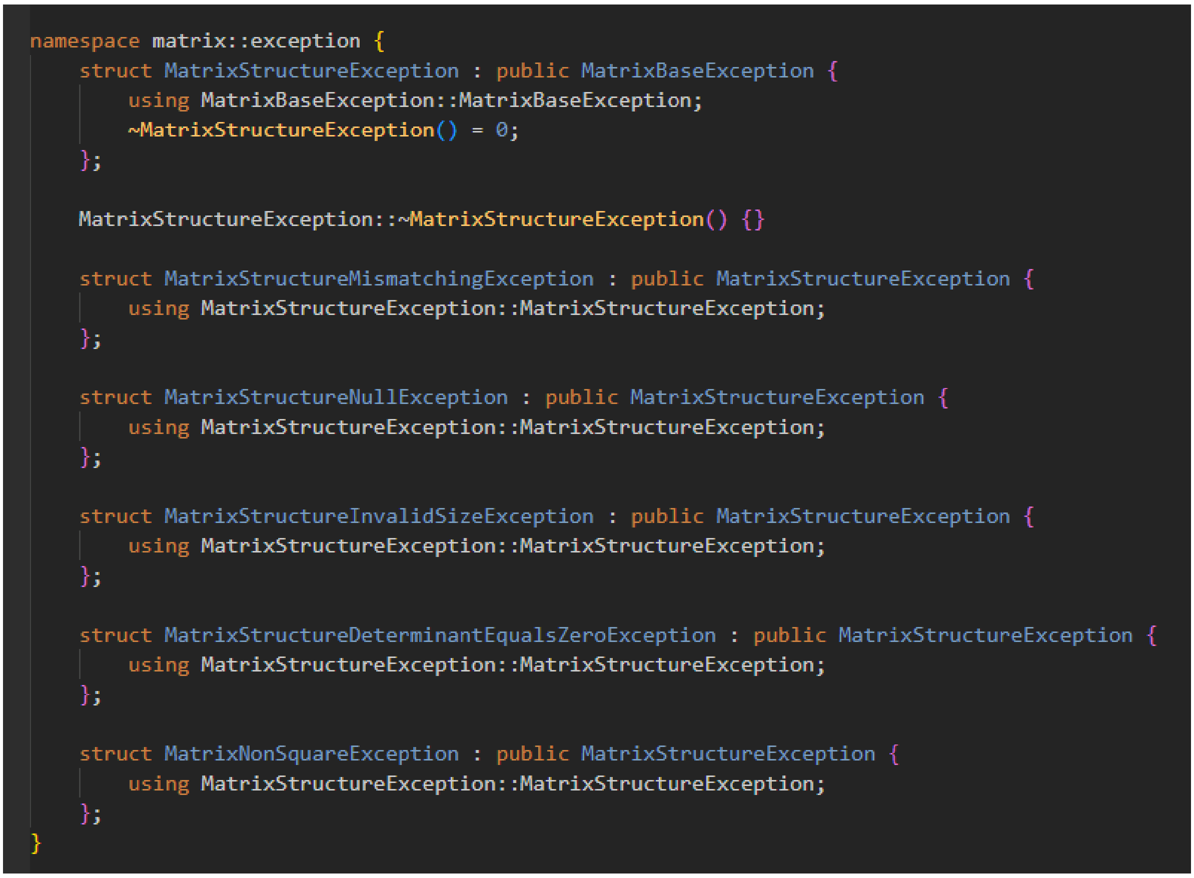


```
TEST_METHOD{
    LinearOwnedMatrix<int> x(3,3);
    x[0][0] = 1;
    x[0][1] = 2;
    x[0][2] = 3;
    x[1][0] = 4;
    x[1][1] = 5;
    x[1][2] = 6;
    x[2][0] = 7;
    x[2][1] = 8;
    x[2][2] = 9;
    auto y = x.convolution(x);
    bassert_eq(y[0][0], 1);
    bassert_eq(y[0][1], 4);
    bassert_eq(y[0][2], 10);
    bassert_eq(y[0][3], 12);
    bassert_eq(y[0][4], 9);
    bassert_eq(y[1][0], 8);
    bassert_eq(y[1][1], 26);
    bassert_eq(y[1][2], 56);
    bassert_eq(y[1][3], 54);
    bassert_eq(y[1][4], 36);
    bassert_eq(y[2][0], 30);
    bassert_eq(y[2][1], 84);
    bassert_eq(y[2][2], 165);
    bassert_eq(y[2][3], 144);
}
```

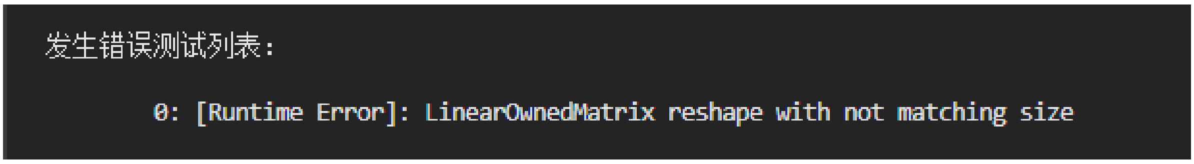
OpenCV Transfer

```
            template <typename InnerType = ValueType, typename CV_Mat>
            static This from_cv_mat(CV_Mat const &self) {
                This ans (self.rows, self.cols);
                for (size_t i = 0; i < self.rows; ++i) {
                    for (size_t j = 0; j < self.cols; ++j) {
                        ans[i][j] = self.template at<InnerType>(i, j);
                    }
                }
            }
```

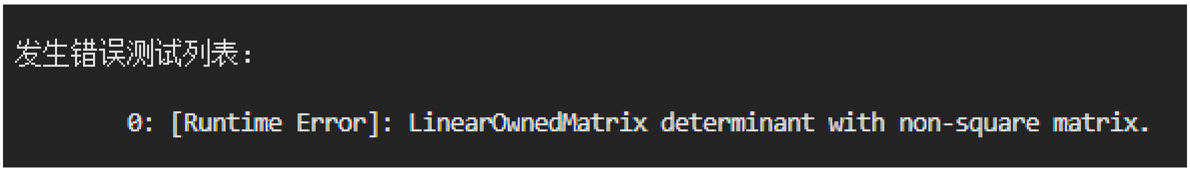
Exceptions



```cpp
namespace matrix::exception {
    struct MatrixStructureException : public MatrixBaseException {
        using MatrixBaseException::MatrixBaseException;
        ~MatrixStructureException() = 0;
    };

    MatrixStructureException::~MatrixStructureException() {}

    struct MatrixStructureMismatchingException : public MatrixStructureException {
        using MatrixStructureException::MatrixStructureException;
    };

    struct MatrixStructureNullException : public MatrixStructureException {
        using MatrixStructureException::MatrixStructureException;
    };

    struct MatrixStructureInvalidSizeException : public MatrixStructureException {
        using MatrixStructureException::MatrixStructureException;
    };

    struct MatrixStructureDeterminantEqualsZeroException : public MatrixStructureException {
        using MatrixStructureException::MatrixStructureException;
    };

    struct MatrixNonSquareException : public MatrixStructureException {
        using MatrixStructureException::MatrixStructureException;
    };
}
```



```
发生错误测试列表:

        0: [Runtime Error]: LinearOwnedMatrix reshape with not matching size
```



```
发生错误测试列表:

        0: [Runtime Error]: LinearOwnedMatrix determinant with non-square matrix.
```