

## Matrix HPP

Generated by Doxygen 1.9.8



|  |          |
|--|----------|
| <b>1 Matrix HPP - C++11 library for matrix class container and linear algebra computations</b> | <b>1</b> |
| 1.1 Installation   | 1        |
| 1.2 Functionality  | 1        |
| 1.3 Hello world example  | 2        |
| 1.4 Tests  | 2        |
| 1.5 License  | 2        |
| <b>2 Hierarchical Index</b>  | <b>3</b> |
| 2.1 Class Hierarchy  | 3        |
| <b>3 Class Index</b>   | <b>5</b> |
| 3.1 Class List   | 5        |
| <b>4 File Index</b>  | <b>7</b> |
| 4.1 File List  | 7        |
| <b>5 Class Documentation</b>   | <b>9</b> |
| 5.1 Mtx::Eigenvalues_result< T > Struct Template Reference                                     | 9        |
| 5.1.1 Detailed Description   | 9        |
| 5.2 Mtx::Hessenberg_result< T > Struct Template Reference                                      | 9        |
| 5.2.1 Detailed Description   | 10       |
| 5.3 Mtx::LDL_result< T > Struct Template Reference   | 10       |
| 5.3.1 Detailed Description   | 10       |
| 5.4 Mtx::LU_result< T > Struct Template Reference  | 11       |
| 5.4.1 Detailed Description   | 11       |
| 5.5 Mtx::LUP_result< T > Struct Template Reference   | 11       |
| 5.5.1 Detailed Description   | 12       |
| 5.6 Mtx::Matrix< T > Class Template Reference  | 12       |
| 5.6.1 Detailed Description   | 14       |
| 5.6.2 Constructor & Destructor Documentation   | 14       |
| 5.6.2.1 Matrix() [1/8]   | 14       |
| 5.6.2.2 Matrix() [2/8]   | 15       |
| 5.6.2.3 Matrix() [3/8]   | 15       |
| 5.6.2.4 Matrix() [4/8]   | 15       |
| 5.6.2.5 Matrix() [5/8]   | 15       |
| 5.6.2.6 Matrix() [6/8]   | 16       |
| 5.6.2.7 Matrix() [7/8]   | 17       |
| 5.6.2.8 Matrix() [8/8]   | 17       |
| 5.6.2.9 ~Matrix()  | 17       |
| 5.6.3 Member Function Documentation  | 17       |
| 5.6.3.1 add() [1/2]  | 17       |
| 5.6.3.2 add() [2/2]  | 18       |
| 5.6.3.3 add_col_to_another()   | 18       |
| 5.6.3.4 add_row_to_another()   | 18       |

|  |    |
|--|----|
| 5.6.3.5 clear()                                    | 19 |
| 5.6.3.6 col_from_vector()                          | 19 |
| 5.6.3.7 col_to_vector()                            | 19 |
| 5.6.3.8 cols()                                     | 20 |
| 5.6.3.9 ctranspose()                               | 20 |
| 5.6.3.10 div()                                     | 20 |
| 5.6.3.11 exists()                                  | 21 |
| 5.6.3.12 fill()                                    | 21 |
| 5.6.3.13 fill_col()                                | 21 |
| 5.6.3.14 fill_row()                                | 21 |
| 5.6.3.15 get_submatrix()                           | 22 |
| 5.6.3.16 isempty()                                 | 22 |
| 5.6.3.17 isequal() [1/2]                           | 22 |
| 5.6.3.18 isequal() [2/2]                           | 23 |
| 5.6.3.19 mult()                                    | 23 |
| 5.6.3.20 mult_col_by_another()                     | 23 |
| 5.6.3.21 mult_hadamard()                           | 23 |
| 5.6.3.22 mult_row_by_another()                     | 24 |
| 5.6.3.23 numel()                                   | 24 |
| 5.6.3.24 operator std::vector< T >()               | 24 |
| 5.6.3.25 operator>() [1/2]                         | 25 |
| 5.6.3.26 operator>() [2/2]                         | 25 |
| 5.6.3.27 operator=() [1/2]                         | 25 |
| 5.6.3.28 operator=() [2/2]                         | 25 |
| 5.6.3.29 ptr() [1/2]                               | 26 |
| 5.6.3.30 ptr() [2/2]                               | 26 |
| 5.6.3.31 reshape()                                 | 26 |
| 5.6.3.32 resize()                                  | 26 |
| 5.6.3.33 row_from_vector()                         | 27 |
| 5.6.3.34 row_to_vector()                           | 27 |
| 5.6.3.35 rows()                                    | 27 |
| 5.6.3.36 set_submatrix()                           | 28 |
| 5.6.3.37 subtract() [1/2]                          | 28 |
| 5.6.3.38 subtract() [2/2]                          | 29 |
| 5.6.3.39 swap_cols()                               | 29 |
| 5.6.3.40 swap_rows()                               | 29 |
| 5.6.3.41 transpose()                               | 30 |
| 5.7 Mtx::QR_result< T > Struct Template Reference  | 30 |
| 5.7.1 Detailed Description                         | 30 |
| 5.8 Mtx::singular_matrix_exception Class Reference | 30 |

|                                   |    |
|-----------------------------------|----|
| 6.1 matrix.hpp File Reference     | 31 |
| 6.1.1 Function Documentation      | 37 |
| 6.1.1.1 add() [1/2]               | 37 |
| 6.1.1.2 add() [2/2]               | 38 |
| 6.1.1.3 adj()                     | 38 |
| 6.1.1.4 cconj()                   | 38 |
| 6.1.1.5 chol()                    | 39 |
| 6.1.1.6 cholinv()                 | 39 |
| 6.1.1.7 circshift()               | 40 |
| 6.1.1.8 circulant() [1/2]         | 40 |
| 6.1.1.9 circulant() [2/2]         | 41 |
| 6.1.1.10 cofactor()               | 41 |
| 6.1.1.11 concatenate_horizontal() | 42 |
| 6.1.1.12 concatenate_vertical()   | 42 |
| 6.1.1.13 cond()                   | 42 |
| 6.1.1.14 csign()                  | 43 |
| 6.1.1.15 ctranspose()             | 43 |
| 6.1.1.16 det()                    | 43 |
| 6.1.1.17 det_lu()                 | 44 |
| 6.1.1.18 diag() [1/3]             | 44 |
| 6.1.1.19 diag() [2/3]             | 45 |
| 6.1.1.20 diag() [3/3]             | 45 |
| 6.1.1.21 div()                    | 46 |
| 6.1.1.22 eigenvalues() [1/2]      | 46 |
| 6.1.1.23 eigenvalues() [2/2]      | 46 |
| 6.1.1.24 eye()                    | 47 |
| 6.1.1.25 foreach_elem()           | 47 |
| 6.1.1.26 foreach_elem_copy()      | 48 |
| 6.1.1.27 hessenberg()             | 48 |
| 6.1.1.28 householder_reflection() | 49 |
| 6.1.1.29 imag()                   | 49 |
| 6.1.1.30 inv()                    | 50 |
| 6.1.1.31 inv_gauss_jordan()       | 50 |
| 6.1.1.32 inv_posdef()             | 50 |
| 6.1.1.33 inv_square()             | 51 |
| 6.1.1.34 inv_tril()               | 51 |
| 6.1.1.35 inv_triu()               | 52 |
| 6.1.1.36 ishess()                 | 52 |
| 6.1.1.37 istril()                 | 53 |
| 6.1.1.38 istriu()                 | 53 |
| 6.1.1.39 kron()                   | 53 |
| 6.1.1.40 ldl()                    | 53 |

|                                  |    |
|----------------------------------|----|
| 6.1.1.41 lu()                    | 54 |
| 6.1.1.42 lup()                   | 54 |
| 6.1.1.43 make_complex() [1/2]    | 55 |
| 6.1.1.44 make_complex() [2/2]    | 55 |
| 6.1.1.45 mult() [1/4]            | 56 |
| 6.1.1.46 mult() [2/4]            | 57 |
| 6.1.1.47 mult() [3/4]            | 57 |
| 6.1.1.48 mult() [4/4]            | 58 |
| 6.1.1.49 mult_hadamard()         | 59 |
| 6.1.1.50 norm_fro() [1/2]        | 60 |
| 6.1.1.51 norm_fro() [2/2]        | 60 |
| 6.1.1.52 ones() [1/2]            | 60 |
| 6.1.1.53 ones() [2/2]            | 61 |
| 6.1.1.54 operator!==( )          | 61 |
| 6.1.1.55 operator*( ) [1/5]      | 61 |
| 6.1.1.56 operator*( ) [2/5]      | 62 |
| 6.1.1.57 operator*( ) [3/5]      | 62 |
| 6.1.1.58 operator*( ) [4/5]      | 62 |
| 6.1.1.59 operator*( ) [5/5]      | 62 |
| 6.1.1.60 operator*==( ) [1/2]    | 63 |
| 6.1.1.61 operator*==( ) [2/2]    | 63 |
| 6.1.1.62 operator+( ) [1/3]      | 63 |
| 6.1.1.63 operator+( ) [2/3]      | 63 |
| 6.1.1.64 operator+( ) [3/3]      | 64 |
| 6.1.1.65 operator+==( ) [1/2]    | 64 |
| 6.1.1.66 operator+==( ) [2/2]    | 64 |
| 6.1.1.67 operator-( ) [1/2]      | 64 |
| 6.1.1.68 operator-( ) [2/2]      | 65 |
| 6.1.1.69 operator-==( ) [1/2]    | 65 |
| 6.1.1.70 operator-==( ) [2/2]    | 65 |
| 6.1.1.71 operator/( )            | 65 |
| 6.1.1.72 operator/==( )          | 66 |
| 6.1.1.73 operator<<( )           | 66 |
| 6.1.1.74 operator==( )           | 66 |
| 6.1.1.75 operator^( )            | 66 |
| 6.1.1.76 operator^==( )          | 67 |
| 6.1.1.77 permute_cols()          | 67 |
| 6.1.1.78 permute_rows()          | 67 |
| 6.1.1.79 permute_rows_and_cols() | 68 |
| 6.1.1.80 pinv()                  | 69 |
| 6.1.1.81 qr()                    | 69 |
| 6.1.1.82 qr_householder()        | 70 |

---

|   |    |
|---|----|
| 6.1.1.83 <code>qr_red_gs()</code> . . . . .       | 70 |
| 6.1.1.84 <code>real()</code> . . . . .            | 71 |
| 6.1.1.85 <code>repmat()</code> . . . . .          | 71 |
| 6.1.1.86 <code>solve_posdef()</code> . . . . .    | 72 |
| 6.1.1.87 <code>solve_square()</code> . . . . .    | 72 |
| 6.1.1.88 <code>solve_tril()</code> . . . . .      | 73 |
| 6.1.1.89 <code>solve_triu()</code> . . . . .      | 74 |
| 6.1.1.90 <code>subtract()</code> [1/2] . . . . .  | 74 |
| 6.1.1.91 <code>subtract()</code> [2/2] . . . . .  | 75 |
| 6.1.1.92 <code>trace()</code> . . . . .           | 75 |
| 6.1.1.93 <code>transpose()</code> . . . . .       | 76 |
| 6.1.1.94 <code>tril()</code> . . . . .            | 76 |
| 6.1.1.95 <code>triu()</code> . . . . .            | 76 |
| 6.1.1.96 <code>wilkinson_shift()</code> . . . . . | 76 |
| 6.1.1.97 <code>zeros()</code> [1/2] . . . . .     | 77 |
| 6.1.1.98 <code>zeros()</code> [2/2] . . . . .     | 77 |
| 6.2 <code>matrix.hpp</code> . . . . .             | 78 |





## Chapter 1

# Matrix HPP - C++11 library for matrix class container and linear algebra computations

This library provides a self-contained and easy to use implementation of matrix container class. The main features include:

- Full template parameterization with support for both real and complex data-types.
- Lightweight and self-contained - single header, no dependencies outside of C++ standard library.
- C++11 based.
- Operator overloading for matrix operations like multiplication and addition.
- Support the basic linear algebra operations, including matrix inversion, factorization and linear equation solving.

### 1.1 Installation

Copy the `matrix.hpp` file into the include directory of your project.

### 1.2 Functionality

This library provides the following functionality (but is not limited to):

- Elementary operations: transposition, addition, subtraction, multiplication and element-wise product.
- Matrix determinant.
- Matrix inverse.
- Frobenius norm.
- LU decomposition.
- Cholesky decomposition.
- LDL decomposition.

- Eigenvalue decomposition.
- Hessenberg decomposition.
- QR decomposition.
- Linear equation solving.

For further details please refer to the documentation: [docs/matrix\\_hpp.pdf](#). The documentation is auto generated directly from the source code by Doxygen.

## 1.3 Hello world example

A simple hello world example is provided below. The program defines two matrices with two rows and three columns each, and initializes their content with constant values. Then, the matrices are added together and the resulting matrix is printed to `stdout`.

Note that the `Matrix` class is a template class defined within the `Mtx` namespace. The template parameter specifies the numeric type to represent elements of the matrix container.

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

void main() {
    Mtx::Matrix<double> A({ 1, 2, 3,
                           4, 5, 6}, 2, 3);

    Mtx::Matrix<double> B({ 7, 8, 9,
                           10,11,12}, 2, 3);

    auto C = A + B;

    std::cout << "A + B = [" << C << "];" << std::endl;
}
```

For more examples, refer to [examples/examples.cpp](#) file. Remark that not all features of the library are used in the provided examples.

## 1.4 Tests

Unit tests are compiled with `make tests`.

## 1.5 License

MIT license is used for this project. Please refer to [LICENSE](LICENSE) for details.

## Chapter 2

# Hierarchical Index

### 2.1 Class Hierarchy

This inheritance list is sorted roughly, but not completely, alphabetically:

|  |    |
|--|----|
| std::domain_error                        |    |
| Mtx::singular_matrix_exception . . . . . | 30 |
| Mtx::Eigenvalues_result< T > . . . . .   | 9  |
| Mtx::Hessenberg_result< T > . . . . .    | 9  |
| Mtx::LDL_result< T > . . . . .           | 10 |
| Mtx::LU_result< T > . . . . .            | 11 |
| Mtx::LUP_result< T > . . . . .           | 11 |
| Mtx::Matrix< T > . . . . .               | 12 |
| Mtx::QR_result< T > . . . . .            | 30 |



## Chapter 3

# Class Index

### 3.1 Class List

Here are the classes, structs, unions and interfaces with brief descriptions:

|  |    |
|--|----|
| <a href="#">Mtx::Eigenvalues_result&lt; T &gt;</a> |    |
| Result of eigenvalues . . . . .                    | 9  |
| <a href="#">Mtx::Hessenberg_result&lt; T &gt;</a>  |    |
| Result of Hessenberg decomposition . . . . .       | 9  |
| <a href="#">Mtx::LDL_result&lt; T &gt;</a>         |    |
| Result of LDL decomposition . . . . .              | 10 |
| <a href="#">Mtx::LU_result&lt; T &gt;</a>          |    |
| Result of LU decomposition . . . . .               | 11 |
| <a href="#">Mtx::LUP_result&lt; T &gt;</a>         |    |
| Result of LU decomposition with pivoting . . . . . | 11 |
| <a href="#">Mtx::Matrix&lt; T &gt;</a> . . . . .   | 12 |
| <a href="#">Mtx::QR_result&lt; T &gt;</a>          |    |
| Result of QR decomposition . . . . .               | 30 |
| <a href="#">Mtx::singular_matrix_exception</a>     |    |
| Singular matrix exception . . . . .                | 30 |



## Chapter 4

# File Index

### 4.1 File List

Here is a list of all documented files with brief descriptions:

|                                      |    |
|--------------------------------------|----|
| <a href="#">matrix.hpp</a> . . . . . | 31 |
|--------------------------------------|----|





## Chapter 5

# Class Documentation

### 5.1 Mtx::Eigenvalues\_result< T > Struct Template Reference

Result of eigenvalues.

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

#### Public Attributes

- `std::vector< std::complex< T > > eig`  
*Vector of eigenvalues.*
- `bool converged`  
*Indicates if the eigenvalue algorithm has converged to assumed precision.*
- `T err`  
*Error of eigenvalue calculation after the last iteration.*

#### 5.1.1 Detailed Description

```
template<typename T>  
struct Mtx::Eigenvalues_result< T >
```

Result of eigenvalues.

This structure stores the result of matrix eigenvalue calculation, returned by [Mtx::eigenvalues\(\)](#) function.

The documentation for this struct was generated from the following file:

- [matrix.hpp](#)

### 5.2 Mtx::Hessenberg\_result< T > Struct Template Reference

Result of Hessenberg decomposition.

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

## Public Attributes

- [Matrix< T > H](#)  
*Matrix with upper Hessenberg form.*
- [Matrix< T > Q](#)  
*Orthogonal matrix.*

### 5.2.1 Detailed Description

```
template<typename T>
struct Mtx::Hessenberg_result< T >
```

Result of Hessenberg decomposition.

This structure stores the result of the Hessenberg decomposition, returned by [Mtx::hessenberg\(\)](#) function.

The documentation for this struct was generated from the following file:

- [matrix.hpp](#)

## 5.3 Mtx::LDL\_result< T > Struct Template Reference

Result of LDL decomposition.

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

## Public Attributes

- [Matrix< T > L](#)  
*Lower triangular matrix.*
- `std::vector< T > d`  
*Vector with diagonal elements of diagonal matrix D.*

### 5.3.1 Detailed Description

```
template<typename T>
struct Mtx::LDL_result< T >
```

Result of LDL decomposition.

This structure stores the result of LDL decomposition, returned by [Mtx::ldl\(\)](#) function.

The documentation for this struct was generated from the following file:

- [matrix.hpp](#)

## 5.4 Mtx::LU\_result< T > Struct Template Reference

Result of LU decomposition.

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

### Public Attributes

- [Matrix< T > L](#)  
*Lower triangular matrix.*
- [Matrix< T > U](#)  
*Upper triangular matrix.*

### 5.4.1 Detailed Description

```
template<typename T>  
struct Mtx::LU_result< T >
```

Result of LU decomposition.

This structure stores the result of LU decomposition, returned by [Mtx::lu\(\)](#) function.

The documentation for this struct was generated from the following file:

- [matrix.hpp](#)

## 5.5 Mtx::LUP\_result< T > Struct Template Reference

Result of LU decomposition with pivoting.

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

### Public Attributes

- [Matrix< T > L](#)  
*Lower triangular matrix.*
- [Matrix< T > U](#)  
*Upper triangular matrix.*
- `std::vector< unsigned > P`  
*Vector with column permutation indices.*

### 5.5.1 Detailed Description

```
template<typename T>
struct Mtx::LUP_result< T >
```

Result of LU decomposition with pivoting.

This structure stores the result of LU decomposition with pivoting, returned by [Mtx::lup\(\)](#) function.

The documentation for this struct was generated from the following file:

- [matrix.hpp](#)

## 5.6 Mtx::Matrix< T > Class Template Reference

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

### Public Member Functions

- [Matrix \(\)](#)  
*Default constructor.*
- [Matrix \(unsigned size\)](#)  
*Square matrix constructor.*
- [Matrix \(unsigned nrows, unsigned ncols\)](#)  
*Rectangular matrix constructor.*
- [Matrix \(T x, unsigned nrows, unsigned ncols\)](#)  
*Rectangular matrix constructor with fill.*
- [Matrix \(const T \\*array, unsigned nrows, unsigned ncols\)](#)  
*Rectangular matrix constructor with initialization.*
- [Matrix \(const std::vector< T > &vec, unsigned nrows, unsigned ncols\)](#)  
*Rectangular matrix constructor with initialization.*
- [Matrix \(std::initializer\\_list< T > init\\_list, unsigned nrows, unsigned ncols\)](#)  
*Rectangular matrix constructor with initialization.*
- [Matrix \(const Matrix &\)](#)
- [virtual ~Matrix \(\)](#)
- [Matrix< T > get\\_submatrix \(unsigned row\\_first, unsigned row\\_last, unsigned col\\_first, unsigned col\\_last\) const](#)  
*Extract a submatrix.*
- [void set\\_submatrix \(const Matrix< T > &smtx, unsigned row\\_first, unsigned col\\_first\)](#)  
*Embed a submatrix.*
- [void clear \(\)](#)  
*Clears the matrix.*
- [void reshape \(unsigned rows, unsigned cols\)](#)  
*Matrix dimension reshape.*
- [void resize \(unsigned rows, unsigned cols\)](#)  
*Resize the matrix.*
- [bool exists \(unsigned row, unsigned col\) const](#)  
*Element exist check.*
- [T \\* ptr \(unsigned row, unsigned col\)](#)

- *Memory pointer.*
- `T * ptr ()`
- *Memory pointer.*
- `void fill (T value)`
- `void fill_col (T value, unsigned col)`
- *Fill column with a scalar.*
- `void fill_row (T value, unsigned row)`
- *Fill row with a scalar.*
- `bool isempty () const`
- *Emptiness check.*
- `bool issquare () const`
- *Squareness check. Check if the matrix is square, i.e., the width of the first and the second dimensions are equal.*
- `bool isequal (const Matrix< T > &) const`
- *Matrix equality check.*
- `bool isequal (const Matrix< T > &, T) const`
- *Matrix equality check with tolerance.*
- `unsigned numel () const`
- *Matrix capacity.*
- `unsigned rows () const`
- *Number of rows.*
- `unsigned cols () const`
- *Number of columns.*
- `Matrix< T > transpose () const`
- *Transpose a matrix.*
- `Matrix< T > ctranspose () const`
- *Transpose a complex matrix.*
- `Matrix< T > & add (const Matrix< T > &)`
- *Matrix sum (in-place).*
- `Matrix< T > & subtract (const Matrix< T > &)`
- *Matrix subtraction (in-place).*
- `Matrix< T > & mult_hadamard (const Matrix< T > &)`
- *Matrix Hadamard product (in-place).*
- `Matrix< T > & add (T)`
- *Matrix sum with scalar (in-place).*
- `Matrix< T > & subtract (T)`
- *Matrix subtraction with scalar (in-place).*
- `Matrix< T > & mult (T)`
- *Matrix product with scalar (in-place).*
- `Matrix< T > & div (T)`
- *Matrix division by scalar (in-place).*
- `Matrix< T > & operator= (const Matrix< T > &)`
- *Matrix assignment.*
- `Matrix< T > & operator= (T)`
- *Matrix fill operator.*
- `operator std::vector< T > () const`
- *Vector cast operator.*
- `std::vector< T > to_vector () const`
- `T & operator() (unsigned nel)`
- *Element access operator (1D)*
- `T operator() (unsigned nel) const`
- `T & at (unsigned nel)`

- [T at \(unsigned nel\) const](#)
- [T & operator\(\) \(unsigned row, unsigned col\)](#)  
*Element access operator (2D)*
- [T operator\(\) \(unsigned row, unsigned col\) const](#)
- [T & at \(unsigned row, unsigned col\)](#)
- [T at \(unsigned row, unsigned col\) const](#)
- [void add\\_row\\_to\\_another \(unsigned to, unsigned from\)](#)  
*Row addition.*
- [void add\\_col\\_to\\_another \(unsigned to, unsigned from\)](#)  
*Column addition.*
- [void mult\\_row\\_by\\_another \(unsigned to, unsigned from\)](#)  
*Row multiplication.*
- [void mult\\_col\\_by\\_another \(unsigned to, unsigned from\)](#)  
*Column multiplication.*
- [void swap\\_rows \(unsigned i, unsigned j\)](#)  
*Row swap.*
- [void swap\\_cols \(unsigned i, unsigned j\)](#)  
*Column swap.*
- [std::vector< T > col\\_to\\_vector \(unsigned col\) const](#)  
*Column to vector.*
- [std::vector< T > row\\_to\\_vector \(unsigned row\) const](#)  
*Row to vector.*
- [void col\\_from\\_vector \(const std::vector< T > &, unsigned col\)](#)  
*Column from vector.*
- [void row\\_from\\_vector \(const std::vector< T > &, unsigned row\)](#)  
*Row from vector.*

### 5.6.1 Detailed Description

```
template<typename T>
class Mtx::Matrix< T >
```

[Matrix](#) class definition.

### 5.6.2 Constructor & Destructor Documentation

#### 5.6.2.1 Matrix() [1/8]

```
template<typename T>
Mtx::Matrix< T >::Matrix ( )
```

Default constructor.

Constructs an empty matrix with zero capacity, taking *rows* = 0 and *cols* = 0.

Referenced by [Mtx::Matrix< T >::add\(\)](#), [Mtx::Matrix< T >::col\\_from\\_vector\(\)](#), [Mtx::Matrix< T >::col\\_to\\_vector\(\)](#), [Mtx::Matrix< T >::ctranspose\(\)](#), [Mtx::Matrix< T >::get\\_submatrix\(\)](#), [Mtx::Matrix< T >::isequal\(\)](#), [Mtx::Matrix< T >::isequal\(\)](#), [Mtx::Matrix< T >::mult\\_hadamard\(\)](#), [Mtx::Matrix< T >::ptr\(\)](#), [Mtx::Matrix< T >::row\\_from\\_vector\(\)](#), [Mtx::Matrix< T >::row\\_to\\_vector\(\)](#), [Mtx::Matrix< T >::set\\_submatrix\(\)](#), [Mtx::Matrix< T >::subtract\(\)](#), [Mtx::Matrix< T >::swap\\_cols\(\)](#), [Mtx::Matrix< T >::swap\\_rows\(\)](#), and [Mtx::Matrix< T >::transpose\(\)](#).

### 5.6.2.2 Matrix() [2/8]

```
template<typename T >
Mtx::Matrix< T >::Matrix (
    unsigned size )
```

Square matrix constructor.

Constructs a square matrix of size *size* x *size*. The content of the matrix is left uninitialized.

### 5.6.2.3 Matrix() [3/8]

```
template<typename T >
Mtx::Matrix< T >::Matrix (
    unsigned nrows,
    unsigned ncols )
```

Rectangular matrix constructor.

Constructs a matrix of size *nrows* x *ncols*. The content of the matrix is left uninitialized.

References [Mtx::Matrix< T >::numel\(\)](#).

### 5.6.2.4 Matrix() [4/8]

```
template<typename T >
Mtx::Matrix< T >::Matrix (
    T x,
    unsigned nrows,
    unsigned ncols )
```

Rectangular matrix constructor with fill.

Constructs a matrix of size *nrows* x *ncols*. All of the matrix elements of are set to value *x*.

References [Mtx::Matrix< T >::fill\(\)](#).

### 5.6.2.5 Matrix() [5/8]

```
template<typename T >
Mtx::Matrix< T >::Matrix (
    const T * array,
    unsigned nrows,
    unsigned ncols )
```

Rectangular matrix constructor with initialization.

Constructs a matrix of size *nrows* x *ncols*. The elements of the matrix are initialized using the elements stored in the input *array*. The elements of the matrix are filled in a column-major order.

References [Mtx::Matrix< T >::numel\(\)](#).

#### 5.6.2.6 Matrix() [6/8]

```
template<typename T >
Mtx::Matrix< T >::Matrix (
    const std::vector< T > & vec,
    unsigned nRows,
    unsigned nCols )
```

Rectangular matrix constructor with initialization.

Constructs a matrix of size *nRows* x *nCols*. The elements of the matrix are initialized using the elements stored in the input `std::vector`. Size of the vector must be equal to the number of matrix elements. The elements of the matrix are filled in a column-major order.



## Exceptions

|                                 |   |
|---------------------------------|---|
| <code>std::runtime_error</code> | when the size of initialization vector is not consistent with matrix dimensions |
|---------------------------------|---|

References [Mtx::Matrix< T >::numel\(\)](#).

**5.6.2.7 Matrix()** [7/8]

```
template<typename T >
Mtx::Matrix< T >::Matrix (
    std::initializer_list< T > init_list,
    unsigned nrows,
    unsigned ncols )
```

Rectangular matrix constructor with initialization.

Constructs a matrix of size *nrows* x *ncols*. The elements of the matrix are initialized using the elements stored in the input `std::initializer_list`. Size of the vector must be equal to the number of matrix elements. The elements of the matrix are filled in a column-major order.

## Exceptions

|                                 |   |
|---------------------------------|---|
| <code>std::runtime_error</code> | when the size of initialization list is not consistent with matrix dimensions |
|---------------------------------|---|

References [Mtx::Matrix< T >::numel\(\)](#).

**5.6.2.8 Matrix()** [8/8]

```
template<typename T >
Mtx::Matrix< T >::Matrix (
    const Matrix< T > & other )
```

Copy constructor.

**5.6.2.9 ~Matrix()**

```
template<typename T >
Mtx::Matrix< T >::~Matrix ( ) [virtual]
```

Destructor.

**5.6.3 Member Function Documentation****5.6.3.1 add()** [1/2]

```
template<typename T >
Matrix< T > & Mtx::Matrix< T >::add (
    const Matrix< T > & m )
```

[Matrix](#) sum (in-place).

Calculates a sum of two matrices  $A + B$ .  $A$  and  $B$  must be the same size. Operation is performed in-place by modifying elements of the matrix.

## Exceptions

|                                 |                                     |
|---------------------------------|-------------------------------------|
| <code>std::runtime_error</code> | when matrix dimensions do not match |
|---------------------------------|-------------------------------------|

References [Mtx::Matrix< T >::cols\(\)](#), [Mtx::Matrix< T >::Matrix\(\)](#), [Mtx::Matrix< T >::numel\(\)](#), and [Mtx::Matrix< T >::rows\(\)](#).

Referenced by [Mtx::operator+=\(\)](#), and [Mtx::operator+=\(\)](#).

**5.6.3.2 add() [2/2]**

```
template<typename T >
Mtx::Matrix< T > & Mtx::Matrix< T >::add (
    T s )
```

[Matrix](#) sum with scalar (in-place).

Adds a scalar *s* to each element of the matrix. Operation is performed in-place by modifying elements of the matrix.

**5.6.3.3 add\_col\_to\_another()**

```
template<typename T >
void Mtx::Matrix< T >::add_col_to_another (
    unsigned to,
    unsigned from )
```

Column addition.

Adds values of elements in column *from* to the elements of column *to*. The elements in column *from* are unchanged.

## Exceptions

|                                |                                   |
|--------------------------------|-----------------------------------|
| <code>std::out_of_range</code> | when column index is out of range |
|--------------------------------|-----------------------------------|

References [Mtx::Matrix< T >::cols\(\)](#), and [Mtx::Matrix< T >::rows\(\)](#).

**5.6.3.4 add\_row\_to\_another()**

```
template<typename T >
void Mtx::Matrix< T >::add_row_to_another (
    unsigned to,
    unsigned from )
```

Row addition.

Adds values of elements in row *from* to the elements of row *to*. The elements in row *from* are unchanged.

## Exceptions

|                                |                                |
|--------------------------------|--------------------------------|
| <code>std::out_of_range</code> | when row index is out of range |
|--------------------------------|--------------------------------|

References [Mtx::Matrix< T >::cols\(\)](#), and [Mtx::Matrix< T >::rows\(\)](#).

**5.6.3.5 clear()**

```
template<typename T >
void Mtx::Matrix< T >::clear ( ) [inline]
```

Clears the matrix.

De-allocates the memory reserved for matrix storage and sets the matrix size to 0.

**5.6.3.6 col\_from\_vector()**

```
template<typename T >
void Mtx::Matrix< T >::col_from_vector (
    const std::vector< T > & vec,
    unsigned col ) [inline]
```

Column from vector.

Assigns values of elements of a column *col* to the values stored in the input vector. Size of the vector must be equal to the number of rows of the matrix.

## Exceptions

|                                 |  |
|---------------------------------|--|
| <code>std::runtime_error</code> | when std::vector size is not equal to number of rows |
| <code>std::out_of_range</code>  | when column index out of range                       |

References [Mtx::Matrix< T >::cols\(\)](#), [Mtx::Matrix< T >::Matrix\(\)](#), and [Mtx::Matrix< T >::rows\(\)](#).

**5.6.3.7 col\_to\_vector()**

```
template<typename T >
std::vector< T > Mtx::Matrix< T >::col_to_vector (
    unsigned col ) const [inline]
```

Column to vector.

Stores elements from column *col* to a std::vector.

## Exceptions

|                                |                                   |
|--------------------------------|-----------------------------------|
| <code>std::out_of_range</code> | when column index is out of range |
|--------------------------------|-----------------------------------|

References [Mtx::Matrix< T >::Matrix\(\)](#), and [Mtx::Matrix< T >::rows\(\)](#).

### 5.6.3.8 cols()

```
template<typename T >
unsigned Mtx::Matrix< T >::cols ( ) const [inline]
```

Number of columns.

Returns the number of columns of the matrix, i.e., the size of the second dimension.

Referenced by [Mtx::Matrix< T >::add\(\)](#), [Mtx::add\(\)](#), [Mtx::add\(\)](#), [Mtx::Matrix< T >::add\\_col\\_to\\_another\(\)](#), [Mtx::Matrix< T >::add\\_row\\_to\\_another\(\)](#), [Mtx::adj\(\)](#), [Mtx::circshift\(\)](#), [Mtx::cofactor\(\)](#), [Mtx::Matrix< T >::col\\_from\\_vector\(\)](#), [Mtx::concatenate\\_horizontal\(\)](#), [Mtx::concatenate\\_vertical\(\)](#), [Mtx::div\(\)](#), [Mtx::Matrix< T >::fill\\_col\(\)](#), [Mtx::Matrix< T >::get\\_submatrix\(\)](#), [Mtx::householder\\_reflection\(\)](#), [Mtx::imag\(\)](#), [Mtx::Matrix< T >::isequal\(\)](#), [Mtx::Matrix< T >::isequal\(\)](#), [Mtx::istril\(\)](#), [Mtx::istriu\(\)](#), [Mtx::kron\(\)](#), [Mtx::lu\(\)](#), [Mtx::lup\(\)](#), [Mtx::make\\_complex\(\)](#), [Mtx::make\\_complex\(\)](#), [Mtx::mult\(\)](#), [Mtx::mult\(\)](#), [Mtx::mult\(\)](#), [Mtx::mult\(\)](#), [Mtx::Matrix< T >::mult\\_col\\_by\\_another\(\)](#), [Mtx::Matrix< T >::mult\\_hadamard\(\)](#), [Mtx::mult\\_hadamard\(\)](#), [Mtx::Matrix< T >::mult\\_row\\_by\\_another\(\)](#), [Mtx::operator<<\(\)](#), [Mtx::permute\\_cols\(\)](#), [Mtx::permute\\_rows\(\)](#), [Mtx::permute\\_rows\\_and\\_cols\(\)](#), [Mtx::pinv\(\)](#), [Mtx::Matrix< T >::ptr\(\)](#), [Mtx::qr\\_householder\(\)](#), [Mtx::qr\\_red\\_gs\(\)](#), [Mtx::real\(\)](#), [Mtx::repmat\(\)](#), [Mtx::Matrix< T >::reshape\(\)](#), [Mtx::Matrix< T >::resize\(\)](#), [Mtx::Matrix< T >::row\\_from\\_vector\(\)](#), [Mtx::Matrix< T >::row\\_to\\_vector\(\)](#), [Mtx::Matrix< T >::set\\_submatrix\(\)](#), [Mtx::solve\\_tril\(\)](#), [Mtx::solve\\_triu\(\)](#), [Mtx::Matrix< T >::subtract\(\)](#), [Mtx::subtract\(\)](#), [Mtx::subtract\(\)](#), [Mtx::Matrix< T >::swap\\_cols\(\)](#), [Mtx::Matrix< T >::swap\\_rows\(\)](#), [Mtx::tril\(\)](#), and [Mtx::triu\(\)](#).

### 5.6.3.9 ctranspose()

```
template<typename T >
Matrix< T > Mtx::Matrix< T >::ctranspose ( ) const [inline]
```

Transpose a complex matrix.

Returns a matrix that is a conjugate (Hermitian) transposition of an input matrix.

Conjugate transpose applies a conjugate operation to all elements in addition to matrix transposition.

References [Mtx::cconj\(\)](#), and [Mtx::Matrix< T >::Matrix\(\)](#).

Referenced by [Mtx::ctranspose\(\)](#).

### 5.6.3.10 div()

```
template<typename T >
Matrix< T > & Mtx::Matrix< T >::div (
    T s )
```

[Matrix](#) division by scalar (in-place).

Divides each element of the matrix by a scalar *s*. Operation is performed in-place by modifying elements of the matrix.

Referenced by [Mtx::operator/=\( \)](#).

### 5.6.3.11 exists()

```
template<typename T >
bool Mtx::Matrix< T >::exists (
    unsigned row,
    unsigned col ) const [inline]
```

Element exist check.

Returns true if the element with specified coordinates exists within the matrix dimension range. For example, calling *exist(4,0)* on a matrix with dimensions 2 x 2 shall yield false.

### 5.6.3.12 fill()

```
template<typename T >
void Mtx::Matrix< T >::fill (
    T value ) [inline]
```

Fill with a scalar. Set all the elements of the matrix to a specified value.

References [Mtx::Matrix< T >::numel\(\)](#).

Referenced by [Mtx::Matrix< T >::Matrix\(\)](#), and [Mtx::Matrix< T >::operator=\(\)](#).

### 5.6.3.13 fill\_col()

```
template<typename T >
void Mtx::Matrix< T >::fill_col (
    T value,
    unsigned col ) [inline]
```

Fill column with a scalar.

Set all the elements in a specified column of the matrix to a specified value.

#### Exceptions

|                                |                                   |
|--------------------------------|-----------------------------------|
| <code>std::out_of_range</code> | when column index is out of range |
|--------------------------------|-----------------------------------|

References [Mtx::Matrix< T >::cols\(\)](#).

### 5.6.3.14 fill\_row()

```
template<typename T >
void Mtx::Matrix< T >::fill_row (
    T value,
    unsigned row ) [inline]
```

Fill row with a scalar.

Set all the elements in a specified row of the matrix to a specified value.

## Exceptions

|                                |                                |
|--------------------------------|--------------------------------|
| <code>std::out_of_range</code> | when row index is out of range |
|--------------------------------|--------------------------------|

References [Mtx::Matrix< T >::rows\(\)](#).

5.6.3.15 `get_submatrix()`

```
template<typename T >
Matrix< T > Mtx::Matrix< T >::get_submatrix (
    unsigned row_first,
    unsigned row_last,
    unsigned col_first,
    unsigned col_last ) const
```

Extract a submatrix.

Constructs a submatrix using the specified range of row and column indices. The submatrix contains a copy of elements placed between row indices indicated by `row_first` and `row_last`, and column indices `col_first` and `col_last`. Both index ranges are inclusive.

## Exceptions

|                                |   |
|--------------------------------|---|
| <code>std::out_of_range</code> | when row or column index is out of range of matrix dimensions |
|--------------------------------|---|

References [Mtx::Matrix< T >::cols\(\)](#), [Mtx::Matrix< T >::Matrix\(\)](#), and [Mtx::Matrix< T >::rows\(\)](#).

Referenced by [Mtx::hessenberg\(\)](#), [Mtx::qr\\_householder\(\)](#), and [Mtx::qr\\_red\\_gs\(\)](#).

5.6.3.16 `isempty()`

```
template<typename T >
bool Mtx::Matrix< T >::isempty ( ) const [inline]
```

Emptiness check.

Check if the matrix is empty, i.e., if both dimensions are equal zero and the matrix stores no elements.

Referenced by [Mtx::Matrix< T >::issquare\(\)](#), and [Mtx::Matrix< T >::set\\_submatrix\(\)](#).

5.6.3.17 `isequal()` [1/2]

```
template<typename T >
bool Mtx::Matrix< T >::isequal (
    const Matrix< T > & A ) const
```

[Matrix](#) equality check.

Returns true, if both matrices are the same size and all of the element are equal value.

References [Mtx::Matrix< T >::cols\(\)](#), [Mtx::Matrix< T >::Matrix\(\)](#), [Mtx::Matrix< T >::numel\(\)](#), and [Mtx::Matrix< T >::rows\(\)](#).

Referenced by [Mtx::operator!=\(\)](#), and [Mtx::operator==\(\)](#).

**5.6.3.18 isequal()** [2/2]

```
template<typename T >
bool Mtx::Matrix< T >::isequal (
    const Matrix< T > & A,
    T tol ) const
```

**Matrix** equality check with tolerance.

Returns true, if both matrices are the same size and all of the element are equal in value under assumed tolerance. The tolerance check is performed for each element:  $tol < |A_{i,j} - B_{i,j}|$ .

References [Mtx::Matrix< T >::cols\(\)](#), [Mtx::Matrix< T >::Matrix\(\)](#), [Mtx::Matrix< T >::numel\(\)](#), and [Mtx::Matrix< T >::rows\(\)](#).

**5.6.3.19 mult()**

```
template<typename T >
Matrix< T > & Mtx::Matrix< T >::mult (
    T s )
```

**Matrix** product with scalar (in-place).

Multiplies each element of the matrix by a scalar *s*. Operation is performed in-place by modifying elements of the matrix.

Referenced by [Mtx::operator\\*=\( \)](#).

**5.6.3.20 mult\_col\_by\_another()**

```
template<typename T >
void Mtx::Matrix< T >::mult_col_by_another (
    unsigned to,
    unsigned from )
```

Column multiplication.

Multiply values of each element in column *to* by the elements of column *from*. The elements in column *from* are unchanged.

**Exceptions**

|                                |                                   |
|--------------------------------|-----------------------------------|
| <code>std::out_of_range</code> | when column index is out of range |
|--------------------------------|-----------------------------------|

References [Mtx::Matrix< T >::cols\(\)](#), and [Mtx::Matrix< T >::rows\(\)](#).

**5.6.3.21 mult\_hadamard()**

```
template<typename T >
Matrix< T > & Mtx::Matrix< T >::mult_hadamard (
    const Matrix< T > & m )
```

[Matrix](#) Hadamard product (in-place).

Calculates a Hadamard product of two matrices  $A \otimes B$ .  $A$  and  $B$  must be the same size. Hadamard product is calculated as an element-wise multiplication between the matrices. Operation is performed in-place by modifying elements of the matrix.

#### Exceptions

|                                 |                                     |
|---------------------------------|-------------------------------------|
| <code>std::runtime_error</code> | when matrix dimensions do not match |
|---------------------------------|-------------------------------------|

References [Mtx::Matrix< T >::cols\(\)](#), [Mtx::Matrix< T >::Matrix\(\)](#), [Mtx::Matrix< T >::numel\(\)](#), and [Mtx::Matrix< T >::rows\(\)](#).

Referenced by [Mtx::operator^=\(\)](#).

### 5.6.3.22 mult\_row\_by\_another()

```
template<typename T >
void Mtx::Matrix< T >::mult_row_by_another (
    unsigned to,
    unsigned from )
```

Row multiplication.

Multiply values of each element in row *to* by the elements of row *from*. The elements in row *from* are unchanged.

#### Exceptions

|                                |                                |
|--------------------------------|--------------------------------|
| <code>std::out_of_range</code> | when row index is out of range |
|--------------------------------|--------------------------------|

References [Mtx::Matrix< T >::cols\(\)](#), and [Mtx::Matrix< T >::rows\(\)](#).

### 5.6.3.23 numel()

```
template<typename T >
unsigned Mtx::Matrix< T >::numel ( ) const [inline]
```

[Matrix](#) capacity.

Returns the number of the elements stored within the matrix, i.e., a product of both dimensions.

Referenced by [Mtx::Matrix< T >::add\(\)](#), [Mtx::add\(\)](#), [Mtx::div\(\)](#), [Mtx::Matrix< T >::fill\(\)](#), [Mtx::foreach\\_elem\(\)](#), [Mtx::householder\\_reflection\(\)](#), [Mtx::imag\(\)](#), [Mtx::inv\(\)](#), [Mtx::Matrix< T >::isequal\(\)](#), [Mtx::Matrix< T >::isequal\(\)](#), [Mtx::lu\(\)](#), [Mtx::lup\(\)](#), [Mtx::make\\_complex\(\)](#), [Mtx::make\\_complex\(\)](#), [Mtx::Matrix< T >::Matrix\(\)](#), [Mtx::Matrix< T >::Matrix\(\)](#), [Mtx::Matrix< T >::Matrix\(\)](#), [Mtx::Matrix< T >::Matrix\(\)](#), [Mtx::mult\(\)](#), [Mtx::Matrix< T >::mult\\_hadamard\(\)](#), [Mtx::norm\\_fro\(\)](#), [Mtx::norm\\_fro\(\)](#), [Mtx::real\(\)](#), [Mtx::Matrix< T >::reshape\(\)](#), [Mtx::solve\\_posdef\(\)](#), [Mtx::solve\\_square\(\)](#), [Mtx::solve\\_tril\(\)](#), [Mtx::solve\\_triu\(\)](#), [Mtx::Matrix< T >::subtract\(\)](#), and [Mtx::subtract\(\)](#).

### 5.6.3.24 operator std::vector< T >()

```
template<typename T >
Mtx::Matrix< T >::operator std::vector< T > ( ) const [inline], [explicit]
```

Vector cast operator.

Converts the matrix to a vector with *nrows* x *ncols* elements. Element order in the vector follow column-major format.



**5.6.3.25 operator>() [1/2]**

```
template<typename T >
T & Mtx::Matrix< T >::operator() (
    unsigned nel ) [inline]
```

Element access operator (1D)

Access specific matrix element using singular index of the element. Follows column-major convention.

**Exceptions**

|                                |                                    |
|--------------------------------|------------------------------------|
| <code>std::out_of_range</code> | when element index is out of range |
|--------------------------------|------------------------------------|

**5.6.3.26 operator()( [2/2]**

```
template<typename T >
T & Mtx::Matrix< T >::operator() (
    unsigned row,
    unsigned col ) [inline]
```

Element access operator (2D)

Access specific matrix element using row and column index of the element.

**Exceptions**

|                                |   |
|--------------------------------|---|
| <code>std::out_of_range</code> | when row or column index is out of range of matrix dimensions |
|--------------------------------|---|

**5.6.3.27 operator=() [1/2]**

```
template<typename T >
Matrix< T > & Mtx::Matrix< T >::operator= (
    const Matrix< T > & other )
```

**Matrix** assignment.

Performs deep-copy of another matrix.

**5.6.3.28 operator=() [2/2]**

```
template<typename T >
Matrix< T > & Mtx::Matrix< T >::operator= (
    T s )
```

**Matrix** fill operator.

Assigns value of each element in the matrix to a given scalar. This method does not affect the shape and capacity of the matrix.

References `Mtx::Matrix< T >::fill()`.

**5.6.3.29 ptr() [1/2]**

```
template<typename T >
T * Mtx::Matrix< T >::ptr ( ) [inline]
```

Memory pointer.

Returns a pointer to the first element in the array used internally by the matrix. The matrix memory is arranged in a column-major order.

**Exceptions**

|                                |  |
|--------------------------------|--|
| <code>std::out_of_range</code> | when row or column index is out of range |
|--------------------------------|--|

**5.6.3.30 ptr() [2/2]**

```
template<typename T >
T * Mtx::Matrix< T >::ptr (
    unsigned row,
    unsigned col ) [inline]
```

Memory pointer.

Returns a pointer to the selected element in the array used internally by the matrix. The matrix memory is arranged in a column-major order.

References [Mtx::Matrix< T >::cols\(\)](#), [Mtx::Matrix< T >::Matrix\(\)](#), and [Mtx::Matrix< T >::rows\(\)](#).

**5.6.3.31 reshape()**

```
template<typename T >
void Mtx::Matrix< T >::reshape (
    unsigned rows,
    unsigned cols )
```

[Matrix](#) dimension reshape.

Modifies the first and the second dimension of the matrix according to the input parameters. A number of elements in the reshaped matrix must be the preserved and not changed comparing to the state before the reshape.

**Exceptions**

|                                 |  |
|---------------------------------|--|
| <code>std::runtime_error</code> | when reshape attempts to change the number of elements |
|---------------------------------|--|

References [Mtx::Matrix< T >::cols\(\)](#), [Mtx::Matrix< T >::numel\(\)](#), and [Mtx::Matrix< T >::rows\(\)](#).

**5.6.3.32 resize()**

```
template<typename T >
void Mtx::Matrix< T >::resize (
```

```

    unsigned rows,
    unsigned cols )

```

Resize the matrix.

Clears the content of the matrix and changes it dimensions to be equal to the specified number of rows and columns. Remark that the content of the matrix is lost after calling the reshape method.

References [Mtx::Matrix< T >::cols\(\)](#), and [Mtx::Matrix< T >::rows\(\)](#).

Referenced by [Mtx::det\\_lu\(\)](#), [Mtx::diag\(\)](#), and [Mtx::lup\(\)](#).

### 5.6.3.33 row\_from\_vector()

```

template<typename T >
void Mtx::Matrix< T >::row_from_vector (
    const std::vector< T > & vec,
    unsigned row ) [inline]

```

Row from vector.

Assigns values of elements of a row *col* to the values stored in the input vector. Size of the vector must be equal to the number of columns of the matrix.

#### Exceptions

|                                 |  |
|---------------------------------|--|
| <code>std::runtime_error</code> | when <code>std::vector</code> size is not equal to number of columns |
| <code>std::out_of_range</code>  | when row index out of range  |

References [Mtx::Matrix< T >::cols\(\)](#), [Mtx::Matrix< T >::Matrix\(\)](#), and [Mtx::Matrix< T >::rows\(\)](#).

### 5.6.3.34 row\_to\_vector()

```

template<typename T >
std::vector< T > Mtx::Matrix< T >::row_to_vector (
    unsigned row ) const [inline]

```

Row to vector.

Stores elements from row *row* to a `std::vector`.

#### Exceptions

|                                |                                |
|--------------------------------|--------------------------------|
| <code>std::out_of_range</code> | when row index is out of range |
|--------------------------------|--------------------------------|

References [Mtx::Matrix< T >::cols\(\)](#), and [Mtx::Matrix< T >::Matrix\(\)](#).

### 5.6.3.35 rows()

```

template<typename T >
unsigned Mtx::Matrix< T >::rows ( ) const [inline]

```

Number of rows.

Returns the number of rows of the matrix, i.e., the size of the first dimension.

Referenced by [Mtx::Matrix< T >::add\(\)](#), [Mtx::add\(\)](#), [Mtx::add\(\)](#), [Mtx::Matrix< T >::add\\_col\\_to\\_another\(\)](#), [Mtx::Matrix< T >::add\\_row\\_to\\_another\(\)](#), [Mtx::adj\(\)](#), [Mtx::chol\(\)](#), [Mtx::cholinv\(\)](#), [Mtx::circshift\(\)](#), [Mtx::cofactor\(\)](#), [Mtx::Matrix< T >::col\\_from\\_vector\(\)](#), [Mtx::Matrix< T >::col\\_to\\_vector\(\)](#), [Mtx::concatenate\\_horizontal\(\)](#), [Mtx::concatenate\\_vertical\(\)](#), [Mtx::det\(\)](#), [Mtx::det\\_lu\(\)](#), [Mtx::diag\(\)](#), [Mtx::div\(\)](#), [Mtx::eigenvalues\(\)](#), [Mtx::Matrix< T >::fill\\_row\(\)](#), [Mtx::Matrix< T >::get\\_submatrix\(\)](#), [Mtx::hessenberg\(\)](#), [Mtx::imag\(\)](#), [Mtx::inv\(\)](#), [Mtx::inv\\_gauss\\_jordan\(\)](#), [Mtx::inv\\_tril\(\)](#), [Mtx::inv\\_triu\(\)](#), [Mtx::Matrix< T >::isequal\(\)](#), [Mtx::Matrix< T >::isequal\(\)](#), [Mtx::ishess\(\)](#), [Mtx::istril\(\)](#), [Mtx::istriu\(\)](#), [Mtx::kron\(\)](#), [Mtx::ldl\(\)](#), [Mtx::lu\(\)](#), [Mtx::lup\(\)](#), [Mtx::make\\_complex\(\)](#), [Mtx::make\\_complex\(\)](#), [Mtx::mult\(\)](#), [Mtx::mult\(\)](#), [Mtx::mult\(\)](#), [Mtx::mult\(\)](#), [Mtx::Matrix< T >::mult\\_col\\_by\\_another\(\)](#), [Mtx::Matrix< T >::mult\\_hadamard\(\)](#), [Mtx::mult\\_hadamard\(\)](#), [Mtx::Matrix< T >::mult\\_row\\_by\\_another\(\)](#), [Mtx::operator<<\(\)](#), [Mtx::permute\\_cols\(\)](#), [Mtx::permute\\_rows\(\)](#), [Mtx::permute\\_rows\\_and\\_cols\(\)](#), [Mtx::pinv\(\)](#), [Mtx::Matrix< T >::ptr\(\)](#), [Mtx::qr\\_householder\(\)](#), [Mtx::qr\\_red\\_gs\(\)](#), [Mtx::real\(\)](#), [Mtx::repmat\(\)](#), [Mtx::Matrix< T >::reshape\(\)](#), [Mtx::Matrix< T >::resize\(\)](#), [Mtx::Matrix< T >::row\\_from\\_vector\(\)](#), [Mtx::Matrix< T >::set\\_submatrix\(\)](#), [Mtx::solve\\_posdef\(\)](#), [Mtx::solve\\_square\(\)](#), [Mtx::solve\\_tril\(\)](#), [Mtx::solve\\_triu\(\)](#), [Mtx::Matrix< T >::subtract\(\)](#), [Mtx::subtract\(\)](#), [Mtx::subtract\(\)](#), [Mtx::Matrix< T >::swap\\_cols\(\)](#), [Mtx::Matrix< T >::swap\\_rows\(\)](#), [Mtx::trace\(\)](#), [Mtx::tril\(\)](#), [Mtx::triu\(\)](#), and [Mtx::wilkinson\\_shift\(\)](#).

### 5.6.3.36 set\_submatrix()

```
template<typename T >
void Mtx::Matrix< T >::set_submatrix (
    const Matrix< T > & smtx,
    unsigned row_first,
    unsigned col_first )
```

Embed a submatrix.

Embed elements of the input submatrix at the specified range of row and column indices. The elements of input submatrix are placed starting at row index incated by *row\_first* and column indices *col\_first*.

#### Exceptions

|                           |   |
|---------------------------|---|
| <i>std::out_of_range</i>  | when row or column index is out of range of matrix dimensions |
| <i>std::runtime_error</i> | when input matrix is empty (i.e., it has zero elements)       |

References [Mtx::Matrix< T >::cols\(\)](#), [Mtx::Matrix< T >::isempty\(\)](#), [Mtx::Matrix< T >::Matrix\(\)](#), and [Mtx::Matrix< T >::rows\(\)](#).

### 5.6.3.37 subtract() [1/2]

```
template<typename T >
Matrix< T > & Mtx::Matrix< T >::subtract (
    const Matrix< T > & m )
```

**Matrix** subtraction (in-place).

Calculates a subtraction of two matrices  $A - B$ .  $A$  and  $B$  must be the same size. Operation is performed in-place by modifying elements of the matrix.

#### Exceptions

|                           |                                     |
|---------------------------|-------------------------------------|
| <i>std::runtime_error</i> | when matrix dimensions do not match |
|---------------------------|-------------------------------------|

References [Mtx::Matrix< T >::cols\(\)](#), [Mtx::Matrix< T >::Matrix\(\)](#), [Mtx::Matrix< T >::numel\(\)](#), and [Mtx::Matrix< T >::rows\(\)](#).

Referenced by [Mtx::operator-=\(\)](#), and [Mtx::operator-=\(\)](#).

### 5.6.3.38 subtract() [2/2]

```
template<typename T >
Matrix< T > & Mtx::Matrix< T >::subtract (
    T s )
```

[Matrix](#) subtraction with scalar (in-place).

Subtracts a scalar *s* from each element of the matrix. Operation is performed in-place by modifying elements of the matrix.

### 5.6.3.39 swap\_cols()

```
template<typename T >
void Mtx::Matrix< T >::swap_cols (
    unsigned i,
    unsigned j )
```

Column swap.

Swaps element values between two columns.

#### Exceptions

|                          |                                   |
|--------------------------|-----------------------------------|
| <i>std::out_of_range</i> | when column index is out of range |
|--------------------------|-----------------------------------|

References [Mtx::Matrix< T >::cols\(\)](#), [Mtx::Matrix< T >::Matrix\(\)](#), and [Mtx::Matrix< T >::rows\(\)](#).

Referenced by [Mtx::lup\(\)](#).

### 5.6.3.40 swap\_rows()

```
template<typename T >
void Mtx::Matrix< T >::swap_rows (
    unsigned i,
    unsigned j )
```

Row swap.

Swaps element values of two columns.

#### Exceptions

|                          |                                |
|--------------------------|--------------------------------|
| <i>std::out_of_range</i> | when row index is out of range |
|--------------------------|--------------------------------|

References [Mtx::Matrix< T >::cols\(\)](#), [Mtx::Matrix< T >::Matrix\(\)](#), and [Mtx::Matrix< T >::rows\(\)](#).

#### 5.6.3.41 transpose()

```
template<typename T >
Matrix< T > Mtx::Matrix< T >::transpose ( ) const [inline]
```

Transpose a matrix.

Returns a matrix that is a transposition of an input matrix.

References [Mtx::Matrix< T >::Matrix\(\)](#).

Referenced by [Mtx::transpose\(\)](#).

The documentation for this class was generated from the following file:

- [matrix.hpp](#)

## 5.7 Mtx::QR\_result< T > Struct Template Reference

Result of QR decomposition.

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

### Public Attributes

- [Matrix< T > Q](#)  
*Orthogonal matrix.*
- [Matrix< T > R](#)  
*Upper triangular matrix.*

### 5.7.1 Detailed Description

```
template<typename T>
struct Mtx::QR_result< T >
```

Result of QR decomposition.

This structure stores the result of QR decomposition, returned by, e.g., from [Mtx::qr\(\)](#) function. Note that the dimensions of *Q* and *R* matrices depends on the employed variant of QR decomposition.

The documentation for this struct was generated from the following file:

- [matrix.hpp](#)

## 5.8 Mtx::singular\_matrix\_exception Class Reference

Singular matrix exception.

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

Inheritance diagram for [Mtx::singular\\_matrix\\_exception](#):

# Chapter 6

## File Documentation

### 6.1 matrix.hpp File Reference

#### Classes

- class [Mtx::singular\\_matrix\\_exception](#)  
*Singular matrix exception.*
- struct [Mtx::LU\\_result< T >](#)  
*Result of LU decomposition.*
- struct [Mtx::LUP\\_result< T >](#)  
*Result of LU decomposition with pivoting.*
- struct [Mtx::QR\\_result< T >](#)  
*Result of QR decomposition.*
- struct [Mtx::Hessenberg\\_result< T >](#)  
*Result of Hessenberg decomposition.*
- struct [Mtx::LDL\\_result< T >](#)  
*Result of LDL decomposition.*
- struct [Mtx::Eigenvalues\\_result< T >](#)  
*Result of eigenvalues.*
- class [Mtx::Matrix< T >](#)

#### Functions

- [template<typename T , typename std::enable\\_if<!is\\_complex< T >::value, int >::type = 0> T Mtx::cconj \(T x\)](#)  
*Complex conjugate helper.*
- [template<typename T , typename std::enable\\_if<!is\\_complex< T >::value, int >::type = 0> T Mtx::csign \(T x\)](#)  
*Complex sign helper.*
- [template<typename T > Matrix< T > Mtx::zeros \(unsigned nrows, unsigned ncols\)](#)  
*Matrix of zeros.*
- [template<typename T > Matrix< T > Mtx::zeros \(unsigned n\)](#)  
*Square matrix of zeros.*

- `template<typename T>`  
`Matrix< T > Mtx::ones (unsigned nrows, unsigned ncols)`  
*Matrix of ones.*
- `template<typename T>`  
`Matrix< T > Mtx::ones (unsigned n)`  
*Square matrix of ones.*
- `template<typename T>`  
`Matrix< T > Mtx::eye (unsigned n)`  
*Identity matrix.*
- `template<typename T>`  
`Matrix< T > Mtx::diag (const T *array, size_t n)`  
*Diagonal matrix from array.*
- `template<typename T>`  
`Matrix< T > Mtx::diag (const std::vector< T > &v)`  
*Diagonal matrix from std::vector.*
- `template<typename T>`  
`std::vector< T > Mtx::diag (const Matrix< T > &A)`  
*Diagonal extraction.*
- `template<typename T>`  
`Matrix< T > Mtx::circulant (const T *array, unsigned n)`  
*Circulant matrix from array.*
- `template<typename T>`  
`Matrix< std::complex< T > > Mtx::make_complex (const Matrix< T > &Re, const Matrix< T > &Im)`  
*Create complex matrix from real and imaginary matrices.*
- `template<typename T>`  
`Matrix< std::complex< T > > Mtx::make_complex (const Matrix< T > &Re)`  
*Create complex matrix from real matrix.*
- `template<typename T>`  
`Matrix< T > Mtx::real (const Matrix< std::complex< T > > &C)`  
*Get real part of complex matrix.*
- `template<typename T>`  
`Matrix< T > Mtx::imag (const Matrix< std::complex< T > > &C)`  
*Get imaginary part of complex matrix.*
- `template<typename T>`  
`Matrix< T > Mtx::circulant (const std::vector< T > &v)`  
*Circulant matrix from std::vector.*
- `template<typename T>`  
`Matrix< T > Mtx::transpose (const Matrix< T > &A)`  
*Transpose a matrix.*
- `template<typename T>`  
`Matrix< T > Mtx::ctranspose (const Matrix< T > &A)`  
*Transpose a complex matrix.*
- `template<typename T>`  
`Matrix< T > Mtx::circshift (const Matrix< T > &A, int row_shift, int col_shift)`  
*Circular shift.*
- `template<typename T>`  
`Matrix< T > Mtx:: repmat (const Matrix< T > &A, unsigned m, unsigned n)`  
*Repeat matrix.*
- `template<typename T>`  
`Matrix< T > Mtx::concatenate_horizontal (const Matrix< T > &A, const Matrix< T > &B)`  
*Horizontal matrix concatenation.*
- `template<typename T>`  
`Matrix< T > Mtx::concatenate_vertical (const Matrix< T > &A, const Matrix< T > &B)`



- Vertical matrix concatenation.*

  - `template<typename T >`  
`double Mtx::norm_fro (const Matrix< T > &A)`

*Frobenius norm.*
- `template<typename T >`  
`double Mtx::norm_fro (const Matrix< std::complex< T > > &A)`

*Frobenius norm of a complex matrix.*
- `template<typename T >`  
`Matrix< T > Mtx::tril (const Matrix< T > &A)`

*Extract triangular lower part.*
- `template<typename T >`  
`Matrix< T > Mtx::triu (const Matrix< T > &A)`

*Extract triangular upper part.*
- `template<typename T >`  
`bool Mtx::istril (const Matrix< T > &A)`

*Lower triangular matrix check.*
- `template<typename T >`  
`bool Mtx::istriu (const Matrix< T > &A)`

*Lower triangular matrix check.*
- `template<typename T >`  
`bool Mtx::ishess (const Matrix< T > &A)`

*Hessenberg matrix check.*
- `template<typename T >`  
`void Mtx::foreach_elem (Matrix< T > &A, std::function< T(T)> func)`

*Applies custom function element-wise in-place.*
- `template<typename T >`  
`Matrix< T > Mtx::foreach_elem_copy (const Matrix< T > &A, std::function< T(T)> func)`

*Applies custom function element-wise with matrix copy.*
- `template<typename T >`  
`Matrix< T > Mtx::permute_rows (const Matrix< T > &A, const std::vector< unsigned > perm)`

*Permute rows of the matrix.*
- `template<typename T >`  
`Matrix< T > Mtx::permute_cols (const Matrix< T > &A, const std::vector< unsigned > perm)`

*Permute columns of the matrix.*
- `template<typename T >`  
`Matrix< T > Mtx::permute_rows_and_cols (const Matrix< T > &A, const std::vector< unsigned > perm_rows, const std::vector< unsigned > perm_cols)`

*Permute both rows and columns of the matrix.*
- `template<typename T , bool transpose_first = false, bool transpose_second = false>`  
`Matrix< T > Mtx::mult (const Matrix< T > &A, const Matrix< T > &B)`

*Matrix multiplication.*
- `template<typename T , bool transpose_first = false, bool transpose_second = false>`  
`Matrix< T > Mtx::mult_hadamard (const Matrix< T > &A, const Matrix< T > &B)`

*Matrix Hadamard (element-wise) multiplication.*
- `template<typename T , bool transpose_first = false, bool transpose_second = false>`  
`Matrix< T > Mtx::add (const Matrix< T > &A, const Matrix< T > &B)`

*Matrix addition.*
- `template<typename T , bool transpose_first = false, bool transpose_second = false>`  
`Matrix< T > Mtx::subtract (const Matrix< T > &A, const Matrix< T > &B)`

*Matrix subtraction.*
- `template<typename T , bool transpose_matrix = false>`  
`std::vector< T > Mtx::mult (const Matrix< T > &A, const std::vector< T > &v)`

*Multiplication of matrix by std::vector.*

- `template<typename T, bool transpose_matrix = false>`  
`std::vector< T > Mtx::mult (const std::vector< T > &v, const Matrix< T > &A)`  
*Multiplication of std::vector by matrix.*
- `template<typename T >`  
`Matrix< T > Mtx::add (const Matrix< T > &A, T s)`  
*Addition of scalar to matrix.*
- `template<typename T >`  
`Matrix< T > Mtx::subtract (const Matrix< T > &A, T s)`  
*Subtraction of scalar from matrix.*
- `template<typename T >`  
`Matrix< T > Mtx::mult (const Matrix< T > &A, T s)`  
*Multiplication of matrix by scalar.*
- `template<typename T >`  
`Matrix< T > Mtx::div (const Matrix< T > &A, T s)`  
*Division of matrix by scalar.*
- `template<typename T >`  
`std::ostream & Mtx::operator<< (std::ostream &os, const Matrix< T > &A)`  
*Matrix ostream operator.*
- `template<typename T >`  
`Matrix< T > Mtx::operator+ (const Matrix< T > &A, const Matrix< T > &B)`  
*Matrix sum.*
- `template<typename T >`  
`Matrix< T > Mtx::operator- (const Matrix< T > &A, const Matrix< T > &B)`  
*Matrix subtraction.*
- `template<typename T >`  
`Matrix< T > Mtx::operator^ (const Matrix< T > &A, const Matrix< T > &B)`  
*Matrix Hadamard product.*
- `template<typename T >`  
`Matrix< T > Mtx::operator* (const Matrix< T > &A, const Matrix< T > &B)`  
*Matrix product.*
- `template<typename T >`  
`std::vector< T > Mtx::operator* (const Matrix< T > &A, const std::vector< T > &v)`  
*Matrix and std::vector product.*
- `template<typename T >`  
`std::vector< T > Mtx::operator* (const std::vector< T > &v, const Matrix< T > &A)`  
*std::vector and matrix product.*
- `template<typename T >`  
`Matrix< T > Mtx::operator+ (const Matrix< T > &A, T s)`  
*Matrix sum with scalar.*
- `template<typename T >`  
`Matrix< T > Mtx::operator- (const Matrix< T > &A, T s)`  
*Matrix subtraction with scalar.*
- `template<typename T >`  
`Matrix< T > Mtx::operator* (const Matrix< T > &A, T s)`  
*Matrix product with scalar.*
- `template<typename T >`  
`Matrix< T > Mtx::operator/ (const Matrix< T > &A, T s)`  
*Matrix division by scalar.*
- `template<typename T >`  
`Matrix< T > Mtx::operator+ (T s, const Matrix< T > &A)`
- `template<typename T >`  
`Matrix< T > Mtx::operator* (T s, const Matrix< T > &A)`  
*Matrix product with scalar.*

- `template<typename T>`  
`Matrix< T> & Mtx::operator+= (Matrix< T> &A, const Matrix< T> &B)`  
*Matrix sum.*
- `template<typename T>`  
`Matrix< T> & Mtx::operator-= (Matrix< T> &A, const Matrix< T> &B)`  
*Matrix subtraction.*
- `template<typename T>`  
`Matrix< T> & Mtx::operator*= (Matrix< T> &A, const Matrix< T> &B)`  
*Matrix product.*
- `template<typename T>`  
`Matrix< T> & Mtx::operator^= (Matrix< T> &A, const Matrix< T> &B)`  
*Matrix Hadamard product.*
- `template<typename T>`  
`Matrix< T> & Mtx::operator+= (Matrix< T> &A, T s)`  
*Matrix sum with scalar.*
- `template<typename T>`  
`Matrix< T> & Mtx::operator-= (Matrix< T> &A, T s)`  
*Matrix subtraction with scalar.*
- `template<typename T>`  
`Matrix< T> & Mtx::operator*= (Matrix< T> &A, T s)`  
*Matrix product with scalar.*
- `template<typename T>`  
`Matrix< T> & Mtx::operator/= (Matrix< T> &A, T s)`  
*Matrix division by scalar.*
- `template<typename T>`  
`bool Mtx::operator== (const Matrix< T> &A, const Matrix< T> &b)`  
*Matrix equality check operator.*
- `template<typename T>`  
`bool Mtx::operator!= (const Matrix< T> &A, const Matrix< T> &b)`  
*Matrix non-equality check operator.*
- `template<typename T>`  
`Matrix< T> Mtx::kron (const Matrix< T> &A, const Matrix< T> &B)`  
*Kronecker product.*
- `template<typename T>`  
`Matrix< T> Mtx::adj (const Matrix< T> &A)`  
*Adjugate matrix.*
- `template<typename T>`  
`Matrix< T> Mtx::cofactor (const Matrix< T> &A, unsigned p, unsigned q)`  
*Cofactor matrix.*
- `template<typename T>`  
`T Mtx::det_lu (const Matrix< T> &A)`  
*Matrix determinant from on LU decomposition.*
- `template<typename T>`  
`T Mtx::det (const Matrix< T> &A)`  
*Matrix determinant.*
- `template<typename T>`  
`LU_result< T> Mtx::lu (const Matrix< T> &A)`  
*LU decomposition.*
- `template<typename T>`  
`LUP_result< T> Mtx::lup (const Matrix< T> &A)`  
*LU decomposition with pivoting.*
- `template<typename T>`  
`Matrix< T> Mtx::inv_gauss_jordan (const Matrix< T> &A)`

- Matrix inverse using Gauss-Jordan elimination.*

  - `template<typename T >`  
`Matrix< T > Mtx::inv_tril (const Matrix< T > &A)`  
*Matrix inverse for lower triangular matrix.*
  - `template<typename T >`  
`Matrix< T > Mtx::inv_triu (const Matrix< T > &A)`  
*Matrix inverse for upper triangular matrix.*
  - `template<typename T >`  
`Matrix< T > Mtx::inv_posdef (const Matrix< T > &A)`  
*Matrix inverse for Hermitian positive-definite matrix.*
  - `template<typename T >`  
`Matrix< T > Mtx::inv_square (const Matrix< T > &A)`  
*Matrix inverse for general square matrix.*
  - `template<typename T >`  
`Matrix< T > Mtx::inv (const Matrix< T > &A)`  
*Matrix inverse (universal).*
  - `template<typename T >`  
`Matrix< T > Mtx::pinv (const Matrix< T > &A)`  
*Moore-Penrose pseudo-inverse.*
  - `template<typename T >`  
`T Mtx::trace (const Matrix< T > &A)`  
*Matrix trace.*
  - `template<typename T >`  
`double Mtx::cond (const Matrix< T > &A)`  
*Condition number of a matrix.*
  - `template<typename T , bool is_upper = false>`  
`Matrix< T > Mtx::chol (const Matrix< T > &A)`  
*Cholesky decomposition.*
  - `template<typename T >`  
`Matrix< T > Mtx::cholinv (const Matrix< T > &A)`  
*Inverse of Cholesky decomposition.*
  - `template<typename T >`  
`LDL_result< T > Mtx::ldl (const Matrix< T > &A)`  
*LDL decomposition.*
  - `template<typename T >`  
`QR_result< T > Mtx::qr_red_gs (const Matrix< T > &A)`  
*Reduced QR decomposition based on Gram-Schmidt method.*
  - `template<typename T >`  
`Matrix< T > Mtx::householder_reflection (const Matrix< T > &a)`  
*Generate Householder reflection.*
  - `template<typename T >`  
`QR_result< T > Mtx::qr_householder (const Matrix< T > &A, bool calculate_Q=true)`  
*QR decomposition based on Householder method.*
  - `template<typename T >`  
`QR_result< T > Mtx::qr (const Matrix< T > &A, bool calculate_Q=true)`  
*QR decomposition.*
  - `template<typename T >`  
`Hessenberg_result< T > Mtx::hessenberg (const Matrix< T > &A, bool calculate_Q=true)`  
*Hessenberg decomposition.*
  - `template<typename T >`  
`std::complex< T > Mtx::wilkinson_shift (const Matrix< std::complex< T > > &H, T tol=1e-10)`  
*Wilkinson's shift for complex eigenvalues.*

- `template<typename T>`  
`Eigenvalues_result< T> Mtx::eigenvalues (const Matrix< std::complex< T> > &A, T tol=1e-12, unsigned max_iter=100)`  
*Matrix eigenvalues of complex matrix.*
- `template<typename T>`  
`Eigenvalues_result< T> Mtx::eigenvalues (const Matrix< T> &A, T tol=1e-12, unsigned max_iter=100)`  
*Matrix eigenvalues of real matrix.*
- `template<typename T>`  
`Matrix< T> Mtx::solve_triu (const Matrix< T> &U, const Matrix< T> &B)`  
*Solves the upper triangular system.*
- `template<typename T>`  
`Matrix< T> Mtx::solve_tril (const Matrix< T> &L, const Matrix< T> &B)`  
*Solves the lower triangular system.*
- `template<typename T>`  
`Matrix< T> Mtx::solve_square (const Matrix< T> &A, const Matrix< T> &B)`  
*Solves the square system.*
- `template<typename T>`  
`Matrix< T> Mtx::solve_posdef (const Matrix< T> &A, const Matrix< T> &B)`  
*Solves the positive definite (Hermitian) system.*

## 6.1.1 Function Documentation

### 6.1.1.1 add() [1/2]

```
template<typename T, bool transpose_first = false, bool transpose_second = false>
Matrix< T> Mtx::add (
    const Matrix< T> & A,
    const Matrix< T> & B )
```

Matrix addition.

Performs addition of two matrices.

This function supports template parameterization of input matrix transposition, providing better efficiency than in case of using `Mtx::ctranspose()` function due to zero-copy operation. In case of complex matrices, conjugate (Hermitian) transpose is used.

#### Template Parameters

|                         |   |
|-------------------------|---|
| <i>transpose_first</i>  | if set to true, the left-side input matrix will be transposed during operation  |
| <i>transpose_second</i> | if set to true, the right-side input matrix will be transposed during operation |

#### Parameters

|          |  |
|----------|--|
| <i>A</i> | left-side matrix of size $N \times M$ (after transposition)  |
| <i>B</i> | right-side matrix of size $N \times M$ (after transposition) |

#### Returns

output matrix of size  $N \times M$

References [Mtx::add\(\)](#), [Mtx::cconj\(\)](#), [Mtx::Matrix< T >::cols\(\)](#), and [Mtx::Matrix< T >::rows\(\)](#).

Referenced by [Mtx::add\(\)](#), [Mtx::add\(\)](#), [Mtx::operator+\(\)](#), [Mtx::operator+\(\)](#), and [Mtx::operator+\(\)](#).

### 6.1.1.2 add() [2/2]

```
template<typename T >
Matrix< T > Mtx::add (
    const Matrix< T > & A,
    T s )
```

Addition of scalar to matrix.

Adds a scalar  $s$  from each element of the input matrix. This method does not modify the input matrix but creates a copy.

References [Mtx::add\(\)](#), [Mtx::Matrix< T >::cols\(\)](#), [Mtx::Matrix< T >::numel\(\)](#), and [Mtx::Matrix< T >::rows\(\)](#).

### 6.1.1.3 adj()

```
template<typename T >
Matrix< T > Mtx::adj (
    const Matrix< T > & A )
```

Adjugate matrix.

Calculates adjugate of the matrix being the transpose of its cofactor matrix.

More information: [https://en.wikipedia.org/wiki/Adjugate\\_matrix](https://en.wikipedia.org/wiki/Adjugate_matrix)

#### Exceptions

|                                 |                                     |
|---------------------------------|-------------------------------------|
| <code>std::runtime_error</code> | when the input matrix is not square |
|---------------------------------|-------------------------------------|

References [Mtx::adj\(\)](#), [Mtx::cofactor\(\)](#), [Mtx::Matrix< T >::cols\(\)](#), [Mtx::det\(\)](#), [Mtx::Matrix< T >::issquare\(\)](#), and [Mtx::Matrix< T >::rows\(\)](#).

Referenced by [Mtx::adj\(\)](#).

### 6.1.1.4 cconj()

```
template<typename T , typename std::enable_if<!is_complex< T >::value, int >::type = 0>
T Mtx::cconj (
    T x ) [inline]
```

Complex conjugate helper.

Helper function to allow for generalization of code for complex and real types.

For real numbers, this function returns the input argument unchanged.

For complex numbers, this function calls `std::conj`.

References [Mtx::cconj\(\)](#).

Referenced by [Mtx::add\(\)](#), [Mtx::cconj\(\)](#), [Mtx::chol\(\)](#), [Mtx::cholinv\(\)](#), [Mtx::Matrix< T >::ctranspose\(\)](#), [Mtx::ldl\(\)](#), [Mtx::mult\(\)](#), [Mtx::mult\(\)](#), [Mtx::mult\(\)](#), [Mtx::mult\\_hadamard\(\)](#), [Mtx::qr\\_red\\_gs\(\)](#), and [Mtx::subtract\(\)](#).

### 6.1.1.5 chol()

```
template<typename T , bool is_upper = false>
Matrix< T > Mtx::chol (
    const Matrix< T > & A )
```

Cholesky decomposition.

The Cholesky decomposition of a Hermitian positive-definite matrix  $A$  is a decomposition of the form  $A = LL^H$ , where  $L$  is a lower triangular matrix with real and positive diagonal entries, and  $^H$  denotes the conjugate transpose. Alternatively, the decomposition can be computed as  $A = U^H U$  with  $U$  being upper-triangular matrix. Selection between lower and upper triangular factor can be done via template parameter.

Input matrix must be square and Hermitian. If the matrix is not Hermitian positive-definite or is ill-conditioned, the result may be unreliable. Only the lower-triangular or upper-triangular and diagonal elements of the input matrix are used for calculations. No checking is performed to verify if the input matrix is Hermitian.

More information: [https://en.wikipedia.org/wiki/Cholesky\\_decomposition](https://en.wikipedia.org/wiki/Cholesky_decomposition)

#### Template Parameters

|                 |  |
|-----------------|--|
| <i>is_upper</i> | if set to true, the result is provided for upper-triangular factor $U$ . If set to false, the result is provided for lower-triangular factor $L$ . |
|-----------------|--|

#### Exceptions

|                                  |  |
|----------------------------------|--|
| <i>std::runtime_error</i>        | when the input matrix is not square  |
| <i>singular_matrix_exception</i> | when the input matrix is singular (detected as division by 0 during computation) |

References [Mtx::conj\(\)](#), [Mtx::chol\(\)](#), [Mtx::Matrix< T >::issquare\(\)](#), [Mtx::Matrix< T >::rows\(\)](#), [Mtx::tril\(\)](#), and [Mtx::triu\(\)](#).

Referenced by [Mtx::chol\(\)](#), and [Mtx::solve\\_posdef\(\)](#).

### 6.1.1.6 cholinv()

```
template<typename T >
Matrix< T > Mtx::cholinv (
    const Matrix< T > & A )
```

Inverse of Cholesky decomposition.

This function directly calculates the inverse of Cholesky decomposition  $L^{-1}$  such that  $A = LL^H$ .

See [Mtx::chol\(\)](#) for reference on Cholesky decomposition.

Input matrix must be square. If the matrix is not Hermitian positive-definite or is ill-conditioned, the result may be unreliable.

More information: [https://en.wikipedia.org/wiki/Cholesky\\_decomposition](https://en.wikipedia.org/wiki/Cholesky_decomposition)

#### Exceptions

|                                  |  |
|----------------------------------|--|
| <i>std::runtime_error</i>        | when the input matrix is not square  |
| <i>singular_matrix_exception</i> | when the input matrix is singular (detected as division by 0 during computation) |

References [Mtx::cconj\(\)](#), [Mtx::cholinv\(\)](#), [Mtx::Matrix< T >::issquare\(\)](#), and [Mtx::Matrix< T >::rows\(\)](#).

Referenced by [Mtx::cholinv\(\)](#), and [Mtx::inv\\_posdef\(\)](#).

#### 6.1.1.7 circshift()

```
template<typename T >
Matrix< T > Mtx::circshift (
    const Matrix< T > & A,
    int row_shift,
    int col_shift )
```

Circular shift.

Returns a matrix that is created by shifting the columns and rows of an input matrix in a circular manner. If the specified shift factor is a positive value, columns of the matrix are shifted towards right or rows are shifted towards the bottom. A negative value may be used to apply shifts in opposite directions.

##### Parameters

|                  |                     |
|------------------|---------------------|
| <i>A</i>         | matrix              |
| <i>row_shift</i> | row shift factor    |
| <i>col_shift</i> | column shift factor |

##### Returns

matrix inverse

References [Mtx::circshift\(\)](#), [Mtx::Matrix< T >::cols\(\)](#), and [Mtx::Matrix< T >::rows\(\)](#).

Referenced by [Mtx::circshift\(\)](#).

#### 6.1.1.8 circulant() [1/2]

```
template<typename T >
Matrix< T > Mtx::circulant (
    const std::vector< T > & v ) [inline]
```

Circulant matrix from std::vector.

Constructs a circulant matrix, whose the elements of the first column are set to the elements stored in the std::vector *v*. Size of the matrix is equal to the vector size.

##### Parameters

|          |                  |
|----------|------------------|
| <i>v</i> | vector with data |
|----------|------------------|

##### Returns

circulant matrix

References [Mtx::circulant\(\)](#).



### 6.1.1.9 `circulant()` [2/2]

```
template<typename T >
Matrix< T > Mtx::circulant (
    const T * array,
    unsigned n )
```

Circulant matrix from array.

Constructs a circulant matrix of size  $n \times n$  by taking the elements from *array* as the first column.

#### Parameters

|              |   |
|--------------|---|
| <i>array</i> | pointer to the first element of the array where the elements of the first column are stored |
| <i>n</i>     | size of the matrix to be constructed. Also, a number of elements stored in <i>array</i>     |

#### Returns

circulant matrix

References [Mtx::circulant\(\)](#).

Referenced by [Mtx::circulant\(\)](#), and [Mtx::circulant\(\)](#).

### 6.1.1.10 `cofactor()`

```
template<typename T >
Matrix< T > Mtx::cofactor (
    const Matrix< T > & A,
    unsigned p,
    unsigned q )
```

Cofactor matrix.

Calculates first minor of the matrix by deleting row *p* and column *q*. Note that this function does not include sign change required by cofactor calculation.

More information: [https://en.wikipedia.org/wiki/Cofactor\\_\(linear\\_algebra\)](https://en.wikipedia.org/wiki/Cofactor_(linear_algebra))

#### Parameters

|          |   |
|----------|---|
| <i>A</i> | input square matrix                       |
| <i>p</i> | row to be deleted in the output matrix    |
| <i>q</i> | column to be deleted in the output matrix |

#### Exceptions

|                                 |   |
|---------------------------------|---|
| <code>std::runtime_error</code> | when the input matrix is not square                               |
| <code>std::out_of_range</code>  | when row index <i>p</i> or column index <i>q</i> are out of range |
| <code>std::runtime_error</code> | when input matrix <i>A</i> has less than 2 rows                   |

References [Mtx::cofactor\(\)](#), [Mtx::Matrix< T >::cols\(\)](#), [Mtx::Matrix< T >::issquare\(\)](#), and [Mtx::Matrix< T >::rows\(\)](#).

Referenced by [Mtx::adj\(\)](#), and [Mtx::cofactor\(\)](#).

#### 6.1.1.11 concatenate\_horizontal()

```
template<typename T >
Matrix< T > Mtx::concatenate_horizontal (
    const Matrix< T > & A,
    const Matrix< T > & B )
```

Horizontal matrix concatenation.

Concatenates two input matrices  $A$  and  $B$  horizontally to form a concatenated matrix  $C = [A|B]$ .

##### Exceptions

|                                 |  |
|---------------------------------|--|
| <code>std::runtime_error</code> | when the number of rows in $A$ and $B$ is not equal. |
|---------------------------------|--|

References [Mtx::Matrix< T >::cols\(\)](#), [Mtx::concatenate\\_horizontal\(\)](#), and [Mtx::Matrix< T >::rows\(\)](#).

Referenced by [Mtx::concatenate\\_horizontal\(\)](#).

#### 6.1.1.12 concatenate\_vertical()

```
template<typename T >
Matrix< T > Mtx::concatenate_vertical (
    const Matrix< T > & A,
    const Matrix< T > & B )
```

Vertical matrix concatenation.

Concatenates two input matrices  $A$  and  $B$  vertically to form a concatenated matrix  $C = [A|B]^T$ .

##### Exceptions

|                                 |   |
|---------------------------------|---|
| <code>std::runtime_error</code> | when the number of columns in $A$ and $B$ is not equal. |
|---------------------------------|---|

References [Mtx::Matrix< T >::cols\(\)](#), [Mtx::concatenate\\_vertical\(\)](#), and [Mtx::Matrix< T >::rows\(\)](#).

Referenced by [Mtx::concatenate\\_vertical\(\)](#).

#### 6.1.1.13 cond()

```
template<typename T >
double Mtx::cond (
    const Matrix< T > & A )
```

Condition number of a matrix.

Calculates condition number of a matrix. The condition number of a matrix measures sensitivity of a solution for system of linear equations to errors in the input data. The condition number is calculated by:

$$\text{cond} = \text{norm}(A) * \text{norm}(A^{-1})$$

Frobenius norm is used for the sake of calculations. See [Mtx::norm\\_fro\(\)](#).

References [Mtx::cond\(\)](#), [Mtx::inv\(\)](#), and [Mtx::norm\\_fro\(\)](#).

Referenced by [Mtx::cond\(\)](#).

#### 6.1.1.14 csign()

```
template<typename T , typename std::enable_if<!is_complex< T >::value, int >::type = 0>
T Mtx::csign (
    T x ) [inline]
```

Complex sign helper.

Helper function to allow for generalization of code for complex and real types.

For real numbers, this function returns sign bit, i.e., 1 when the value is non-negative and -1 otherwise.

For complex numbers, this function calculates  $e^{i \cdot \arg(x)}$ .

References [Mtx::csign\(\)](#).

Referenced by [Mtx::csign\(\)](#), and [Mtx::householder\\_reflection\(\)](#).

#### 6.1.1.15 ctranspose()

```
template<typename T >
Matrix< T > Mtx::ctranspose (
    const Matrix< T > & A ) [inline]
```

Transpose a complex matrix.

Returns a matrix that is a conjugate (Hermitian) transposition of an input matrix.

Conjugate transpose applies a conjugate operation to all elements in addition to matrix transposition.

References [Mtx::Matrix< T >::ctranspose\(\)](#), and [Mtx::ctranspose\(\)](#).

Referenced by [Mtx::ctranspose\(\)](#).

#### 6.1.1.16 det()

```
template<typename T >
T Mtx::det (
    const Matrix< T > & A )
```

Matrix determinant.

Calculates determinant of a square matrix. If the size of the matrix is smaller than 4, the determinant is calculated using hard-coded formulas. For matrix sizes equal to 4 and more, determinant is calculated recursively using Laplace expansion.

More information: <https://en.wikipedia.org/wiki/Determinant>

## Exceptions

|                                 |                                     |
|---------------------------------|-------------------------------------|
| <code>std::runtime_error</code> | when the input matrix is not square |
|---------------------------------|-------------------------------------|

References [Mtx::det\(\)](#), [Mtx::det\\_lu\(\)](#), [Mtx::Matrix< T >::issquare\(\)](#), and [Mtx::Matrix< T >::rows\(\)](#).

Referenced by [Mtx::adj\(\)](#), [Mtx::det\(\)](#), and [Mtx::inv\(\)](#).

**6.1.1.17 det\_lu()**

```
template<typename T >
T Mtx::det_lu (
    const Matrix< T > & A )
```

Matrix determinant from on LU decomposition.

Calculates the determinant of a matrix using LU decomposition with pivoting.

Note that determinant is calculated as a product:  $\det(L) \cdot \det(U) \cdot \det(P)$ , where determinants of  $L$  and  $U$  are calculated as the product of their diagonal elements, when the determinant of  $P$  is either 1 or -1 depending on the number of row swaps performed during the pivoting process.

More information: <https://en.wikipedia.org/wiki/Determinant>

## Exceptions

|                                 |                                     |
|---------------------------------|-------------------------------------|
| <code>std::runtime_error</code> | when the input matrix is not square |
|---------------------------------|-------------------------------------|

References [Mtx::det\\_lu\(\)](#), [Mtx::Matrix< T >::issquare\(\)](#), [Mtx::lup\(\)](#), [Mtx::Matrix< T >::resize\(\)](#), and [Mtx::Matrix< T >::rows\(\)](#).

Referenced by [Mtx::det\(\)](#), and [Mtx::det\\_lu\(\)](#).

**6.1.1.18 diag()** [1/3]

```
template<typename T >
std::vector< T > Mtx::diag (
    const Matrix< T > & A )
```

Diagonal extraction.

Store diagonal elements of a square matrix in `std::vector`.

## Parameters

|                |               |
|----------------|---------------|
| <code>A</code> | square matrix |
|----------------|---------------|

## Returns

vector of diagonal elements

## Exceptions

|                                 |                                     |
|---------------------------------|-------------------------------------|
| <code>std::runtime_error</code> | when the input matrix is not square |
|---------------------------------|-------------------------------------|

References [Mtx::diag\(\)](#), [Mtx::Matrix< T >::issquare\(\)](#), [Mtx::Matrix< T >::resize\(\)](#), and [Mtx::Matrix< T >::rows\(\)](#).

**6.1.1.19 diag() [2/3]**

```
template<typename T >
Matrix< T > Mtx::diag (
    const std::vector< T > & v ) [inline]
```

Diagonal matrix from `std::vector`.

Constructs a diagonal matrix, whose diagonal elements are set to the elements stored in the `std::vector` `v`. Size of the matrix is equal to the vector size.

## Parameters

|                |                             |
|----------------|-----------------------------|
| <code>v</code> | vector of diagonal elements |
|----------------|-----------------------------|

## Returns

diagonal matrix

References [Mtx::diag\(\)](#).

**6.1.1.20 diag() [3/3]**

```
template<typename T >
Matrix< T > Mtx::diag (
    const T * array,
    size_t n )
```

Diagonal matrix from array.

Constructs a diagonal matrix of size  $n \times n$ , whose diagonal elements are set to the elements stored in the `array`.

## Parameters

|                    |   |
|--------------------|---|
| <code>array</code> | pointer to the first element of the array where the diagonal elements are stored              |
| <code>n</code>     | size of the matrix to be constructed. Also, a number of elements stored in <code>array</code> |

## Returns

diagonal matrix

References [Mtx::diag\(\)](#).

Referenced by [Mtx::diag\(\)](#), [Mtx::diag\(\)](#), [Mtx::diag\(\)](#), and [Mtx::eigenvalues\(\)](#).

### 6.1.1.21 div()

```
template<typename T >
Matrix< T > Mtx::div (
    const Matrix< T > & A,
    T s )
```

Division of matrix by scalar.

Divides each element of the input matrix by a scalar  $s$ . This method does not modify the input matrix but creates a copy.

References [Mtx::Matrix< T >::cols\(\)](#), [Mtx::div\(\)](#), [Mtx::Matrix< T >::numel\(\)](#), and [Mtx::Matrix< T >::rows\(\)](#).

Referenced by [Mtx::div\(\)](#), and [Mtx::operator/\(\)](#).

### 6.1.1.22 eigenvalues() [1/2]

```
template<typename T >
Eigenvalues_result< T > Mtx::eigenvalues (
    const Matrix< std::complex< T > > & A,
    T tol = 1e-12,
    unsigned max_iter = 100 )
```

Matrix eigenvalues of complex matrix.

Computes eigenvalues of input square matrix using the QR method with shifts.

#### Parameters

|                 |  |
|-----------------|--|
| <i>A</i>        | input complex matrix to be decomposed            |
| <i>tol</i>      | numerical precision tolerance for stop condition |
| <i>max_iter</i> | maximum number of iterations                     |

#### Returns

structure containing the result and status of eigenvalue calculation

#### Exceptions

|                           |                                     |
|---------------------------|-------------------------------------|
| <i>std::runtime_error</i> | when the input matrix is not square |
|---------------------------|-------------------------------------|

References [Mtx::diag\(\)](#), [Mtx::eigenvalues\(\)](#), [Mtx::hessenberg\(\)](#), [Mtx::Matrix< T >::issquare\(\)](#), [Mtx::qr\(\)](#), [Mtx::Matrix< T >::rows\(\)](#), and [Mtx::wilkinson\\_shift\(\)](#).

Referenced by [Mtx::eigenvalues\(\)](#), and [Mtx::eigenvalues\(\)](#).

### 6.1.1.23 eigenvalues() [2/2]

```
template<typename T >
Eigenvalues_result< T > Mtx::eigenvalues (
```

```
const Matrix< T > & A,
T tol = 1e-12,
unsigned max_iter = 100 )
```

Matrix eigenvalues of real matrix.

Computes eigenvalues of input square matrix using the QR method with shifts.

#### Parameters

|                 |  |
|-----------------|--|
| <i>A</i>        | input real matrix to be decomposed               |
| <i>tol</i>      | numerical precision tolerance for stop condition |
| <i>max_iter</i> | maximum number of iterations                     |

#### Returns

structure containing the result and status of eigenvalue calculation

References [Mtx::eigenvalues\(\)](#), and [Mtx::make\\_complex\(\)](#).

#### 6.1.1.24 eye()

```
template<typename T >
Matrix< T > Mtx::eye (
    unsigned n )
```

Identity matrix.

Construct a square identity matrix. In case of complex datatype, the diagonal elements are set to  $1 + 0i$ .

#### Parameters

|          |  |
|----------|--|
| <i>n</i> | size of the square matrix (the first and the second dimension) |
|----------|--|

#### Returns

zeros matrix

References [Mtx::eye\(\)](#).

Referenced by [Mtx::eye\(\)](#).

#### 6.1.1.25 foreach\_elem()

```
template<typename T >
void Mtx::foreach_elem (
    Matrix< T > & A,
    std::function< T(T)> func ) [inline]
```

Applies custom function element-wise in-place.

Applies specified function *func* to all elements of the input matrix.

This function applies operation to the elements in-place (zero-copy). In order to apply the function to the copy of the matrix without modifying the input one, use [Mtx::foreach\\_elem\\_copy\(\)](#).

## Parameters

|             |   |
|-------------|---|
| <i>A</i>    | input matrix to be modified   |
| <i>func</i> | function to be applied element-wise to <i>A</i> . It inputs one variable of template type <i>T</i> and returns variable of the same type. |

References [Mtx::foreach\\_elem\(\)](#), and [Mtx::Matrix< T >::numel\(\)](#).

Referenced by [Mtx::foreach\\_elem\(\)](#), and [Mtx::foreach\\_elem\\_copy\(\)](#).

#### 6.1.1.26 foreach\_elem\_copy()

```
template<typename T >
Matrix< T > Mtx::foreach_elem_copy (
    const Matrix< T > & A,
    std::function< T(T)> func ) [inline]
```

Applies custom function element-wise with matrix copy.

Applies the specified function *func* to all elements of the input matrix.

This function applies operation to the copy of the input matrix. For in-place (zero-copy) operation, use [Mtx::foreach\\_elem\(\)](#).

## Parameters

|             |  |
|-------------|--|
| <i>A</i>    | input matrix   |
| <i>func</i> | function to be applied element-wise to <i>A</i> . It inputs one variable of template type <i>T</i> and returns variable of the same type |

## Returns

output matrix whose elements were modified by the function *func*

References [Mtx::foreach\\_elem\(\)](#), and [Mtx::foreach\\_elem\\_copy\(\)](#).

Referenced by [Mtx::foreach\\_elem\\_copy\(\)](#).

#### 6.1.1.27 hessenberg()

```
template<typename T >
Hessenberg_result< T > Mtx::hessenberg (
    const Matrix< T > & A,
    bool calculate_Q = true )
```

Hessenberg decomposition.

Finds the Hessenberg decomposition of  $A = QHQ^*$ . Hessenberg matrix  $H$  has zero entries below the first subdiagonal. More information: [https://en.wikipedia.org/wiki/Hessenberg\\_matrix](https://en.wikipedia.org/wiki/Hessenberg_matrix)



## Parameters

|                    |  |
|--------------------|--|
| <i>A</i>           | input matrix to be decomposed          |
| <i>calculate_Q</i> | indicates if <i>Q</i> to be calculated |

## Returns

structure encapsulating calculated *H* and *Q*. *Q* is calculated only when *calculate\_Q* = True.

## Exceptions

|                           |                                     |
|---------------------------|-------------------------------------|
| <i>std::runtime_error</i> | when the input matrix is not square |
|---------------------------|-------------------------------------|

References [Mtx::Matrix< T >::get\\_submatrix\(\)](#), [Mtx::hessenberg\(\)](#), [Mtx::householder\\_reflection\(\)](#), [Mtx::Matrix< T >::issquare\(\)](#), and [Mtx::Matrix< T >::rows\(\)](#).

Referenced by [Mtx::eigenvalues\(\)](#), and [Mtx::hessenberg\(\)](#).

## 6.1.1.28 householder\_reflection()

```
template<typename T >
Matrix< T > Mtx::householder_reflection (
    const Matrix< T > & a )
```

Generate Householder reflection.

Generates Householder reflection for a given vector. The function returns vector *v* normalized to square root of 2.

## Parameters

|          |                                    |
|----------|------------------------------------|
| <i>a</i> | column vector of size <i>N</i> x 1 |
|----------|------------------------------------|

## Returns

column vector with Householder reflection of *a*

References [Mtx::Matrix< T >::cols\(\)](#), [Mtx::csign\(\)](#), [Mtx::householder\\_reflection\(\)](#), [Mtx::norm\\_fro\(\)](#), and [Mtx::Matrix< T >::numel\(\)](#).

Referenced by [Mtx::hessenberg\(\)](#), [Mtx::householder\\_reflection\(\)](#), and [Mtx::qr\\_householder\(\)](#).

## 6.1.1.29 imag()

```
template<typename T >
Matrix< T > Mtx::imag (
    const Matrix< std::complex< T > > & C )
```

Get imaginary part of complex matrix.

Constructs a matrix of real type from `std::complex` matrix by taking its imaginary part.

References [Mtx::Matrix< T >::cols\(\)](#), [Mtx::imag\(\)](#), [Mtx::Matrix< T >::numel\(\)](#), and [Mtx::Matrix< T >::rows\(\)](#).

Referenced by [Mtx::imag\(\)](#).

### 6.1.1.30 inv()

```
template<typename T >
Matrix< T > Mtx::inv (
    const Matrix< T > & A )
```

Matrix inverse (universal).

Calculates an inverse of a square matrix. If the size of the matrix is smaller than 4, inverse is calculated using hard-coded formulas. For matrix sizes equal to 4 and more, determinant is calculated recursively using Gauss-Jordan elimination.

If the inverse doesn't exist, e.g., because the input matrix was singular, an empty matrix is returned.

More information: [https://en.wikipedia.org/wiki/Gaussian\\_elimination](https://en.wikipedia.org/wiki/Gaussian_elimination)

#### Exceptions

|                                  |  |
|----------------------------------|--|
| <i>std::runtime_error</i>        | when the input matrix is not square  |
| <i>singular_matrix_exception</i> | when the input matrix is singular (detected as division by 0 during computation) |

References [Mtx::det\(\)](#), [Mtx::inv\(\)](#), [Mtx::inv\\_square\(\)](#), [Mtx::Matrix< T >::issquare\(\)](#), [Mtx::Matrix< T >::numel\(\)](#), and [Mtx::Matrix< T >::rows\(\)](#).

Referenced by [Mtx::cond\(\)](#), and [Mtx::inv\(\)](#).

### 6.1.1.31 inv\_gauss\_jordan()

```
template<typename T >
Matrix< T > Mtx::inv_gauss_jordan (
    const Matrix< T > & A )
```

Matrix inverse using Gauss-Jordan elimination.

Calculates an inverse of a square matrix recursively using Gauss-Jordan elimination.

If the inverse doesn't exist, e.g., because the input matrix was singular, an empty matrix is returned.

More information: [https://en.wikipedia.org/wiki/Gaussian\\_elimination](https://en.wikipedia.org/wiki/Gaussian_elimination)

Using this function is generally not recommended, please refer to [Mtx::inv\(\)](#) instead.

#### Exceptions

|                                  |                                     |
|----------------------------------|-------------------------------------|
| <i>std::runtime_error</i>        | when the input matrix is not square |
| <i>singular_matrix_exception</i> | when input matrix is singular       |

References [Mtx::inv\\_gauss\\_jordan\(\)](#), [Mtx::Matrix< T >::issquare\(\)](#), and [Mtx::Matrix< T >::rows\(\)](#).

Referenced by [Mtx::inv\\_gauss\\_jordan\(\)](#).

### 6.1.1.32 inv\_posdef()

```
template<typename T >
Matrix< T > Mtx::inv_posdef (
    const Matrix< T > & A )
```

Matrix inverse for Hermitian positive-definite matrix.

Calculates an inverse of symmetric (for real input) or Hermitian (for complex input) positive definite matrix using Cholesky decomposition.

This function provides more optimal performance than `Mtx::inv()` for symmetric matrices. However, validation of input matrix structure is not performed. It is up to the user to decide when this function can be used and, if needed, perform required validations.

More information: [https://en.wikipedia.org/wiki/Gaussian\\_elimination](https://en.wikipedia.org/wiki/Gaussian_elimination)

#### Exceptions

|  |  |
|--|--|
| <code>std::runtime_error</code>        | when the input matrix is not square  |
| <code>singular_matrix_exception</code> | when the input matrix is singular (detected as division by 0 during computation) |

References `Mtx::cholinv()`, and `Mtx::inv_posdef()`.

Referenced by `Mtx::inv_posdef()`, and `Mtx::pinv()`.

#### 6.1.1.33 inv\_square()

```
template<typename T >
Matrix< T > Mtx::inv_square (
    const Matrix< T > & A )
```

Matrix inverse for general square matrix.

Calculates an inverse of square matrix using matrix.

This function provides more optimal performance than `Mtx::inv()` for upper triangular matrices. However, validation of input matrix structure is not performed. It is up to the user to decide when this function can be used and, if needed, perform required validations.

#### Exceptions

|  |  |
|--|--|
| <code>std::runtime_error</code>        | when the input matrix is not square  |
| <code>singular_matrix_exception</code> | when the input matrix is singular (detected as division by 0 during computation) |

References `Mtx::inv_square()`, `Mtx::inv_tril()`, `Mtx::inv_triu()`, `Mtx::Matrix< T >::issquare()`, `Mtx::lup()`, and `Mtx::permute_rows()`.

Referenced by `Mtx::inv()`, and `Mtx::inv_square()`.

#### 6.1.1.34 inv\_tril()

```
template<typename T >
Matrix< T > Mtx::inv_tril (
    const Matrix< T > & A )
```

Matrix inverse for lower triangular matrix.

Calculates an inverse of lower triangular matrix.

This function provides more optimal performance than [Mtx::inv\(\)](#) for lower triangular matrices. However, validation of triangular input matrix structure is not performed. It is up to the user to decide when this function can be used and, if needed, perform required validations.

#### Exceptions

|                                  |  |
|----------------------------------|--|
| <i>std::runtime_error</i>        | when the input matrix is not square  |
| <i>singular_matrix_exception</i> | when the input matrix is singular (detected as division by 0 during computation) |

References [Mtx::inv\\_tril\(\)](#), [Mtx::Matrix< T >::issquare\(\)](#), and [Mtx::Matrix< T >::rows\(\)](#).

Referenced by [Mtx::inv\\_square\(\)](#), and [Mtx::inv\\_tril\(\)](#).

#### 6.1.1.35 inv\_triu()

```
template<typename T >
Matrix< T > Mtx::inv_triu (
    const Matrix< T > & A )
```

Matrix inverse for upper triangular matrix.

Calculates an inverse of upper triangular matrix.

This function provides more optimal performance than [Mtx::inv\(\)](#) for upper triangular matrices. However, validation of triangular input matrix structure is not performed. It is up to the user to decide when this function can be used and, if needed, perform required validations.

#### Exceptions

|                                  |  |
|----------------------------------|--|
| <i>std::runtime_error</i>        | when the input matrix is not square  |
| <i>singular_matrix_exception</i> | when the input matrix is singular (detected as division by 0 during computation) |

References [Mtx::inv\\_triu\(\)](#), [Mtx::Matrix< T >::issquare\(\)](#), and [Mtx::Matrix< T >::rows\(\)](#).

Referenced by [Mtx::inv\\_square\(\)](#), and [Mtx::inv\\_triu\(\)](#).

#### 6.1.1.36 ishess()

```
template<typename T >
bool Mtx::ishess (
    const Matrix< T > & A )
```

Hessenberg matrix check.

Return true if *A* is an upper Hessenberg matrix, i.e., it is square and has only zero entries below the first subdiagonal. This function uses hard decision for equality check.

References [Mtx::ishess\(\)](#), [Mtx::Matrix< T >::issquare\(\)](#), and [Mtx::Matrix< T >::rows\(\)](#).

Referenced by [Mtx::ishess\(\)](#).

**6.1.1.37 istril()**

```
template<typename T >
bool Mtx::istril (
    const Matrix< T > & A )
```

Lower triangular matrix check.

Return true if A is a lower triangular matrix, i.e., when it has nonzero entries only on the main diagonal and below. This function uses hard decision for equality check.

References [Mtx::Matrix< T >::cols\(\)](#), [Mtx::istril\(\)](#), and [Mtx::Matrix< T >::rows\(\)](#).

Referenced by [Mtx::istril\(\)](#).

**6.1.1.38 istriu()**

```
template<typename T >
bool Mtx::istriu (
    const Matrix< T > & A )
```

Lower triangular matrix check.

Return true if A is a lower triangular matrix, i.e., when it has nonzero entries only on the main diagonal and below. This function uses hard decision for equality check.

References [Mtx::Matrix< T >::cols\(\)](#), [Mtx::istriu\(\)](#), and [Mtx::Matrix< T >::rows\(\)](#).

Referenced by [Mtx::istriu\(\)](#).

**6.1.1.39 kron()**

```
template<typename T >
Matrix< T > Mtx::kron (
    const Matrix< T > & A,
    const Matrix< T > & B )
```

Kronecker product.

Form the Kronecker product of two matrices. Kronecker product is defined block by block as  $C = [A(i, j) \cdot B]$ .

More information: [https://en.wikipedia.org/wiki/Kronecker\\_product](https://en.wikipedia.org/wiki/Kronecker_product)

References [Mtx::Matrix< T >::cols\(\)](#), [Mtx::kron\(\)](#), and [Mtx::Matrix< T >::rows\(\)](#).

Referenced by [Mtx::kron\(\)](#).

**6.1.1.40 ldl()**

```
template<typename T >
LDL_result< T > Mtx::ldl (
    const Matrix< T > & A )
```

LDL decomposition.

The LDL decomposition of a Hermitian positive-definite matrix A, is a decomposition of the form:

$$A = LDL^H$$

where L is a lower unit triangular matrix with ones at the diagonal,  $L^H$  denotes the conjugate transpose of L, and D denotes diagonal matrix.

Input matrix must be square. If the matrix is not Hermitian positive-definite or is ill-conditioned, the result may be unreliable.

More information: [https://en.wikipedia.org/wiki/Cholesky\\_decomposition#LDL\\_decomposition](https://en.wikipedia.org/wiki/Cholesky_decomposition#LDL_decomposition)

**Parameters**

|          |   |
|----------|---|
| <i>A</i> | input positive-definite matrix to be decomposed |
|----------|---|

**Returns**

structure encapsulating calculated  $L$  and  $D$

**Exceptions**

|                                  |  |
|----------------------------------|--|
| <i>std::runtime_error</i>        | when the input matrix is not square  |
| <i>singular_matrix_exception</i> | when the input matrix is singular (detected as division by 0 during computation) |

References [Mtx::cconj\(\)](#), [Mtx::Matrix< T >::issquare\(\)](#), [Mtx::ldl\(\)](#), and [Mtx::Matrix< T >::rows\(\)](#).

Referenced by [Mtx::ldl\(\)](#).

**6.1.1.41 lu()**

```
template<typename T >
LU_result< T > Mtx::lu (
    const Matrix< T > & A )
```

LU decomposition.

Performs LU factorization of the matrix into the the product of a lower triangular matrix  $L$  and an upper triangular matrix  $U$ .

This function implements LU factorization without pivoting. Use [Mtx::lup\(\)](#) if pivoting is required.

More information: [https://en.wikipedia.org/wiki/LU\\_decomposition](https://en.wikipedia.org/wiki/LU_decomposition)

**Parameters**

|          |                                      |
|----------|--------------------------------------|
| <i>A</i> | input square matrix to be decomposed |
|----------|--------------------------------------|

**Returns**

structure containing calculated  $L$  and  $U$  matrices

References [Mtx::Matrix< T >::cols\(\)](#), [Mtx::lu\(\)](#), [Mtx::Matrix< T >::numel\(\)](#), and [Mtx::Matrix< T >::rows\(\)](#).

Referenced by [Mtx::lu\(\)](#).

**6.1.1.42 lup()**

```
template<typename T >
LUP_result< T > Mtx::lup (
    const Matrix< T > & A )
```

LU decomposition with pivoting.

Performs LU factorization with partial pivoting, employing column permutations.

The input matrix can be re-created from  $L$ ,  $U$  and  $P$  using `permute_cols()` accordingly:

```
auto r = lup(A);
auto A_rec = permute_cols(r.L * r.U, r.P);
```

More information: [https://en.wikipedia.org/wiki/LU\\_decomposition#LU\\_factorization\\_with\\_partial\\_pivoting](https://en.wikipedia.org/wiki/LU_decomposition#LU_factorization_with_partial_pivoting)

#### Parameters

|     |                                      |
|-----|--------------------------------------|
| $A$ | input square matrix to be decomposed |
|-----|--------------------------------------|

#### Returns

structure containing  $L$ ,  $U$  and  $P$ .

References [Mtx::Matrix< T >::cols\(\)](#), [Mtx::lup\(\)](#), [Mtx::Matrix< T >::numel\(\)](#), [Mtx::Matrix< T >::resize\(\)](#), [Mtx::Matrix< T >::rows\(\)](#), and [Mtx::Matrix< T >::swap\\_cols\(\)](#).

Referenced by [Mtx::det\\_lu\(\)](#), [Mtx::inv\\_square\(\)](#), [Mtx::lup\(\)](#), and [Mtx::solve\\_square\(\)](#).

#### 6.1.1.43 make\_complex() [1/2]

```
template<typename T >
Matrix< std::complex< T > > Mtx::make_complex (
    const Matrix< T > & Re )
```

Create complex matrix from real matrix.

Constructs a matrix of `std::complex` type from real and imaginary matrices.

#### Parameters

|      |                  |
|------|------------------|
| $Re$ | real part matrix |
|------|------------------|

#### Returns

complex matrix with real part set to  $Re$  and imaginary part to zero

References [Mtx::Matrix< T >::cols\(\)](#), [Mtx::make\\_complex\(\)](#), [Mtx::Matrix< T >::numel\(\)](#), and [Mtx::Matrix< T >::rows\(\)](#).

#### 6.1.1.44 make\_complex() [2/2]

```
template<typename T >
Matrix< std::complex< T > > Mtx::make_complex (
    const Matrix< T > & Re,
    const Matrix< T > & Im )
```

Create complex matrix from real and imaginary matrices.

Constructs a matrix of `std::complex` type from real matrices providing real and imaginary parts.  $Re$  and  $Im$  matrices must have the same dimensions.

## Parameters

|           |                       |
|-----------|-----------------------|
| <i>Re</i> | real part matrix      |
| <i>Im</i> | imaginary part matrix |

## Returns

complex matrix with real part set to *Re* and imaginary part to *Im*

## Exceptions

|                                 |  |
|---------------------------------|--|
| <code>std::runtime_error</code> | when <i>Re</i> and <i>Im</i> have different dimensions |
|---------------------------------|--|

References [Mtx::Matrix< T >::cols\(\)](#), [Mtx::make\\_complex\(\)](#), [Mtx::Matrix< T >::numel\(\)](#), and [Mtx::Matrix< T >::rows\(\)](#).

Referenced by [Mtx::eigenvalues\(\)](#), [Mtx::make\\_complex\(\)](#), and [Mtx::make\\_complex\(\)](#).

6.1.1.45 **mult()** [1/4]

```
template<typename T , bool transpose_first = false, bool transpose_second = false>
Matrix< T > Mtx::mult (
    const Matrix< T > & A,
    const Matrix< T > & B )
```

Matrix multiplication.

Performs multiplication of two matrices.

This function supports template parameterization of input matrix transposition, providing better efficiency than in case of using [Mtx::ctranspose\(\)](#) function due to zero-copy operation. In case of complex matrices, conjugate (Hermitian) transpose is used.

## Template Parameters

|                         |   |
|-------------------------|---|
| <i>transpose_first</i>  | if set to true, the left-side input matrix will be transposed during operation  |
| <i>transpose_second</i> | if set to true, the right-side input matrix will be transposed during operation |

## Parameters

|          |  |
|----------|--|
| <i>A</i> | left-side matrix of size $N \times M$ (after transposition)  |
| <i>B</i> | right-side matrix of size $M \times K$ (after transposition) |

## Returns

output matrix of size  $N \times K$

References [Mtx::conj\(\)](#), [Mtx::Matrix< T >::cols\(\)](#), [Mtx::mult\(\)](#), and [Mtx::Matrix< T >::rows\(\)](#).

Referenced by [Mtx::mult\(\)](#), [Mtx::mult\(\)](#), [Mtx::mult\(\)](#), [Mtx::mult\(\)](#), [Mtx::operator\\*\(\)](#), [Mtx::operator\\*\(\)](#), [Mtx::operator\\*\(\)](#), [Mtx::operator\\*\(\)](#), [Mtx::operator\\*\(\)](#), and [Mtx::operator\\*\(\)=\(\)](#).



**6.1.1.46 mult()** [2/4]

```
template<typename T , bool transpose_matrix = false>
std::vector< T > Mtx::mult (
    const Matrix< T > & A,
    const std::vector< T > & v )
```

Multiplication of matrix by std::vector.

Performs the right multiplication of a matrix with a column vector represented by std::vector. The result of this operation is also a std::vector.

This function supports template parameterization of input matrix transposition, providing better efficiency than in case of using [Mtx::ctranspose\(\)](#) function due to zero-copy operation. In case of complex matrices, conjugate (Hermitian) transpose is used.

**Template Parameters**

|                         |  |
|-------------------------|--|
| <i>transpose_matrix</i> | if set to true, the matrix will be transposed during operation |
|-------------------------|--|

**Parameters**

|          |                                   |
|----------|-----------------------------------|
| <i>A</i> | input matrix of size $N \times M$ |
| <i>v</i> | std::vector of size $M$           |

**Returns**

std::vector of size  $N$  being the result of multiplication

References [Mtx::cconj\(\)](#), [Mtx::Matrix< T >::cols\(\)](#), [Mtx::mult\(\)](#), and [Mtx::Matrix< T >::rows\(\)](#).

**6.1.1.47 mult()** [3/4]

```
template<typename T >
Matrix< T > Mtx::mult (
    const Matrix< T > & A,
    T s )
```

Multiplication of matrix by scalar.

Multiplies each element of the input matrix by a scalar  $s$ . This method does not modify the input matrix but creates a copy.

References [Mtx::Matrix< T >::cols\(\)](#), [Mtx::mult\(\)](#), [Mtx::Matrix< T >::numel\(\)](#), and [Mtx::Matrix< T >::rows\(\)](#).

#### 6.1.1.48 mult() [4/4]

```
template<typename T , bool transpose_matrix = false>
std::vector< T > Mtx::mult (
    const std::vector< T > & v,
    const Matrix< T > & A )
```

Multiplication of std::vector by matrix.

Performs the left multiplication of a std::vector with a matrix. The result of this operation is also a std::vector.

This function supports template parameterization of input matrix transposition, providing better efficiency than in case of using [Mtx::ctranspose\(\)](#) function due to zero-copy operation. In case of complex matrices, conjugate (Hermitian) transpose is used.

## Template Parameters

|                         |  |
|-------------------------|--|
| <i>transpose_matrix</i> | if set to true, the matrix will be transposed during operation |
|-------------------------|--|

## Parameters

|          |  |
|----------|--|
| <i>v</i> | std::vector of size <i>N</i>             |
| <i>A</i> | input matrix of size <i>N</i> x <i>M</i> |

## Returns

std::vector of size *M* being the result of multiplication

References [Mtx::cconj\(\)](#), [Mtx::Matrix< T >::cols\(\)](#), [Mtx::mult\(\)](#), and [Mtx::Matrix< T >::rows\(\)](#).

## 6.1.1.49 mult\_hadamard()

```
template<typename T , bool transpose_first = false, bool transpose_second = false>
Matrix< T > Mtx::mult_hadamard (
    const Matrix< T > & A,
    const Matrix< T > & B )
```

Matrix Hadamard (element-wise) multiplication.

Performs Hadamard (element-wise) multiplication of two matrices.

This function supports template parameterization of input matrix transposition, providing better efficiency than in case of using [Mtx::ctranspose\(\)](#) function due to zero-copy operation. In case of complex matrices, conjugate (Hermitian) transpose is used.

## Template Parameters

|                         |   |
|-------------------------|---|
| <i>transpose_first</i>  | if set to true, the left-side input matrix will be transposed during operation  |
| <i>transpose_second</i> | if set to true, the right-side input matrix will be transposed during operation |

## Parameters

|          |   |
|----------|---|
| <i>A</i> | left-side matrix of size <i>N</i> x <i>M</i> (after transposition)  |
| <i>B</i> | right-side matrix of size <i>N</i> x <i>M</i> (after transposition) |

## Returns

output matrix of size *N* x *M*

References [Mtx::cconj\(\)](#), [Mtx::Matrix< T >::cols\(\)](#), [Mtx::mult\\_hadamard\(\)](#), and [Mtx::Matrix< T >::rows\(\)](#).

Referenced by [Mtx::mult\\_hadamard\(\)](#), and [Mtx::operator^\(\)](#).

**6.1.1.50 norm\_fro()** [1/2]

```
template<typename T >
double Mtx::norm_fro (
    const Matrix< std::complex< T > > & A )
```

Frobenius norm of a complex matrix.

Calculates Frobenius norm of complex matrix.

More information: [https://en.wikipedia.org/wiki/Matrix\\_norm#Frobenius\\_norm](https://en.wikipedia.org/wiki/Matrix_norm#Frobenius_norm)

References [Mtx::norm\\_fro\(\)](#), and [Mtx::Matrix< T >::numel\(\)](#).

**6.1.1.51 norm\_fro()** [2/2]

```
template<typename T >
double Mtx::norm_fro (
    const Matrix< T > & A )
```

Frobenius norm.

Calculates Frobenius norm of a real matrix.

More information [https://en.wikipedia.org/wiki/Matrix\\_norm#Frobenius\\_norm](https://en.wikipedia.org/wiki/Matrix_norm#Frobenius_norm)

References [Mtx::norm\\_fro\(\)](#), and [Mtx::Matrix< T >::numel\(\)](#).

Referenced by [Mtx::cond\(\)](#), [Mtx::householder\\_reflection\(\)](#), [Mtx::norm\\_fro\(\)](#), [Mtx::norm\\_fro\(\)](#), and [Mtx::qr\\_red\\_gs\(\)](#).

**6.1.1.52 ones()** [1/2]

```
template<typename T >
Matrix< T > Mtx::ones (
    unsigned n ) [inline]
```

Square matrix of ones.

Construct a square matrix of size  $n \times n$  and fill it with all elements set to 1.

In case of complex datatype, matrix is filled with  $1 + 0i$ .

**Parameters**

|     |  |
|-----|--|
| $n$ | size of the square matrix (the first and the second dimension) |
|-----|--|

**Returns**

zeros matrix

References [Mtx::ones\(\)](#).

**6.1.1.53 ones()** [2/2]

```
template<typename T >
Matrix< T > Mtx::ones (
    unsigned nrows,
    unsigned ncols ) [inline]
```

Matrix of ones.

Construct a matrix of size *nrows* x *ncols* and fill it with all elements set to 1.  
In case of complex data types, matrix is filled with  $1 + 0i$ .

**Parameters**

|              |  |
|--------------|--|
| <i>nrows</i> | number of rows (the first dimension)     |
| <i>ncols</i> | number of columns (the second dimension) |

**Returns**

ones matrix

References [Mtx::ones\(\)](#).

Referenced by [Mtx::ones\(\)](#), and [Mtx::ones\(\)](#).

**6.1.1.54 operator!=(())**

```
template<typename T >
bool Mtx::operator!= (
    const Matrix< T > & A,
    const Matrix< T > & b ) [inline]
```

Matrix non-equality check operator.

Returns true, if both matrices are not the same size or not all of the elements are equal value.

References [Mtx::Matrix< T >::isequal\(\)](#), and [Mtx::operator!=\(\)](#).

Referenced by [Mtx::operator!=\(\)](#).

**6.1.1.55 operator\*()** [1/5]

```
template<typename T >
Matrix< T > Mtx::operator* (
    const Matrix< T > & A,
    const Matrix< T > & B ) [inline]
```

Matrix product.

Calculates matrix product of two matrices  $A \cdot B$ .  $A$  and  $B$  must be the same size.

References [Mtx::mult\(\)](#), and [Mtx::operator\\*\(\)](#).

Referenced by [Mtx::operator\\*\(\)](#), [Mtx::operator\\*\(\)](#), [Mtx::operator\\*\(\)](#), [Mtx::operator\\*\(\)](#), and [Mtx::operator\\*\(\)](#).

**6.1.1.56 operator\*() [2/5]**

```
template<typename T >
std::vector< T > Mtx::operator* (
    const Matrix< T > & A,
    const std::vector< T > & v ) [inline]
```

Matrix and std::vector product.

Calculates product between matrix and std::vector  $A \cdot v$ . The input vector is assumed to be a column vector.

References [Mtx::mult\(\)](#), and [Mtx::operator\\*\(\)](#).

**6.1.1.57 operator\*() [3/5]**

```
template<typename T >
Matrix< T > Mtx::operator* (
    const Matrix< T > & A,
    T s ) [inline]
```

Matrix product with scalar.

Multiplies each element of the matrix by a scalar  $s$ .

References [Mtx::mult\(\)](#), and [Mtx::operator\\*\(\)](#).

**6.1.1.58 operator\*() [4/5]**

```
template<typename T >
std::vector< T > Mtx::operator* (
    const std::vector< T > & v,
    const Matrix< T > & A ) [inline]
```

std::vector and matrix product.

Calculates product between std::vector and matrix  $v \cdot A$ . The input vector is assumed to be a row vector.

References [Mtx::mult\(\)](#), and [Mtx::operator\\*\(\)](#).

**6.1.1.59 operator\*() [5/5]**

```
template<typename T >
Matrix< T > Mtx::operator* (
    T s,
    const Matrix< T > & A ) [inline]
```

Matrix product with scalar.

Multiplies each element of the matrix by a scalar  $s$ .

References [Mtx::mult\(\)](#), and [Mtx::operator\\*\(\)](#).

**6.1.1.60 operator\*=( ) [1/2]**

```
template<typename T >
Matrix< T > & Mtx::operator*= (
    Matrix< T > & A,
    const Matrix< T > & B ) [inline]
```

Matrix product.

Calculates matrix product of two matrices  $A \cdot B$ .  $A$  and  $B$  must be the same size.

References [Mtx::mult\(\)](#), and [Mtx::operator\\*=\( \)](#).

Referenced by [Mtx::operator\\*=\( \)](#), and [Mtx::operator\\*=\( \)](#).

**6.1.1.61 operator\*=( ) [2/2]**

```
template<typename T >
Matrix< T > & Mtx::operator*= (
    Matrix< T > & A,
    T s ) [inline]
```

Matrix product with scalar.

Multiplies each element of the matrix by a scalar  $s$ .

References [Mtx::Matrix< T >::mult\(\)](#), and [Mtx::operator\\*=\( \)](#).

**6.1.1.62 operator+( ) [1/3]**

```
template<typename T >
Matrix< T > Mtx::operator+ (
    const Matrix< T > & A,
    const Matrix< T > & B ) [inline]
```

Matrix sum.

Calculates a sum of two matrices  $A + B$ .  $A$  and  $B$  must be the same size.

References [Mtx::add\(\)](#), and [Mtx::operator+\( \)](#).

Referenced by [Mtx::operator+\( \)](#), [Mtx::operator+\( \)](#), and [Mtx::operator+\( \)](#).

**6.1.1.63 operator+( ) [2/3]**

```
template<typename T >
Matrix< T > Mtx::operator+ (
    const Matrix< T > & A,
    T s ) [inline]
```

Matrix sum with scalar.

Adds a scalar  $s$  to each element of the matrix.

References [Mtx::add\(\)](#), and [Mtx::operator+\( \)](#).

**6.1.1.64 operator+()** [3/3]

```
template<typename T >
Matrix< T > Mtx::operator+ (
    T s,
    const Matrix< T > & A ) [inline]
```

Matrix sum with scalar. Adds a scalar  $s$  to each element of the matrix.

References [Mtx::add\(\)](#), and [Mtx::operator+\(\)](#).

**6.1.1.65 operator+=()** [1/2]

```
template<typename T >
Matrix< T > & Mtx::operator+= (
    Matrix< T > & A,
    const Matrix< T > & B ) [inline]
```

Matrix sum.

Calculates a sum of two matrices  $A + B$ .  $A$  and  $B$  must be the same size.

References [Mtx::Matrix< T >::add\(\)](#), and [Mtx::operator+=\(\)](#).

Referenced by [Mtx::operator+=\(\)](#), and [Mtx::operator+=\(\)](#).

**6.1.1.66 operator+=()** [2/2]

```
template<typename T >
Matrix< T > & Mtx::operator+= (
    Matrix< T > & A,
    T s ) [inline]
```

Matrix sum with scalar.

Adds a scalar  $s$  to each element of the matrix.

References [Mtx::Matrix< T >::add\(\)](#), and [Mtx::operator+=\(\)](#).

**6.1.1.67 operator-()** [1/2]

```
template<typename T >
Matrix< T > Mtx::operator- (
    const Matrix< T > & A,
    const Matrix< T > & B ) [inline]
```

Matrix subtraction.

Calculates a subtraction of two matrices  $A - B$ .  $A$  and  $B$  must be the same size.

References [Mtx::operator-\(\)](#), and [Mtx::subtract\(\)](#).

Referenced by [Mtx::operator-\(\)](#), and [Mtx::operator-\(\)](#).



**6.1.1.68 operator-() [2/2]**

```
template<typename T >
Matrix< T > Mtx::operator- (
    const Matrix< T > & A,
    T s ) [inline]
```

Matrix subtraction with scalar.

Subtracts a scalar  $s$  from each element of the matrix.

References [Mtx::operator-\(\)](#), and [Mtx::subtract\(\)](#).

**6.1.1.69 operator-=() [1/2]**

```
template<typename T >
Matrix< T > & Mtx::operator-= (
    Matrix< T > & A,
    const Matrix< T > & B ) [inline]
```

Matrix subtraction.

Subtracts two matrices  $A - B$ .  $A$  and  $B$  must be the same size.

References [Mtx::operator-=\(\)](#), and [Mtx::Matrix< T >::subtract\(\)](#).

Referenced by [Mtx::operator-=\(\)](#), and [Mtx::operator-=\(\)](#).

**6.1.1.70 operator-=() [2/2]**

```
template<typename T >
Matrix< T > & Mtx::operator-= (
    Matrix< T > & A,
    T s ) [inline]
```

Matrix subtraction with scalar.

Subtracts a scalar  $s$  from each element of the matrix.

References [Mtx::operator-=\(\)](#), and [Mtx::Matrix< T >::subtract\(\)](#).

**6.1.1.71 operator/()**

```
template<typename T >
Matrix< T > Mtx::operator/ (
    const Matrix< T > & A,
    T s ) [inline]
```

Matrix division by scalar.

Divides each element of the matrix by a scalar  $s$ .

References [Mtx::div\(\)](#), and [Mtx::operator/\(\)](#).

Referenced by [Mtx::operator/\(\)](#).

### 6.1.1.72 operator/=( )

```
template<typename T >
Matrix< T > & Mtx::operator/= (
    Matrix< T > & A,
    T s ) [inline]
```

Matrix division by scalar.

Divides each element of the matrix by a scalar  $s$ .

References [Mtx::Matrix< T >::div\(\)](#), and [Mtx::operator/=\( \)](#).

Referenced by [Mtx::operator/=\( \)](#).

### 6.1.1.73 operator<<( )

```
template<typename T >
std::ostream & Mtx::operator<< (
    std::ostream & os,
    const Matrix< T > & A )
```

Matrix ostream operator.

Formats a string incorporating the elements of a matrix. Elements within the same row are separated by space sign ' '. Different rows are separated by the newline delimiters.

References [Mtx::Matrix< T >::cols\(\)](#), and [Mtx::Matrix< T >::rows\(\)](#).

### 6.1.1.74 operator==( )

```
template<typename T >
bool Mtx::operator==(
    const Matrix< T > & A,
    const Matrix< T > & b ) [inline]
```

Matrix equality check operator.

Returns true, if both matrices are the same size and all of the element are equal value.

References [Mtx::Matrix< T >::isequal\(\)](#), and [Mtx::operator==\( \)](#).

Referenced by [Mtx::operator==\( \)](#).

### 6.1.1.75 operator^()

```
template<typename T >
Matrix< T > Mtx::operator^ (
    const Matrix< T > & A,
    const Matrix< T > & B ) [inline]
```

Matrix Hadamard product.

Calculates a Hadamard product of two matrices  $A \otimes B$ .  $A$  and  $B$  must be the same size. Hadamard product is calculated as an element-wise multiplication between the matrices.

References [Mtx::mult\\_hadamard\(\)](#), and [Mtx::operator^\(\)](#).

Referenced by [Mtx::operator^\(\)](#).

**6.1.1.76 operator^=()**

```
template<typename T >
Matrix< T > & Mtx::operator^= (
    Matrix< T > & A,
    const Matrix< T > & B ) [inline]
```

Matrix Hadamard product.

Calculates a Hadamard product of two matrices  $A \otimes B$ .  $A$  and  $B$  must be the same size. Hadamard product is calculated as an element-wise multiplication between the matrices.

References [Mtx::Matrix< T >::mult\\_hadamard\(\)](#), and [Mtx::operator^=\(\)](#).

Referenced by [Mtx::operator^=\(\)](#).

**6.1.1.77 permute\_cols()**

```
template<typename T >
Matrix< T > Mtx::permute_cols (
    const Matrix< T > & A,
    const std::vector< unsigned > perm )
```

Permute columns of the matrix.

Creates a copy of the matrix with permutation of columns specified as input parameter. Each column in the new matrix is a copy of respective column from the input matrix indexed by permutation vector. The size of the output matrix is  $A.rows() \times perm.size()$ .

**Parameters**

|             |  |
|-------------|--|
| <i>A</i>    | input matrix                           |
| <i>perm</i> | permutation vector with column indices |

**Returns**

output matrix created by column permutation of  $A$

**Exceptions**

|                           |  |
|---------------------------|--|
| <i>std::runtime_error</i> | when permutation vector is empty                     |
| <i>std::out_of_range</i>  | when any index in permutation vector is out of range |

References [Mtx::Matrix< T >::cols\(\)](#), [Mtx::permute\\_cols\(\)](#), and [Mtx::Matrix< T >::rows\(\)](#).

Referenced by [Mtx::permute\\_cols\(\)](#).

**6.1.1.78 permute\_rows()**

```
template<typename T >
Matrix< T > Mtx::permute_rows (
```

```
const Matrix< T > & A,
const std::vector< unsigned > perm )
```

Permute rows of the matrix.

Creates a copy of the matrix with permutation of rows specified as input parameter. Each row in the new matrix is a copy of respective row from the input matrix indexed by permutation vector. The size of the output matrix is *perm.size()* x *A.cols()*.

#### Parameters

|             |                                     |
|-------------|-------------------------------------|
| <i>A</i>    | input matrix                        |
| <i>perm</i> | permutation vector with row indices |

#### Returns

output matrix created by row permutation of *A*

#### Exceptions

|                           |  |
|---------------------------|--|
| <i>std::runtime_error</i> | when permutation vector is empty                     |
| <i>std::out_of_range</i>  | when any index in permutation vector is out of range |

References [Mtx::Matrix< T >::cols\(\)](#), [Mtx::permute\\_rows\(\)](#), and [Mtx::Matrix< T >::rows\(\)](#).

Referenced by [Mtx::inv\\_square\(\)](#), [Mtx::permute\\_rows\(\)](#), and [Mtx::solve\\_square\(\)](#).

#### 6.1.1.79 permute\_rows\_and\_cols()

```
template<typename T >
Matrix< T > Mtx::permute_rows_and_cols (
    const Matrix< T > & A,
    const std::vector< unsigned > perm_rows,
    const std::vector< unsigned > perm_cols )
```

Permute both rows and columns of the matrix.

Creates a copy of the matrix with permutation of rows and columns specified as input parameter. The result of this function is equivalent to performing row and column permutations separately - see [Mtx::permute\\_rows\(\)](#) and [Mtx::permute\\_cols\(\)](#).

The size of the output matrix is *perm\_rows.size()* x *perm\_cols.size()*.

#### Parameters

|                  |  |
|------------------|--|
| <i>A</i>         | input matrix                           |
| <i>perm_rows</i> | permutation vector with row indices    |
| <i>perm_cols</i> | permutation vector with column indices |

## Returns

output matrix created by row and column permutation of  $A$

## Exceptions

|                                 |  |
|---------------------------------|--|
| <code>std::runtime_error</code> | when any of permutation vectors is empty             |
| <code>std::out_of_range</code>  | when any index in permutation vector is out of range |

References [Mtx::Matrix< T >::cols\(\)](#), [Mtx::permute\\_rows\\_and\\_cols\(\)](#), and [Mtx::Matrix< T >::rows\(\)](#).

Referenced by [Mtx::permute\\_rows\\_and\\_cols\(\)](#).

## 6.1.1.80 pinv()

```
template<typename T >
Matrix< T > Mtx::pinv (
    const Matrix< T > & A )
```

Moore-Penrose pseudo-inverse.

Calculates the Moore-Penrose pseudo-inverse  $A^+$  of a matrix  $A$ .

If  $A$  has linearly independent columns, the pseudo-inverse is a left inverse, that is  $A^+A = I$ , and  $A^+ = (A'A)^{-1}A'$ .

If  $A$  has linearly independent rows, the pseudo-inverse is a right inverse, that is  $AA^+ = I$ , and  $A^+ = A'(AA')^{-1}$ .

More information: [https://en.wikipedia.org/wiki/Moore%E2%80%93Penrose\\_inverse](https://en.wikipedia.org/wiki/Moore%E2%80%93Penrose_inverse)

References [Mtx::Matrix< T >::cols\(\)](#), [Mtx::inv\\_posdef\(\)](#), [Mtx::pinv\(\)](#), and [Mtx::Matrix< T >::rows\(\)](#).

Referenced by [Mtx::pinv\(\)](#).

## 6.1.1.81 qr()

```
template<typename T >
QR_result< T > Mtx::qr (
    const Matrix< T > & A,
    bool calculate_Q = true ) [inline]
```

QR decomposition.

The QR decomposition is a decomposition of a matrix  $A$  into a product  $A = QR$  of an orthonormal matrix  $Q$  and an upper triangular matrix  $R$ .

Currently, this function is a wrapper around [Mtx::qr\\_householder\(\)](#). Refer to [qr\\_red\\_gs\(\)](#) for alternative implementation.

## Parameters

|                          |                                   |
|--------------------------|-----------------------------------|
| $A$                      | input matrix to be decomposed     |
| <code>calculate_Q</code> | indicates if $Q$ to be calculated |

**Returns**

structure encapsulating calculated  $Q$  of size  $n \times n$  and  $R$  of size  $n \times m$ .  $Q$  is calculated only when `calculate_Q = True`.

References [Mtx::qr\(\)](#), and [Mtx::qr\\_householder\(\)](#).

Referenced by [Mtx::eigenvalues\(\)](#), and [Mtx::qr\(\)](#).

**6.1.1.82 qr\_householder()**

```
template<typename T >
QR_result< T > Mtx::qr_householder (
    const Matrix< T > & A,
    bool calculate_Q = true )
```

QR decomposition based on Householder method.

The QR decomposition is a decomposition of a matrix  $A$  into a product  $A = QR$  of an orthonormal matrix  $Q$  and an upper triangular matrix  $R$ .

This function implements QR decomposition based on Householder reflections method.

More information: [https://en.wikipedia.org/wiki/QR\\_decomposition](https://en.wikipedia.org/wiki/QR_decomposition)

**Parameters**

|                          |  |
|--------------------------|--|
| $A$                      | input matrix to be decomposed, size $n \times m$ |
| <code>calculate_Q</code> | indicates if $Q$ to be calculated                |

**Returns**

structure encapsulating calculated  $Q$  of size  $n \times n$  and  $R$  of size  $n \times m$ .  $Q$  is calculated only when `calculate_Q = True`.

References [Mtx::Matrix< T >::cols\(\)](#), [Mtx::Matrix< T >::get\\_submatrix\(\)](#), [Mtx::householder\\_reflection\(\)](#), [Mtx::qr\\_householder\(\)](#), and [Mtx::Matrix< T >::rows\(\)](#).

Referenced by [Mtx::qr\(\)](#), and [Mtx::qr\\_householder\(\)](#).

**6.1.1.83 qr\_red\_gs()**

```
template<typename T >
QR_result< T > Mtx::qr_red_gs (
    const Matrix< T > & A )
```

Reduced QR decomposition based on Gram-Schmidt method.

The QR decomposition is a decomposition of a matrix  $A$  into a product  $A = QR$  of an orthonormal matrix  $Q$  and an upper triangular matrix  $R$ .

This function implements the reduced QR decomposition based on Gram-Schmidt method.

More information: [https://en.wikipedia.org/wiki/QR\\_decomposition](https://en.wikipedia.org/wiki/QR_decomposition)

## Parameters

|          |  |
|----------|--|
| <i>A</i> | input matrix to be decomposed, size $n \times m$ |
|----------|--|

## Returns

structure encapsulating calculated  $Q$  of size  $n \times m$ , and  $R$  of size  $m \times m$ .

## Exceptions

|                                  |  |
|----------------------------------|--|
| <i>singular_matrix_exception</i> | when division by 0 is encountered during computation |
|----------------------------------|--|

References [Mtx::conj\(\)](#), [Mtx::Matrix< T >::cols\(\)](#), [Mtx::Matrix< T >::get\\_submatrix\(\)](#), [Mtx::norm\\_fro\(\)](#), [Mtx::qr\\_red\\_gs\(\)](#), and [Mtx::Matrix< T >::rows\(\)](#).

Referenced by [Mtx::qr\\_red\\_gs\(\)](#).

**6.1.1.84 real()**

```
template<typename T >
Matrix< T > Mtx::real (
    const Matrix< std::complex< T > > & C )
```

Get real part of complex matrix.

Constructs a matrix of real type from `std::complex` matrix by taking its real part.

References [Mtx::Matrix< T >::cols\(\)](#), [Mtx::Matrix< T >::numel\(\)](#), [Mtx::real\(\)](#), and [Mtx::Matrix< T >::rows\(\)](#).

Referenced by [Mtx::real\(\)](#).

**6.1.1.85 repmat()**

```
template<typename T >
Matrix< T > Mtx::repmat (
    const Matrix< T > & A,
    unsigned m,
    unsigned n )
```

Repeat matrix.

Form a block matrix of size  $m$  by  $n$ , with a copy of matrix  $A$  as each element.

## Parameters

|          |  |
|----------|--|
| <i>A</i> | input matrix to be repeated  |
| <i>m</i> | number of times to repeat matrix $A$ in vertical dimension (rows)      |
| <i>n</i> | number of times to repeat matrix $A$ in horizontal dimension (columns) |

References [Mtx::Matrix< T >::cols\(\)](#), [Mtx::repmat\(\)](#), and [Mtx::Matrix< T >::rows\(\)](#).

Referenced by [Mtx::repmat\(\)](#).

#### 6.1.1.86 solve\_posdef()

```
template<typename T >
Matrix< T > Mtx::solve_posdef (
    const Matrix< T > & A,
    const Matrix< T > & B )
```

Solves the positive definite (Hermitian) system.

Return the matrix left division of  $A$  and  $B$ , where  $A$  is positive definite matrix. It is equivalent to solving the system  $A \cdot X = B$  with respect to  $X$ . The system is solved for each column of  $B$  using Cholesky decomposition followed by forward and backward propagation.

A minimum norm solution is computed if the coefficient matrix is singular.

##### Parameters

|     |   |
|-----|---|
| $A$ | left side matrix of size $N \times N$ . Must be square and positive definite. |
| $B$ | right hand side matrix of size $N \times M$ .                                 |

##### Returns

solution matrix of size  $N \times M$ .

##### Exceptions

|                                  |  |
|----------------------------------|--|
| <i>std::runtime_error</i>        | when the input matrix is not square  |
| <i>std::runtime_error</i>        | when number of rows is not equal between input matrices                          |
| <i>singular_matrix_exception</i> | when the input matrix is singular (detected as division by 0 during computation) |

References [Mtx::chol\(\)](#), [Mtx::Matrix< T >::issquare\(\)](#), [Mtx::Matrix< T >::numel\(\)](#), [Mtx::Matrix< T >::rows\(\)](#), [Mtx::solve\\_posdef\(\)](#), [Mtx::solve\\_tril\(\)](#), and [Mtx::solve\\_triu\(\)](#).

Referenced by [Mtx::solve\\_posdef\(\)](#).

#### 6.1.1.87 solve\_square()

```
template<typename T >
Matrix< T > Mtx::solve_square (
    const Matrix< T > & A,
    const Matrix< T > & B )
```

Solves the square system.

Return the matrix left division of  $A$  and  $B$ , where  $A$  is square. It is equivalent to solving the system  $A \cdot X = B$  with respect to  $X$ . The system is solved for each column of  $B$  using LU decomposition followed by forward and backward propagation.

A minimum norm solution is computed if the coefficient matrix is singular.



## Parameters

|          |   |
|----------|---|
| <i>A</i> | left side matrix of size $N \times N$ . Must be square. |
| <i>B</i> | right hand side matrix of size $N \times M$ .           |

## Returns

solution matrix of size  $N \times M$ .

## Exceptions

|                                  |  |
|----------------------------------|--|
| <i>std::runtime_error</i>        | when the input matrix is not square  |
| <i>std::runtime_error</i>        | when number of rows is not equal between input matrices                          |
| <i>singular_matrix_exception</i> | when the input matrix is singular (detected as division by 0 during computation) |

References [Mtx::Matrix< T >::issquare\(\)](#), [Mtx::lup\(\)](#), [Mtx::Matrix< T >::numel\(\)](#), [Mtx::permute\\_rows\(\)](#), [Mtx::Matrix< T >::rows\(\)](#), [Mtx::solve\\_square\(\)](#), [Mtx::solve\\_tril\(\)](#), and [Mtx::solve\\_triu\(\)](#).

Referenced by [Mtx::solve\\_square\(\)](#).

## 6.1.1.88 solve\_tril()

```
template<typename T >
Matrix< T > Mtx::solve_tril (
    const Matrix< T > & L,
    const Matrix< T > & B )
```

Solves the lower triangular system.

Return the matrix left division of  $L$  and  $B$ , where  $L$  is square and lower triangular. It is equivalent to solving the system  $L \cdot X = B$  with respect to  $X$ . The system is solved for each column of  $B$  using forwards substitution. A minimum norm solution is computed if the coefficient matrix is singular.

## Parameters

|          |   |
|----------|---|
| <i>L</i> | left side matrix of size $N \times N$ . Must be square and lower triangular |
| <i>B</i> | right hand side matrix of size $N \times M$ .                               |

## Returns

$X$  solution matrix of size  $N \times M$ .

## Exceptions

|                                  |  |
|----------------------------------|--|
| <i>std::runtime_error</i>        | when the input matrix is not square  |
| <i>std::runtime_error</i>        | when number of rows is not equal between input matrices                          |
| <i>singular_matrix_exception</i> | when the input matrix is singular (detected as division by 0 during computation) |

References [Mtx::Matrix< T >::cols\(\)](#), [Mtx::Matrix< T >::issquare\(\)](#), [Mtx::Matrix< T >::numel\(\)](#), [Mtx::Matrix< T >::rows\(\)](#), and [Mtx::solve\\_tril\(\)](#).

Referenced by [Mtx::solve\\_posdef\(\)](#), [Mtx::solve\\_square\(\)](#), and [Mtx::solve\\_tril\(\)](#).

#### 6.1.1.89 solve\_triu()

```
template<typename T >
Matrix< T > Mtx::solve_triu (
    const Matrix< T > & U,
    const Matrix< T > & B )
```

Solves the upper triangular system.

Return the matrix left division of  $U$  and  $B$ , where  $U$  is square and upper triangular. It is equivalent to solving the system  $U \cdot X = B$  with respect to  $X$ . The system is solved for each column of  $B$  using backwards substitution. A minimum norm solution is computed if the coefficient matrix is singular.

##### Parameters

|     |   |
|-----|---|
| $U$ | left side matrix of size $N \times N$ . Must be square and upper triangular |
| $B$ | right hand side matrix of size $N \times M$ .                               |

##### Returns

solution matrix of size  $N \times M$ .

##### Exceptions

|                                  |  |
|----------------------------------|--|
| <i>std::runtime_error</i>        | when the input matrix is not square  |
| <i>std::runtime_error</i>        | when number of rows is not equal between input matrices                          |
| <i>singular_matrix_exception</i> | when the input matrix is singular (detected as division by 0 during computation) |

References [Mtx::Matrix< T >::cols\(\)](#), [Mtx::Matrix< T >::issquare\(\)](#), [Mtx::Matrix< T >::numel\(\)](#), [Mtx::Matrix< T >::rows\(\)](#), and [Mtx::solve\\_triu\(\)](#).

Referenced by [Mtx::solve\\_posdef\(\)](#), [Mtx::solve\\_square\(\)](#), and [Mtx::solve\\_triu\(\)](#).

#### 6.1.1.90 subtract() [1/2]

```
template<typename T , bool transpose_first = false, bool transpose_second = false>
Matrix< T > Mtx::subtract (
    const Matrix< T > & A,
    const Matrix< T > & B )
```

Matrix subtraction.

Performs subtraction of two matrices.

This function supports template parameterization of input matrix transposition, providing better efficiency than in case of using [Mtx::ctranspose\(\)](#) function due to zero-copy operation. In case of complex matrices, conjugate (Hermitian) transpose is used.

## Template Parameters

|                         |   |
|-------------------------|---|
| <i>transpose_first</i>  | if set to true, the left-side input matrix will be transposed during operation  |
| <i>transpose_second</i> | if set to true, the right-side input matrix will be transposed during operation |

## Parameters

|          |  |
|----------|--|
| <i>A</i> | left-side matrix of size $N \times M$ (after transposition)  |
| <i>B</i> | right-side matrix of size $N \times M$ (after transposition) |

## Returns

output matrix of size  $N \times M$

References [Mtx::cconj\(\)](#), [Mtx::Matrix< T >::cols\(\)](#), [Mtx::Matrix< T >::rows\(\)](#), and [Mtx::subtract\(\)](#).

Referenced by [Mtx::operator-\(\)](#), [Mtx::operator-\(\)](#), [Mtx::subtract\(\)](#), and [Mtx::subtract\(\)](#).

**6.1.1.91 subtract()** [2/2]

```
template<typename T >
Matrix< T > Mtx::subtract (
    const Matrix< T > & A,
    T s )
```

Subtraction of scalar from matrix.

Subtracts a scalar  $s$  from each element of the input matrix. This method does not modify the input matrix but creates a copy.

References [Mtx::Matrix< T >::cols\(\)](#), [Mtx::Matrix< T >::numel\(\)](#), [Mtx::Matrix< T >::rows\(\)](#), and [Mtx::subtract\(\)](#).

**6.1.1.92 trace()**

```
template<typename T >
T Mtx::trace (
    const Matrix< T > & A )
```

Matrix trace.

Calculates trace of a matrix by summing the elements on the diagonal.

$$\text{tr}(A) = \sum_{n=0}^{N-1} [A]_{n,n}$$

References [Mtx::Matrix< T >::rows\(\)](#), and [Mtx::trace\(\)](#).

Referenced by [Mtx::trace\(\)](#).

### 6.1.1.93 transpose()

```
template<typename T >
Matrix< T > Mtx::transpose (
    const Matrix< T > & A ) [inline]
```

Transpose a matrix.

Returns a matrix that is a transposition of an input matrix.

References [Mtx::Matrix< T >::transpose\(\)](#), and [Mtx::transpose\(\)](#).

Referenced by [Mtx::transpose\(\)](#).

### 6.1.1.94 tril()

```
template<typename T >
Matrix< T > Mtx::tril (
    const Matrix< T > & A )
```

Extract triangular lower part.

Return a new matrix formed by extracting the lower triangular part of the input matrix, and setting all other elements to zero.

References [Mtx::Matrix< T >::cols\(\)](#), [Mtx::Matrix< T >::rows\(\)](#), and [Mtx::tril\(\)](#).

Referenced by [Mtx::chol\(\)](#), and [Mtx::tril\(\)](#).

### 6.1.1.95 triu()

```
template<typename T >
Matrix< T > Mtx::triu (
    const Matrix< T > & A )
```

Extract triangular upper part.

Return a new matrix formed by extracting the upper triangular part of the input matrix, and setting all other elements to zero.

References [Mtx::Matrix< T >::cols\(\)](#), [Mtx::Matrix< T >::rows\(\)](#), and [Mtx::triu\(\)](#).

Referenced by [Mtx::chol\(\)](#), and [Mtx::triu\(\)](#).

### 6.1.1.96 wilkinson\_shift()

```
template<typename T >
std::complex< T > Mtx::wilkinson_shift (
    const Matrix< std::complex< T > > & H,
    T tol = 1e-10 )
```

Wilkinson's shift for complex eigenvalues.

Computes Wilkinson's shift value *mu* for complex eigenvalues of input matrix. Wilkinson's shift is calculated as eigenvalue of the bottom 2 x 2 principal minor closest to the corner entry of the matrix. Input must be a square matrix in Hessenberg form.

## Exceptions

|                                 |                                     |
|---------------------------------|-------------------------------------|
| <code>std::runtime_error</code> | when the input matrix is not square |
|---------------------------------|-------------------------------------|

References [Mtx::Matrix< T >::rows\(\)](#), and [Mtx::wilkinson\\_shift\(\)](#).

Referenced by [Mtx::eigenvalues\(\)](#), and [Mtx::wilkinson\\_shift\(\)](#).

**6.1.1.97 zeros()** [1/2]

```
template<typename T >
Matrix< T > Mtx::zeros (
    unsigned n ) [inline]
```

Square matrix of zeros.

Construct a square matrix of size  $n \times n$  and fill it with all elements set to 0.

## Parameters

|          |  |
|----------|--|
| <i>n</i> | size of the square matrix (the first and the second dimension) |
|----------|--|

## Returns

zeros matrix

References [Mtx::zeros\(\)](#).

**6.1.1.98 zeros()** [2/2]

```
template<typename T >
Matrix< T > Mtx::zeros (
    unsigned nrows,
    unsigned ncols ) [inline]
```

Matrix of zeros.

Create a matrix of size  $nrows \times ncols$  and fill it with all elements set to 0.

## Parameters

|              |  |
|--------------|--|
| <i>nrows</i> | number of rows (the first dimension)     |
| <i>ncols</i> | number of columns (the second dimension) |

## Returns

zeros matrix

References [Mtx::zeros\(\)](#).

Referenced by [Mtx::zeros\(\)](#), and [Mtx::zeros\(\)](#).

## 6.2 matrix.hpp

[Go to the documentation of this file.](#)

```

00001
00002
00003 /* MIT License
00004 *
00005 * Copyright (c) 2024 gc1905
00006 *
00007 * Permission is hereby granted, free of charge, to any person obtaining a copy
00008 * of this software and associated documentation files (the "Software"), to deal
00009 * in the Software without restriction, including without limitation the rights
00010 * to use, copy, modify, merge, publish, distribute, sublicense, and/or sell
00011 * copies of the Software, and to permit persons to whom the Software is
00012 * furnished to do so, subject to the following conditions:
00013 *
00014 * The above copyright notice and this permission notice shall be included in all
00015 * copies or substantial portions of the Software.
00016 *
00017 * THE SOFTWARE IS PROVIDED "AS IS", WITHOUT WARRANTY OF ANY KIND, EXPRESS OR
00018 * IMPLIED, INCLUDING BUT NOT LIMITED TO THE WARRANTIES OF MERCHANTABILITY,
00019 * FITNESS FOR A PARTICULAR PURPOSE AND NONINFRINGEMENT. IN NO EVENT SHALL THE
00020 * AUTHORS OR COPYRIGHT HOLDERS BE LIABLE FOR ANY CLAIM, DAMAGES OR OTHER
00021 * LIABILITY, WHETHER IN AN ACTION OF CONTRACT, TORT OR OTHERWISE, ARISING FROM,
00022 * OUT OF OR IN CONNECTION WITH THE SOFTWARE OR THE USE OR OTHER DEALINGS IN THE
00023 * SOFTWARE.
00024 */
00025
00026 #ifndef __MATRIX_HPP__
00027 #define __MATRIX_HPP__
00028
00029 #include <ostream>
00030 #include <complex>
00031 #include <vector>
00032 #include <initializer_list>
00033 #include <limits>
00034 #include <functional>
00035 #include <algorithm>
00036
00037 namespace Mtx {
00038
00039 template<typename T> class Matrix;
00040
00041 template<class T> struct is_complex : std::false_type {};
00042 template<class T> struct is_complex<std::complex<T> > : std::true_type {};
00043
00044 template<typename T, typename std::enable_if<!is_complex<T>::value, int>::type = 0>
00045 inline T cconj(T x) {
00046     return x;
00047 }
00048
00049 template<typename T, typename std::enable_if<is_complex<T>::value, int>::type = 0>
00050 inline T cconj(T x) {
00051     return std::conj(x);
00052 }
00053
00054 template<typename T, typename std::enable_if<!is_complex<T>::value, int>::type = 0>
00055 inline T csign(T x) {
00056     return (x > static_cast<T>(0)) ? static_cast<T>(1) : static_cast<T>(-1);
00057 }
00058
00059 template<typename T, typename std::enable_if<is_complex<T>::value, int>::type = 0>
00060 inline T csign(T x) {
00061     auto x_arg = std::arg(x);
00062     T y(0, x_arg);
00063     return std::exp(y);
00064 }
00065
00066 class singular_matrix_exception : public std::domain_error {
00067 public:
00068     singular_matrix_exception(const std::string& message) : std::domain_error(message) {}
00069 };
00070
00071 template<typename T>
00072 struct LU_result {
00073     Matrix<T> L;
00074     Matrix<T> U;
00075 };
00076
00077 template<typename T>
00078 struct LUP_result {
00079     Matrix<T> L;
00080     Matrix<T> U;
00081 
```

```

00120
00123     std::vector<unsigned> P;
00124 };
00125
00131 template<typename T>
00132 struct QR_result {
00133     Matrix<T> Q;
00136
00139     Matrix<T> R;
00140 };
00141
00146 template<typename T>
00147 struct Hessenberg_result {
00150     Matrix<T> H;
00151
00154     Matrix<T> Q;
00155 };
00156
00161 template<typename T>
00162 struct LDL_result {
00165     Matrix<T> L;
00166
00169     std::vector<T> d;
00170 };
00171
00176 template<typename T>
00177 struct Eigenvalues_result {
00180     std::vector<std::complex<T>> eig;
00181
00184     bool converged;
00185
00188     T err;
00189 };
00190
00191
00199 template<typename T>
00200 inline Matrix<T> zeros(unsigned nrows, unsigned ncols) {
00201     return Matrix<T>(static_cast<T>(0), nrows, ncols);
00202 }
00203
00210 template<typename T>
00211 inline Matrix<T> zeros(unsigned n) {
00212     return zeros<T>(n,n);
00213 }
00214
00223 template<typename T>
00224 inline Matrix<T> ones(unsigned nrows, unsigned ncols) {
00225     return Matrix<T>(static_cast<T>(1), nrows, ncols);
00226 }
00227
00235 template<typename T>
00236 inline Matrix<T> ones(unsigned n) {
00237     return ones<T>(n,n);
00238 }
00239
00247 template<typename T>
00248 Matrix<T> eye(unsigned n) {
00249     Matrix<T> A(static_cast<T>(0), n, n);
00250     for (unsigned i = 0; i < n; i++)
00251         A(i,i) = static_cast<T>(1);
00252     return A;
00253 }
00254
00262 template<typename T>
00263 Matrix<T> diag(const T* array, size_t n) {
00264     Matrix<T> A(static_cast<T>(0), n, n);
00265     for (unsigned i = 0; i < n; i++) {
00266         A(i,i) = array[i];
00267     }
00268     return A;
00269 }
00270
00278 template<typename T>
00279 inline Matrix<T> diag(const std::vector<T>& v) {
00280     return diag(v.data(), v.size());
00281 }
00282
00291 template<typename T>
00292 std::vector<T> diag(const Matrix<T>& A) {
00293     if (! A.issquare()) throw std::runtime_error("Input matrix is not square");
00294
00295     std::vector<T> v;
00296     v.resize(A.rows());
00297
00298     for (unsigned i = 0; i < A.rows(); i++)
00299         v[i] = A(i,i);
00300     return v;

```

```

00301 }
00302
00310 template<typename T>
00311 Matrix<T> circulant(const T* array, unsigned n) {
00312     Matrix<T> A(n, n);
00313     for (unsigned j = 0; j < n; j++)
00314         for (unsigned i = 0; i < n; i++)
00315             A((i+j) % n, j) = array[i];
00316     return A;
00317 }
00318
00329 template<typename T>
00330 Matrix<std::complex<T>> make_complex(const Matrix<T>& Re, const Matrix<T>& Im) {
00331     if (Re.rows() != Im.rows() || Re.cols() != Im.cols()) throw std::runtime_error("Size of input
matrices does not match");
00332
00333     Matrix<std::complex<T>> C(Re.rows(), Re.cols());
00334     for (unsigned n = 0; n < Re.numel(); n++) {
00335         C(n).real(Re(n));
00336         C(n).imag(Im(n));
00337     }
00338
00339     return C;
00340 }
00341
00348 template<typename T>
00349 Matrix<std::complex<T>> make_complex(const Matrix<T>& Re) {
00350     Matrix<std::complex<T>> C(Re.rows(), Re.cols());
00351
00352     for (unsigned n = 0; n < Re.numel(); n++) {
00353         C(n).real(Re(n));
00354         C(n).imag(static_cast<T>(0));
00355     }
00356
00357     return C;
00358 }
00359
00364 template<typename T>
00365 Matrix<T> real(const Matrix<std::complex<T>& C) {
00366     Matrix<T> Re(C.rows(), C.cols());
00367
00368     for (unsigned n = 0; n < C.numel(); n++)
00369         Re(n) = C(n).real();
00370
00371     return Re;
00372 }
00373
00378 template<typename T>
00379 Matrix<T> imag(const Matrix<std::complex<T>& C) {
00380     Matrix<T> Re(C.rows(), C.cols());
00381
00382     for (unsigned n = 0; n < C.numel(); n++)
00383         Re(n) = C(n).imag();
00384
00385     return Re;
00386 }
00387
00395 template<typename T>
00396 inline Matrix<T> circulant(const std::vector<T>& v) {
00397     return circulant(v.data(), v.size());
00398 }
00399
00404 template<typename T>
00405 inline Matrix<T> transpose(const Matrix<T>& A) {
00406     return A.transpose();
00407 }
00408
00414 template<typename T>
00415 inline Matrix<T> ctranspose(const Matrix<T>& A) {
00416     return A.ctranspose();
00417 }
00418
00429 template<typename T>
00430 Matrix<T> circshift(const Matrix<T>& A, int row_shift, int col_shift) {
00431     Matrix<T> B(A.rows(), A.cols());
00432     for (int i = 0; i < A.rows(); i++) {
00433         int ii = (i + row_shift) % A.rows();
00434         for (int j = 0; j < A.cols(); j++) {
00435             int jj = (j + col_shift) % A.cols();
00436             B(ii, jj) = A(i, j);
00437         }
00438     }
00439     return B;
00440 }
00441
00449 template<typename T>
00450 Matrix<T> repmat(const Matrix<T>& A, unsigned m, unsigned n) {

```



```

00451     Matrix<T> B(m * A.rows(), n * A.cols());
00452
00453     for (unsigned cb = 0; cb < n; cb++)
00454         for (unsigned rb = 0; rb < m; rb++)
00455             for (unsigned c = 0; c < A.cols(); c++)
00456                 for (unsigned r = 0; r < A.rows(); r++)
00457                     B(rb*A.rows() + r, cb*A.cols() + c) = A(r, c);
00458
00459     return B;
00460 }
00461
00462 template<typename T>
00463 Matrix<T> concatenate_horizontal(const Matrix<T>& A, const Matrix<T>& B) {
00470     if (A.rows() != B.rows()) throw std::runtime_error("Unmatching number of rows for horizontal
concatenation");
00471
00472     Matrix<T> C(A.rows(), A.cols() + B.cols());
00473
00474     for (unsigned c = 0; c < A.cols(); c++)
00475         for (unsigned r = 0; r < A.rows(); r++)
00476             C(r,c) = A(r,c);
00477
00478     for (unsigned c = 0; c < B.cols(); c++)
00479         for (unsigned r = 0; r < B.rows(); r++)
00480             C(r,c+A.cols()) = B(r,c);
00481
00482     return C;
00483 }
00484
00491 template<typename T>
00492 Matrix<T> concatenate_vertical(const Matrix<T>& A, const Matrix<T>& B) {
00493     if (A.cols() != B.cols()) throw std::runtime_error("Unmatching number of columns for vertical
concatenation");
00494
00495     Matrix<T> C(A.rows() + B.rows(), A.cols());
00496
00497     for (unsigned c = 0; c < A.cols(); c++)
00498         for (unsigned r = 0; r < A.rows(); r++)
00499             C(r,c) = A(r,c);
00500
00501     for (unsigned c = 0; c < B.cols(); c++)
00502         for (unsigned r = 0; r < B.rows(); r++)
00503             C(r+A.rows(),c) = B(r,c);
00504
00505     return C;
00506 }
00507
00513 template<typename T>
00514 double norm_fro(const Matrix<T>& A) {
00515     double sum = 0;
00516
00517     for (unsigned i = 0; i < A.numel(); i++)
00518         sum += A(i) * A(i);
00519
00520     return std::sqrt(sum);
00521 }
00522
00528 template<typename T>
00529 double norm_fro(const Matrix<std::complex<T>>& A) {
00530     double sum = 0;
00531
00532     for (unsigned i = 0; i < A.numel(); i++) {
00533         T x = std::abs(A(i));
00534         sum += x * x;
00535     }
00536
00537     return std::sqrt(sum);
00538 }
00539
00545 template<typename T>
00546 Matrix<T> tril(const Matrix<T>& A) {
00547     Matrix<T> B(A);
00548
00549     for (unsigned row = 0; row < B.rows(); row++)
00550         for (unsigned col = row+1; col < B.cols(); col++)
00551             B(row,col) = 0;
00552
00553     return B;
00554 }
00555
00561 template<typename T>
00562 Matrix<T> triu(const Matrix<T>& A) {
00563     Matrix<T> B(A);
00564
00565     for (unsigned col = 0; col < B.cols(); col++)
00566         for (unsigned row = col+1; row < B.rows(); row++)
00567             B(row,col) = 0;

```

```

00568
00569     return B;
00570 }
00571
00572 template<typename T>
00573 bool istril(const Matrix<T>& A) {
00574     for (unsigned row = 0; row < A.rows(); row++)
00575         for (unsigned col = row+1; col < A.cols(); col++)
00576             if (A(row,col) != static_cast<T>(0)) return false;
00577     return true;
00578 }
00579
00580 template<typename T>
00581 bool istriu(const Matrix<T>& A) {
00582     for (unsigned col = 0; col < A.cols(); col++)
00583         for (unsigned row = col+1; row < A.rows(); row++)
00584             if (A(row,col) != static_cast<T>(0)) return false;
00585     return true;
00586 }
00587
00588 template<typename T>
00589 bool ishess(const Matrix<T>& A) {
00590     if (!A.issquare())
00591         return false;
00592     for (unsigned row = 2; row < A.rows(); row++)
00593         for (unsigned col = 0; col < row-2; col++)
00594             if (A(row,col) != static_cast<T>(0)) return false;
00595     return true;
00596 }
00597
00598 template<typename T>
00599 inline void foreach_elem(Matrix<T>& A, std::function<T(T)> func) {
00600     for (unsigned i = 0; i < A.numel(); i++)
00601         A(i) = func(A(i));
00602 }
00603
00604 template<typename T>
00605 inline Matrix<T> foreach_elem_copy(const Matrix<T>& A, std::function<T(T)> func) {
00606     Matrix<T> B(A);
00607     foreach_elem(B, func);
00608     return B;
00609 }
00610
00611 template<typename T>
00612 Matrix<T> permute_rows(const Matrix<T>& A, const std::vector<unsigned> perm) {
00613     if (perm.empty()) throw std::runtime_error("Permutation vector is empty");
00614     for (unsigned p = 0; p < perm.size(); p++)
00615         if (!perm[p] < A.rows()) throw std::out_of_range("Index in permutation vector out of range");
00616     Matrix<T> B(perm.size(), A.cols());
00617     for (unsigned p = 0; p < perm.size(); p++)
00618         for (unsigned c = 0; c < A.cols(); c++)
00619             B(p,c) = A(perm[p],c);
00620     return B;
00621 }
00622
00623 template<typename T>
00624 Matrix<T> permute_cols(const Matrix<T>& A, const std::vector<unsigned> perm) {
00625     if (perm.empty()) throw std::runtime_error("Permutation vector is empty");
00626     for (unsigned p = 0; p < perm.size(); p++)
00627         if (!perm[p] < A.cols()) throw std::out_of_range("Index in permutation vector out of range");
00628     Matrix<T> B(A.rows(), perm.size());
00629     for (unsigned p = 0; p < perm.size(); p++)
00630         for (unsigned r = 0; r < A.rows(); r++)
00631             B(r,p) = A(r,perm[p]);
00632     return B;
00633 }
00634
00635 template<typename T>
00636 Matrix<T> permute_rows_and_cols(const Matrix<T>& A, const std::vector<unsigned> perm_rows, const
std::vector<unsigned> perm_cols) {
00637     if (perm_rows.empty()) throw std::runtime_error("Row permutation vector is empty");
00638     if (perm_cols.empty()) throw std::runtime_error("Column permutation vector is empty");
00639     for (unsigned pc = 0; pc < perm_cols.size(); pc++)
00640         if (!perm_cols[pc] < A.cols()) throw std::out_of_range("Column index in permutation vector out
of range");
00641     for (unsigned pr = 0; pr < perm_rows.size(); pr++)
00642         if (!perm_rows[pr] < A.rows()) throw std::out_of_range("Row index in permutation vector out of

```

```

    range");
00725
00726     Matrix<T> B(perm_rows.size(), perm_cols.size());
00727
00728     for (unsigned pc = 0; pc < perm_cols.size(); pc++)
00729         for (unsigned pr = 0; pr < perm_rows.size(); pr++)
00730             B(pr,pc) = A(perm_rows[pr],perm_cols[pc]);
00731
00732     return B;
00733 }
00734
00750 template<typename T, bool transpose_first = false, bool transpose_second = false>
00751 Matrix<T> mult(const Matrix<T>& A, const Matrix<T>& B) {
00752     // Adjust dimensions based on transpositions
00753     unsigned rows_A = transpose_first ? A.cols() : A.rows();
00754     unsigned cols_A = transpose_first ? A.rows() : A.cols();
00755     unsigned rows_B = transpose_second ? B.cols() : B.rows();
00756     unsigned cols_B = transpose_second ? B.rows() : B.cols();
00757
00758     if (cols_A != rows_B) throw std::runtime_error("Unmatching matrix dimensions for mult");
00759
00760     Matrix<T> C(static_cast<T>(0), rows_A, cols_B);
00761
00762     for (unsigned i = 0; i < rows_A; i++)
00763         for (unsigned j = 0; j < cols_B; j++)
00764             for (unsigned k = 0; k < cols_A; k++)
00765                 C(i,j) += (transpose_first ? cconj(A(k,i)) : A(i,k)) *
00766                     (transpose_second ? cconj(B(j,k)) : B(k,j));
00767
00768     return C;
00769 }
00770
00786 template<typename T, bool transpose_first = false, bool transpose_second = false>
00787 Matrix<T> mult_hadamard(const Matrix<T>& A, const Matrix<T>& B) {
00788     // Adjust dimensions based on transpositions
00789     unsigned rows_A = transpose_first ? A.cols() : A.rows();
00790     unsigned cols_A = transpose_first ? A.rows() : A.cols();
00791     unsigned rows_B = transpose_second ? B.cols() : B.rows();
00792     unsigned cols_B = transpose_second ? B.rows() : B.cols();
00793
00794     if ((rows_A != rows_B) || (cols_A != cols_B)) throw std::runtime_error("Unmatching matrix dimensions
for mult_hadamard");
00795
00796     Matrix<T> C(static_cast<T>(0), rows_A, cols_A);
00797
00798     for (unsigned i = 0; i < rows_A; i++)
00799         for (unsigned j = 0; j < cols_A; j++)
00800             C(i,j) += (transpose_first ? cconj(A(j,i)) : A(i,j)) *
00801                 (transpose_second ? cconj(B(j,i)) : B(i,j));
00802
00803     return C;
00804 }
00805
00821 template<typename T, bool transpose_first = false, bool transpose_second = false>
00822 Matrix<T> add(const Matrix<T>& A, const Matrix<T>& B) {
00823     // Adjust dimensions based on transpositions
00824     unsigned rows_A = transpose_first ? A.cols() : A.rows();
00825     unsigned cols_A = transpose_first ? A.rows() : A.cols();
00826     unsigned rows_B = transpose_second ? B.cols() : B.rows();
00827     unsigned cols_B = transpose_second ? B.rows() : B.cols();
00828
00829     if ((rows_A != rows_B) || (cols_A != cols_B)) throw std::runtime_error("Unmatching matrix dimensions
for add");
00830
00831     Matrix<T> C(static_cast<T>(0), rows_A, cols_A);
00832
00833     for (unsigned i = 0; i < rows_A; i++)
00834         for (unsigned j = 0; j < cols_A; j++)
00835             C(i,j) += (transpose_first ? cconj(A(j,i)) : A(i,j)) +
00836                 (transpose_second ? cconj(B(j,i)) : B(i,j));
00837
00838     return C;
00839 }
00840
00856 template<typename T, bool transpose_first = false, bool transpose_second = false>
00857 Matrix<T> subtract(const Matrix<T>& A, const Matrix<T>& B) {
00858     // Adjust dimensions based on transpositions
00859     unsigned rows_A = transpose_first ? A.cols() : A.rows();
00860     unsigned cols_A = transpose_first ? A.rows() : A.cols();
00861     unsigned rows_B = transpose_second ? B.cols() : B.rows();
00862     unsigned cols_B = transpose_second ? B.rows() : B.cols();
00863
00864     if ((rows_A != rows_B) || (cols_A != cols_B)) throw std::runtime_error("Unmatching matrix dimensions
for subtract");
00865
00866     Matrix<T> C(static_cast<T>(0), rows_A, cols_A);
00867

```

```

00868     for (unsigned i = 0; i < rows_A; i++)
00869         for (unsigned j = 0; j < cols_A; j++)
00870             C(i,j) += (transpose_first ? cconj(A(j,i)) : A(i,j)) -
00871                 (transpose_second ? cconj(B(j,i)) : B(i,j));
00872
00873     return C;
00874 }
00875
00891 template<typename T, bool transpose_matrix = false>
00892 std::vector<T> mult(const Matrix<T>& A, const std::vector<T>& v) {
00893     // Adjust dimensions based on transpositions
00894     unsigned rows_A = transpose_matrix ? A.cols() : A.rows();
00895     unsigned cols_A = transpose_matrix ? A.rows() : A.cols();
00896
00897     if (cols_A != v.size()) throw std::runtime_error("Unmatching matrix dimensions for mult");
00898
00899     std::vector<T> u(rows_A, static_cast<T>(0));
00900     for (unsigned r = 0; r < rows_A; r++)
00901         for (unsigned c = 0; c < cols_A; c++)
00902             u[r] += v[c] * (transpose_matrix ? cconj(A(c,r)) : A(r,c));
00903
00904     return u;
00905 }
00906
00922 template<typename T, bool transpose_matrix = false>
00923 std::vector<T> mult(const std::vector<T>& v, const Matrix<T>& A) {
00924     // Adjust dimensions based on transpositions
00925     unsigned rows_A = transpose_matrix ? A.cols() : A.rows();
00926     unsigned cols_A = transpose_matrix ? A.rows() : A.cols();
00927
00928     if (rows_A != v.size()) throw std::runtime_error("Unmatching matrix dimensions for mult");
00929
00930     std::vector<T> u(cols_A, static_cast<T>(0));
00931     for (unsigned c = 0; c < cols_A; c++)
00932         for (unsigned r = 0; r < rows_A; r++)
00933             u[c] += v[r] * (transpose_matrix ? cconj(A(c,r)) : A(r,c));
00934
00935     return u;
00936 }
00937
00943 template<typename T>
00944 Matrix<T> add(const Matrix<T>& A, T s) {
00945     Matrix<T> B(A.rows(), A.cols());
00946     for (unsigned i = 0; i < A.numel(); i++)
00947         B(i) = A(i) + s;
00948     return B;
00949 }
00950
00956 template<typename T>
00957 Matrix<T> subtract(const Matrix<T>& A, T s) {
00958     Matrix<T> B(A.rows(), A.cols());
00959     for (unsigned i = 0; i < A.numel(); i++)
00960         B(i) = A(i) - s;
00961     return B;
00962 }
00963
00969 template<typename T>
00970 Matrix<T> mult(const Matrix<T>& A, T s) {
00971     Matrix<T> B(A.rows(), A.cols());
00972     for (unsigned i = 0; i < A.numel(); i++)
00973         B(i) = A(i) * s;
00974     return B;
00975 }
00976
00982 template<typename T>
00983 Matrix<T> div(const Matrix<T>& A, T s) {
00984     Matrix<T> B(A.rows(), A.cols());
00985     for (unsigned i = 0; i < A.numel(); i++)
00986         B(i) = A(i) / s;
00987     return B;
00988 }
00989
00995 template<typename T>
00996 std::ostream& operator<<(std::ostream& os, const Matrix<T>& A) {
00997     for (unsigned row = 0; row < A.rows(); row++) {
00998         for (unsigned col = 0; col < A.cols(); col++)
00999             os << A(row,col) << " ";
01000         if (row < static_cast<unsigned>(A.rows()-1)) os << std::endl;
01001     }
01002     return os;
01003 }
01004
01009 template<typename T>
01010 inline Matrix<T> operator+(const Matrix<T>& A, const Matrix<T>& B) {
01011     return add(A,B);
01012 }
01013

```

```

01018 template<typename T>
01019 inline Matrix<T> operator-(const Matrix<T>& A, const Matrix<T>& B) {
01020     return subtract(A,B);
01021 }
01022
01028 template<typename T>
01029 inline Matrix<T> operator^(const Matrix<T>& A, const Matrix<T>& B) {
01030     return mult_hadamard(A,B);
01031 }
01032
01037 template<typename T>
01038 inline Matrix<T> operator*(const Matrix<T>& A, const Matrix<T>& B) {
01039     return mult(A,B);
01040 }
01041
01046 template<typename T>
01047 inline std::vector<T> operator*(const Matrix<T>& A, const std::vector<T>& v) {
01048     return mult(A,v);
01049 }
01050
01055 template<typename T>
01056 inline std::vector<T> operator*(const std::vector<T>& v, const Matrix<T>& A) {
01057     return mult(v,A);
01058 }
01059
01064 template<typename T>
01065 inline Matrix<T> operator+(const Matrix<T>& A, T s) {
01066     return add(A,s);
01067 }
01068
01073 template<typename T>
01074 inline Matrix<T> operator-(const Matrix<T>& A, T s) {
01075     return subtract(A,s);
01076 }
01077
01082 template<typename T>
01083 inline Matrix<T> operator*(const Matrix<T>& A, T s) {
01084     return mult(A,s);
01085 }
01086
01091 template<typename T>
01092 inline Matrix<T> operator/(const Matrix<T>& A, T s) {
01093     return div(A,s);
01094 }
01095
01099 template<typename T>
01100 inline Matrix<T> operator+(T s, const Matrix<T>& A) {
01101     return add(A,s);
01102 }
01103
01108 template<typename T>
01109 inline Matrix<T> operator*(T s, const Matrix<T>& A) {
01110     return mult(A,s);
01111 }
01112
01117 template<typename T>
01118 inline Matrix<T>& operator+=(Matrix<T>& A, const Matrix<T>& B) {
01119     return A.add(B);
01120 }
01121
01126 template<typename T>
01127 inline Matrix<T>& operator-=(Matrix<T>& A, const Matrix<T>& B) {
01128     return A.subtract(B);
01129 }
01130
01135 template<typename T>
01136 inline Matrix<T>& operator*=(Matrix<T>& A, const Matrix<T>& B) {
01137     A = mult(A,B);
01138     return A;
01139 }
01140
01146 template<typename T>
01147 inline Matrix<T>& operator^=(Matrix<T>& A, const Matrix<T>& B) {
01148     return A.mult_hadamard(B);
01149 }
01150
01155 template<typename T>
01156 inline Matrix<T>& operator+=(Matrix<T>& A, T s) {
01157     return A.add(s);
01158 }
01159
01164 template<typename T>
01165 inline Matrix<T>& operator-=(Matrix<T>& A, T s) {
01166     return A.subtract(s);
01167 }
01168
01173 template<typename T>

```

```

01174 inline Matrix<T>& operator*=(Matrix<T>& A, T s) {
01175     return A.mult(s);
01176 }
01177
01182 template<typename T>
01183 inline Matrix<T>& operator/=(Matrix<T>& A, T s) {
01184     return A.div(s);
01185 }
01186
01191 template<typename T>
01192 inline bool operator==(const Matrix<T>& A, const Matrix<T>& b) {
01193     return A.isequal(b);
01194 }
01195
01200 template<typename T>
01201 inline bool operator!=(const Matrix<T>& A, const Matrix<T>& b) {
01202     return !(A.isequal(b));
01203 }
01204
01211 template<typename T>
01212 Matrix<T> kron(const Matrix<T>& A, const Matrix<T>& B) {
01213     const unsigned rows_A = A.rows();
01214     const unsigned cols_A = A.cols();
01215     const unsigned rows_B = B.rows();
01216     const unsigned cols_B = B.cols();
01217
01218     unsigned rows_C = rows_A * rows_B;
01219     unsigned cols_C = cols_A * cols_B;
01220
01221     Matrix<T> C(rows_C, cols_C);
01222
01223     for (unsigned i = 0; i < rows_A; i++)
01224         for (unsigned j = 0; j < cols_A; j++)
01225             for (unsigned k = 0; k < rows_B; k++)
01226                 for (unsigned l = 0; l < cols_B; l++)
01227                     C(i+rows_B * k, j+cols_B * l) = A(i, j) * B(k, l);
01228
01229     return C;
01230 }
01231
01239 template<typename T>
01240 Matrix<T> adj(const Matrix<T>& A) {
01241     if (!A.issquare()) throw std::runtime_error("Input matrix is not square");
01242
01243     Matrix<T> B(A.rows(), A.cols());
01244     if (A.rows() == 1) {
01245         B(0) = 1.0;
01246     } else {
01247         for (unsigned i = 0; i < A.rows(); i++) {
01248             for (unsigned j = 0; j < A.cols(); j++) {
01249                 T sgn = ((i + j) % 2 == 0) ? 1.0 : -1.0;
01250                 B(j, i) = sgn * det(cofactor(A, i, j));
01251             }
01252         }
01253     }
01254     return B;
01255 }
01256
01270 template<typename T>
01271 Matrix<T> cofactor(const Matrix<T>& A, unsigned p, unsigned q) {
01272     if (!A.issquare()) throw std::runtime_error("Input matrix is not square");
01273     if (p < A.rows()) throw std::out_of_range("Row index out of range");
01274     if (q < A.cols()) throw std::out_of_range("Column index out of range");
01275     if (A.cols() < 2) throw std::runtime_error("Cofactor calculation requested for matrix with less than
2 rows");
01276
01277     Matrix<T> c(A.rows()-1, A.cols()-1);
01278     unsigned i = 0;
01279     unsigned j = 0;
01280
01281     for (unsigned row = 0; row < A.rows(); row++) {
01282         if (row != p) {
01283             for (unsigned col = 0; col < A.cols(); col++)
01284                 if (col != q) c(i, j++) = A(row, col);
01285             j = 0;
01286             i++;
01287         }
01288     }
01289
01290     return c;
01291 }
01292
01304 template<typename T>
01305 T det_lu(const Matrix<T>& A) {
01306     if (!A.issquare()) throw std::runtime_error("Input matrix is not square");
01307
01308     // LU decomposition with pivoting

```

```

01309     auto res = lup(A);
01310
01311     // Determinants of LU
01312     T detLU = static_cast<T>(1);
01313
01314     for (unsigned i = 0; i < res.L.rows(); i++)
01315         detLU *= res.L(i,i) * res.U(i,i);
01316
01317     // Determinant of P
01318     unsigned len = res.P.size();
01319     T detP = 1;
01320
01321     std::vector<unsigned> p(res.P);
01322     std::vector<unsigned> q;
01323     q.resize(len);
01324
01325     for (unsigned i = 0; i < len; i++)
01326         q[p[i]] = i;
01327
01328     for (unsigned i = 0; i < len; i++) {
01329         unsigned j = p[i];
01330         unsigned k = q[i];
01331         if (j != i) {
01332             p[k] = p[i];
01333             q[j] = q[i];
01334             detP = - detP;
01335         }
01336     }
01337
01338     return detLU * detP;
01339 }
01340
01350 template<typename T>
01351 T det(const Matrix<T>& A) {
01352     if (! A.issquare()) throw std::runtime_error("Input matrix is not square");
01353
01354     if (A.rows() == 1)
01355         return A(0,0);
01356     else if (A.rows() == 2)
01357         return A(0,0)*A(1,1) - A(0,1)*A(1,0);
01358     else if (A.rows() == 3)
01359         return A(0,0)*(A(1,1)*A(2,2) - A(1,2)*A(2,1)) -
01360             A(0,1)*(A(1,0)*A(2,2) - A(1,2)*A(2,0)) +
01361             A(0,2)*(A(1,0)*A(2,1) - A(1,1)*A(2,0));
01362     else
01363         return det_lu(A);
01364 }
01365
01375 template<typename T>
01376 LU_result<T> lu(const Matrix<T>& A) {
01377     const unsigned M = A.rows();
01378     const unsigned N = A.cols();
01379
01380     LU_result<T> res;
01381     res.L = eye<T>(M);
01382     res.U = Matrix<T>(A);
01383
01384     // aliases
01385     auto& L = res.L;
01386     auto& U = res.U;
01387
01388     if (A.numel() == 0)
01389         return res;
01390
01391     for (unsigned k = 0; k < M-1; k++) {
01392         for (unsigned i = k+1; i < M; i++) {
01393             L(i,k) = U(i,k) / U(k,k);
01394             for (unsigned l = k+1; l < N; l++) {
01395                 U(i,l) -= L(i,k) * U(k,l);
01396             }
01397         }
01398     }
01399
01400     for (unsigned col = 0; col < N; col++)
01401         for (unsigned row = col+1; row < M; row++)
01402             U(row,col) = 0;
01403
01404     return res;
01405 }
01406
01420 template<typename T>
01421 LUP_result<T> lup(const Matrix<T>& A) {
01422     const unsigned M = A.rows();
01423     const unsigned N = A.cols();
01424
01425     // Initialize L, U, and PP
01426     LUP_result<T> res;

```

```

01427
01428     if (A.numel() == 0)
01429         return res;
01430
01431     res.L = eye<T>(M);
01432     res.U = Matrix<T>(A);
01433     std::vector<unsigned> PP;
01434
01435     // aliases
01436     auto& L = res.L;
01437     auto& U = res.U;
01438
01439     PP.resize(N);
01440     for (unsigned i = 0; i < N; i++)
01441         PP[i] = i;
01442
01443     for (unsigned k = 0; k < M-1; k++) {
01444         // Find the column with the largest absolute value in the current row
01445         auto max_col_value = std::abs(U(k,k));
01446         unsigned max_col_index = k;
01447         for (unsigned l = k+1; l < N; l++) {
01448             auto val = std::abs(U(k,l));
01449             if (val > max_col_value) {
01450                 max_col_value = val;
01451                 max_col_index = l;
01452             }
01453         }
01454
01455         // Swap columns k and max_col_index in U and update P
01456         if (max_col_index != k) {
01457             U.swap_cols(k, max_col_index); // TODO: This could be reworked to avoid column swap in U during
every iteration by:
01458                                     //      1. using PP[k] for column indexing across iterations
01459                                     //      2. doing just one permutation of U at the end
01460             std::swap(PP[k], PP[max_col_index]);
01461         }
01462
01463         // Update L and U
01464         for (unsigned i = k+1; i < M; i++) {
01465             L(i,k) = U(i,k) / U(k,k);
01466             for (unsigned l = k+1; l < N; l++) {
01467                 U(i,l) -= L(i,k) * U(k,l);
01468             }
01469         }
01470     }
01471
01472     // Set elements in lower triangular part of U to zero
01473     for (unsigned col = 0; col < N; col++)
01474         for (unsigned row = col+1; row < M; row++)
01475             U(row,col) = 0;
01476
01477     // Transpose indices in permutation vector
01478     res.P.resize(N);
01479     for (unsigned i = 0; i < N; i++)
01480         res.P[PP[i]] = i;
01481
01482     return res;
01483 }
01484
01495 template<typename T>
01496 Matrix<T> inv_gauss_jordan(const Matrix<T>& A) {
01497     if (! A.issquare()) throw std::runtime_error("Input matrix is not square");
01498
01499     const unsigned N = A.rows();
01500     Matrix<T> AA(A);
01501     auto IA = eye<T>(N);
01502
01503     bool found_nonzero;
01504     for (unsigned j = 0; j < N; j++) {
01505         found_nonzero = false;
01506         for (unsigned i = j; i < N; i++) {
01507             if (AA(i,j) != static_cast<T>(0)) {
01508                 found_nonzero = true;
01509                 for (unsigned k = 0; k < N; k++) {
01510                     std::swap(AA(j,k), AA(i,k));
01511                     std::swap(IA(j,k), IA(i,k));
01512                 }
01513                 if (AA(j,j) != static_cast<T>(1)) {
01514                     T s = static_cast<T>(1) / AA(j,j);
01515                     for (unsigned k = 0; k < N; k++) {
01516                         AA(j,k) *= s;
01517                         IA(j,k) *= s;
01518                     }
01519                 }
01520                 for (unsigned l = 0; l < N; l++) {
01521                     if (l != j) {
01522                         T s = AA(l,j);

```



```

01523         for (unsigned k = 0; k < N; k++) {
01524             AA(l,k) -= s * AA(j,k);
01525             IA(l,k) -= s * IA(j,k);
01526         }
01527     }
01528 }
01529 }
01530     break;
01531 }
01532 // if a row full of zeros is found, the input matrix was singular
01533 if (!found_nonzero) throw singular_matrix_exception("Singular matrix in inv_gauss_jordan");
01534 }
01535 return IA;
01536 }
01537
01538 template<typename T>
01539 Matrix<T> inv_tril(const Matrix<T>& A) {
01540     if (!A.issquare()) throw std::runtime_error("Input matrix is not square");
01541     const unsigned N = A.rows();
01542     auto IA = zeros<T>(N);
01543     for (unsigned i = 0; i < N; i++) {
01544         if (A(i,i) == 0.0) throw singular_matrix_exception("Division by zero in inv_tril");
01545         IA(i,i) = static_cast<T>(1.0) / A(i,i);
01546         for (unsigned j = 0; j < i; j++) {
01547             T s = 0.0;
01548             for (unsigned k = j; k < i; k++)
01549                 s += A(i,k) * IA(k,j);
01550             IA(i,j) = -s * IA(i,i);
01551         }
01552     }
01553     return IA;
01554 }
01555
01556 template<typename T>
01557 Matrix<T> inv_triu(const Matrix<T>& A) {
01558     if (!A.issquare()) throw std::runtime_error("Input matrix is not square");
01559     const unsigned N = A.rows();
01560     auto IA = zeros<T>(N);
01561     for (int i = N - 1; i >= 0; i--) {
01562         if (A(i,i) == 0.0) throw singular_matrix_exception("Division by zero in inv_triu");
01563         IA(i,i) = static_cast<T>(1.0) / A(i,i);
01564         for (int j = N - 1; j > i; j--) {
01565             T s = 0.0;
01566             for (int k = i + 1; k <= j; k++)
01567                 s += A(i,k) * IA(k,j);
01568             IA(i,j) = -s * IA(i,i);
01569         }
01570     }
01571     return IA;
01572 }
01573
01574 template<typename T>
01575 Matrix<T> inv_posdef(const Matrix<T>& A) {
01576     auto L = cholinv(A);
01577     return mult<T,true,false>(L,L);
01578 }
01579
01580 template<typename T>
01581 Matrix<T> inv_square(const Matrix<T>& A) {
01582     if (!A.issquare()) throw std::runtime_error("Input matrix is not square");
01583     // LU decomposition with pivoting
01584     auto LU = lup(A);
01585     auto IL = inv_tril(LU.L);
01586     auto IU = inv_triu(LU.U);
01587     return permute_rows(IU * IL, LU.P);
01588 }
01589
01590 template<typename T>
01591 Matrix<T> inv(const Matrix<T>& A) {
01592     if (!A.issquare()) throw std::runtime_error("Input matrix is not square");
01593     if (A.numel() == 0) {
01594         return Matrix<T>();
01595     } else if (A.rows() < 4) {
01596         T d = det(A);

```

```

01663
01664     if (d == 0.0) throw singular_matrix_exception("Singular matrix in inv");
01665
01666     Matrix<T> IA(A.rows(), A.rows());
01667     T invdet = static_cast<T>(1.0) / d;
01668
01669     if (A.rows() == 1) {
01670         IA(0,0) = invdet;
01671     } else if (A.rows() == 2) {
01672         IA(0,0) = A(1,1) * invdet;
01673         IA(0,1) = - A(0,1) * invdet;
01674         IA(1,0) = - A(1,0) * invdet;
01675         IA(1,1) = A(0,0) * invdet;
01676     } else if (A.rows() == 3) {
01677         IA(0,0) = (A(1,1)*A(2,2) - A(2,1)*A(1,2)) * invdet;
01678         IA(0,1) = (A(0,2)*A(2,1) - A(0,1)*A(2,2)) * invdet;
01679         IA(0,2) = (A(0,1)*A(1,2) - A(0,2)*A(1,1)) * invdet;
01680         IA(1,0) = (A(1,2)*A(2,0) - A(1,0)*A(2,2)) * invdet;
01681         IA(1,1) = (A(0,0)*A(2,2) - A(0,2)*A(2,0)) * invdet;
01682         IA(1,2) = (A(1,0)*A(0,2) - A(0,0)*A(1,2)) * invdet;
01683         IA(2,0) = (A(1,0)*A(2,1) - A(2,0)*A(1,1)) * invdet;
01684         IA(2,1) = (A(2,0)*A(0,1) - A(0,0)*A(2,1)) * invdet;
01685         IA(2,2) = (A(0,0)*A(1,1) - A(1,0)*A(0,1)) * invdet;
01686     }
01687
01688     return IA;
01689 } else {
01690     return inv_square(A);
01691 }
01692 }
01693
01703 template<typename T>
01704 Matrix<T> pinv(const Matrix<T>& A) {
01705     if (A.rows() > A.cols()) {
01706         auto AH_A = mult<T,true,false>(A, A);
01707         auto Linv = inv_posdef(AH_A);
01708         return mult<T,false,true>(Linv, A);
01709     } else {
01710         auto AA_H = mult<T,false,true>(A, A);
01711         auto Linv = inv_posdef(AA_H);
01712         return mult<T,true,false>(A, Linv);
01713     }
01714 }
01715
01721 template<typename T>
01722 T trace(const Matrix<T>& A) {
01723     T t = static_cast<T>(0);
01724     for (int i = 0; i < A.rows(); i++)
01725         t += A(i,i);
01726     return t;
01727 }
01728
01737 template<typename T>
01738 double cond(const Matrix<T>& A) {
01739     try {
01740         auto A_inv = inv(A);
01741         return norm_fro(A) * norm_fro(A_inv);
01742     } catch (singular_matrix_exception& e) {
01743         return std::numeric_limits<double>::max();
01744     }
01745 }
01746
01765 template<typename T, bool is_upper = false>
01766 Matrix<T> chol(const Matrix<T>& A) {
01767     if (!A.issquare()) throw std::runtime_error("Input matrix is not square");
01768
01769     const unsigned N = A.rows();
01770
01771     // Calculate lower or upper triangular, depending on template parameter.
01772     // Calculation is the same - the difference is in transposed row and column indexing.
01773     Matrix<T> C = is_upper ? triu(A) : tril(A);
01774
01775     for (unsigned j = 0; j < N; j++) {
01776         if (C(j,j) == 0.0) throw singular_matrix_exception("Singular matrix in chol");
01777         C(j,j) = std::sqrt(C(j,j));
01778
01779         for (unsigned k = j+1; k < N; k++)
01780             if (is_upper)
01781                 C(j,k) /= C(j,j);
01782             else
01783                 C(k,j) /= C(j,j);
01784
01785         for (unsigned k = j+1; k < N; k++)
01786             for (unsigned i = k; i < N; i++)
01787                 if (is_upper)
01788                     C(k,i) -= C(j,i) * cconj(C(j,k));
01789

```

```

01790         else
01791             C(i,k) -= C(i,j) * cconj(C(k,j));
01792     }
01793     return C;
01794 }
01795
01796 template<typename T>
01800 Matrix<T> cholinv(const Matrix<T>& A) {
01801     if (! A.issquare()) throw std::runtime_error("Input matrix is not square");
01802
01803     const unsigned N = A.rows();
01804     Matrix<T> L(A);
01805     auto Linv = eye<T>(N);
01806
01807     for (unsigned j = 0; j < N; j++) {
01808         if (L(j,j) == 0.0) throw singular_matrix_exception("Singular matrix in cholinv");
01809
01810         L(j,j) = 1.0 / std::sqrt(L(j,j));
01811
01812         for (unsigned k = j+1; k < N; k++)
01813             L(k,j) = L(k,j) * L(j,j);
01814
01815         for (unsigned k = j+1; k < N; k++)
01816             for (unsigned i = k; i < N; i++)
01817                 L(i,k) = L(i,k) - L(i,j) * cconj(L(k,j));
01818     }
01819
01820     for (unsigned k = 0; k < N; k++) {
01821         for (unsigned i = k; i < N; i++) {
01822             Linv(i,k) = Linv(i,k) * L(i,i);
01823             for (unsigned j = i+1; j < N; j++)
01824                 Linv(j,k) = Linv(j,k) - L(j,i) * Linv(i,k);
01825         }
01826     }
01827     return Linv;
01828 }
01829
01830 template<typename T>
01831 LDL_result<T> ldl(const Matrix<T>& A) {
01832     if (! A.issquare()) throw std::runtime_error("Input matrix is not square");
01833
01834     const unsigned N = A.rows();
01835     LDL_result<T> res;
01836
01837     // aliases
01838     auto& L = res.L;
01839     auto& d = res.d;
01840
01841     L = eye<T>(N);
01842     d.resize(N);
01843
01844     for (unsigned m = 0; m < N; m++) {
01845         d[m] = A(m,m);
01846
01847         for (unsigned k = 0; k < m; k++)
01848             d[m] -= L(m,k) * cconj(L(m,k)) * d[k];
01849
01850         if (d[m] == 0.0) throw singular_matrix_exception("Singular matrix in ldl");
01851
01852         for (unsigned n = m+1; n < N; n++) {
01853             L(n,m) = A(n,m);
01854             for (unsigned k = 0; k < m; k++)
01855                 L(n,m) -= L(n,k) * cconj(L(m,k)) * d[k];
01856             L(n,m) /= d[m];
01857         }
01858     }
01859     return res;
01860 }
01861
01862 template<typename T>
01863 QR_result<T> qr_red_gs(const Matrix<T>& A) {
01864     const int rows = A.rows();
01865     const int cols = A.cols();
01866     QR_result<T> res;
01867
01868     //aliases
01869     auto& Q = res.Q;
01870     auto& R = res.R;
01871
01872     Q = zeros<T>(rows, cols);
01873     R = zeros<T>(cols, cols);
01874 }

```

```

01914     for (int c = 0; c < cols; c++) {
01915         Matrix<T> v = A.get_submatrix(0, rows-1, c, c);
01916         for (int r = 0; r < c; r++) {
01917             for (int k = 0; k < rows; k++)
01918                 R(r,c) = R(r,c) + cconj(Q(k,r)) * A(k,c);
01919             for (int k = 0; k < rows; k++)
01920                 v(k) = v(k) - R(r,c) * Q(k,r);
01921         }
01922
01923         R(c,c) = static_cast<T>(norm_fro(v));
01924
01925         if (R(c,c) == 0.0) throw singular_matrix_exception("Division by 0 in QR GS");
01926
01927         for (int k = 0; k < rows; k++)
01928             Q(k,c) = v(k) / R(c,c);
01929     }
01930
01931     return res;
01932 }
01933
01941 template<typename T>
01942 Matrix<T> householder_reflection(const Matrix<T>& a) {
01943     if (a.cols() != 1) throw std::runtime_error("Input not a column vector");
01944
01945     static const T ISQRT2 = static_cast<T>(0.707106781186547);
01946
01947     Matrix<T> v(a);
01948     v(0) += csign(v(0)) * norm_fro(v);
01949     auto vn = norm_fro(v) * ISQRT2;
01950     for (unsigned i = 0; i < v.numel(); i++)
01951         v(i) /= vn;
01952     return v;
01953 }
01954
01966 template<typename T>
01967 QR_result<T> qr_householder(const Matrix<T>& A, bool calculate_Q = true) {
01968     const unsigned rows = A.rows();
01969     const unsigned cols = A.cols();
01970
01971     QR_result<T> res;
01972
01973     //aliases
01974     auto& Q = res.Q;
01975     auto& R = res.R;
01976
01977     R = Matrix<T>(A);
01978
01979     if (calculate_Q)
01980         Q = eye<T>(rows);
01981
01982     const unsigned N = (rows > cols) ? cols : rows;
01983
01984     for (unsigned j = 0; j < N; j++) {
01985         auto v = householder_reflection(R.get_submatrix(j, rows-1, j, j));
01986
01987         auto R1 = R.get_submatrix(j, rows-1, j, cols-1);
01988         auto WR = v * mult<T,true,false>(v, R1);
01989         for (unsigned c = j; c < cols; c++)
01990             for (unsigned r = j; r < rows; r++)
01991                 R(r,c) -= WR(r-j,c-j);
01992
01993         if (calculate_Q) {
01994             auto Q1 = Q.get_submatrix(0, rows-1, j, rows-1);
01995             auto WQ = mult<T,false,true>(Q1 * v, v);
01996             for (unsigned c = j; c < rows; c++)
01997                 for (unsigned r = 0; r < rows; r++)
01998                     Q(r,c) -= WQ(r,c-j);
01999         }
02000     }
02001
02002     for (unsigned col = 0; col < R.cols(); col++)
02003         for (unsigned row = col+1; row < R.rows(); row++)
02004             R(row,col) = 0;
02005
02006     return res;
02007 }
02008
02020 template<typename T>
02021 inline QR_result<T> qr(const Matrix<T>& A, bool calculate_Q = true) {
02022     return qr_householder(A, calculate_Q);
02023 }
02024
02035 template<typename T>
02036 Hessenberg_result<T> hessenberg(const Matrix<T>& A, bool calculate_Q = true) {
02037     if (! A.issquare()) throw std::runtime_error("Input matrix is not square");
02038
02039     Hessenberg_result<T> res;

```

```

02040
02041 // aliases
02042 auto& H = res.H;
02043 auto& Q = res.Q;
02044
02045 const unsigned N = A.rows();
02046 H = Matrix<T>(A);
02047
02048 if (calculate_Q)
02049     Q = eye<T>(N);
02050
02051 for (unsigned k = 1; k < N-1; k++) {
02052     auto v = householder_reflection(H.get_submatrix(k, N-1, k-1, k-1));
02053
02054     auto H1 = H.get_submatrix(k, N-1, 0, N-1);
02055     auto W1 = v * mult<T,true,false>(v, H1);
02056     for (unsigned c = 0; c < N; c++)
02057         for (unsigned r = k; r < N; r++)
02058             H(r,c) -= W1(r-k,c);
02059
02060     auto H2 = H.get_submatrix(0, N-1, k, N-1);
02061     auto W2 = mult<T,false,true>(H2 * v, v);
02062     for (unsigned c = k; c < N; c++)
02063         for (unsigned r = 0; r < N; r++)
02064             H(r,c) -= W2(r,c-k);
02065
02066     if (calculate_Q) {
02067         auto Q1 = Q.get_submatrix(0, N-1, k, N-1);
02068         auto W3 = mult<T,false,true>(Q1 * v, v);
02069         for (unsigned c = k; c < N; c++)
02070             for (unsigned r = 0; r < N; r++)
02071                 Q(r,c) -= W3(r,c-k);
02072     }
02073 }
02074
02075 for (unsigned row = 2; row < N; row++)
02076     for (unsigned col = 0; col < row-2; col++)
02077         H(row,col) = static_cast<T>(0);
02078
02079 return res;
02080 }
02081
02090 template<typename T>
02091 std::complex<T> wilkinson_shift(const Matrix<std::complex<T>& H, T tol = 1e-10) {
02092     if (! H.issquare()) throw std::runtime_error("Input matrix is not square");
02093
02094     const unsigned n = H.rows();
02095     std::complex<T> mu;
02096
02097     if (std::abs(H(n-1,n-2)) < tol) {
02098         mu = H(n-2,n-2);
02099     } else {
02100         auto trA = H(n-2,n-2) + H(n-1,n-1);
02101         auto detA = H(n-2,n-2) * H(n-1,n-1) - H(n-2, n-1) * H(n-1, n-2);
02102         mu = (trA + std::sqrt(trA*trA - 4.0*detA)) / 2.0;
02103     }
02104
02105     return mu;
02106 }
02107
02119 template<typename T>
02120 Eigenvalues_result<T> eigenvalues(const Matrix<std::complex<T>& A, T tol = 1e-12, unsigned max_iter =
100) {
02121     if (! A.issquare()) throw std::runtime_error("Input matrix is not square");
02122
02123     const unsigned N = A.rows();
02124     Matrix<std::complex<T>> H;
02125     bool success = false;
02126
02127     QR_result<std::complex<T>> QR;
02128
02129     // aliases
02130     auto& Q = QR.Q;
02131     auto& R = QR.R;
02132
02133     // Transfer A to Hessenberg form to improve convergence (skip calculation of Q)
02134     H = hessenberg(A, false).H;
02135
02136     for (unsigned iter = 0; iter < max_iter; iter++) {
02137         auto mu = wilkinson_shift(H, tol);
02138
02139         // subtract mu from diagonal
02140         for (unsigned n = 0; n < N; n++)
02141             H(n,n) -= mu;
02142
02143         // QR factorization with shifted H
02144         QR = qr(H);

```

```

02145     H = R * Q;
02146
02147     // add back mu to diagonal
02148     for (unsigned n = 0; n < N; n++)
02149         H(n,n) += mu;
02150
02151     // Check for convergence
02152     if (std::abs(H(N-2,N-1)) <= tol) {
02153         success = true;
02154         break;
02155     }
02156 }
02157
02158 Eigenvalues_result<T> res;
02159 res.eig = diag(H);
02160 res.err = std::abs(H(N-2,N-1));
02161 res.converged = success;
02162
02163 return res;
02164 }
02165
02175 template<typename T>
02176 Eigenvalues_result<T> eigenvalues(const Matrix<T>& A, T tol = 1e-12, unsigned max_iter = 100) {
02177     auto A_cplx = make_complex(A);
02178     return eigenvalues(A_cplx, tol, max_iter);
02179 }
02180
02196 template<typename T>
02197 Matrix<T> solve_triu(const Matrix<T>& U, const Matrix<T>& B) {
02198     if (! U.issquare()) throw std::runtime_error("Input matrix is not square");
02199     if (U.rows() != B.rows()) throw std::runtime_error("Unmatching matrix dimensions for solve");
02200
02201     const unsigned N = U.rows();
02202     const unsigned M = B.cols();
02203
02204     if (U.numel() == 0)
02205         return Matrix<T>();
02206
02207     Matrix<T> X(B);
02208
02209     for (unsigned m = 0; m < M; m++) {
02210         // backwards substitution for each column of B
02211         for (int n = N-1; n >= 0; n--) {
02212             for (unsigned j = n + 1; j < N; j++)
02213                 X(n,m) -= U(n,j) * X(j,m);
02214
02215             if (U(n,n) == 0.0) throw singular_matrix_exception("Singular matrix in solve_triu");
02216
02217             X(n,m) /= U(n,n);
02218         }
02219     }
02220
02221     return X;
02222 }
02223
02239 template<typename T>
02240 Matrix<T> solve_tril(const Matrix<T>& L, const Matrix<T>& B) {
02241     if (! L.issquare()) throw std::runtime_error("Input matrix is not square");
02242     if (L.rows() != B.rows()) throw std::runtime_error("Unmatching matrix dimensions for solve");
02243
02244     const unsigned N = L.rows();
02245     const unsigned M = B.cols();
02246
02247     if (L.numel() == 0)
02248         return Matrix<T>();
02249
02250     Matrix<T> X(B);
02251
02252     for (unsigned m = 0; m < M; m++) {
02253         // forwards substitution for each column of B
02254         for (unsigned n = 0; n < N; n++) {
02255             for (unsigned j = 0; j < n; j++)
02256                 X(n,m) -= L(n,j) * X(j,m);
02257
02258             if (L(n,n) == 0.0) throw singular_matrix_exception("Singular matrix in solve_tril");
02259
02260             X(n,m) /= L(n,n);
02261         }
02262     }
02263
02264     return X;
02265 }
02266
02282 template<typename T>
02283 Matrix<T> solve_square(const Matrix<T>& A, const Matrix<T>& B) {
02284     if (! A.issquare()) throw std::runtime_error("Input matrix is not square");
02285     if (A.rows() != B.rows()) throw std::runtime_error("Unmatching matrix dimensions for solve");

```

```

02286
02287     if (A.numel() == 0)
02288         return Matrix<T>();
02289
02290     Matrix<T> L;
02291     Matrix<T> U;
02292     std::vector<unsigned> P;
02293
02294     // LU decomposition with pivoting
02295     auto lup_res = lup(A);
02296
02297     auto y = solve_tril(lup_res.L, B);
02298     auto x = solve_triu(lup_res.U, y);
02299
02300     return permute_rows(x, lup_res.P);
02301 }
02302
02318 template<typename T>
02319 Matrix<T> solve_posdef(const Matrix<T>& A, const Matrix<T>& B) {
02320     if (!A.issquare()) throw std::runtime_error("Input matrix is not square");
02321     if (A.rows() != B.rows()) throw std::runtime_error("Unmatching matrix dimensions for solve");
02322
02323     if (A.numel() == 0)
02324         return Matrix<T>();
02325
02326     // LU decomposition with pivoting
02327     auto L = chol(A);
02328
02329     auto Y = solve_tril(L, B);
02330     return solve_triu(L.ctranspose(), Y);
02331 }
02332
02337 template<typename T>
02338 class Matrix {
02339     public:
02340         Matrix();
02341
02342         Matrix(unsigned size);
02343
02344         Matrix(unsigned nrows, unsigned ncols);
02345
02346         Matrix(T x, unsigned nrows, unsigned ncols);
02347
02348         Matrix(const T* array, unsigned nrows, unsigned ncols);
02349
02350         Matrix(const std::vector<T>& vec, unsigned nrows, unsigned ncols);
02351
02352         Matrix(std::initializer_list<T> init_list, unsigned nrows, unsigned ncols);
02353
02354         Matrix(const Matrix &);
02355
02356         virtual ~Matrix();
02357
02358         Matrix<T> get_submatrix(unsigned row_first, unsigned row_last, unsigned col_first, unsigned
02359         col_last) const;
02360
02361         void set_submatrix(const Matrix<T>& smtx, unsigned row_first, unsigned col_first);
02362
02363         void clear();
02364
02365         void reshape(unsigned rows, unsigned cols);
02366
02367         void resize(unsigned rows, unsigned cols);
02368
02369         bool exists(unsigned row, unsigned col) const;
02370
02371         T* ptr(unsigned row, unsigned col);
02372
02373         T* ptr();
02374
02375         void fill(T value);
02376
02377         void fill_col(T value, unsigned col);
02378
02379         void fill_row(T value, unsigned row);
02380
02381         bool isempty() const;
02382
02383         bool issquare() const;
02384
02385         bool isequal(const Matrix<T>&) const;
02386
02387         bool isequal(const Matrix<T>&, T) const;
02388
02389         unsigned numel() const;
02390
02391         unsigned rows() const;

```

```

02523
02528     unsigned cols() const;
02529
02534     Matrix<T> transpose() const;
02535
02541     Matrix<T> ctranspose() const;
02542
02550     Matrix<T>& add(const Matrix<T>&);
02551
02559     Matrix<T>& subtract(const Matrix<T>&);
02560
02569     Matrix<T>& mult_hadamard(const Matrix<T>&);
02570
02576     Matrix<T>& add(T);
02577
02583     Matrix<T>& subtract(T);
02584
02590     Matrix<T>& mult(T);
02591
02597     Matrix<T>& div(T);
02598
02603     Matrix<T>& operator=(const Matrix<T>&);
02604
02610     Matrix<T>& operator=(T);
02611
02617     explicit operator std::vector<T>() const;
02618     std::vector<T> to_vector() const;
02619
02626     T& operator()(unsigned nel);
02627     T operator()(unsigned nel) const;
02628     T& at(unsigned nel);
02629     T at(unsigned nel) const;
02630
02637     T& operator()(unsigned row, unsigned col);
02638     T operator()(unsigned row, unsigned col) const;
02639     T& at(unsigned row, unsigned col);
02640     T at(unsigned row, unsigned col) const;
02641
02649     void add_row_to_another(unsigned to, unsigned from);
02650
02658     void add_col_to_another(unsigned to, unsigned from);
02659
02667     void mult_row_by_another(unsigned to, unsigned from);
02668
02676     void mult_col_by_another(unsigned to, unsigned from);
02677
02684     void swap_rows(unsigned i, unsigned j);
02685
02692     void swap_cols(unsigned i, unsigned j);
02693
02700     std::vector<T> col_to_vector(unsigned col) const;
02701
02708     std::vector<T> row_to_vector(unsigned row) const;
02709
02718     void col_from_vector(const std::vector<T>&, unsigned col);
02719
02728     void row_from_vector(const std::vector<T>&, unsigned row);
02729
02730 private:
02731     unsigned nrows;
02732     unsigned ncols;
02733     std::vector<T> data;
02734 };
02735
02736 /*
02737  * Implementation of Matrix class methods
02738  */
02739
02740 template<typename T>
02741 Matrix<T>::Matrix() : nrows(0), ncols(0), data() { }
02742
02743 template<typename T>
02744 Matrix<T>::Matrix(unsigned size) : Matrix(size, size) { }
02745
02746 template<typename T>
02747 Matrix<T>::Matrix(unsigned rows, unsigned cols) : nrows(rows), ncols(cols) {
02748     data.resize(numel());
02749 }
02750
02751 template<typename T>
02752 Matrix<T>::Matrix(T x, unsigned rows, unsigned cols) : Matrix(rows, cols) {
02753     fill(x);
02754 }
02755
02756 template<typename T>
02757 Matrix<T>::Matrix(const T* array, unsigned rows, unsigned cols) : Matrix(rows, cols) {
02758     data.assign(array, array + numel());

```



```

02759 }
02760
02761 template<typename T>
02762 Matrix<T>::Matrix(const std::vector<T>& vec, unsigned rows, unsigned cols) : Matrix(rows, cols) {
02763     if (vec.size() != numel()) throw std::runtime_error("Size of initialization vector not consistent
    with matrix dimensions");
02764
02765     data.assign(vec.begin(), vec.end());
02766 }
02767
02768 template<typename T>
02769 Matrix<T>::Matrix(std::initializer_list<T> init_list, unsigned rows, unsigned cols) : Matrix(rows,
    cols) {
02770     if (init_list.size() != numel()) throw std::runtime_error("Size of initialization list not
    consistent with matrix dimensions");
02771
02772     auto it = init_list.begin();
02773
02774     for (unsigned row = 0; row < this->nrows; row++)
02775         for (unsigned col = 0; col < this->ncols; col++)
02776             this->at(row,col) = *(it++);
02777 }
02778
02779 template<typename T>
02780 Matrix<T>::Matrix(const Matrix & other) : Matrix(other.nrows, other.ncols) {
02781     this->data.assign(other.data.begin(), other.data.end());
02782 }
02783
02784 template<typename T>
02785 Matrix<T>& Matrix<T>::operator=(const Matrix<T>& other) {
02786     this->nrows = other.nrows;
02787     this->ncols = other.ncols;
02788     this->data.assign(other.data.begin(), other.data.end());
02789     return *this;
02790 }
02791
02792 template<typename T>
02793 Matrix<T>& Matrix<T>::operator=(T s) {
02794     fill(s);
02795     return *this;
02796 }
02797
02798 template<typename T>
02799 inline Matrix<T>::operator std::vector<T>() const {
02800     return data;
02801 }
02802
02803 template<typename T>
02804 inline void Matrix<T>::clear() {
02805     this->nrows = 0;
02806     this->ncols = 0;
02807     data.resize(0);
02808 }
02809
02810 template<typename T>
02811 void Matrix<T>::reshape(unsigned rows, unsigned cols) {
02812     if (this->numel() != rows * cols) throw std::runtime_error("Illegal attempt to change number of
    elements via reshape");
02813
02814     this->nrows = rows;
02815     this->ncols = cols;
02816 }
02817
02818 template<typename T>
02819 void Matrix<T>::resize(unsigned rows, unsigned cols) {
02820     this->nrows = rows;
02821     this->ncols = cols;
02822     data.resize(nrows*ncols);
02823 }
02824
02825 template<typename T>
02826 Matrix<T> Matrix<T>::get_submatrix(unsigned row_base, unsigned row_lim, unsigned col_base, unsigned
    col_lim) const {
02827     if (row_base > row_lim) throw std::out_of_range("Row index of submatrix out of range");
02828     if (col_base > col_lim) throw std::out_of_range("Column index of submatrix out of range");
02829     if (row_lim >= this->nrows()) throw std::out_of_range("Row index of submatrix out of range");
02830     if (col_lim >= this->ncols()) throw std::out_of_range("Column index of submatrix out of range");
02831
02832     unsigned num_rows = row_lim - row_base + 1;
02833     unsigned num_cols = col_lim - col_base + 1;
02834     Matrix<T> S(num_rows, num_cols);
02835     for (unsigned i = 0; i < num_rows; i++) {
02836         for (unsigned j = 0; j < num_cols; j++) {
02837             S(i,j) = at(row_base + i, col_base + j);
02838         }
02839     }
02840     return S;

```

```

02841 }
02842
02843 template<typename T>
02844 void Matrix<T>::set_submatrix(const Matrix<T>& S, unsigned row_base, unsigned col_base) {
02845     if (this->isempty()) throw std::runtime_error("Invalid attempt to set submatrix in empty matrix");
02846
02847     const unsigned row_lim = row_base + S.rows() - 1;
02848     const unsigned col_lim = col_base + S.cols() - 1;
02849
02850     if (row_base > row_lim) throw std::out_of_range("Row index of submatrix out of range");
02851     if (col_base > col_lim) throw std::out_of_range("Column index of submatrix out of range");
02852     if (row_lim >= this->rows()) throw std::out_of_range("Row index of submatrix out of range");
02853     if (col_lim >= this->cols()) throw std::out_of_range("Column index of submatrix out of range");
02854
02855     unsigned num_rows = row_lim - row_base + 1;
02856     unsigned num_cols = col_lim - col_base + 1;
02857     for (unsigned i = 0; i < num_rows; i++)
02858         for (unsigned j = 0; j < num_cols; j++)
02859             at(row_base + i, col_base + j) = S(i, j);
02860 }
02861
02862 template<typename T>
02863 inline T & Matrix<T>::operator()(unsigned nel) {
02864     return at(nel);
02865 }
02866
02867 template<typename T>
02868 inline T & Matrix<T>::operator()(unsigned row, unsigned col) {
02869     return at(row, col);
02870 }
02871
02872 template<typename T>
02873 inline T Matrix<T>::operator()(unsigned nel) const {
02874     return at(nel);
02875 }
02876
02877 template<typename T>
02878 inline T Matrix<T>::operator()(unsigned row, unsigned col) const {
02879     return at(row, col);
02880 }
02881
02882 template<typename T>
02883 inline T & Matrix<T>::at(unsigned nel) {
02884     if (!(nel < numel())) throw std::out_of_range("Element index out of range");
02885
02886     return data[nel];
02887 }
02888
02889 template<typename T>
02890 inline T & Matrix<T>::at(unsigned row, unsigned col) {
02891     if (!(row < rows() && col < cols())) throw std::out_of_range("Element index out of range");
02892
02893     return data[nrows * col + row];
02894 }
02895
02896 template<typename T>
02897 inline T Matrix<T>::at(unsigned nel) const {
02898     if (!(nel < numel())) throw std::out_of_range("Element index out of range");
02899
02900     return data[nel];
02901 }
02902
02903 template<typename T>
02904 inline T Matrix<T>::at(unsigned row, unsigned col) const {
02905     if (!(row < rows())) throw std::out_of_range("Row index out of range");
02906     if (!(col < cols())) throw std::out_of_range("Column index out of range");
02907
02908     return data[nrows * col + row];
02909 }
02910
02911 template<typename T>
02912 inline void Matrix<T>::fill(T value) {
02913     for (unsigned i = 0; i < numel(); i++)
02914         data[i] = value;
02915 }
02916
02917 template<typename T>
02918 inline void Matrix<T>::fill_col(T value, unsigned col) {
02919     if (!(col < cols())) throw std::out_of_range("Column index out of range");
02920
02921     for (unsigned i = col * nrows; i < (col+1) * nrows; i++)
02922         data[i] = value;
02923 }
02924
02925 template<typename T>
02926 inline void Matrix<T>::fill_row(T value, unsigned row) {
02927     if (!(row < rows())) throw std::out_of_range("Row index out of range");

```

```

02928
02929     for (unsigned i = 0; i < ncols; i++)
02930         data[row + i * nrows] = value;
02931 }
02932
02933 template<typename T>
02934 inline bool Matrix<T>::exists(unsigned row, unsigned col) const {
02935     return (row < nrows && col < ncols);
02936 }
02937
02938 template<typename T>
02939 inline T* Matrix<T>::ptr(unsigned row, unsigned col) {
02940     if (!(row < rows())) throw std::out_of_range("Row index out of range");
02941     if (!(col < cols())) throw std::out_of_range("Column index out of range");
02942
02943     return data.data() + nrows * col + row;
02944 }
02945
02946 template<typename T>
02947 inline T* Matrix<T>::ptr() {
02948     return data.data();
02949 }
02950
02951 template<typename T>
02952 inline bool Matrix<T>::isempty() const {
02953     return (nrows == 0) || (ncols == 0);
02954 }
02955
02956 template<typename T>
02957 inline bool Matrix<T>::issquare() const {
02958     return (nrows == ncols) && !isempty();
02959 }
02960
02961 template<typename T>
02962 bool Matrix<T>::isequal(const Matrix<T>& A) const {
02963     bool ret = true;
02964     if (nrows != A.rows() || ncols != A.cols()) {
02965         ret = false;
02966     } else {
02967         for (unsigned i = 0; i < numel(); i++) {
02968             if (at(i) != A(i)) {
02969                 ret = false;
02970                 break;
02971             }
02972         }
02973     }
02974     return ret;
02975 }
02976
02977 template<typename T>
02978 bool Matrix<T>::isequal(const Matrix<T>& A, T tol) const {
02979     bool ret = true;
02980     if (rows() != A.rows() || cols() != A.cols()) {
02981         ret = false;
02982     } else {
02983         auto abs_tol = std::abs(tol); // workaround for complex
02984         for (unsigned i = 0; i < A.numel(); i++) {
02985             if (abs_tol < std::abs(at(i) - A(i))) {
02986                 ret = false;
02987                 break;
02988             }
02989         }
02990     }
02991     return ret;
02992 }
02993
02994 template<typename T>
02995 inline unsigned Matrix<T>::numel() const {
02996     return nrows * ncols;
02997 }
02998
02999 template<typename T>
03000 inline unsigned Matrix<T>::rows() const {
03001     return nrows;
03002 }
03003
03004 template<typename T>
03005 inline unsigned Matrix<T>::cols() const {
03006     return ncols;
03007 }
03008
03009 template<typename T>
03010 inline Matrix<T> Matrix<T>::transpose() const {
03011     Matrix<T> res(ncols, nrows);
03012     for (unsigned c = 0; c < ncols; c++)
03013         for (unsigned r = 0; r < nrows; r++)
03014             res(c,r) = at(r,c);

```

```

03015     return res;
03016 }
03017
03018 template<typename T>
03019 inline Matrix<T> Matrix<T>::ctranspose() const {
03020     Matrix<T> res(ncols, nrows);
03021     for (unsigned c = 0; c < ncols; c++)
03022         for (unsigned r = 0; r < nrows; r++)
03023             res(c,r) = cconj(at(r,c));
03024     return res;
03025 }
03026
03027 template<typename T>
03028 Matrix<T>& Matrix<T>::add(const Matrix<T>& m) {
03029     if (!(m.rows() == rows() && m.cols() == cols())) throw std::runtime_error("Unmatching matrix
dimensions for iadd");
03030
03031     for (unsigned i = 0; i < numel(); i++)
03032         data[i] += m(i);
03033     return *this;
03034 }
03035
03036 template<typename T>
03037 Matrix<T>& Matrix<T>::subtract(const Matrix<T>& m) {
03038     if (!(m.rows() == rows() && m.cols() == cols())) throw std::runtime_error("Unmatching matrix
dimensions for isubtract");
03039
03040     for (unsigned i = 0; i < numel(); i++)
03041         data[i] -= m(i);
03042     return *this;
03043 }
03044
03045 template<typename T>
03046 Matrix<T>& Matrix<T>::mult_hadamard(const Matrix<T>& m) {
03047     if (!(m.rows() == rows() && m.cols() == cols())) throw std::runtime_error("Unmatching matrix
dimensions for ihprod");
03048
03049     for (unsigned i = 0; i < numel(); i++)
03050         data[i] *= m(i);
03051     return *this;
03052 }
03053
03054 template<typename T>
03055 Matrix<T>& Matrix<T>::add(T s) {
03056     for (auto& x : data)
03057         x += s;
03058     return *this;
03059 }
03060
03061 template<typename T>
03062 Matrix<T>& Matrix<T>::subtract(T s) {
03063     for (auto& x : data)
03064         x -= s;
03065     return *this;
03066 }
03067
03068 template<typename T>
03069 Matrix<T>& Matrix<T>::mult(T s) {
03070     for (auto& x : data)
03071         x *= s;
03072     return *this;
03073 }
03074
03075 template<typename T>
03076 Matrix<T>& Matrix<T>::div(T s) {
03077     for (auto& x : data)
03078         x /= s;
03079     return *this;
03080 }
03081
03082 template<typename T>
03083 void Matrix<T>::add_row_to_another(unsigned to, unsigned from) {
03084     if (!(to < rows() && from < rows())) throw std::out_of_range("Row index out of range");
03085
03086     for (unsigned k = 0; k < cols(); k++)
03087         at(to, k) += at(from, k);
03088 }
03089
03090 template<typename T>
03091 void Matrix<T>::add_col_to_another(unsigned to, unsigned from) {
03092     if (!(to < cols() && from < cols())) throw std::out_of_range("Column index out of range");
03093
03094     for (unsigned k = 0; k < rows(); k++)
03095         at(k, to) += at(k, from);
03096 }
03097
03098 template<typename T>

```

```

03099 void Matrix<T>::mult_row_by_another(unsigned to, unsigned from) {
03100     if (!(to < rows() && from < rows())) throw std::out_of_range("Row index out of range");
03101
03102     for (unsigned k = 0; k < cols(); k++)
03103         at(to, k) *= at(from, k);
03104 }
03105
03106 template<typename T>
03107 void Matrix<T>::mult_col_by_another(unsigned to, unsigned from) {
03108     if (!(to < cols() && from < cols())) throw std::out_of_range("Column index out of range");
03109
03110     for (unsigned k = 0; k < rows(); k++)
03111         at(k, to) *= at(k, from);
03112 }
03113
03114 template<typename T>
03115 void Matrix<T>::swap_rows(unsigned i, unsigned j) {
03116     if (!(i < rows() && j < rows())) throw std::out_of_range("Row index out of range");
03117
03118     for (unsigned k = 0; k < cols(); k++) {
03119         T tmp = at(i, k);
03120         at(i, k) = at(j, k);
03121         at(j, k) = tmp;
03122     }
03123 }
03124
03125 template<typename T>
03126 void Matrix<T>::swap_cols(unsigned i, unsigned j) {
03127     if (!(i < cols() && j < cols())) throw std::out_of_range("Column index out of range");
03128
03129     for (unsigned k = 0; k < rows(); k++) {
03130         T tmp = at(k, i);
03131         at(k, i) = at(k, j);
03132         at(k, j) = tmp;
03133     }
03134 }
03135
03136 template<typename T>
03137 inline std::vector<T> Matrix<T>::to_vector() const {
03138     return data;
03139 }
03140
03141 template<typename T>
03142 inline std::vector<T> Matrix<T>::col_to_vector(unsigned col) const {
03143     std::vector<T> vec(rows());
03144     for (unsigned i = 0; i < rows(); i++)
03145         vec[i] = at(i, col);
03146     return vec;
03147 }
03148
03149 template<typename T>
03150 inline std::vector<T> Matrix<T>::row_to_vector(unsigned row) const {
03151     std::vector<T> vec(cols());
03152     for (unsigned i = 0; i < cols(); i++)
03153         vec[i] = at(row, i);
03154     return vec;
03155 }
03156
03157 template<typename T>
03158 inline void Matrix<T>::col_from_vector(const std::vector<T>& vec, unsigned col) {
03159     if (vec.size() != rows()) throw std::runtime_error("Vector size is not equal to number of rows");
03160     if (col >= cols()) throw std::out_of_range("Column index out of range");
03161
03162     for (unsigned i = 0; i < rows(); i++)
03163         data[col*rows() + i] = vec[i];
03164 }
03165
03166 template<typename T>
03167 inline void Matrix<T>::row_from_vector(const std::vector<T>& vec, unsigned row) {
03168     if (vec.size() != cols()) throw std::runtime_error("Vector size is not equal to number of columns");
03169     if (row >= rows()) throw std::out_of_range("Row index out of range");
03170
03171     for (unsigned i = 0; i < cols(); i++)
03172         data[row + i*rows()] = vec[i];
03173 }
03174
03175 template<typename T>
03176 Matrix<T>::~Matrix() { }
03177
03178 } // namespace Matrix_hpp
03179
03180 #endif // __MATRIX_HPP__

```

