

## 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()	19
5.6.3.9 ctranspose()	20
5.6.3.10 div()	20
5.6.3.11 exists()	20
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()	21
5.6.3.16 isempty()	22
5.6.3.17 isequal() [1/2]	22
5.6.3.18 isequal() [2/2]	22
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]	24
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]	25
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]	28
5.6.3.39 swap_cols()	29
5.6.3.40 swap_rows()	29
5.6.3.41 transpose()	29
5.7 Mtx::QR_result< T > Struct Template Reference	29
5.7.1 Detailed Description	30
5.8 Mtx::singular_matrix_exception Class Reference	30

6.1 examples.cpp File Reference	31
6.1.1 Detailed Description	31
6.2 matrix.hpp File Reference	31
6.2.1 Function Documentation	37
6.2.1.1 add() [1/2]	37
6.2.1.2 add() [2/2]	38
6.2.1.3 adj()	38
6.2.1.4 cconj()	38
6.2.1.5 chol()	39
6.2.1.6 cholinv()	39
6.2.1.7 circshift()	40
6.2.1.8 circulant() [1/2]	40
6.2.1.9 circulant() [2/2]	41
6.2.1.10 cofactor()	41
6.2.1.11 concatenate_horizontal()	42
6.2.1.12 concatenate_vertical()	42
6.2.1.13 cond()	43
6.2.1.14 csign()	43
6.2.1.15 ctranspose()	43
6.2.1.16 det()	43
6.2.1.17 det_lu()	44
6.2.1.18 diag() [1/3]	44
6.2.1.19 diag() [2/3]	45
6.2.1.20 diag() [3/3]	45
6.2.1.21 div()	46
6.2.1.22 eigenvalues() [1/2]	46
6.2.1.23 eigenvalues() [2/2]	46
6.2.1.24 eye()	47
6.2.1.25 foreach_elem()	47
6.2.1.26 foreach_elem_copy()	48
6.2.1.27 hessenberg()	48
6.2.1.28 householder_reflection()	49
6.2.1.29 imag()	49
6.2.1.30 inv()	50
6.2.1.31 inv_gauss_jordan()	50
6.2.1.32 inv_posdef()	50
6.2.1.33 inv_square()	51
6.2.1.34 inv_tril()	51
6.2.1.35 inv_triu()	52
6.2.1.36 ishess()	52
6.2.1.37 istril()	53
6.2.1.38 istriu()	53

6.2.1.39 kron()	53
6.2.1.40 ldl()	53
6.2.1.41 lu()	54
6.2.1.42 lup()	54
6.2.1.43 make_complex() [1/2]	55
6.2.1.44 make_complex() [2/2]	55
6.2.1.45 mult() [1/4]	56
6.2.1.46 mult() [2/4]	57
6.2.1.47 mult() [3/4]	57
6.2.1.48 mult() [4/4]	58
6.2.1.49 mult_hadamard()	59
6.2.1.50 norm_fro() [1/2]	60
6.2.1.51 norm_fro() [2/2]	60
6.2.1.52 ones() [1/2]	60
6.2.1.53 ones() [2/2]	61
6.2.1.54 operator!=()	61
6.2.1.55 operator*() [1/5]	61
6.2.1.56 operator*() [2/5]	62
6.2.1.57 operator*() [3/5]	62
6.2.1.58 operator*() [4/5]	62
6.2.1.59 operator*() [5/5]	62
6.2.1.60 operator*=( ) [1/2]	63
6.2.1.61 operator*=( ) [2/2]	63
6.2.1.62 operator+( ) [1/3]	63
6.2.1.63 operator+( ) [2/3]	63
6.2.1.64 operator+( ) [3/3]	64
6.2.1.65 operator+=( ) [1/2]	64
6.2.1.66 operator+=( ) [2/2]	64
6.2.1.67 operator-( ) [1/2]	64
6.2.1.68 operator-( ) [2/2]	65
6.2.1.69 operator-=( ) [1/2]	65
6.2.1.70 operator-=( ) [2/2]	65
6.2.1.71 operator/( )	65
6.2.1.72 operator/=( )	66
6.2.1.73 operator<<()	66
6.2.1.74 operator==( )	66
6.2.1.75 operator^()	66
6.2.1.76 operator^=( )	67
6.2.1.77 permute_cols()	67
6.2.1.78 permute_rows()	67
6.2.1.79 pinv()	68
6.2.1.80 qr()	68

---

6.2.1.81 qr_householder()	69
6.2.1.82 qr_red_gs()	69
6.2.1.83 real()	70
6.2.1.84 repmat()	70
6.2.1.85 solve_posdef()	71
6.2.1.86 solve_square()	71
6.2.1.87 solve_tril()	72
6.2.1.88 solve_triu()	73
6.2.1.89 subtract() [1/2]	73
6.2.1.90 subtract() [2/2]	74
6.2.1.91 trace()	74
6.2.1.92 transpose()	75
6.2.1.93 tril()	75
6.2.1.94 triu()	75
6.2.1.95 wilkinson_shift()	75
6.2.1.96 zeros() [1/2]	76
6.2.1.97 zeros() [2/2]	76
6.3 matrix.hpp	77





## 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: [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.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 > . . . . .	29



## 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 . . . . .	29
<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="#">examples.cpp</a>	.....	31
<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 `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 [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 [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 [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 [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.

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

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

## 5.6.2.8 Matrix() [8/8]

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

Copy constructor.

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

## 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\(\)](#), 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
--------------------------------	-----------------------------------

**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
--------------------------------	--------------------------------

### 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.

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

### 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

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

### 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

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

### 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 value of the second dimension.

Referenced by [Mtx::Matrix< T >::add\(\)](#), [Mtx::add\(\)](#), [Mtx::add\(\)](#), [Mtx::adj\(\)](#), [Mtx::circshift\(\)](#), [Mtx::cofactor\(\)](#), [Mtx::concatenate\\_horizontal\(\)](#), [Mtx::concatenate\\_vertical\(\)](#), [Mtx::div\(\)](#), [Mtx::householder\\_reflection\(\)](#), [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\\_hadamard\(\)](#), [Mtx::mult\\_hadamard\(\)](#), [Mtx::operator<<\(\)](#), [Mtx::permute\\_cols\(\)](#), [Mtx::permute\\_rows\(\)](#), [Mtx::pinv\(\)](#), [Mtx::qr\\_householder\(\)](#), [Mtx::qr\\_red\\_gs\(\)](#), [Mtx::repmat\(\)](#), [Mtx::Matrix< T >::set\\_submatrix\(\)](#), [Mtx::solve\\_tril\(\)](#), [Mtx::solve\\_triu\(\)](#), [Mtx::Matrix< T >::subtract\(\)](#), [Mtx::subtract\(\)](#), [Mtx::subtract\(\)](#), [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\(\)](#).

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.

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

**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
--------------------------------	-----------------------------------

**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
--------------------------------	--------------------------------

**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
--------------------------------	---

Referenced by [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.

#### 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\(\)](#), 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 >::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::Matrix< T >::Matrix\(\)](#), [Mtx::Matrix< T >::Matrix\(\)](#), [Mtx::Matrix< T >::Matrix\(\)](#), [Mtx::Matrix< T >::Matrix\(\)](#), [Mtx::Matrix< T >::Matrix\(\)](#), and [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
--------------------------------	-----------------------------------

## 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\(\)](#), 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
--------------------------------	--------------------------------

### 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::add\(\)](#), [Mtx::div\(\)](#), [Mtx::foreach\\_elem\(\)](#), [Mtx::householder\\_reflection\(\)](#), [Mtx::inv\(\)](#), [Mtx::Matrix< T >::isegal\(\)](#), [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::norm\\_fro\(\)](#), [Mtx::solve\\_posdef\(\)](#), [Mtx::solve\\_square\(\)](#), [Mtx::solve\\_tril\(\)](#), [Mtx::solve\\_triu\(\)](#), 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.

## 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.

**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
---------------------------------	--

**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.

Referenced by [Mtx::Matrix< T >::clear\(\)](#).

### 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 columnc
<code>std::out_of_range</code>	when row index out of range

### 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
--------------------------------	--------------------------------

### 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 value of the first dimension.

Referenced by `Mtx::Matrix< T >::add()`, `Mtx::add()`, `Mtx::add()`, `Mtx::adj()`, `Mtx::chol()`, `Mtx::cholinv()`, `Mtx::circshift()`, `Mtx::cofactor()`, `Mtx::concatenate_horizontal()`, `Mtx::concatenate_vertical()`, `Mtx::det()`, `Mtx::diag()`, `Mtx::div()`, `Mtx::eigenvalues()`, `Mtx::hessenberg()`, `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_hadamard()`, `Mtx::mult_hadamard()`, `Mtx::operator<<()`, `Mtx::permute_cols()`, `Mtx::permute_rows()`, `Mtx::pinv()`, `Mtx::qr_householder()`, `Mtx::qr_red_gs()`, `Mtx::repmat()`, `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::trace()`, `Mtx::tril()`, and `Mtx::triu()`.

### 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

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

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

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

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

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

### 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

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

### 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.

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 [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 examples.cpp File Reference

#### 6.1.1 Detailed Description

Provides various examples of [matrix.hpp](#) library usage.

### 6.2 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 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, 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 (elementwise) 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 pseudoinverse.*
- `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::cholin (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.*

## 6.2.1 Function Documentation

### 6.2.1.1 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 `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+\(\)](#).

**6.2.1.2 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\(\)](#).

**6.2.1.3 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\(\)](#).

**6.2.1.4 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\(\)](#).

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

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

### 6.2.1.6 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 `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

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

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

#### 6.2.1.7 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 bottom. A negative value may be used to apply shifts in opposite directions.

#### Parameters

<code>A</code>	matrix
<code>row_shift</code>	row shift factor
<code>col_shift</code>	column shift factor

#### Returns

matrix inverse

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

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

#### 6.2.1.8 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\(\)](#).

**6.2.1.9 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\(\)](#).

**6.2.1.10 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

$A$	input square matrix
$p$	row to be deleted in the output matrix
$q$	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 $p$ or column index $q$ are out of range
<code>std::runtime_error</code>	when input matrix $A$ 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\(\)](#).

**6.2.1.11 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\(\)](#).

**6.2.1.12 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\(\)](#).

### 6.2.1.13 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 the sensitivity of a system solution of linear equations to errors in the 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.

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

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

### 6.2.1.14 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\(\)](#).

### 6.2.1.15 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\(\)](#).

### 6.2.1.16 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\(\)](#).

**6.2.1.17 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\(\)](#), and [Mtx::lup\(\)](#).

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

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

**6.2.1.19 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\(\)](#).

**6.2.1.20 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\(\)](#).

### 6.2.1.21 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/\(\)](#).

### 6.2.1.22 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::Eigenvalues\\_result< T >::converged](#), [Mtx::diag\(\)](#), [Mtx::Eigenvalues\\_result< T >::eig](#), [Mtx::eigenvalues\(\)](#), [Mtx::Eigenvalues\\_result< T >::err](#), [Mtx::hessenberg\(\)](#), [Mtx::Matrix< T >::issquare\(\)](#), [Mtx::QR\\_result< T >::Q](#), [Mtx::qr\(\)](#), [Mtx::QR\\_result< T >::R](#), [Mtx::Matrix< T >::rows\(\)](#), and [Mtx::wilkinson\\_shift\(\)](#).

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

### 6.2.1.23 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\(\)](#).

#### 6.2.1.24 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\(\)](#).

#### 6.2.1.25 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 `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\(\)](#).

**6.2.1.26 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 `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\(\)](#).

**6.2.1.27 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::Hessenberg\\_result< T >::H](#), [Mtx::hessenberg\(\)](#), [Mtx::householder\\_reflection\(\)](#), [Mtx::Matrix< T >::issquare\(\)](#), [Mtx::Hessenberg\\_result< T >::Q](#), and [Mtx::Matrix< T >::rows\(\)](#).

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

## 6.2.1.28 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\(\)](#).

## 6.2.1.29 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::imag\(\)](#).

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

### 6.2.1.30 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\(\)](#).

### 6.2.1.31 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 [inv\(\)](#) function instead of this one offers better performance for matrices of size smaller than 4.

#### 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\(\)](#).

### 6.2.1.32 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 `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\(\)](#).

#### 6.2.1.33 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 `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\(\)](#).

#### 6.2.1.34 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 `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\(\)](#).

#### 6.2.1.35 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 `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\(\)](#).

#### 6.2.1.36 ishess()

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

Hessenberg matrix check.

Return true if A is a, 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\(\)](#).

### 6.2.1.37 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\(\)](#).

### 6.2.1.38 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\(\)](#).

### 6.2.1.39 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\(\)](#).

### 6.2.1.40 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::conj\(\)](#), [Mtx::LDL\\_result< T >::d](#), [Mtx::Matrix< T >::issquare\(\)](#), [Mtx::LDL\\_result< T >::L](#), [Mtx::ldl\(\)](#), and [Mtx::Matrix< T >::rows\(\)](#).

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

## 6.2.1.41 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 `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\\_result< T >::L](#), [Mtx::lu\(\)](#), [Mtx::Matrix< T >::numel\(\)](#), [Mtx::Matrix< T >::rows\(\)](#), and [Mtx::LU\\_result< T >::U](#).

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

## 6.2.1.42 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\\_result< T >::L](#), [Mtx::lup\(\)](#), [Mtx::Matrix< T >::numel\(\)](#), [Mtx::LUP\\_result< T >::P](#), [Mtx::Matrix< T >::rows\(\)](#), and [Mtx::LUP\\_result< T >::U](#).

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

#### 6.2.1.43 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\(\)](#).

#### 6.2.1.44 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\(\)](#).

**6.2.1.45 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 `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::cconj\(\)](#), [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\\*\(\)=\(\)](#).



**6.2.1.46 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 the operation is also a std::vector.

This function supports template parameterization of input matrix transposition, providing better efficiency than in case of using transpose() 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::conj\(\)](#), [Mtx::Matrix< T >::cols\(\)](#), [Mtx::mult\(\)](#), and [Mtx::Matrix< T >::rows\(\)](#).

**6.2.1.47 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\(\)](#).

#### 6.2.1.48 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 the operation is also a std::vector.

This function supports template parameterization of input matrix transposition, providing better efficiency than in case of using 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\(\)](#).

## 6.2.1.49 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 (elementwise) multiplication.

Performs Hadamard (elementwise) multiplication of two matrices.

This function supports template parameterization of input matrix transposition, providing better efficiency than in case of using `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^\(\)](#).

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

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

Frobenius norm of 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\(\)](#).

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

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

Frobenius norm.

Calculates Frobenius norm of 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\(\)](#).

**6.2.1.52 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\(\)](#).

**6.2.1.53 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\(\)](#).

**6.2.1.54 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!=\(\(\)\)](#).

**6.2.1.55 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\\*\(\)](#).

**6.2.1.56 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\\*\(\)](#).

**6.2.1.57 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\\*\(\)](#).

**6.2.1.58 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\\*\(\)](#).

**6.2.1.59 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\\*\(\)](#).

**6.2.1.60 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\\*=\( \)](#).

**6.2.1.61 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\\*=\( \)](#).

**6.2.1.62 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+\( \)](#).

**6.2.1.63 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+\( \)](#).

**6.2.1.64 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+\(\)](#).

**6.2.1.65 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+=\(\)](#).

**6.2.1.66 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+=\(\)](#).

**6.2.1.67 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-\(\)](#).



**6.2.1.68 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\(\)](#).

**6.2.1.69 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-=\(\)](#).

**6.2.1.70 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\(\)](#).

**6.2.1.71 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/\(\)](#).

**6.2.1.72 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/=\(\)](#).

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

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

**6.2.1.74 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==\(\)](#).

**6.2.1.75 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^\(\)](#).

**6.2.1.76 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^=\(\)](#).

**6.2.1.77 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**

$A$	input matrix
$perm$	permutation vector with column indices

**Returns**

output matrix created by column permutation of  $A$

**Exceptions**

<code>std::runtime_error</code>	when permutation vector 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\\_cols\(\)](#), and [Mtx::Matrix< T >::rows\(\)](#).

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

**6.2.1.78 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\(\)](#).

#### 6.2.1.79 pinv()

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

Moore-Penrose pseudoinverse.

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

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

#### 6.2.1.80 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 `qr_householder()`. Refer to `qr_red_gs()` for alternative implementation.

## Parameters

$A$	input matrix to be decomposed
$\text{calculate\_}Q$	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  $\text{calculate\_}Q = \text{True}$ .

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

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

## 6.2.1.81 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 $n \times m$
$\text{calculate\_}Q$	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  $\text{calculate\_}Q = \text{True}$ .

References [Mtx::Matrix< T >::cols\(\)](#), [Mtx::householder\\_reflection\(\)](#), [Mtx::QR\\_result< T >::Q](#), [Mtx::qr\\_householder\(\)](#), [Mtx::QR\\_result< T >::R](#), and [Mtx::Matrix< T >::rows\(\)](#).

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

## 6.2.1.82 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

$A$	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\\_result< T >::Q](#), [Mtx::qr\\_red\\_gs\(\)](#), [Mtx::QR\\_result< T >::R](#), and [Mtx::Matrix< T >::rows\(\)](#).

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

#### 6.2.1.83 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::real\(\)](#).

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

#### 6.2.1.84 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\(\)](#).

**6.2.1.85 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

<i>A</i>	left side matrix of size $N \times N$ . Must be square and positive definite.
<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::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\(\)](#).

**6.2.1.86 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

$A$	left side matrix of size $N \times N$ . Must be square.
$B$	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 >::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\(\)](#).

#### 6.2.1.87 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

$L$	left side matrix of size $N \times N$ . Must be square and lower triangular
$B$	right hand side matrix of size $N \times M$ .



## Returns

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

## Exceptions

<code>std::runtime_error</code>	when the input matrix is not square
<code>std::runtime_error</code>	when number of rows is not equal between input matrices
<code>singular_matrix_exception</code>	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\(\)](#).

## 6.2.1.88 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

<code>std::runtime_error</code>	when the input matrix is not square
<code>std::runtime_error</code>	when number of rows is not equal between input matrices
<code>singular_matrix_exception</code>	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\(\)](#).

## 6.2.1.89 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 `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\(\)](#).

#### 6.2.1.90 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\(\)](#).

#### 6.2.1.91 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\(\)](#).

### 6.2.1.92 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\(\)](#).

### 6.2.1.93 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\(\)](#).

### 6.2.1.94 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\(\)](#).

### 6.2.1.95 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::wilkinson\\_shift\(\)](#).

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

**6.2.1.96 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\(\)](#).

**6.2.1.97 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\(\)](#).

## 6.3 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 
```

```

00118
00121     std::vector<unsigned> P;
00122 };
00123
00129 template<typename T>
00130 struct QR_result {
00133     Matrix<T> Q;
00134
00137     Matrix<T> R;
00138 };
00139
00144 template<typename T>
00145 struct Hessenberg_result {
00148     Matrix<T> H;
00149
00152     Matrix<T> Q;
00153 };
00154
00159 template<typename T>
00160 struct LDL_result {
00163     Matrix<T> L;
00164
00167     std::vector<T> d;
00168 };
00169
00174 template<typename T>
00175 struct Eigenvalues_result {
00178     std::vector<std::complex<T>> eig;
00179
00182     bool converged;
00183
00186     T err;
00187 };
00188
00189
00197 template<typename T>
00198 inline Matrix<T> zeros(unsigned nrows, unsigned ncols) {
00199     return Matrix<T>(static_cast<T>(0), nrows, ncols);
00200 }
00201
00208 template<typename T>
00209 inline Matrix<T> zeros(unsigned n) {
00210     return zeros<T>(n,n);
00211 }
00212
00221 template<typename T>
00222 inline Matrix<T> ones(unsigned nrows, unsigned ncols) {
00223     return Matrix<T>(static_cast<T>(1), nrows, ncols);
00224 }
00225
00233 template<typename T>
00234 inline Matrix<T> ones(unsigned n) {
00235     return ones<T>(n,n);
00236 }
00237
00245 template<typename T>
00246 Matrix<T> eye(unsigned n) {
00247     Matrix<T> A(static_cast<T>(0), n, n);
00248     for (unsigned i = 0; i < n; i++)
00249         A(i,i) = static_cast<T>(1);
00250     return A;
00251 }
00252
00260 template<typename T>
00261 Matrix<T> diag(const T* array, size_t n) {
00262     Matrix<T> A(static_cast<T>(0), n, n);
00263     for (unsigned i = 0; i < n; i++) {
00264         A(i,i) = array[i];
00265     }
00266     return A;
00267 }
00268
00276 template<typename T>
00277 inline Matrix<T> diag(const std::vector<T>& v) {
00278     return diag(v.data(), v.size());
00279 }
00280
00289 template<typename T>
00290 std::vector<T> diag(const Matrix<T>& A) {
00291     if (! A.issquare()) throw std::runtime_error("Input matrix is not square");
00292
00293     std::vector<T> v;
00294     v.resize(A.rows());
00295
00296     for (unsigned i = 0; i < A.rows(); i++)
00297         v[i] = A(i,i);
00298     return v;

```

```

00299 }
00300
00308 template<typename T>
00309 Matrix<T> circulant(const T* array, unsigned n) {
00310     Matrix<T> A(n, n);
00311     for (unsigned j = 0; j < n; j++)
00312         for (unsigned i = 0; i < n; i++)
00313             A((i+j) % n, j) = array[i];
00314     return A;
00315 }
00316
00327 template<typename T>
00328 Matrix<std::complex<T>> make_complex(const Matrix<T>& Re, const Matrix<T>& Im) {
00329     if (Re.rows() != Im.rows() || Re.cols() != Im.cols()) throw std::runtime_error("Size of input
matrices does not match");
00330
00331     Matrix<std::complex<T>> C(Re.rows(), Re.cols());
00332     for (unsigned n = 0; n < Re.numel(); n++) {
00333         C(n).real(Re(n));
00334         C(n).imag(Im(n));
00335     }
00336
00337     return C;
00338 }
00339
00346 template<typename T>
00347 Matrix<std::complex<T>> make_complex(const Matrix<T>& Re) {
00348     Matrix<std::complex<T>> C(Re.rows(), Re.cols());
00349
00350     for (unsigned n = 0; n < Re.numel(); n++) {
00351         C(n).real(Re(n));
00352         C(n).imag(static_cast<T>(0));
00353     }
00354
00355     return C;
00356 }
00357
00362 template<typename T>
00363 Matrix<T> real(const Matrix<std::complex<T>& C) {
00364     Matrix<T> Re(C.rows(), C.cols());
00365
00366     for (unsigned n = 0; n < C.numel(); n++)
00367         Re(n) = C(n).real();
00368
00369     return Re;
00370 }
00371
00376 template<typename T>
00377 Matrix<T> imag(const Matrix<std::complex<T>& C) {
00378     Matrix<T> Re(C.rows(), C.cols());
00379
00380     for (unsigned n = 0; n < C.numel(); n++)
00381         Re(n) = C(n).imag();
00382
00383     return Re;
00384 }
00385
00393 template<typename T>
00394 inline Matrix<T> circulant(const std::vector<T>& v) {
00395     return circulant(v.data(), v.size());
00396 }
00397
00402 template<typename T>
00403 inline Matrix<T> transpose(const Matrix<T>& A) {
00404     return A.transpose();
00405 }
00406
00412 template<typename T>
00413 inline Matrix<T> ctranspose(const Matrix<T>& A) {
00414     return A.ctranspose();
00415 }
00416
00427 template<typename T>
00428 Matrix<T> circshift(const Matrix<T>& A, int row_shift, int col_shift) {
00429     Matrix<T> B(A.rows(), A.cols());
00430     for (int i = 0; i < A.rows(); i++) {
00431         int ii = (i + row_shift) % A.rows();
00432         for (int j = 0; j < A.cols(); j++) {
00433             int jj = (j + col_shift) % A.cols();
00434             B(ii, jj) = A(i, j);
00435         }
00436     }
00437     return B;
00438 }
00439
00447 template<typename T>
00448 Matrix<T> repmat(const Matrix<T>& A, unsigned m, unsigned n) {

```

```

00449     Matrix<T> B(m * A.rows(), n * A.cols());
00450
00451     for (unsigned cb = 0; cb < n; cb++)
00452         for (unsigned rb = 0; rb < m; rb++)
00453             for (unsigned c = 0; c < A.cols(); c++)
00454                 for (unsigned r = 0; r < A.rows(); r++)
00455                     B(rb*A.rows() + r, cb*A.cols() + c) = A(r, c);
00456
00457     return B;
00458 }
00459
00466 template<typename T>
00467 Matrix<T> concatenate_horizontal(const Matrix<T>& A, const Matrix<T>& B) {
00468     if (A.rows() != B.rows()) throw std::runtime_error("Unmatching number of rows for horizontal
concatenation");
00469
00470     Matrix<T> C(A.rows(), A.cols() + B.cols());
00471
00472     for (unsigned c = 0; c < A.cols(); c++)
00473         for (unsigned r = 0; r < A.rows(); r++)
00474             C(r,c) = A(r,c);
00475
00476     for (unsigned c = 0; c < B.cols(); c++)
00477         for (unsigned r = 0; r < B.rows(); r++)
00478             C(r,c+A.cols()) = B(r,c);
00479
00480     return C;
00481 }
00482
00489 template<typename T>
00490 Matrix<T> concatenate_vertical(const Matrix<T>& A, const Matrix<T>& B) {
00491     if (A.cols() != B.cols()) throw std::runtime_error("Unmatching number of columns for vertical
concatenation");
00492
00493     Matrix<T> C(A.rows() + B.rows(), A.cols());
00494
00495     for (unsigned c = 0; c < A.cols(); c++)
00496         for (unsigned r = 0; r < A.rows(); r++)
00497             C(r,c) = A(r,c);
00498
00499     for (unsigned c = 0; c < B.cols(); c++)
00500         for (unsigned r = 0; r < B.rows(); r++)
00501             C(r+A.rows(),c) = B(r,c);
00502
00503     return C;
00504 }
00505
00511 template<typename T>
00512 double norm_fro(const Matrix<T>& A) {
00513     double sum = 0;
00514
00515     for (unsigned i = 0; i < A.numel(); i++)
00516         sum += A(i) * A(i);
00517
00518     return std::sqrt(sum);
00519 }
00520
00526 template<typename T>
00527 double norm_fro(const Matrix<std::complex<T>>& A) {
00528     double sum = 0;
00529
00530     for (unsigned i = 0; i < A.numel(); i++) {
00531         T x = std::abs(A(i));
00532         sum += x * x;
00533     }
00534
00535     return std::sqrt(sum);
00536 }
00537
00542 template<typename T>
00543 Matrix<T> tril(const Matrix<T>& A) {
00544     Matrix<T> B(A);
00545
00546     for (unsigned row = 0; row < B.rows(); row++)
00547         for (unsigned col = row+1; col < B.cols(); col++)
00548             B(row,col) = 0;
00549
00550     return B;
00551 }
00552
00557 template<typename T>
00558 Matrix<T> triu(const Matrix<T>& A) {
00559     Matrix<T> B(A);
00560
00561     for (unsigned col = 0; col < B.cols(); col++)
00562         for (unsigned row = col+1; row < B.rows(); row++)
00563             B(row,col) = 0;

```



```

00564
00565     return B;
00566 }
00567
00573 template<typename T>
00574 bool istril(const Matrix<T>& A) {
00575     for (unsigned row = 0; row < A.rows(); row++)
00576         for (unsigned col = row+1; col < A.cols(); col++)
00577             if (A(row,col) != static_cast<T>(0)) return false;
00578     return true;
00579 }
00580
00586 template<typename T>
00587 bool istriu(const Matrix<T>& A) {
00588     for (unsigned col = 0; col < A.cols(); col++)
00589         for (unsigned row = col+1; row < A.rows(); row++)
00590             if (A(row,col) != static_cast<T>(0)) return false;
00591     return true;
00592 }
00593
00599 template<typename T>
00600 bool ishess(const Matrix<T>& A) {
00601     if (!A.issquare())
00602         return false;
00603     for (unsigned row = 2; row < A.rows(); row++)
00604         for (unsigned col = 0; col < row-2; col++)
00605             if (A(row,col) != static_cast<T>(0)) return false;
00606     return true;
00607 }
00608
00617 template<typename T>
00618 inline void foreach_elem(Matrix<T>& A, std::function<T(T)> func) {
00619     for (unsigned i = 0; i < A.numel(); i++)
00620         A(i) = func(A(i));
00621 }
00622
00631 template<typename T>
00632 inline Matrix<T> foreach_elem_copy(const Matrix<T>& A, std::function<T(T)> func) {
00633     Matrix<T> B(A);
00634     foreach_elem(B, func);
00635     return B;
00636 }
00637
00650 template<typename T>
00651 Matrix<T> permute_rows(const Matrix<T>& A, const std::vector<unsigned> perm) {
00652     if (perm.empty()) throw std::runtime_error("Permutation vector is empty");
00653     Matrix<T> B(perm.size(), A.cols());
00654     for (unsigned p = 0; p < perm.size(); p++) {
00655         if (!perm[p] < A.rows()) throw std::out_of_range("Index in permutation vector out of range");
00656         for (unsigned c = 0; c < A.cols(); c++)
00657             B(p,c) = A(perm[p],c);
00658     }
00659     return B;
00660 }
00661
00678 template<typename T>
00679 Matrix<T> permute_cols(const Matrix<T>& A, const std::vector<unsigned> perm) {
00680     if (perm.empty()) throw std::runtime_error("Permutation vector is empty");
00681     Matrix<T> B(A.rows(), perm.size());
00682     for (unsigned p = 0; p < perm.size(); p++) {
00683         if (!perm[p] < A.cols()) throw std::out_of_range("Index in permutation vector out of range");
00684         for (unsigned r = 0; r < A.rows(); r++)
00685             B(r,p) = A(r,perm[p]);
00686     }
00687     return B;
00688 }
00689
00708 template<typename T, bool transpose_first = false, bool transpose_second = false>
00709 Matrix<T> mult(const Matrix<T>& A, const Matrix<T>& B) {
00710     // Adjust dimensions based on transpositions
00711     unsigned rows_A = transpose_first ? A.cols() : A.rows();
00712     unsigned cols_A = transpose_first ? A.rows() : A.cols();
00713     unsigned rows_B = transpose_second ? B.cols() : B.rows();
00714     unsigned cols_B = transpose_second ? B.rows() : B.cols();
00715     if (cols_A != rows_B) throw std::runtime_error("Unmatching matrix dimensions for mult");
00716     Matrix<T> C(static_cast<T>(0), rows_A, cols_B);
00717 }

```

```

00720     for (unsigned i = 0; i < rows_A; i++)
00721         for (unsigned j = 0; j < cols_B; j++)
00722             for (unsigned k = 0; k < cols_A; k++)
00723                 C(i,j) += (transpose_first ? cconj(A(k,i)) : A(i,k)) *
00724                     (transpose_second ? cconj(B(j,k)) : B(k,j));
00725
00726     return C;
00727 }
00728
00743 template<typename T, bool transpose_first = false, bool transpose_second = false>
00744 Matrix<T> mult_hadamard(const Matrix<T>& A, const Matrix<T>& B) {
00745     // Adjust dimensions based on transpositions
00746     unsigned rows_A = transpose_first ? A.cols() : A.rows();
00747     unsigned cols_A = transpose_first ? A.rows() : A.cols();
00748     unsigned rows_B = transpose_second ? B.cols() : B.rows();
00749     unsigned cols_B = transpose_second ? B.rows() : B.cols();
00750
00751     if ((rows_A != rows_B) || (cols_A != cols_B)) throw std::runtime_error("Unmatching matrix dimensions
for mult_hadamard");
00752
00753     Matrix<T> C(static_cast<T>(0), rows_A, cols_A);
00754
00755     for (unsigned i = 0; i < rows_A; i++)
00756         for (unsigned j = 0; j < cols_A; j++)
00757             C(i,j) += (transpose_first ? cconj(A(j,i)) : A(i,j)) *
00758                 (transpose_second ? cconj(B(j,i)) : B(i,j));
00759
00760     return C;
00761 }
00762
00777 template<typename T, bool transpose_first = false, bool transpose_second = false>
00778 Matrix<T> add(const Matrix<T>& A, const Matrix<T>& B) {
00779     // Adjust dimensions based on transpositions
00780     unsigned rows_A = transpose_first ? A.cols() : A.rows();
00781     unsigned cols_A = transpose_first ? A.rows() : A.cols();
00782     unsigned rows_B = transpose_second ? B.cols() : B.rows();
00783     unsigned cols_B = transpose_second ? B.rows() : B.cols();
00784
00785     if ((rows_A != rows_B) || (cols_A != cols_B)) throw std::runtime_error("Unmatching matrix dimensions
for add");
00786
00787     Matrix<T> C(static_cast<T>(0), rows_A, cols_A);
00788
00789     for (unsigned i = 0; i < rows_A; i++)
00790         for (unsigned j = 0; j < cols_A; j++)
00791             C(i,j) += (transpose_first ? cconj(A(j,i)) : A(i,j)) +
00792                 (transpose_second ? cconj(B(j,i)) : B(i,j));
00793
00794     return C;
00795 }
00796
00811 template<typename T, bool transpose_first = false, bool transpose_second = false>
00812 Matrix<T> subtract(const Matrix<T>& A, const Matrix<T>& B) {
00813     // Adjust dimensions based on transpositions
00814     unsigned rows_A = transpose_first ? A.cols() : A.rows();
00815     unsigned cols_A = transpose_first ? A.rows() : A.cols();
00816     unsigned rows_B = transpose_second ? B.cols() : B.rows();
00817     unsigned cols_B = transpose_second ? B.rows() : B.cols();
00818
00819     if ((rows_A != rows_B) || (cols_A != cols_B)) throw std::runtime_error("Unmatching matrix dimensions
for subtract");
00820
00821     Matrix<T> C(static_cast<T>(0), rows_A, cols_A);
00822
00823     for (unsigned i = 0; i < rows_A; i++)
00824         for (unsigned j = 0; j < cols_A; j++)
00825             C(i,j) += (transpose_first ? cconj(A(j,i)) : A(i,j)) -
00826                 (transpose_second ? cconj(B(j,i)) : B(i,j));
00827
00828     return C;
00829 }
00830
00844 template<typename T, bool transpose_matrix = false>
00845 std::vector<T> mult(const Matrix<T>& A, const std::vector<T>& v) {
00846     // Adjust dimensions based on transpositions
00847     unsigned rows_A = transpose_matrix ? A.cols() : A.rows();
00848     unsigned cols_A = transpose_matrix ? A.rows() : A.cols();
00849
00850     if (cols_A != v.size()) throw std::runtime_error("Unmatching matrix dimensions for mult");
00851
00852     std::vector<T> u(rows_A, static_cast<T>(0));
00853     for (unsigned r = 0; r < rows_A; r++)
00854         for (unsigned c = 0; c < cols_A; c++)
00855             u[r] += v[c] * (transpose_matrix ? cconj(A(c,r)) : A(r,c));
00856
00857     return u;
00858 }

```

```

00859
00873 template<typename T, bool transpose_matrix = false>
00874 std::vector<T> mult(const std::vector<T>& v, const Matrix<T>& A) {
00875     // Adjust dimensions based on transpositions
00876     unsigned rows_A = transpose_matrix ? A.cols() : A.rows();
00877     unsigned cols_A = transpose_matrix ? A.rows() : A.cols();
00878
00879     if (rows_A != v.size()) throw std::runtime_error("Unmatching matrix dimensions for mult");
00880
00881     std::vector<T> u(cols_A, static_cast<T>(0));
00882     for (unsigned c = 0; c < cols_A; c++)
00883         for (unsigned r = 0; r < rows_A; r++)
00884             u[c] += v[r] * (transpose_matrix ? cconj(A(c,r)) : A(r,c));
00885
00886     return u;
00887 }
00888
00894 template<typename T>
00895 Matrix<T> add(const Matrix<T>& A, T s) {
00896     Matrix<T> B(A.rows(), A.cols());
00897     for (unsigned i = 0; i < A.numel(); i++)
00898         B(i) = A(i) + s;
00899     return B;
00900 }
00901
00907 template<typename T>
00908 Matrix<T> subtract(const Matrix<T>& A, T s) {
00909     Matrix<T> B(A.rows(), A.cols());
00910     for (unsigned i = 0; i < A.numel(); i++)
00911         B(i) = A(i) - s;
00912     return B;
00913 }
00914
00920 template<typename T>
00921 Matrix<T> mult(const Matrix<T>& A, T s) {
00922     Matrix<T> B(A.rows(), A.cols());
00923     for (unsigned i = 0; i < A.numel(); i++)
00924         B(i) = A(i) * s;
00925     return B;
00926 }
00927
00933 template<typename T>
00934 Matrix<T> div(const Matrix<T>& A, T s) {
00935     Matrix<T> B(A.rows(), A.cols());
00936     for (unsigned i = 0; i < A.numel(); i++)
00937         B(i) = A(i) / s;
00938     return B;
00939 }
00940
00946 template<typename T>
00947 std::ostream& operator<<(std::ostream& os, const Matrix<T>& A) {
00948     for (unsigned row = 0; row < A.rows(); row++) {
00949         for (unsigned col = 0; col < A.cols(); col++)
00950             os << A(row,col) << " ";
00951         if (row < static_cast<unsigned>(A.rows()-1)) os << std::endl;
00952     }
00953     return os;
00954 }
00955
00960 template<typename T>
00961 inline Matrix<T> operator+(const Matrix<T>& A, const Matrix<T>& B) {
00962     return add(A,B);
00963 }
00964
00969 template<typename T>
00970 inline Matrix<T> operator-(const Matrix<T>& A, const Matrix<T>& B) {
00971     return subtract(A,B);
00972 }
00973
00979 template<typename T>
00980 inline Matrix<T> operator^(const Matrix<T>& A, const Matrix<T>& B) {
00981     return mult_hadamard(A,B);
00982 }
00983
00988 template<typename T>
00989 inline Matrix<T> operator*(const Matrix<T>& A, const Matrix<T>& B) {
00990     return mult(A,B);
00991 }
00992
00997 template<typename T>
00998 inline std::vector<T> operator*(const Matrix<T>& A, const std::vector<T>& v) {
00999     return mult(A,v);
01000 }
01001
01006 template<typename T>
01007 inline std::vector<T> operator*(const std::vector<T>& v, const Matrix<T>& A) {
01008     return mult(v,A);

```

```

01009 }
01010
01015 template<typename T>
01016 inline Matrix<T> operator+(const Matrix<T>& A, T s) {
01017     return add(A,s);
01018 }
01019
01024 template<typename T>
01025 inline Matrix<T> operator-(const Matrix<T>& A, T s) {
01026     return subtract(A,s);
01027 }
01028
01033 template<typename T>
01034 inline Matrix<T> operator*(const Matrix<T>& A, T s) {
01035     return mult(A,s);
01036 }
01037
01042 template<typename T>
01043 inline Matrix<T> operator/(const Matrix<T>& A, T s) {
01044     return div(A,s);
01045 }
01046
01050 template<typename T>
01051 inline Matrix<T> operator+(T s, const Matrix<T>& A) {
01052     return add(A,s);
01053 }
01054
01059 template<typename T>
01060 inline Matrix<T> operator*(T s, const Matrix<T>& A) {
01061     return mult(A,s);
01062 }
01063
01068 template<typename T>
01069 inline Matrix<T>& operator+=(Matrix<T>& A, const Matrix<T>& B) {
01070     return A.add(B);
01071 }
01072
01077 template<typename T>
01078 inline Matrix<T>& operator-=(Matrix<T>& A, const Matrix<T>& B) {
01079     return A.subtract(B);
01080 }
01081
01086 template<typename T>
01087 inline Matrix<T>& operator*=(Matrix<T>& A, const Matrix<T>& B) {
01088     A = mult(A,B);
01089     return A;
01090 }
01091
01097 template<typename T>
01098 inline Matrix<T>& operator^=(Matrix<T>& A, const Matrix<T>& B) {
01099     return A.mult_hadamard(B);
01100 }
01101
01106 template<typename T>
01107 inline Matrix<T>& operator+=(Matrix<T>& A, T s) {
01108     return A.add(s);
01109 }
01110
01115 template<typename T>
01116 inline Matrix<T>& operator-=(Matrix<T>& A, T s) {
01117     return A.subtract(s);
01118 }
01119
01124 template<typename T>
01125 inline Matrix<T>& operator*=(Matrix<T>& A, T s) {
01126     return A.mult(s);
01127 }
01128
01133 template<typename T>
01134 inline Matrix<T>& operator/=(Matrix<T>& A, T s) {
01135     return A.div(s);
01136 }
01137
01142 template<typename T>
01143 inline bool operator==(const Matrix<T>& A, const Matrix<T>& b) {
01144     return A.isequal(b);
01145 }
01146
01151 template<typename T>
01152 inline bool operator!=(const Matrix<T>& A, const Matrix<T>& b) {
01153     return !(A.isequal(b));
01154 }
01155
01161 template<typename T>
01162 Matrix<T> kron(const Matrix<T>& A, const Matrix<T>& B) {
01163     const unsigned rows_A = A.rows();
01164     const unsigned cols_A = A.cols();

```

```

01165     const unsigned rows_B = B.rows();
01166     const unsigned cols_B = B.cols();
01167
01168     unsigned rows_C = rows_A * rows_B;
01169     unsigned cols_C = cols_A * cols_B;
01170
01171     Matrix<T> C(rows_C, cols_C);
01172
01173     for (unsigned i = 0; i < rows_A; i++)
01174         for (unsigned j = 0; j < cols_A; j++)
01175             for (unsigned k = 0; k < rows_B; k++)
01176                 for (unsigned l = 0; l < cols_B; l++)
01177                     C(i*rows_B + k, j*cols_B + l) = A(i, j) * B(k, l);
01178
01179     return C;
01180 }
01181
01182 template<typename T>
01183 Matrix<T> adj(const Matrix<T>& A) {
01184     if (! A.issquare()) throw std::runtime_error("Input matrix is not square");
01185
01186     Matrix<T> B(A.rows(), A.cols());
01187     if (A.rows() == 1) {
01188         B(0) = 1.0;
01189     } else {
01190         for (unsigned i = 0; i < A.rows(); i++) {
01191             for (unsigned j = 0; j < A.cols(); j++) {
01192                 T sgn = ((i + j) % 2 == 0) ? 1.0 : -1.0;
01193                 B(j, i) = sgn * det(cofactor(A, i, j));
01194             }
01195         }
01196     }
01197     return B;
01198 }
01199
01200 template<typename T>
01201 Matrix<T> cofactor(const Matrix<T>& A, unsigned p, unsigned q) {
01202     if (! A.issquare()) throw std::runtime_error("Input matrix is not square");
01203     if (!(p < A.rows())) throw std::out_of_range("Row index out of range");
01204     if (!(q < A.cols())) throw std::out_of_range("Column index out of range");
01205     if (A.cols() < 2) throw std::runtime_error("Cofactor calculation requested for matrix with less than 2 rows");
01206
01207     Matrix<T> c(A.rows()-1, A.cols()-1);
01208     unsigned i = 0;
01209     unsigned j = 0;
01210
01211     for (unsigned row = 0; row < A.rows(); row++) {
01212         if (row != p) {
01213             for (unsigned col = 0; col < A.cols(); col++)
01214                 if (col != q) c(i, j++) = A(row, col);
01215             j = 0;
01216             i++;
01217         }
01218     }
01219     return c;
01220 }
01221
01222 template<typename T>
01223 T det_lu(const Matrix<T>& A) {
01224     if (! A.issquare()) throw std::runtime_error("Input matrix is not square");
01225
01226     // LU decomposition with pivoting
01227     auto res = lup(A);
01228
01229     // Determinants of LU
01230     T detLU = static_cast<T>(1);
01231
01232     for (unsigned i = 0; i < res.L.rows(); i++)
01233         detLU *= res.L(i, i) * res.U(i, i);
01234
01235     // Determinant of P
01236     unsigned len = res.P.size();
01237     T detP = 1;
01238
01239     std::vector<unsigned> p(res.P);
01240     std::vector<unsigned> q;
01241     q.resize(len);
01242
01243     for (unsigned i = 0; i < len; i++)
01244         q[p[i]] = i;
01245
01246     for (unsigned i = 0; i < len; i++) {
01247         unsigned j = p[i];
01248         unsigned k = q[i];
01249         if (j != i) {

```

```

01281         p[k] = p[i];
01282         q[j] = q[i];
01283         detP = - detP;
01284     }
01285 }
01286
01287 return detLU * detP;
01288 }
01289
01290 template<typename T>
01291 T det(const Matrix<T>& A) {
01300     if (! A.issquare()) throw std::runtime_error("Input matrix is not square");
01301
01302     if (A.rows() == 1)
01303         return A(0,0);
01304     else if (A.rows() == 2)
01305         return A(0,0)*A(1,1) - A(0,1)*A(1,0);
01306     else if (A.rows() == 3)
01307         return A(0,0)*(A(1,1)*A(2,2) - A(1,2)*A(2,1)) -
01308             A(0,1)*(A(1,0)*A(2,2) - A(1,2)*A(2,0)) +
01309             A(0,2)*(A(1,0)*A(2,1) - A(1,1)*A(2,0));
01310     else
01311         return det_lu(A);
01312 }
01313
01322 template<typename T>
01323 LU_result<T> lu(const Matrix<T>& A) {
01324     const unsigned M = A.rows();
01325     const unsigned N = A.cols();
01326
01327     LU_result<T> res;
01328     res.L = eye<T>(M);
01329     res.U = Matrix<T>(A);
01330
01331     // aliases
01332     auto& L = res.L;
01333     auto& U = res.U;
01334
01335     if (A.numel() == 0)
01336         return res;
01337
01338     for (unsigned k = 0; k < M-1; k++) {
01339         for (unsigned i = k+1; i < M; i++) {
01340             L(i,k) = U(i,k) / U(k,k);
01341             for (unsigned l = k+1; l < N; l++) {
01342                 U(i,l) -= L(i,k) * U(k,l);
01343             }
01344         }
01345     }
01346
01347     for (unsigned col = 0; col < N; col++)
01348         for (unsigned row = col+1; row < M; row++)
01349             U(row,col) = 0;
01350
01351     return res;
01352 }
01353
01367 template<typename T>
01368 LUP_result<T> lup(const Matrix<T>& A) {
01369     const unsigned M = A.rows();
01370     const unsigned N = A.cols();
01371
01372     // Initialize L, U, and PP
01373     LUP_result<T> res;
01374
01375     if (A.numel() == 0)
01376         return res;
01377
01378     res.L = eye<T>(M);
01379     res.U = Matrix<T>(A);
01380     std::vector<unsigned> PP;
01381
01382     // aliases
01383     auto& L = res.L;
01384     auto& U = res.U;
01385
01386     PP.resize(N);
01387     for (unsigned i = 0; i < N; i++)
01388         PP[i] = i;
01389
01390     for (unsigned k = 0; k < M-1; k++) {
01391         // Find the column with the largest absolute value in the current row
01392         auto max_col_value = std::abs(U(k,k));
01393         unsigned max_col_index = k;
01394         for (unsigned l = k+1; l < N; l++) {
01395             auto val = std::abs(U(k,l));
01396             if (val > max_col_value) {

```

```

01397         max_col_value = val;
01398         max_col_index = l;
01399     }
01400 }
01401
01402 // Swap columns k and max_col_index in U and update P
01403 if (max_col_index != k) {
01404     U.swap_cols(k, max_col_index); // TODO: This could be reworked to avoid column swap in U during
every iteration by:
01405                                     // 1. using PP[k] for column indexing across iterations
01406                                     // 2. doing just one permutation of U at the end
01407     std::swap(PP[k], PP[max_col_index]);
01408 }
01409
01410 // Update L and U
01411 for (unsigned i = k+1; i < M; i++) {
01412     L(i,k) = U(i,k) / U(k,k);
01413     for (unsigned l = k+1; l < N; l++) {
01414         U(i,l) -= L(i,k) * U(k,l);
01415     }
01416 }
01417 }
01418
01419 // Set elements in lower triangular part of U to zero
01420 for (unsigned col = 0; col < N; col++)
01421     for (unsigned row = col+1; row < M; row++)
01422         U(row,col) = 0;
01423
01424 // Transpose indices in permutation vector
01425 res.P.resize(N);
01426 for (unsigned i = 0; i < N; i++)
01427     res.P[PP[i]] = i;
01428
01429 return res;
01430 }
01431
01432 template<typename T>
01433 Matrix<T> inv_gauss_jordan(const Matrix<T>& A) {
01434     if (! A.issquare()) throw std::runtime_error("Input matrix is not square");
01435
01436     const unsigned N = A.rows();
01437     Matrix<T> AA(A);
01438     auto IA = eye<T>(N);
01439
01440     bool found_nonzero;
01441     for (unsigned j = 0; j < N; j++) {
01442         found_nonzero = false;
01443         for (unsigned i = j; i < N; i++) {
01444             if (AA(i,j) != static_cast<T>(0)) {
01445                 found_nonzero = true;
01446                 for (unsigned k = 0; k < N; k++) {
01447                     std::swap(AA(j,k), AA(i,k));
01448                     std::swap(IA(j,k), IA(i,k));
01449                 }
01450                 if (AA(j,j) != static_cast<T>(1)) {
01451                     T s = static_cast<T>(1) / AA(j,j);
01452                     for (unsigned k = 0; k < N; k++) {
01453                         AA(j,k) *= s;
01454                         IA(j,k) *= s;
01455                     }
01456                 }
01457                 for (unsigned l = 0; l < N; l++) {
01458                     if (l != j) {
01459                         T s = AA(l,j);
01460                         for (unsigned k = 0; k < N; k++) {
01461                             AA(l,k) -= s * AA(j,k);
01462                             IA(l,k) -= s * IA(j,k);
01463                         }
01464                     }
01465                 }
01466             }
01467         }
01468         if (! found_nonzero) break;
01469     }
01470     // if a row full of zeros is found, the input matrix was singular
01471     if (! found_nonzero) throw singular_matrix_exception("Singular matrix in inv_gauss_jordan");
01472 }
01473 return IA;
01474 }
01475
01476 template<typename T>
01477 Matrix<T> inv_tril(const Matrix<T>& A) {
01478     if (! A.issquare()) throw std::runtime_error("Input matrix is not square");
01479
01480     const unsigned N = A.rows();
01481     auto IA = zeros<T>(N);
01482 }

```

```

01503     for (unsigned i = 0; i < N; i++) {
01504         if (A(i,i) == 0.0) throw singular_matrix_exception("Division by zero in inv_tril");
01505
01506         IA(i,i) = static_cast<T>(1.0) / A(i,i);
01507         for (unsigned j = 0; j < i; j++) {
01508             T s = 0.0;
01509             for (unsigned k = j; k < i; k++)
01510                 s += A(i,k) * IA(k,j);
01511             IA(i,j) = -s * IA(i,i) ;
01512         }
01513     }
01514
01515     return IA;
01516 }
01517
01528 template<typename T>
01529 Matrix<T> inv_triu(const Matrix<T>& A) {
01530     if (!A.issquare()) throw std::runtime_error("Input matrix is not square");
01531
01532     const unsigned N = A.rows();
01533
01534     auto IA = zeros<T>(N);
01535
01536     for (int i = N - 1; i >= 0; i--) {
01537         if (A(i,i) == 0.0) throw singular_matrix_exception("Division by zero in inv_triu");
01538
01539         IA(i, i) = static_cast<T>(1.0) / A(i,i);
01540         for (int j = N - 1; j > i; j--) {
01541             T s = 0.0;
01542             for (int k = i + 1; k <= j; k++)
01543                 s += A(i,k) * IA(k,j);
01544             IA(i,j) = -s * IA(i,i);
01545         }
01546     }
01547
01548     return IA;
01549 }
01550
01563 template<typename T>
01564 Matrix<T> inv_posdef(const Matrix<T>& A) {
01565     auto L = cholinv(A);
01566     return mult<T,true,false>(L,L);
01567 }
01568
01579 template<typename T>
01580 Matrix<T> inv_square(const Matrix<T>& A) {
01581     if (! A.issquare()) throw std::runtime_error("Input matrix is not square");
01582
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
01588     return permute_rows(IU * IL, LU.P);
01589 }
01590
01601 template<typename T>
01602 Matrix<T> inv(const Matrix<T>& A) {
01603     if (! A.issquare()) throw std::runtime_error("Input matrix is not square");
01604
01605     if (A.numel() == 0) {
01606         return Matrix<T>();
01607     } else if (A.rows() < 4) {
01608         T d = det(A);
01609
01610         if (d == 0.0) throw singular_matrix_exception("Singular matrix in inv");
01611
01612         Matrix<T> IA(A.rows(), A.rows());
01613         T invdet = static_cast<T>(1.0) / d;
01614
01615         if (A.rows() == 1) {
01616             IA(0,0) = invdet;
01617         } else if (A.rows() == 2) {
01618             IA(0,0) = A(1,1) * invdet;
01619             IA(0,1) = - A(0,1) * invdet;
01620             IA(1,0) = - A(1,0) * invdet;
01621             IA(1,1) = A(0,0) * invdet;
01622         } else if (A.rows() == 3) {
01623             IA(0,0) = (A(1,1)*A(2,2) - A(2,1)*A(1,2)) * invdet;
01624             IA(0,1) = (A(0,2)*A(2,1) - A(0,1)*A(2,2)) * invdet;
01625             IA(0,2) = (A(0,1)*A(1,2) - A(0,2)*A(1,1)) * invdet;
01626             IA(1,0) = (A(1,2)*A(2,0) - A(1,0)*A(2,2)) * invdet;
01627             IA(1,1) = (A(0,0)*A(2,2) - A(0,2)*A(2,0)) * invdet;
01628             IA(1,2) = (A(1,0)*A(0,2) - A(0,0)*A(1,2)) * invdet;
01629             IA(2,0) = (A(1,0)*A(2,1) - A(2,0)*A(1,1)) * invdet;
01630             IA(2,1) = (A(2,0)*A(0,1) - A(0,0)*A(2,1)) * invdet;
01631             IA(2,2) = (A(0,0)*A(1,1) - A(1,0)*A(0,1)) * invdet;

```



```

01632     }
01633
01634     return IA;
01635 } else {
01636     return inv_square(A);
01637 }
01638 }
01639
01640 template<typename T>
01641 Matrix<T> pinv(const Matrix<T>& A) {
01642     if (A.rows() > A.cols()) {
01643         auto AH_A = mult<T,true,false>(A, A);
01644         auto Linv = inv_posdef(AH_A);
01645         return mult<T,false,true>(Linv, A);
01646     } else {
01647         auto AA_H = mult<T,false,true>(A, A);
01648         auto Linv = inv_posdef(AA_H);
01649         return mult<T,true,false>(A, Linv);
01650     }
01651 }
01652
01653 template<typename T>
01654 T trace(const Matrix<T>& A) {
01655     T t = static_cast<T>(0);
01656     for (int i = 0; i < A.rows(); i++)
01657         t += A(i,i);
01658     return t;
01659 }
01660
01661 template<typename T>
01662 double cond(const Matrix<T>& A) {
01663     try {
01664         auto A_inv = inv(A);
01665         return norm_fro(A) * norm_fro(A_inv);
01666     } catch (singular_matrix_exception& e) {
01667         return std::numeric_limits<double>::max();
01668     }
01669 }
01670
01671 template<typename T, bool is_upper = false>
01672 Matrix<T> chol(const Matrix<T>& A) {
01673     if (! A.issquare()) throw std::runtime_error("Input matrix is not square");
01674
01675     const unsigned N = A.rows();
01676
01677     // Calculate lower or upper triangular, depending on template parameter.
01678     // Calculation is the same - the difference is in transposed row and column indexing.
01679     Matrix<T> C = is_upper ? triu(A) : tril(A);
01680
01681     for (unsigned j = 0; j < N; j++) {
01682         if (C(j,j) == 0.0) throw singular_matrix_exception("Singular matrix in chol");
01683         C(j,j) = std::sqrt(C(j,j));
01684
01685         for (unsigned k = j+1; k < N; k++)
01686             if (is_upper)
01687                 C(j,k) /= C(j,j);
01688             else
01689                 C(k,j) /= C(j,j);
01690
01691         for (unsigned k = j+1; k < N; k++)
01692             for (unsigned i = k; i < N; i++)
01693                 if (is_upper)
01694                     C(k,i) -= C(j,i) * cconj(C(j,k));
01695                 else
01696                     C(i,k) -= C(i,j) * cconj(C(k,j));
01697     }
01698
01699     return C;
01700 }
01701
01702 template<typename T>
01703 Matrix<T> cholinv(const Matrix<T>& A) {
01704     if (! A.issquare()) throw std::runtime_error("Input matrix is not square");
01705
01706     const unsigned N = A.rows();
01707     Matrix<T> L(A);
01708     auto Linv = eye<T>(N);
01709
01710     for (unsigned j = 0; j < N; j++) {
01711         if (L(j,j) == 0.0) throw singular_matrix_exception("Singular matrix in cholinv");
01712         L(j,j) = 1.0 / std::sqrt(L(j,j));
01713
01714         for (unsigned k = j+1; k < N; k++)
01715             L(k,j) = L(k,j) * L(j,j);
01716     }
01717 }

```

```

01765     for (unsigned k = j+1; k < N; k++)
01766     for (unsigned i = k; i < N; i++)
01767         L(i,k) = L(i,k) - L(i,j) * cconj(L(k,j));
01768 }
01769
01770 for (unsigned k = 0; k < N; k++) {
01771     for (unsigned i = k; i < N; i++) {
01772         Linv(i,k) = Linv(i,k) * L(i,i);
01773         for (unsigned j = i+1; j < N; j++)
01774             Linv(j,k) = Linv(j,k) - L(j,i) * Linv(i,k);
01775     }
01776 }
01777
01778 return Linv;
01779 }
01780
01795 template<typename T>
01796 LDL_result<T> ldl(const Matrix<T>& A) {
01797     if (! A.issquare()) throw std::runtime_error("Input matrix is not square");
01798
01799     const unsigned N = A.rows();
01800
01801     LDL_result<T> res;
01802
01803     // aliases
01804     auto& L = res.L;
01805     auto& d = res.d;
01806
01807     L = eye<T>(N);
01808     d.resize(N);
01809
01810     for (unsigned m = 0; m < N; m++) {
01811         d[m] = A(m,m);
01812
01813         for (unsigned k = 0; k < m; k++)
01814             d[m] -= L(m,k) * cconj(L(m,k)) * d[k];
01815
01816         if (d[m] == 0.0) throw singular_matrix_exception("Singular matrix in ldl");
01817
01818         for (unsigned n = m+1; n < N; n++) {
01819             L(n,m) = A(n,m);
01820             for (unsigned k = 0; k < m; k++)
01821                 L(n,m) -= L(n,k) * cconj(L(m,k)) * d[k];
01822             L(n,m) /= d[m];
01823         }
01824     }
01825
01826     return res;
01827 }
01828
01840 template<typename T>
01841 QR_result<T> qr_red_gs(const Matrix<T>& A) {
01842     const int rows = A.rows();
01843     const int cols = A.cols();
01844
01845     QR_result<T> res;
01846
01847     //aliases
01848     auto& Q = res.Q;
01849     auto& R = res.R;
01850
01851     Q = zeros<T>(rows, cols);
01852     R = zeros<T>(cols, cols);
01853
01854     for (int c = 0; c < cols; c++) {
01855         Matrix<T> v = A.get_submatrix(0, rows-1, c, c);
01856         for (int r = 0; r < c; r++) {
01857             for (int k = 0; k < rows; k++)
01858                 R(r,c) = R(r,c) + cconj(Q(k,r)) * A(k,c);
01859             for (int k = 0; k < rows; k++)
01860                 v(k) = v(k) - R(r,c) * Q(k,r);
01861         }
01862
01863         R(c,c) = static_cast<T>(norm_fro(v));
01864
01865         if (R(c,c) == 0.0) throw singular_matrix_exception("Division by 0 in QR GS");
01866
01867         for (int k = 0; k < rows; k++)
01868             Q(k,c) = v(k) / R(c,c);
01869     }
01870
01871     return res;
01872 }
01873
01881 template<typename T>
01882 Matrix<T> householder_reflection(const Matrix<T>& a) {
01883     if (a.cols() != 1) throw std::runtime_error("Input not a column vector");

```

```

01884
01885     static const T ISQRT2 = static_cast<T>(0.707106781186547);
01886
01887     Matrix<T> v(a);
01888     v(0) += csign(v(0)) * norm_fro(v);
01889     auto vn = norm_fro(v) * ISQRT2;
01890     for (unsigned i = 0; i < v.numel(); i++)
01891         v(i) /= vn;
01892     return v;
01893 }
01894
01906 template<typename T>
01907 QR_result<T> qr_householder(const Matrix<T>& A, bool calculate_Q = true) {
01908     const unsigned rows = A.rows();
01909     const unsigned cols = A.cols();
01910
01911     QR_result<T> res;
01912
01913     //aliases
01914     auto& Q = res.Q;
01915     auto& R = res.R;
01916
01917     R = Matrix<T>(A);
01918
01919     if (calculate_Q)
01920         Q = eye<T>(rows);
01921
01922     const unsigned N = (rows > cols) ? cols : rows;
01923
01924     for (unsigned j = 0; j < N; j++) {
01925         auto v = householder_reflection(R.get_submatrix(j, rows-1, j, j));
01926
01927         auto R1 = R.get_submatrix(j, rows-1, j, cols-1);
01928         auto WR = v * mult<T,true,false>(v, R1);
01929         for (unsigned c = j; c < cols; c++)
01930             for (unsigned r = j; r < rows; r++)
01931                 R(r,c) -= WR(r-j,c-j);
01932
01933         if (calculate_Q) {
01934             auto Q1 = Q.get_submatrix(0, rows-1, j, rows-1);
01935             auto WQ = mult<T,false,true>(Q1 * v, v);
01936             for (unsigned c = j; c < rows; c++)
01937                 for (unsigned r = 0; r < rows; r++)
01938                     Q(r,c) -= WQ(r,c-j);
01939         }
01940     }
01941
01942     for (unsigned col = 0; col < R.cols(); col++)
01943         for (unsigned row = col+1; row < R.rows(); row++)
01944             R(row,col) = 0;
01945
01946     return res;
01947 }
01948
01959 template<typename T>
01960 inline QR_result<T> qr(const Matrix<T>& A, bool calculate_Q = true) {
01961     return qr_householder(A, calculate_Q);
01962 }
01963
01974 template<typename T>
01975 Hessenberg_result<T> hessenberg(const Matrix<T>& A, bool calculate_Q = true) {
01976     if (! A.issquare()) throw std::runtime_error("Input matrix is not square");
01977
01978     Hessenberg_result<T> res;
01979
01980     // aliases
01981     auto& H = res.H;
01982     auto& Q = res.Q;
01983
01984     const unsigned N = A.rows();
01985     H = Matrix<T>(A);
01986
01987     if (calculate_Q)
01988         Q = eye<T>(N);
01989
01990     for (unsigned k = 1; k < N-1; k++) {
01991         auto v = householder_reflection(H.get_submatrix(k, N-1, k-1, k-1));
01992
01993         auto H1 = H.get_submatrix(k, N-1, 0, N-1);
01994         auto W1 = v * mult<T,true,false>(v, H1);
01995         for (unsigned c = 0; c < N; c++)
01996             for (unsigned r = k; r < N; r++)
01997                 H(r,c) -= W1(r-k,c);
01998
01999         auto H2 = H.get_submatrix(0, N-1, k, N-1);
02000         auto W2 = mult<T,false,true>(H2 * v, v);
02001         for (unsigned c = k; c < N; c++)

```

```

02002         for (unsigned r = 0; r < N; r++)
02003             H(r,c) -= W2(r,c-k);
02004
02005         if (calculate_Q) {
02006             auto Q1 = Q.get_submatrix(0, N-1, k, N-1);
02007             auto W3 = mult<T,false,true>(Q1 * v, v);
02008             for (unsigned c = k; c < N; c++)
02009                 for (unsigned r = 0; r < N; r++)
02010                     Q(r,c) -= W3(r,c-k);
02011         }
02012     }
02013
02014     for (unsigned row = 2; row < N; row++)
02015         for (unsigned col = 0; col < row-2; col++)
02016             H(row,col) = static_cast<T>(0);
02017
02018     return res;
02019 }
02020
02021 template<typename T>
02022 std::complex<T> wilkinson_shift(const Matrix<std::complex<T>& H, T tol = 1e-10) {
02023     if (! H.issquare()) throw std::runtime_error("Input matrix is not square");
02024
02025     const unsigned n = H.rows();
02026     std::complex<T> mu;
02027
02028     if (std::abs(H(n-1,n-2)) < tol) {
02029         mu = H(n-2,n-2);
02030     } else {
02031         auto trA = H(n-2,n-2) + H(n-1,n-1);
02032         auto detA = H(n-2,n-2) * H(n-1,n-1) - H(n-2, n-1) * H(n-1, n-2);
02033         mu = (trA + std::sqrt(trA*trA - 4.0*detA)) / 2.0;
02034     }
02035
02036     return mu;
02037 }
02038
02039 template<typename T>
02040 Eigenvalues_result<T> eigenvalues(const Matrix<std::complex<T>& A, T tol = 1e-12, unsigned max_iter =
100) {
02041     if (! A.issquare()) throw std::runtime_error("Input matrix is not square");
02042
02043     const unsigned N = A.rows();
02044     Matrix<std::complex<T>> H;
02045     bool success = false;
02046
02047     QR_result<std::complex<T>> QR;
02048
02049     // aliases
02050     auto& Q = QR.Q;
02051     auto& R = QR.R;
02052
02053     // Transfer A to Hessenberg form to improve convergence (skip calculation of Q)
02054     H = hessenberg(A, false).H;
02055
02056     for (unsigned iter = 0; iter < max_iter; iter++) {
02057         auto mu = wilkinson_shift(H, tol);
02058
02059         // subtract mu from diagonal
02060         for (unsigned n = 0; n < N; n++)
02061             H(n,n) -= mu;
02062
02063         // QR factorization with shifted H
02064         QR = qr(H);
02065         H = R * Q;
02066
02067         // add back mu to diagonal
02068         for (unsigned n = 0; n < N; n++)
02069             H(n,n) += mu;
02070
02071         // Check for convergence
02072         if (std::abs(H(N-2,N-1)) <= tol) {
02073             success = true;
02074             break;
02075         }
02076     }
02077
02078     Eigenvalues_result<T> res;
02079     res.eig = diag(H);
02080     res.err = std::abs(H(N-2,N-1));
02081     res.converged = success;
02082
02083     return res;
02084 }
02085
02086 template<typename T>
02087 Eigenvalues_result<T> eigenvalues(const Matrix<T>& A, T tol = 1e-12, unsigned max_iter = 100) {

```

```

02116     auto A_cplx = make_complex(A);
02117     return eigenvalues(A_cplx, tol, max_iter);
02118 }
02119
02120 template<typename T>
02121 Matrix<T> solve_triu(const Matrix<T>& U, const Matrix<T>& B) {
02122     if (! U.issquare()) throw std::runtime_error("Input matrix is not square");
02123     if (U.rows() != B.rows()) throw std::runtime_error("Unmatching matrix dimensions for solve");
02124
02125     const unsigned N = U.rows();
02126     const unsigned M = B.cols();
02127
02128     if (U.numel() == 0)
02129         return Matrix<T>();
02130
02131     Matrix<T> X(B);
02132
02133     for (unsigned m = 0; m < M; m++) {
02134         // backwards substitution for each column of B
02135         for (int n = N-1; n >= 0; n--) {
02136             for (unsigned j = n + 1; j < N; j++)
02137                 X(n,m) -= U(n,j) * X(j,m);
02138
02139             if (U(n,n) == 0.0) throw singular_matrix_exception("Singular matrix in solve_triu");
02140
02141             X(n,m) /= U(n,n);
02142         }
02143     }
02144
02145     return X;
02146 }
02147
02148 template<typename T>
02149 Matrix<T> solve_tril(const Matrix<T>& L, const Matrix<T>& B) {
02150     if (! L.issquare()) throw std::runtime_error("Input matrix is not square");
02151     if (L.rows() != B.rows()) throw std::runtime_error("Unmatching matrix dimensions for solve");
02152
02153     const unsigned N = L.rows();
02154     const unsigned M = B.cols();
02155
02156     if (L.numel() == 0)
02157         return Matrix<T>();
02158
02159     Matrix<T> X(B);
02160
02161     for (unsigned m = 0; m < M; m++) {
02162         // forwards substitution for each column of B
02163         for (unsigned n = 0; n < N; n++) {
02164             for (unsigned j = 0; j < n; j++)
02165                 X(n,m) -= L(n,j) * X(j,m);
02166
02167             if (L(n,n) == 0.0) throw singular_matrix_exception("Singular matrix in solve_tril");
02168
02169             X(n,m) /= L(n,n);
02170         }
02171     }
02172
02173     return X;
02174 }
02175
02176 template<typename T>
02177 Matrix<T> solve_square(const Matrix<T>& A, const Matrix<T>& B) {
02178     if (! A.issquare()) throw std::runtime_error("Input matrix is not square");
02179     if (A.rows() != B.rows()) throw std::runtime_error("Unmatching matrix dimensions for solve");
02180
02181     if (A.numel() == 0)
02182         return Matrix<T>();
02183
02184     Matrix<T> L;
02185     Matrix<T> U;
02186     std::vector<unsigned> P;
02187
02188     // LU decomposition with pivoting
02189     auto lup_res = lup(A);
02190
02191     auto y = solve_tril(lup_res.L, B);
02192     auto x = solve_triu(lup_res.U, y);
02193
02194     return permute_rows(x, lup_res.P);
02195 }
02196
02197 template<typename T>
02198 Matrix<T> solve_posdef(const Matrix<T>& A, const Matrix<T>& B) {
02199     if (! A.issquare()) throw std::runtime_error("Input matrix is not square");
02200     if (A.rows() != B.rows()) throw std::runtime_error("Unmatching matrix dimensions for solve");
02201
02202     if (A.numel() == 0)

```

```

02259     return Matrix<T>();
02260
02261     // LU decomposition with pivoting
02262     auto L = chol(A);
02263
02264     auto Y = solve_tril(L, B);
02265     return solve_triu(L.ctranspose(), Y);
02266 }
02267
02272 template<typename T>
02273 class Matrix {
02274 public:
02275     Matrix();
02280
02285     Matrix(unsigned size);
02286
02291     Matrix(unsigned nrow, unsigned ncol);
02292
02297     Matrix(T x, unsigned nrow, unsigned ncol);
02298
02304     Matrix(const T* array, unsigned nrow, unsigned ncol);
02305
02313     Matrix(const std::vector<T>& vec, unsigned nrow, unsigned ncol);
02314
02322     Matrix(std::initializer_list<T> init_list, unsigned nrow, unsigned ncol);
02323
02326     Matrix(const Matrix &);
02327
02330     virtual ~Matrix();
02331
02339     Matrix<T> get_submatrix(unsigned row_first, unsigned row_last, unsigned col_first, unsigned
col_last) const;
02340
02349     void set_submatrix(const Matrix<T>& smtx, unsigned row_first, unsigned col_first);
02350
02355     void clear();
02356
02364     void reshape(unsigned rows, unsigned cols);
02365
02371     void resize(unsigned rows, unsigned cols);
02372
02378     bool exists(unsigned row, unsigned col) const;
02379
02384     T* ptr(unsigned row, unsigned col);
02385
02392     T* ptr();
02393
02397     void fill(T value);
02398
02405     void fill_col(T value, unsigned col);
02406
02413     void fill_row(T value, unsigned row);
02414
02419     bool isempty() const;
02420
02424     bool issquare() const;
02425
02430     bool isequal(const Matrix<T>&) const;
02431
02437     bool isequal(const Matrix<T>&, T) const;
02438
02443     unsigned numel() const;
02444
02449     unsigned rows() const;
02450
02455     unsigned cols() const;
02456
02461     Matrix<T> transpose() const;
02462
02468     Matrix<T> ctranspose() const;
02469
02477     Matrix<T>& add(const Matrix<T>&);
02478
02486     Matrix<T>& subtract(const Matrix<T>&);
02487
02496     Matrix<T>& mult_hadamard(const Matrix<T>&);
02497
02503     Matrix<T>& add(T);
02504
02510     Matrix<T>& subtract(T);
02511
02517     Matrix<T>& mult(T);
02518
02524     Matrix<T>& div(T);
02525
02530     Matrix<T>& operator=(const Matrix<T>&);
02531

```

```

02536     Matrix<T>& operator=(T);
02537
02542     explicit operator std::vector<T>() const;
02543     std::vector<T> to_vector() const;
02544
02551     T& operator()(unsigned nel);
02552     T operator()(unsigned nel) const;
02553     T& at(unsigned nel);
02554     T at(unsigned nel) const;
02555
02562     T& operator()(unsigned row, unsigned col);
02563     T operator()(unsigned row, unsigned col) const;
02564     T& at(unsigned row, unsigned col);
02565     T at(unsigned row, unsigned col) const;
02566
02574     void add_row_to_another(unsigned to, unsigned from);
02575
02583     void add_col_to_another(unsigned to, unsigned from);
02584
02592     void mult_row_by_another(unsigned to, unsigned from);
02593
02601     void mult_col_by_another(unsigned to, unsigned from);
02602
02609     void swap_rows(unsigned i, unsigned j);
02610
02617     void swap_cols(unsigned i, unsigned j);
02618
02625     std::vector<T> col_to_vector(unsigned col) const;
02626
02633     std::vector<T> row_to_vector(unsigned row) const;
02634
02642     void col_from_vector(const std::vector<T>&, unsigned col);
02643
02651     void row_from_vector(const std::vector<T>&, unsigned row);
02652
02653 private:
02654     unsigned nrows;
02655     unsigned ncols;
02656     std::vector<T> data;
02657 };
02658
02659 /*
02660  * Implementation of Matrix class methods
02661  */
02662
02663 template<typename T>
02664 Matrix<T>::Matrix() : nrows(0), ncols(0), data() { }
02665
02666 template<typename T>
02667 Matrix<T>::Matrix(unsigned size) : Matrix(size, size) { }
02668
02669 template<typename T>
02670 Matrix<T>::Matrix(unsigned rows, unsigned cols) : nrows(rows), ncols(cols) {
02671     data.resize(numel());
02672 }
02673
02674 template<typename T>
02675 Matrix<T>::Matrix(T x, unsigned rows, unsigned cols) : Matrix(rows, cols) {
02676     fill(x);
02677 }
02678
02679 template<typename T>
02680 Matrix<T>::Matrix(const T* array, unsigned rows, unsigned cols) : Matrix(rows, cols) {
02681     data.assign(array, array + numel());
02682 }
02683
02684 template<typename T>
02685 Matrix<T>::Matrix(const std::vector<T>& vec, unsigned rows, unsigned cols) : Matrix(rows, cols) {
02686     if (vec.size() != numel()) throw std::runtime_error("Size of initialization vector not consistent
with matrix dimensions");
02687
02688     data.assign(vec.begin(), vec.end());
02689 }
02690
02691 template<typename T>
02692 Matrix<T>::Matrix(std::initializer_list<T> init_list, unsigned rows, unsigned cols) : Matrix(rows,
cols) {
02693     if (init_list.size() != numel()) throw std::runtime_error("Size of initialization list not
consistent with matrix dimensions");
02694
02695     auto it = init_list.begin();
02696
02697     for (unsigned row = 0; row < this->nrows; row++)
02698         for (unsigned col = 0; col < this->ncols; col++)
02699             this->at(row, col) = *(it++);
02700 }
02701

```

```

02702 template<typename T>
02703 Matrix<T>::Matrix(const Matrix & other) : Matrix(other.nrows, other.ncols) {
02704     this->data.assign(other.data.begin(), other.data.end());
02705 }
02706
02707 template<typename T>
02708 Matrix<T>& Matrix<T>::operator=(const Matrix<T>& other) {
02709     this->nrows = other.nrows;
02710     this->ncols = other.ncols;
02711     this->data.assign(other.data.begin(), other.data.end());
02712     return *this;
02713 }
02714
02715 template<typename T>
02716 Matrix<T>& Matrix<T>::operator=(T s) {
02717     fill(s);
02718     return *this;
02719 }
02720
02721 template<typename T>
02722 inline Matrix<T>::operator std::vector<T>() const {
02723     return data;
02724 }
02725
02726 template<typename T>
02727 inline void Matrix<T>::clear() {
02728     this->nrows = 0;
02729     this->ncols = 0;
02730     data.resize(0);
02731 }
02732
02733 template<typename T>
02734 void Matrix<T>::reshape(unsigned rows, unsigned cols) {
02735     if (this->numel() != rows * cols) throw std::runtime_error("Illegal attempt to change number of
elements via reshape");
02736
02737     this->nrows = rows;
02738     this->ncols = cols;
02739 }
02740
02741 template<typename T>
02742 void Matrix<T>::resize(unsigned rows, unsigned cols) {
02743     this->nrows = rows;
02744     this->ncols = cols;
02745     data.resize(nrows*ncols);
02746 }
02747
02748 template<typename T>
02749 Matrix<T> Matrix<T>::get_submatrix(unsigned row_base, unsigned row_lim, unsigned col_base, unsigned
col_lim) const {
02750     if (row_base > row_lim) throw std::out_of_range("Row index of submatrix out of range");
02751     if (col_base > col_lim) throw std::out_of_range("Column index of submatrix out of range");
02752     if (row_lim >= this->rows()) throw std::out_of_range("Row index of submatrix out of range");
02753     if (col_lim >= this->cols()) throw std::out_of_range("Column index of submatrix out of range");
02754
02755     unsigned num_rows = row_lim - row_base + 1;
02756     unsigned num_cols = col_lim - col_base + 1;
02757     Matrix<T> S(num_rows, num_cols);
02758     for (unsigned i = 0; i < num_rows; i++) {
02759         for (unsigned j = 0; j < num_cols; j++) {
02760             S(i,j) = at(row_base + i, col_base + j);
02761         }
02762     }
02763     return S;
02764 }
02765
02766 template<typename T>
02767 void Matrix<T>::set_submatrix(const Matrix<T>& S, unsigned row_base, unsigned col_base) {
02768     if (this->isempty()) throw std::runtime_error("Invalid attempt to set submatrix in empty matrix");
02769
02770     const unsigned row_lim = row_base + S.rows() - 1;
02771     const unsigned col_lim = col_base + S.cols() - 1;
02772
02773     if (row_base > row_lim) throw std::out_of_range("Row index of submatrix out of range");
02774     if (col_base > col_lim) throw std::out_of_range("Column index of submatrix out of range");
02775     if (row_lim >= this->rows()) throw std::out_of_range("Row index of submatrix out of range");
02776     if (col_lim >= this->cols()) throw std::out_of_range("Column index of submatrix out of range");
02777
02778     unsigned num_rows = row_lim - row_base + 1;
02779     unsigned num_cols = col_lim - col_base + 1;
02780     for (unsigned i = 0; i < num_rows; i++)
02781         for (unsigned j = 0; j < num_cols; j++)
02782             at(row_base + i, col_base + j) = S(i,j);
02783 }
02784
02785 template<typename T>
02786 inline T & Matrix<T>::operator()(unsigned nel) {

```



```

02787     return at(nel);
02788 }
02789
02790 template<typename T>
02791 inline T & Matrix<T>::operator()(unsigned row, unsigned col) {
02792     return at(row, col);
02793 }
02794
02795 template<typename T>
02796 inline T Matrix<T>::operator()(unsigned nel) const {
02797     return at(nel);
02798 }
02799
02800 template<typename T>
02801 inline T Matrix<T>::operator()(unsigned row, unsigned col) const {
02802     return at(row, col);
02803 }
02804
02805 template<typename T>
02806 inline T & Matrix<T>::at(unsigned nel) {
02807     if (!(nel < numel())) throw std::out_of_range("Element index out of range");
02808     return data[nel];
02809 }
02810
02811
02812 template<typename T>
02813 inline T & Matrix<T>::at(unsigned row, unsigned col) {
02814     if (!(row < rows() && col < cols())) throw std::out_of_range("Element index out of range");
02815     return data[nrows * col + row];
02816 }
02817
02818
02819 template<typename T>
02820 inline T Matrix<T>::at(unsigned nel) const {
02821     if (!(nel < numel())) throw std::out_of_range("Element index out of range");
02822     return data[nel];
02823 }
02824
02825
02826 template<typename T>
02827 inline T Matrix<T>::at(unsigned row, unsigned col) const {
02828     if (!(row < rows())) throw std::out_of_range("Row index out of range");
02829     if (!(col < cols())) throw std::out_of_range("Column index out of range");
02830     return data[nrows * col + row];
02831 }
02832
02833
02834 template<typename T>
02835 inline void Matrix<T>::fill(T value) {
02836     for (unsigned i = 0; i < numel(); i++)
02837         data[i] = value;
02838 }
02839
02840 template<typename T>
02841 inline void Matrix<T>::fill_col(T value, unsigned col) {
02842     if (!(col < cols())) throw std::out_of_range("Column index out of range");
02843     for (unsigned i = col * nrows; i < (col+1) * nrows; i++)
02844         data[i] = value;
02845 }
02846
02847
02848 template<typename T>
02849 inline void Matrix<T>::fill_row(T value, unsigned row) {
02850     if (!(row < rows())) throw std::out_of_range("Row index out of range");
02851     for (unsigned i = 0; i < ncols; i++)
02852         data[row + i * nrows] = value;
02853 }
02854
02855
02856 template<typename T>
02857 inline bool Matrix<T>::exists(unsigned row, unsigned col) const {
02858     return (row < nrows && col < ncols);
02859 }
02860
02861 template<typename T>
02862 inline T* Matrix<T>::ptr(unsigned row, unsigned col) {
02863     if (!(row < rows())) throw std::out_of_range("Row index out of range");
02864     if (!(col < cols())) throw std::out_of_range("Column index out of range");
02865     return data.data() + nrows * col + row;
02866 }
02867
02868
02869 template<typename T>
02870 inline T* Matrix<T>::ptr() {
02871     return data.data();
02872 }
02873

```

```

02874 template<typename T>
02875 inline bool Matrix<T>::isempty() const {
02876     return (nrows == 0) || (ncols == 0);
02877 }
02878
02879 template<typename T>
02880 inline bool Matrix<T>::issquare() const {
02881     return (nrows == ncols) && !isempty();
02882 }
02883
02884 template<typename T>
02885 bool Matrix<T>::isequal(const Matrix<T>& A) const {
02886     bool ret = true;
02887     if (nrows != A.rows() || ncols != A.cols()) {
02888         ret = false;
02889     } else {
02890         for (unsigned i = 0; i < numel(); i++) {
02891             if (at(i) != A(i)) {
02892                 ret = false;
02893                 break;
02894             }
02895         }
02896     }
02897     return ret;
02898 }
02899
02900 template<typename T>
02901 bool Matrix<T>::isequal(const Matrix<T>& A, T tol) const {
02902     bool ret = true;
02903     if (rows() != A.rows() || cols() != A.cols()) {
02904         ret = false;
02905     } else {
02906         auto abs_tol = std::abs(tol); // workaround for complex
02907         for (unsigned i = 0; i < A.numel(); i++) {
02908             if (abs_tol < std::abs(at(i) - A(i))) {
02909                 ret = false;
02910                 break;
02911             }
02912         }
02913     }
02914     return ret;
02915 }
02916
02917 template<typename T>
02918 inline unsigned Matrix<T>::numel() const {
02919     return nrows * ncols;
02920 }
02921
02922 template<typename T>
02923 inline unsigned Matrix<T>::rows() const {
02924     return nrows;
02925 }
02926
02927 template<typename T>
02928 inline unsigned Matrix<T>::cols() const {
02929     return ncols;
02930 }
02931
02932 template<typename T>
02933 inline Matrix<T> Matrix<T>::transpose() const {
02934     Matrix<T> res(ncols, nrows);
02935     for (unsigned c = 0; c < ncols; c++)
02936         for (unsigned r = 0; r < nrows; r++)
02937             res(c,r) = at(r,c);
02938     return res;
02939 }
02940
02941 template<typename T>
02942 inline Matrix<T> Matrix<T>::ctranspose() const {
02943     Matrix<T> res(ncols, nrows);
02944     for (unsigned c = 0; c < ncols; c++)
02945         for (unsigned r = 0; r < nrows; r++)
02946             res(c,r) = cconj(at(r,c));
02947     return res;
02948 }
02949
02950 template<typename T>
02951 Matrix<T>& Matrix<T>::add(const Matrix<T>& m) {
02952     if (!(m.rows() == rows() && m.cols() == cols())) throw std::runtime_error("Unmatching matrix
dimensions for iadd");
02953     for (unsigned i = 0; i < numel(); i++)
02954         data[i] += m(i);
02955     return *this;
02956 }
02957
02958
02959 template<typename T>

```

```

02960 Matrix<T>& Matrix<T>::subtract(const Matrix<T>& m) {
02961     if (!(m.rows() == rows() && m.cols() == cols())) throw std::runtime_error("Unmatching matrix
    dimensions for isubtract");
02962
02963     for (unsigned i = 0; i < numel(); i++)
02964         data[i] -= m(i);
02965     return *this;
02966 }
02967
02968 template<typename T>
02969 Matrix<T>& Matrix<T>::mult_hadamard(const Matrix<T>& m) {
02970     if (!(m.rows() == rows() && m.cols() == cols())) throw std::runtime_error("Unmatching matrix
    dimensions for ihprod");
02971
02972     for (unsigned i = 0; i < numel(); i++)
02973         data[i] *= m(i);
02974     return *this;
02975 }
02976
02977 template<typename T>
02978 Matrix<T>& Matrix<T>::add(T s) {
02979     for (auto& x : data)
02980         x += s;
02981     return *this;
02982 }
02983
02984 template<typename T>
02985 Matrix<T>& Matrix<T>::subtract(T s) {
02986     for (auto& x : data)
02987         x -= s;
02988     return *this;
02989 }
02990
02991 template<typename T>
02992 Matrix<T>& Matrix<T>::mult(T s) {
02993     for (auto& x : data)
02994         x *= s;
02995     return *this;
02996 }
02997
02998 template<typename T>
02999 Matrix<T>& Matrix<T>::div(T s) {
03000     for (auto& x : data)
03001         x /= s;
03002     return *this;
03003 }
03004
03005 template<typename T>
03006 void Matrix<T>::add_row_to_another(unsigned to, unsigned from) {
03007     if (!(to < rows() && from < rows())) throw std::out_of_range("Row index out of range");
03008
03009     for (unsigned k = 0; k < cols(); k++)
03010         at(to, k) += at(from, k);
03011 }
03012
03013 template<typename T>
03014 void Matrix<T>::add_col_to_another(unsigned to, unsigned from) {
03015     if (!(to < cols() && from < cols())) throw std::out_of_range("Column index out of range");
03016
03017     for (unsigned k = 0; k < rows(); k++)
03018         at(k, to) += at(k, from);
03019 }
03020
03021 template<typename T>
03022 void Matrix<T>::mult_row_by_another(unsigned to, unsigned from) {
03023     if (!(to < rows() && from < rows())) throw std::out_of_range("Row index out of range");
03024
03025     for (unsigned k = 0; k < cols(); k++)
03026         at(to, k) *= at(from, k);
03027 }
03028
03029 template<typename T>
03030 void Matrix<T>::mult_col_by_another(unsigned to, unsigned from) {
03031     if (!(to < cols() && from < cols())) throw std::out_of_range("Column index out of range");
03032
03033     for (unsigned k = 0; k < rows(); k++)
03034         at(k, to) *= at(k, from);
03035 }
03036
03037 template<typename T>
03038 void Matrix<T>::swap_rows(unsigned i, unsigned j) {
03039     if (!(i < rows() && j < rows())) throw std::out_of_range("Row index out of range");
03040
03041     for (unsigned k = 0; k < cols(); k++) {
03042         T tmp = at(i, k);
03043         at(i, k) = at(j, k);
03044         at(j, k) = tmp;
    }

```

```

03045     }
03046 }
03047
03048 template<typename T>
03049 void Matrix<T>::swap_cols(unsigned i, unsigned j) {
03050     if (!(i < cols() && j < cols())) throw std::out_of_range("Column index out of range");
03051     for (unsigned k = 0; k < rows(); k++) {
03052         T tmp = at(k,i);
03053         at(k,i) = at(k,j);
03054         at(k,j) = tmp;
03055     }
03056 }
03057 }
03058
03059 template<typename T>
03060 inline std::vector<T> Matrix<T>::to_vector() const {
03061     return data;
03062 }
03063
03064 template<typename T>
03065 inline std::vector<T> Matrix<T>::col_to_vector(unsigned col) const {
03066     std::vector<T> vec(rows());
03067     for (unsigned i = 0; i < rows(); i++)
03068         vec[i] = at(i,col);
03069     return vec;
03070 }
03071
03072 template<typename T>
03073 inline std::vector<T> Matrix<T>::row_to_vector(unsigned row) const {
03074     std::vector<T> vec(cols());
03075     for (unsigned i = 0; i < cols(); i++)
03076         vec[i] = at(row,i);
03077     return vec;
03078 }
03079
03080 template<typename T>
03081 inline void Matrix<T>::col_from_vector(const std::vector<T>& vec, unsigned col) {
03082     if (vec.size() != rows()) throw std::runtime_error("Vector size is not equal to number of rows");
03083     if (col >= cols()) throw std::out_of_range("Column index out of range");
03084     for (unsigned i = 0; i < rows(); i++)
03085         data[col*rows() + i] = vec[i];
03086 }
03087 }
03088
03089 template<typename T>
03090 inline void Matrix<T>::row_from_vector(const std::vector<T>& vec, unsigned row) {
03091     if (vec.size() != cols()) throw std::runtime_error("Vector size is not equal to number of columns");
03092     if (row >= rows()) throw std::out_of_range("Row index out of range");
03093     for (unsigned i = 0; i < cols(); i++)
03094         data[row + i*rows()] = vec[i];
03095 }
03096 }
03097
03098 template<typename T>
03099 Matrix<T>::~Matrix() { }
03100
03101 } // namespace Matrix_hpp
03102
03103 #endif // __MATRIX_HPP__

```