

Matrix HPP

Generated by Doxygen 1.9.8

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

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

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

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

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

Chapter 1

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

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

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

1.1 Installation

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

1.2 Functionality

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

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

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

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

1.3 Hello world example

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

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

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

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

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

    auto C = A + B;

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

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

1.4 Tests

Unit tests are compiled with `make tests`.

1.5 License

MIT license is used for this project. Please refer to LICENSE for details.

Chapter 2

Hierarchical Index

2.1 Class Hierarchy

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

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

Chapter 3

Class Index

3.1 Class List

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

Mtx::Eigenvalues_result< T >	
Result of eigenvalues	9
Mtx::Hessenberg_result< T >	
Result of Hessenberg decomposition	9
Mtx::LDL_result< T >	
Result of LDL decomposition	10
Mtx::LU_result< T >	
Result of LU decomposition	11
Mtx::LUP_result< T >	
Result of LU decomposition with pivoting	11
Mtx::Matrix< T >	12
Mtx::QR_result< T >	
Result of QR decomposition	30
Mtx::singular_matrix_exception	
Singular matrix exception	30

Chapter 4

File Index

4.1 File List

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

matrix.hpp	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.

Referenced by [Mtx::Matrix< T >::add\(\)](#), [Mtx::Matrix< T >::col_from_vector\(\)](#), [Mtx::Matrix< T >::col_to_vector\(\)](#), [Mtx::Matrix< T >::ctranspose\(\)](#), [Mtx::Matrix< T >::get_submatrix\(\)](#), [Mtx::Matrix< T >::isequal\(\)](#), [Mtx::Matrix< T >::isequal\(\)](#), [Mtx::Matrix< T >::mult_hadamard\(\)](#), [Mtx::Matrix< T >::ptr\(\)](#), [Mtx::Matrix< T >::row_from_vector\(\)](#), [Mtx::Matrix< T >::row_to_vector\(\)](#), [Mtx::Matrix< T >::set_submatrix\(\)](#), [Mtx::Matrix< T >::subtract\(\)](#), [Mtx::Matrix< T >::swap_cols\(\)](#), [Mtx::Matrix< T >::swap_rows\(\)](#), and [Mtx::Matrix< T >::transpose\(\)](#).

5.6.2.2 Matrix() [2/8]

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

Square matrix constructor.

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

5.6.2.3 Matrix() [3/8]

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

Rectangular matrix constructor.

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

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

5.6.2.4 Matrix() [4/8]

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

Rectangular matrix constructor with fill.

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

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

5.6.2.5 Matrix() [5/8]

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

Rectangular matrix constructor with initialization.

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

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

5.6.2.6 Matrix() [6/8]

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

Rectangular matrix constructor with initialization.

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

Exceptions

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

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

5.6.2.7 Matrix() [7/8]

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

Rectangular matrix constructor with initialization.

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

Exceptions

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

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

5.6.2.8 Matrix() [8/8]

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

Copy constructor.

5.6.2.9 ~Matrix()

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

Destructor.

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

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

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

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

Exceptions

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

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

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

5.6.3.2 add() [2/2]

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

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

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

5.6.3.3 add_col_to_another()

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

Column addition.

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

Exceptions

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

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

5.6.3.4 add_row_to_another()

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

Row addition.

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

Exceptions

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

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

5.6.3.5 clear()

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

Clears the matrix.

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

5.6.3.6 col_from_vector()

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

Column from vector.

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

Exceptions

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

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

5.6.3.7 col_to_vector()

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

Column to vector.

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

Exceptions

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

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

5.6.3.8 cols()

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

Number of columns.

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

Referenced by [Mtx::Matrix< T >::add\(\)](#), [Mtx::add\(\)](#), [Mtx::add\(\)](#), [Mtx::Matrix< T >::add_col_to_another\(\)](#), [Mtx::Matrix< T >::add_row_to_another\(\)](#), [Mtx::adj\(\)](#), [Mtx::circshift\(\)](#), [Mtx::cofactor\(\)](#), [Mtx::Matrix< T >::col_from_vector\(\)](#), [Mtx::concatenate_horizontal\(\)](#), [Mtx::concatenate_vertical\(\)](#), [Mtx::div\(\)](#), [Mtx::Matrix< T >::fill_col\(\)](#), [Mtx::Matrix< T >::get_submatrix\(\)](#), [Mtx::householder_reflection\(\)](#), [Mtx::imag\(\)](#), [Mtx::Matrix< T >::isequal\(\)](#), [Mtx::Matrix< T >::isequal\(\)](#), [Mtx::istril\(\)](#), [Mtx::istriu\(\)](#), [Mtx::kron\(\)](#), [Mtx::lu\(\)](#), [Mtx::lup\(\)](#), [Mtx::make_complex\(\)](#), [Mtx::make_complex\(\)](#), [Mtx::mult\(\)](#), [Mtx::mult\(\)](#), [Mtx::mult\(\)](#), [Mtx::mult\(\)](#), [Mtx::Matrix< T >::mult_col_by_another\(\)](#), [Mtx::Matrix< T >::mult_hadamard\(\)](#), [Mtx::mult_hadamard\(\)](#), [Mtx::Matrix< T >::mult_row_by_another\(\)](#), [Mtx::operator<<\(\)](#), [Mtx::permute_cols\(\)](#), [Mtx::permute_rows\(\)](#), [Mtx::permute_rows_and_cols\(\)](#), [Mtx::pinv\(\)](#), [Mtx::Matrix< T >::ptr\(\)](#), [Mtx::qr_householder\(\)](#), [Mtx::qr_red_gs\(\)](#), [Mtx::real\(\)](#), [Mtx::repmat\(\)](#), [Mtx::Matrix< T >::reshape\(\)](#), [Mtx::Matrix< T >::resize\(\)](#), [Mtx::Matrix< T >::row_from_vector\(\)](#), [Mtx::Matrix< T >::row_to_vector\(\)](#), [Mtx::Matrix< T >::set_submatrix\(\)](#), [Mtx::solve_tril\(\)](#), [Mtx::solve_triu\(\)](#), [Mtx::Matrix< T >::subtract\(\)](#), [Mtx::subtract\(\)](#), [Mtx::subtract\(\)](#), [Mtx::Matrix< T >::swap_cols\(\)](#), [Mtx::Matrix< T >::swap_rows\(\)](#), [Mtx::tril\(\)](#), and [Mtx::triu\(\)](#).

5.6.3.9 ctranspose()

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

Transpose a complex matrix.

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

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

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

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

5.6.3.10 div()

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

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

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

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

5.6.3.11 exists()

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

Element exist check.

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

5.6.3.12 fill()

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

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

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

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

5.6.3.13 fill_col()

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

Fill column with a scalar.

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

Exceptions

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

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

5.6.3.14 fill_row()

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

Fill row with a scalar.

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

Exceptions

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

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

5.6.3.15 `get_submatrix()`

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

Extract a submatrix.

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

Exceptions

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

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

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

5.6.3.16 `isempty()`

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

Emptiness check.

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

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

5.6.3.17 `isequal()` [1/2]

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

[Matrix](#) equality check.

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

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

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

5.6.3.18 isequal() [2/2]

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

Matrix equality check with tolerance.

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

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

5.6.3.19 mult()

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

Matrix product with scalar (in-place).

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

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

5.6.3.20 mult_col_by_another()

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

Column multiplication.

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

Exceptions

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

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

5.6.3.21 mult_hadamard()

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

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

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

Exceptions

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

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

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

5.6.3.22 mult_row_by_another()

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

Row multiplication.

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

Exceptions

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

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

5.6.3.23 numel()

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

[Matrix](#) capacity.

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

Referenced by [Mtx::Matrix< T >::add\(\)](#), [Mtx::add\(\)](#), [Mtx::div\(\)](#), [Mtx::Matrix< T >::fill\(\)](#), [Mtx::foreach_elem\(\)](#), [Mtx::householder_reflection\(\)](#), [Mtx::imag\(\)](#), [Mtx::inv\(\)](#), [Mtx::Matrix< T >::isequal\(\)](#), [Mtx::Matrix< T >::isequal\(\)](#), [Mtx::lu\(\)](#), [Mtx::lup\(\)](#), [Mtx::make_complex\(\)](#), [Mtx::make_complex\(\)](#), [Mtx::Matrix< T >::Matrix\(\)](#), [Mtx::Matrix< T >::Matrix\(\)](#), [Mtx::Matrix< T >::Matrix\(\)](#), [Mtx::Matrix< T >::Matrix\(\)](#), [Mtx::mult\(\)](#), [Mtx::Matrix< T >::mult_hadamard\(\)](#), [Mtx::norm_fro\(\)](#), [Mtx::norm_fro\(\)](#), [Mtx::real\(\)](#), [Mtx::Matrix< T >::reshape\(\)](#), [Mtx::solve_posdef\(\)](#), [Mtx::solve_square\(\)](#), [Mtx::solve_tril\(\)](#), [Mtx::solve_triu\(\)](#), [Mtx::Matrix< T >::subtract\(\)](#), and [Mtx::subtract\(\)](#).

5.6.3.24 operator std::vector< T >()

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

Vector cast operator.

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

5.6.3.25 operator>() [1/2]

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

Element access operator (1D)

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

Exceptions

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

5.6.3.26 operator>() [2/2]

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

Element access operator (2D)

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

Exceptions

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

5.6.3.27 operator=() [1/2]

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

Matrix assignment.

Performs deep-copy of another matrix.

5.6.3.28 operator=() [2/2]

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

Matrix fill operator.

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

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

5.6.3.29 ptr() [1/2]

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

Memory pointer.

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

Exceptions

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

5.6.3.30 ptr() [2/2]

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

Memory pointer.

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

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

5.6.3.31 reshape()

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

[Matrix](#) dimension reshape.

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

Exceptions

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

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

5.6.3.32 resize()

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

```

    unsigned rows,
    unsigned cols )

```

Resize the matrix.

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

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

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

5.6.3.33 row_from_vector()

```

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

```

Row from vector.

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

Exceptions

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

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

5.6.3.34 row_to_vector()

```

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

```

Row to vector.

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

Exceptions

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

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

5.6.3.35 rows()

```

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

```

Number of rows.

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

Referenced by [Mtx::Matrix< T >::add\(\)](#), [Mtx::add\(\)](#), [Mtx::add\(\)](#), [Mtx::Matrix< T >::add_col_to_another\(\)](#), [Mtx::Matrix< T >::add_row_to_another\(\)](#), [Mtx::adj\(\)](#), [Mtx::chol\(\)](#), [Mtx::cholinv\(\)](#), [Mtx::circshift\(\)](#), [Mtx::cofactor\(\)](#), [Mtx::Matrix< T >::col_from_vector\(\)](#), [Mtx::Matrix< T >::col_to_vector\(\)](#), [Mtx::concatenate_horizontal\(\)](#), [Mtx::concatenate_vertical\(\)](#), [Mtx::det\(\)](#), [Mtx::det_lu\(\)](#), [Mtx::diag\(\)](#), [Mtx::div\(\)](#), [Mtx::eigenvalues\(\)](#), [Mtx::Matrix< T >::fill_row\(\)](#), [Mtx::Matrix< T >::get_submatrix\(\)](#), [Mtx::hessenberg\(\)](#), [Mtx::imag\(\)](#), [Mtx::inv\(\)](#), [Mtx::inv_gauss_jordan\(\)](#), [Mtx::inv_tril\(\)](#), [Mtx::inv_triu\(\)](#), [Mtx::Matrix< T >::isequal\(\)](#), [Mtx::Matrix< T >::isequal\(\)](#), [Mtx::ishess\(\)](#), [Mtx::istril\(\)](#), [Mtx::istriu\(\)](#), [Mtx::kron\(\)](#), [Mtx::ldl\(\)](#), [Mtx::lu\(\)](#), [Mtx::lup\(\)](#), [Mtx::make_complex\(\)](#), [Mtx::make_complex\(\)](#), [Mtx::mult\(\)](#), [Mtx::mult\(\)](#), [Mtx::mult\(\)](#), [Mtx::mult\(\)](#), [Mtx::Matrix< T >::mult_col_by_another\(\)](#), [Mtx::Matrix< T >::mult_hadamard\(\)](#), [Mtx::mult_hadamard\(\)](#), [Mtx::Matrix< T >::mult_row_by_another\(\)](#), [Mtx::operator<<\(\)](#), [Mtx::permute_cols\(\)](#), [Mtx::permute_rows\(\)](#), [Mtx::permute_rows_and_cols\(\)](#), [Mtx::pinv\(\)](#), [Mtx::Matrix< T >::ptr\(\)](#), [Mtx::qr_householder\(\)](#), [Mtx::qr_red_gs\(\)](#), [Mtx::real\(\)](#), [Mtx::repmat\(\)](#), [Mtx::Matrix< T >::reshape\(\)](#), [Mtx::Matrix< T >::resize\(\)](#), [Mtx::Matrix< T >::row_from_vector\(\)](#), [Mtx::Matrix< T >::set_submatrix\(\)](#), [Mtx::solve_posdef\(\)](#), [Mtx::solve_square\(\)](#), [Mtx::solve_tril\(\)](#), [Mtx::solve_triu\(\)](#), [Mtx::Matrix< T >::subtract\(\)](#), [Mtx::subtract\(\)](#), [Mtx::subtract\(\)](#), [Mtx::Matrix< T >::swap_cols\(\)](#), [Mtx::Matrix< T >::swap_rows\(\)](#), [Mtx::trace\(\)](#), [Mtx::tril\(\)](#), [Mtx::triu\(\)](#), and [Mtx::wilkinson_shift\(\)](#).

5.6.3.36 set_submatrix()

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

Embed a submatrix.

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

Exceptions

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

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

5.6.3.37 subtract() [1/2]

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

Matrix subtraction (in-place).

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

Exceptions

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

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

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

5.6.3.38 subtract() [2/2]

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

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

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

5.6.3.39 swap_cols()

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

Column swap.

Swaps element values between two columns.

Exceptions

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

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

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

5.6.3.40 swap_rows()

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

Row swap.

Swaps element values of two columns.

Exceptions

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

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

5.6.3.41 transpose()

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

Transpose a matrix.

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

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

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

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

- [matrix.hpp](#)

5.7 Mtx::QR_result< T > Struct Template Reference

Result of QR decomposition.

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

Public Attributes

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

5.7.1 Detailed Description

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

Result of QR decomposition.

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

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

- [matrix.hpp](#)

5.8 Mtx::singular_matrix_exception Class Reference

Singular matrix exception.

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

Inheritance diagram for Mtx::singular_matrix_exception:

Chapter 6

File Documentation

6.1 matrix.hpp File Reference

Classes

- class [Mtx::singular_matrix_exception](#)
Singular matrix exception.
- struct [Mtx::LU_result< T >](#)
Result of LU decomposition.
- struct [Mtx::LUP_result< T >](#)
Result of LU decomposition with pivoting.
- struct [Mtx::QR_result< T >](#)
Result of QR decomposition.
- struct [Mtx::Hessenberg_result< T >](#)
Result of Hessenberg decomposition.
- struct [Mtx::LDL_result< T >](#)
Result of LDL decomposition.
- struct [Mtx::Eigenvalues_result< T >](#)
Result of eigenvalues.
- class [Mtx::Matrix< T >](#)

Functions

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

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

- Vertical matrix concatenation.*

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

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

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

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

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

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

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

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

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

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

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

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

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

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

Matrix Hadamard (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::cholinv (const Matrix< T > &A)`
Inverse of Cholesky decomposition.
 - `template<typename T >`
`LDL_result< T > Mtx::ldl (const Matrix< T > &A)`
LDL decomposition.
 - `template<typename T >`
`QR_result< T > Mtx::qr_red_gs (const Matrix< T > &A)`
Reduced QR decomposition based on Gram-Schmidt method.
 - `template<typename T >`
`Matrix< T > Mtx::householder_reflection (const Matrix< T > &a)`
Generate Householder reflection.
 - `template<typename T >`
`QR_result< T > Mtx::qr_householder (const Matrix< T > &A, bool calculate_Q=true)`
QR decomposition based on Householder method.
 - `template<typename T >`
`QR_result< T > Mtx::qr (const Matrix< T > &A, bool calculate_Q=true)`
QR decomposition.
 - `template<typename T >`
`Hessenberg_result< T > Mtx::hessenberg (const Matrix< T > &A, bool calculate_Q=true)`
Hessenberg decomposition.
 - `template<typename T >`
`std::complex< T > Mtx::wilkinson_shift (const Matrix< std::complex< T > > &H, T tol=1e-10)`
Wilkinson's shift for complex eigenvalues.

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

6.1.1 Function Documentation

6.1.1.1 add() [1/2]

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

Matrix addition.

Performs addition of two matrices.

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

Template Parameters

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

Parameters

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

Returns

output matrix of size $N \times M$

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

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

6.1.1.2 add() [2/2]

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

Addition of scalar to matrix.

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

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

6.1.1.3 adj()

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

Adjugate matrix.

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

More information: https://en.wikipedia.org/wiki/Adjugate_matrix

Exceptions

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

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

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

6.1.1.4 cconj()

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

Complex conjugate helper.

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

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

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

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

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

6.1.1.5 chol()

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

Cholesky decomposition.

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

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

More information: https://en.wikipedia.org/wiki/Cholesky_decomposition

Template Parameters

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

Exceptions

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

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

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

6.1.1.6 cholinv()

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

Inverse of Cholesky decomposition.

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

See [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

Exceptions

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

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

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

6.1.1.7 circshift()

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

Circular shift.

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

Parameters

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

Returns

matrix inverse

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

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

6.1.1.8 circulant() [1/2]

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

Circulant matrix from std::vector.

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

Parameters

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

Returns

circulant matrix

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

6.1.1.9 `circulant()` [2/2]

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

Circulant matrix from array.

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

Parameters

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

Returns

circulant matrix

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

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

6.1.1.10 `cofactor()`

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

Cofactor matrix.

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

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

Parameters

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

Exceptions

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

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

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

6.1.1.11 concatenate_horizontal()

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

Horizontal matrix concatenation.

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

Exceptions

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

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

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

6.1.1.12 concatenate_vertical()

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

Vertical matrix concatenation.

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

Exceptions

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

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

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

6.1.1.13 cond()

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

Condition number of a matrix.

Calculates condition number of a matrix. The condition number of a matrix measures 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\(\)](#).

6.1.1.14 csign()

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

Complex sign helper.

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

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

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

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

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

6.1.1.15 ctranspose()

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

Transpose a complex matrix.

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

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

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

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

6.1.1.16 det()

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

Matrix determinant.

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

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

Exceptions

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

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

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

6.1.1.17 det_lu()

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

Matrix determinant from on LU decomposition.

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

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

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

Exceptions

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

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

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

6.1.1.18 diag() [1/3]

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

Diagonal extraction.

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

Parameters

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

Returns

vector of diagonal elements

Exceptions

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

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

6.1.1.19 diag() [2/3]

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

Diagonal matrix from `std::vector`.

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

Parameters

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

Returns

diagonal matrix

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

6.1.1.20 diag() [3/3]

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

Diagonal matrix from array.

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

Parameters

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

Returns

diagonal matrix

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

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

6.1.1.21 div()

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

Division of matrix by scalar.

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

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

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

6.1.1.22 eigenvalues() [1/2]

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

Matrix eigenvalues of complex matrix.

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

Parameters

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

Returns

structure containing the result and status of eigenvalue calculation

Exceptions

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

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

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

6.1.1.23 eigenvalues() [2/2]

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

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

Matrix eigenvalues of real matrix.

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

Parameters

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

Returns

structure containing the result and status of eigenvalue calculation

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

6.1.1.24 eye()

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

Identity matrix.

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

Parameters

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

Returns

zeros matrix

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

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

6.1.1.25 foreach_elem()

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

Applies custom function element-wise in-place.

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

This function applies operation to the elements in-place (zero-copy). In order to apply the function to the copy of the matrix without modifying the input one, use `foreach_elem_copy()`.

Parameters

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

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

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

6.1.1.26 foreach_elem_copy()

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

Applies custom function element-wise with matrix copy.

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

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

Parameters

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

Returns

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

References [Mtx::foreach_elem\(\)](#), and [Mtx::foreach_elem_copy\(\)](#).

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

6.1.1.27 hessenberg()

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

Hessenberg decomposition.

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

Parameters

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

Returns

structure encapsulating calculated *H* and *Q*. *Q* is calculated only when *calculate_Q* = True.

Exceptions

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

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

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

6.1.1.28 householder_reflection()

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

Generate Householder reflection.

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

Parameters

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

Returns

column vector with Householder reflection of *a*

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

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

6.1.1.29 imag()

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

Get imaginary part of complex matrix.

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

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

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

6.1.1.30 inv()

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

Matrix inverse (universal).

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

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

More information: https://en.wikipedia.org/wiki/Gaussian_elimination

Exceptions

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

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

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

6.1.1.31 inv_gauss_jordan()

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

Matrix inverse using Gauss-Jordan elimination.

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

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

More information: https://en.wikipedia.org/wiki/Gaussian_elimination

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\(\)](#).

6.1.1.32 inv_posdef()

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

Matrix inverse for Hermitian positive-definite matrix.

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

This function provides more optimal performance than `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

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::cholinv\(\)](#), and [Mtx::inv_posdef\(\)](#).

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

6.1.1.33 inv_square()

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

Matrix inverse for general square matrix.

Calculates an inverse of square matrix using matrix.

This function provides more optimal performance than `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

<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_square\(\)](#), [Mtx::inv_tril\(\)](#), [Mtx::inv_triu\(\)](#), [Mtx::Matrix< T >::issquare\(\)](#), [Mtx::lup\(\)](#), and [Mtx::permute_rows\(\)](#).

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

6.1.1.34 inv_tril()

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

Matrix inverse for lower triangular matrix.

Calculates an inverse of lower triangular matrix.

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

Exceptions

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

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

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

6.1.1.35 inv_triu()

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

Matrix inverse for upper triangular matrix.

Calculates an inverse of upper triangular matrix.

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

Exceptions

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

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

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

6.1.1.36 ishess()

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

Hessenberg matrix check.

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

6.1.1.37 istril()

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

Lower triangular matrix check.

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

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

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

6.1.1.38 istriu()

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

Lower triangular matrix check.

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

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

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

6.1.1.39 kron()

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

Kronecker product.

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

More information: https://en.wikipedia.org/wiki/Kronecker_product

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

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

6.1.1.40 ldl()

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

LDL decomposition.

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

$$A = LDL^H$$

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

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

More information: https://en.wikipedia.org/wiki/Cholesky_decomposition#LDL_decomposition

Parameters

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

Returns

structure encapsulating calculated L and D

Exceptions

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

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

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

6.1.1.41 lu()

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

LU decomposition.

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

This function implements LU factorization without pivoting. Use `lup()` if pivoting is required.

More information: https://en.wikipedia.org/wiki/LU_decomposition

Parameters

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

Returns

structure containing calculated L and U matrices

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

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

6.1.1.42 lup()

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

LU decomposition with pivoting.

Performs LU factorization with partial pivoting, employing column permutations.

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

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

More information: https://en.wikipedia.org/wiki/LU_decomposition#LU_factorization_with_partial_pivoting

Parameters

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

Returns

structure containing L , U and P .

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

Referenced by [Mtx::det_lu\(\)](#), [Mtx::inv_square\(\)](#), [Mtx::lup\(\)](#), and [Mtx::solve_square\(\)](#).

6.1.1.43 make_complex() [1/2]

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

Create complex matrix from real matrix.

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

Parameters

Re	real part matrix
------	------------------

Returns

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

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

6.1.1.44 make_complex() [2/2]

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

Create complex matrix from real and imaginary matrices.

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

Parameters

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

Returns

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

Exceptions

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

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

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

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

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

Matrix multiplication.

Performs multiplication of two matrices.

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

Template Parameters

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

Parameters

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

Returns

output matrix of size $N \times K$

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

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

6.1.1.46 mult() [2/4]

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

Multiplication of matrix by std::vector.

Performs the right multiplication of a matrix with a column vector represented by std::vector. The result of 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\(\)](#).

6.1.1.47 mult() [3/4]

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

Multiplication of matrix by scalar.

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

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

6.1.1.48 mult() [4/4]

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

Multiplication of std::vector by matrix.

Performs the left multiplication of a std::vector with a matrix. The result of 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\(\)](#).

6.1.1.49 mult_hadamard()

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

Matrix Hadamard (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^\(\)](#).

6.1.1.50 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

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

6.1.1.51 norm_fro() [2/2]

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

Frobenius norm.

Calculates Frobenius norm of real matrix.

More information https://en.wikipedia.org/wiki/Matrix_norm#Frobenius_norm

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

Referenced by [Mtx::cond\(\)](#), [Mtx::householder_reflection\(\)](#), [Mtx::norm_fro\(\)](#), [Mtx::norm_fro\(\)](#), and [Mtx::qr_red_gs\(\)](#).

6.1.1.52 ones() [1/2]

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

Square matrix of ones.

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

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

Parameters

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

Returns

zeros matrix

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

6.1.1.53 ones() [2/2]

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

Matrix of ones.

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

Parameters

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

Returns

ones matrix

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

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

6.1.1.54 operator!=(())

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

Matrix non-equality check operator.

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

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

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

6.1.1.55 operator*() [1/5]

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

Matrix product.

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

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

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

6.1.1.56 operator*() [2/5]

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

Matrix and std::vector product.

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

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

6.1.1.57 operator*() [3/5]

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

Matrix product with scalar.

Multiplies each element of the matrix by a scalar s .

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

6.1.1.58 operator*() [4/5]

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

std::vector and matrix product.

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

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

6.1.1.59 operator*() [5/5]

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

Matrix product with scalar.

Multiplies each element of the matrix by a scalar s .

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

6.1.1.60 operator*=() [1/2]

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

Matrix product.

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

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

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

6.1.1.61 operator*=() [2/2]

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

Matrix product with scalar.

Multiplies each element of the matrix by a scalar s .

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

6.1.1.62 operator+() [1/3]

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

Matrix sum.

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

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

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

6.1.1.63 operator+() [2/3]

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

Matrix sum with scalar.

Adds a scalar s to each element of the matrix.

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

6.1.1.64 operator+() [3/3]

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

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

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

6.1.1.65 operator+=() [1/2]

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

Matrix sum.

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

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

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

6.1.1.66 operator+=() [2/2]

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

Matrix sum with scalar.

Adds a scalar s to each element of the matrix.

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

6.1.1.67 operator-() [1/2]

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

Matrix subtraction.

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

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

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

6.1.1.68 operator-() [2/2]

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

Matrix subtraction with scalar.

Subtracts a scalar s from each element of the matrix.

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

6.1.1.69 operator-=() [1/2]

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

Matrix subtraction.

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

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

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

6.1.1.70 operator-=() [2/2]

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

Matrix subtraction with scalar.

Subtracts a scalar s from each element of the matrix.

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

6.1.1.71 operator/()

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

Matrix division by scalar.

Divides each element of the matrix by a scalar s .

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

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

6.1.1.72 operator/=()

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

Matrix division by scalar.

Divides each element of the matrix by a scalar s .

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

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

6.1.1.73 operator<<()

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

Matrix ostream operator.

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

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

6.1.1.74 operator==()

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

Matrix equality check operator.

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

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

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

6.1.1.75 operator^()

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

Matrix Hadamard product.

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

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

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

6.1.1.76 operator^=()

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

Matrix Hadamard product.

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

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

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

6.1.1.77 permute_cols()

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

Permute columns of the matrix.

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

Parameters

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

Returns

output matrix created by column permutation of A

Exceptions

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

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

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

6.1.1.78 permute_rows()

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

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

Permute rows of the matrix.

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

Parameters

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

Returns

output matrix created by row permutation of *A*

Exceptions

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

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

Referenced by [Mtx::inv_square\(\)](#), [Mtx::permute_rows\(\)](#), and [Mtx::solve_square\(\)](#).

6.1.1.79 permute_rows_and_cols()

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

Permute both rows and columns of the matrix.

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

The size of the output matrix is *perm_rows.size()* x *perm_cols.size()*.

Parameters

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

Returns

output matrix created by row and column permutation of A

Exceptions

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

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

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

6.1.1.80 pinv()

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

Moore-Penrose 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

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

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

6.1.1.81 qr()

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

QR decomposition.

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

Currently, this function is a wrapper around `qr_householder()`. Refer to `qr_red_gs()` for alternative implementation.

Parameters

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

Returns

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

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

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

6.1.1.82 qr_householder()

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

QR decomposition based on Householder method.

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

This function implements QR decomposition based on Householder reflections method.

More information: https://en.wikipedia.org/wiki/QR_decomposition

Parameters

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

Returns

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

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

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

6.1.1.83 qr_red_gs()

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

Reduced QR decomposition based on Gram-Schmidt method.

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

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

More information: https://en.wikipedia.org/wiki/QR_decomposition

Parameters

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

Returns

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

Exceptions

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

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

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

6.1.1.84 real()

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

Get real part of complex matrix.

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

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

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

6.1.1.85 repmat()

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

Repeat matrix.

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

Parameters

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

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

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

6.1.1.86 solve_posdef()

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

Solves the positive definite (Hermitian) system.

Return the matrix left division of A and B , where A is positive definite matrix. It is equivalent to solving the system $A \cdot X = B$

with respect to X . The system is solved for each column of B using Cholesky decomposition followed by forward and backward propagation.

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

Parameters

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

Returns

solution matrix of size $N \times M$.

Exceptions

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

References [Mtx::chol\(\)](#), [Mtx::Matrix< T >::issquare\(\)](#), [Mtx::Matrix< T >::numel\(\)](#), [Mtx::Matrix< T >::rows\(\)](#), [Mtx::solve_posdef\(\)](#), [Mtx::solve_tril\(\)](#), and [Mtx::solve_triu\(\)](#).

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

6.1.1.87 solve_square()

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

Solves the square system.

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

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

Parameters

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

Returns

solution matrix of size $N \times M$.

Exceptions

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

References [Mtx::Matrix< T >::issquare\(\)](#), [Mtx::lup\(\)](#), [Mtx::Matrix< T >::numel\(\)](#), [Mtx::permute_rows\(\)](#), [Mtx::Matrix< T >::rows\(\)](#), [Mtx::solve_square\(\)](#), [Mtx::solve_tril\(\)](#), and [Mtx::solve_triu\(\)](#).

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

6.1.1.88 solve_tril()

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

Solves the lower triangular system.

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

Parameters

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

Returns

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

Exceptions

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

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

Referenced by [Mtx::solve_posdef\(\)](#), [Mtx::solve_square\(\)](#), and [Mtx::solve_tril\(\)](#).

6.1.1.89 solve_triu()

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

Solves the upper triangular system.

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

Parameters

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

Returns

solution matrix of size $N \times M$.

Exceptions

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

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

Referenced by [Mtx::solve_posdef\(\)](#), [Mtx::solve_square\(\)](#), and [Mtx::solve_triu\(\)](#).

6.1.1.90 subtract() [1/2]

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

Matrix subtraction.

Performs subtraction of two matrices.

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

Template Parameters

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

Parameters

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

Returns

output matrix of size $N \times M$

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

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

6.1.1.91 subtract() [2/2]

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

Subtraction of scalar from matrix.

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

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

6.1.1.92 trace()

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

Matrix trace.

Calculates trace of a matrix by summing the elements on the diagonal.

$$\text{tr}(A) = \sum_{n=0}^{N-1} [A]_{n,n}$$

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

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

6.1.1.93 transpose()

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

Transpose a matrix.

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

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

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

6.1.1.94 tril()

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

Extract triangular lower part.

Return a new matrix formed by extracting the lower triangular part of the input matrix, and setting all other elements to zero.

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

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

6.1.1.95 triu()

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

Extract triangular upper part.

Return a new matrix formed by extracting the upper triangular part of the input matrix, and setting all other elements to zero.

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

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

6.1.1.96 wilkinson_shift()

```
template<typename T >
std::complex< T > Mtx::wilkinson_shift (
    const Matrix< std::complex< T > > & H,
    T tol = 1e-10 )
```

Wilkinson's shift for complex eigenvalues.

Computes Wilkinson's shift value *mu* for complex eigenvalues of input matrix. Wilkinson's shift is calculated as eigenvalue of the bottom 2 x 2 principal minor closest to the corner entry of the matrix. Input must be a square matrix in Hessenberg form.

Exceptions

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

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

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

6.1.1.97 zeros() [1/2]

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

Square matrix of zeros.

Construct a square matrix of size $n \times n$ and fill it with all elements set to 0.

Parameters

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

Returns

zeros matrix

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

6.1.1.98 zeros() [2/2]

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

Matrix of zeros.

Create a matrix of size $nrows \times ncols$ and fill it with all elements set to 0.

Parameters

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

Returns

zeros matrix

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

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

6.2 matrix.hpp

[Go to the documentation of this file.](#)

```

00001
00002
00003 /* MIT License
00004 *
00005 * Copyright (c) 2024 gc1905
00006 *
00007 * Permission is hereby granted, free of charge, to any person obtaining a copy
00008 * of this software and associated documentation files (the "Software"), to deal
00009 * in the Software without restriction, including without limitation the rights
00010 * to use, copy, modify, merge, publish, distribute, sublicense, and/or sell
00011 * copies of the Software, and to permit persons to whom the Software is
00012 * furnished to do so, subject to the following conditions:
00013 *
00014 * The above copyright notice and this permission notice shall be included in all
00015 * copies or substantial portions of the Software.
00016 *
00017 * THE SOFTWARE IS PROVIDED "AS IS", WITHOUT WARRANTY OF ANY KIND, EXPRESS OR
00018 * IMPLIED, INCLUDING BUT NOT LIMITED TO THE WARRANTIES OF MERCHANTABILITY,
00019 * FITNESS FOR A PARTICULAR PURPOSE AND NONINFRINGEMENT. IN NO EVENT SHALL THE
00020 * AUTHORS OR COPYRIGHT HOLDERS BE LIABLE FOR ANY CLAIM, DAMAGES OR OTHER
00021 * LIABILITY, WHETHER IN AN ACTION OF CONTRACT, TORT OR OTHERWISE, ARISING FROM,
00022 * OUT OF OR IN CONNECTION WITH THE SOFTWARE OR THE USE OR OTHER DEALINGS IN THE
00023 * SOFTWARE.
00024 */
00025
00026 #ifndef __MATRIX_HPP__
00027 #define __MATRIX_HPP__
00028
00029 #include <ostream>
00030 #include <complex>
00031 #include <vector>
00032 #include <initializer_list>
00033 #include <limits>
00034 #include <functional>
00035 #include <algorithm>
00036
00037 namespace Mtx {
00038
00039 template<typename T> class Matrix;
00040
00041 template<class T> struct is_complex : std::false_type {};
00042 template<class T> struct is_complex<std::complex<T> > : std::true_type {};
00043
00044 template<typename T, typename std::enable_if<!is_complex<T>::value, int>::type = 0>
00045 inline T cconj(T x) {
00046     return x;
00047 }
00048
00049 template<typename T, typename std::enable_if<is_complex<T>::value, int>::type = 0>
00050 inline T cconj(T x) {
00051     return std::conj(x);
00052 }
00053
00054 template<typename T, typename std::enable_if<!is_complex<T>::value, int>::type = 0>
00055 inline T csign(T x) {
00056     return (x > static_cast<T>(0)) ? static_cast<T>(1) : static_cast<T>(-1);
00057 }
00058
00059 template<typename T, typename std::enable_if<is_complex<T>::value, int>::type = 0>
00060 inline T csign(T x) {
00061     auto x_arg = std::arg(x);
00062     T y(0, x_arg);
00063     return std::exp(y);
00064 }
00065
00066 class singular_matrix_exception : public std::domain_error {
00067 public:
00068     singular_matrix_exception(const std::string& message) : std::domain_error(message) {}
00069 };
00070
00071 template<typename T>
00072 struct LU_result {
00073     Matrix<T> L;
00074     Matrix<T> U;
00075 };
00076
00077 template<typename T>
00078 struct LUP_result {
00079     Matrix<T> L;
00080     Matrix<T> U;
00081 
```

```

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
00707 template<typename T>
00708 Matrix<T> permute_rows_and_cols(const Matrix<T>& A, const std::vector<unsigned> perm_rows, const
std::vector<unsigned> perm_cols) {
00709     if (perm_rows.empty()) throw std::runtime_error("Row permutation vector is empty");
00710     if (perm_cols.empty()) throw std::runtime_error("Column permutation vector is empty");
00711     Matrix<T> B(perm_rows.size(), perm_cols.size());
00712     for (unsigned pc = 0; pc < perm_cols.size(); pc++) {
00713         if (!perm_cols[pc] < A.cols()) throw std::out_of_range("Column index in permutation vector out
of range");
00714     }
00715 }
00716

```

```

00717     for (unsigned pr = 0; pr < perm_rows.size(); pr++) {
00718         if (! (perm_rows[pr] < A.rows())) throw std::out_of_range("Row index in permutation vector out of
range");
00719     }
00720     B(pr,pc) = A(perm_rows[pr],perm_cols[pc]);
00721 }
00722 }
00723
00724 return B;
00725 }
00726
00741 template<typename T, bool transpose_first = false, bool transpose_second = false>
00742 Matrix<T> mult(const Matrix<T>& A, const Matrix<T>& B) {
00743     // Adjust dimensions based on transpositions
00744     unsigned rows_A = transpose_first ? A.cols() : A.rows();
00745     unsigned cols_A = transpose_first ? A.rows() : A.cols();
00746     unsigned rows_B = transpose_second ? B.cols() : B.rows();
00747     unsigned cols_B = transpose_second ? B.rows() : B.cols();
00748
00749     if (cols_A != rows_B) throw std::runtime_error("Unmatching matrix dimensions for mult");
00750
00751     Matrix<T> C(static_cast<T>(0), rows_A, cols_B);
00752
00753     for (unsigned i = 0; i < rows_A; i++)
00754         for (unsigned j = 0; j < cols_B; j++)
00755             for (unsigned k = 0; k < cols_A; k++)
00756                 C(i,j) += (transpose_first ? cconj(A(k,i)) : A(i,k)) *
00757                     (transpose_second ? cconj(B(j,k)) : B(k,j));
00758
00759     return C;
00760 }
00761
00776 template<typename T, bool transpose_first = false, bool transpose_second = false>
00777 Matrix<T> mult_hadamard(const Matrix<T>& A, const Matrix<T>& B) {
00778     // Adjust dimensions based on transpositions
00779     unsigned rows_A = transpose_first ? A.cols() : A.rows();
00780     unsigned cols_A = transpose_first ? A.rows() : A.cols();
00781     unsigned rows_B = transpose_second ? B.cols() : B.rows();
00782     unsigned cols_B = transpose_second ? B.rows() : B.cols();
00783
00784     if ((rows_A != rows_B) || (cols_A != cols_B)) throw std::runtime_error("Unmatching matrix dimensions
for mult_hadamard");
00785
00786     Matrix<T> C(static_cast<T>(0), rows_A, cols_A);
00787
00788     for (unsigned i = 0; i < rows_A; i++)
00789         for (unsigned j = 0; j < cols_A; j++)
00790             C(i,j) += (transpose_first ? cconj(A(j,i)) : A(i,j)) *
00791                 (transpose_second ? cconj(B(j,i)) : B(i,j));
00792
00793     return C;
00794 }
00795
00810 template<typename T, bool transpose_first = false, bool transpose_second = false>
00811 Matrix<T> add(const Matrix<T>& A, const Matrix<T>& B) {
00812     // Adjust dimensions based on transpositions
00813     unsigned rows_A = transpose_first ? A.cols() : A.rows();
00814     unsigned cols_A = transpose_first ? A.rows() : A.cols();
00815     unsigned rows_B = transpose_second ? B.cols() : B.rows();
00816     unsigned cols_B = transpose_second ? B.rows() : B.cols();
00817
00818     if ((rows_A != rows_B) || (cols_A != cols_B)) throw std::runtime_error("Unmatching matrix dimensions
for add");
00819
00820     Matrix<T> C(static_cast<T>(0), rows_A, cols_A);
00821
00822     for (unsigned i = 0; i < rows_A; i++)
00823         for (unsigned j = 0; j < cols_A; j++)
00824             C(i,j) += (transpose_first ? cconj(A(j,i)) : A(i,j)) +
00825                 (transpose_second ? cconj(B(j,i)) : B(i,j));
00826
00827     return C;
00828 }
00829
00844 template<typename T, bool transpose_first = false, bool transpose_second = false>
00845 Matrix<T> subtract(const Matrix<T>& A, const Matrix<T>& B) {
00846     // Adjust dimensions based on transpositions
00847     unsigned rows_A = transpose_first ? A.cols() : A.rows();
00848     unsigned cols_A = transpose_first ? A.rows() : A.cols();
00849     unsigned rows_B = transpose_second ? B.cols() : B.rows();
00850     unsigned cols_B = transpose_second ? B.rows() : B.cols();
00851
00852     if ((rows_A != rows_B) || (cols_A != cols_B)) throw std::runtime_error("Unmatching matrix dimensions
for subtract");
00853
00854     Matrix<T> C(static_cast<T>(0), rows_A, cols_A);
00855

```

```

00856     for (unsigned i = 0; i < rows_A; i++)
00857         for (unsigned j = 0; j < cols_A; j++)
00858             C(i,j) += (transpose_first ? cconj(A(j,i)) : A(i,j)) -
00859                        (transpose_second ? cconj(B(j,i)) : B(i,j));
00860
00861     return C;
00862 }
00863
00877 template<typename T, bool transpose_matrix = false>
00878 std::vector<T> mult(const Matrix<T>& A, const std::vector<T>& v) {
00879     // Adjust dimensions based on transpositions
00880     unsigned rows_A = transpose_matrix ? A.cols() : A.rows();
00881     unsigned cols_A = transpose_matrix ? A.rows() : A.cols();
00882
00883     if (cols_A != v.size()) throw std::runtime_error("Unmatching matrix dimensions for mult");
00884
00885     std::vector<T> u(rows_A, static_cast<T>(0));
00886     for (unsigned r = 0; r < rows_A; r++)
00887         for (unsigned c = 0; c < cols_A; c++)
00888             u[r] += v[c] * (transpose_matrix ? cconj(A(c,r)) : A(r,c));
00889
00890     return u;
00891 }
00892
00906 template<typename T, bool transpose_matrix = false>
00907 std::vector<T> mult(const std::vector<T>& v, const Matrix<T>& A) {
00908     // Adjust dimensions based on transpositions
00909     unsigned rows_A = transpose_matrix ? A.cols() : A.rows();
00910     unsigned cols_A = transpose_matrix ? A.rows() : A.cols();
00911
00912     if (rows_A != v.size()) throw std::runtime_error("Unmatching matrix dimensions for mult");
00913
00914     std::vector<T> u(cols_A, static_cast<T>(0));
00915     for (unsigned c = 0; c < cols_A; c++)
00916         for (unsigned r = 0; r < rows_A; r++)
00917             u[c] += v[r] * (transpose_matrix ? cconj(A(c,r)) : A(r,c));
00918
00919     return u;
00920 }
00921
00927 template<typename T>
00928 Matrix<T> add(const Matrix<T>& A, T s) {
00929     Matrix<T> B(A.rows(), A.cols());
00930     for (unsigned i = 0; i < A.numel(); i++)
00931         B(i) = A(i) + s;
00932     return B;
00933 }
00934
00940 template<typename T>
00941 Matrix<T> subtract(const Matrix<T>& A, T s) {
00942     Matrix<T> B(A.rows(), A.cols());
00943     for (unsigned i = 0; i < A.numel(); i++)
00944         B(i) = A(i) - s;
00945     return B;
00946 }
00947
00953 template<typename T>
00954 Matrix<T> mult(const Matrix<T>& A, T s) {
00955     Matrix<T> B(A.rows(), A.cols());
00956     for (unsigned i = 0; i < A.numel(); i++)
00957         B(i) = A(i) * s;
00958     return B;
00959 }
00960
00966 template<typename T>
00967 Matrix<T> div(const Matrix<T>& A, T s) {
00968     Matrix<T> B(A.rows(), A.cols());
00969     for (unsigned i = 0; i < A.numel(); i++)
00970         B(i) = A(i) / s;
00971     return B;
00972 }
00973
00979 template<typename T>
00980 std::ostream& operator<<(std::ostream& os, const Matrix<T>& A) {
00981     for (unsigned row = 0; row < A.rows(); row++) {
00982         for (unsigned col = 0; col < A.cols(); col++)
00983             os << A(row,col) << " ";
00984         if (row < static_cast<unsigned>(A.rows()-1)) os << std::endl;
00985     }
00986     return os;
00987 }
00988
00993 template<typename T>
00994 inline Matrix<T> operator+(const Matrix<T>& A, const Matrix<T>& B) {
00995     return add(A,B);
00996 }
00997

```

```

01002 template<typename T>
01003 inline Matrix<T> operator-(const Matrix<T>& A, const Matrix<T>& B) {
01004     return subtract(A,B);
01005 }
01006
01012 template<typename T>
01013 inline Matrix<T> operator^(const Matrix<T>& A, const Matrix<T>& B) {
01014     return mult_hadamard(A,B);
01015 }
01016
01021 template<typename T>
01022 inline Matrix<T> operator*(const Matrix<T>& A, const Matrix<T>& B) {
01023     return mult(A,B);
01024 }
01025
01030 template<typename T>
01031 inline std::vector<T> operator*(const Matrix<T>& A, const std::vector<T>& v) {
01032     return mult(A,v);
01033 }
01034
01039 template<typename T>
01040 inline std::vector<T> operator*(const std::vector<T>& v, const Matrix<T>& A) {
01041     return mult(v,A);
01042 }
01043
01048 template<typename T>
01049 inline Matrix<T> operator+(const Matrix<T>& A, T s) {
01050     return add(A,s);
01051 }
01052
01057 template<typename T>
01058 inline Matrix<T> operator-(const Matrix<T>& A, T s) {
01059     return subtract(A,s);
01060 }
01061
01066 template<typename T>
01067 inline Matrix<T> operator*(const Matrix<T>& A, T s) {
01068     return mult(A,s);
01069 }
01070
01075 template<typename T>
01076 inline Matrix<T> operator/(const Matrix<T>& A, T s) {
01077     return div(A,s);
01078 }
01079
01083 template<typename T>
01084 inline Matrix<T> operator+(T s, const Matrix<T>& A) {
01085     return add(A,s);
01086 }
01087
01092 template<typename T>
01093 inline Matrix<T> operator*(T s, const Matrix<T>& A) {
01094     return mult(A,s);
01095 }
01096
01101 template<typename T>
01102 inline Matrix<T>& operator+=(Matrix<T>& A, const Matrix<T>& B) {
01103     return A.add(B);
01104 }
01105
01110 template<typename T>
01111 inline Matrix<T>& operator-=(Matrix<T>& A, const Matrix<T>& B) {
01112     return A.subtract(B);
01113 }
01114
01119 template<typename T>
01120 inline Matrix<T>& operator*=(Matrix<T>& A, const Matrix<T>& B) {
01121     A = mult(A,B);
01122     return A;
01123 }
01124
01130 template<typename T>
01131 inline Matrix<T>& operator^=(Matrix<T>& A, const Matrix<T>& B) {
01132     return A.mult_hadamard(B);
01133 }
01134
01139 template<typename T>
01140 inline Matrix<T>& operator+=(Matrix<T>& A, T s) {
01141     return A.add(s);
01142 }
01143
01148 template<typename T>
01149 inline Matrix<T>& operator-=(Matrix<T>& A, T s) {
01150     return A.subtract(s);
01151 }
01152
01157 template<typename T>

```

```

01158 inline Matrix<T>& operator*=(Matrix<T>& A, T s) {
01159     return A.mult(s);
01160 }
01161
01166 template<typename T>
01167 inline Matrix<T>& operator/=(Matrix<T>& A, T s) {
01168     return A.div(s);
01169 }
01170
01175 template<typename T>
01176 inline bool operator==(const Matrix<T>& A, const Matrix<T>& b) {
01177     return A.isequal(b);
01178 }
01179
01184 template<typename T>
01185 inline bool operator!=(const Matrix<T>& A, const Matrix<T>& b) {
01186     return !(A.isequal(b));
01187 }
01188
01194 template<typename T>
01195 Matrix<T> kron(const Matrix<T>& A, const Matrix<T>& B) {
01196     const unsigned rows_A = A.rows();
01197     const unsigned cols_A = A.cols();
01198     const unsigned rows_B = B.rows();
01199     const unsigned cols_B = B.cols();
01200
01201     unsigned rows_C = rows_A * rows_B;
01202     unsigned cols_C = cols_A * cols_B;
01203
01204     Matrix<T> C(rows_C, cols_C);
01205
01206     for (unsigned i = 0; i < rows_A; i++)
01207         for (unsigned j = 0; j < cols_A; j++)
01208             for (unsigned k = 0; k < rows_B; k++)
01209                 for (unsigned l = 0; l < cols_B; l++)
01210                     C(i+rows_B * k, j+cols_B * l) = A(i, j) * B(k, l);
01211
01212     return C;
01213 }
01214
01222 template<typename T>
01223 Matrix<T> adj(const Matrix<T>& A) {
01224     if (!A.issquare()) throw std::runtime_error("Input matrix is not square");
01225
01226     Matrix<T> B(A.rows(), A.cols());
01227     if (A.rows() == 1) {
01228         B(0) = 1.0;
01229     } else {
01230         for (unsigned i = 0; i < A.rows(); i++) {
01231             for (unsigned j = 0; j < A.cols(); j++) {
01232                 T sgn = ((i + j) % 2 == 0) ? 1.0 : -1.0;
01233                 B(j, i) = sgn * det(cofactor(A, i, j));
01234             }
01235         }
01236     }
01237     return B;
01238 }
01239
01252 template<typename T>
01253 Matrix<T> cofactor(const Matrix<T>& A, unsigned p, unsigned q) {
01254     if (!A.issquare()) throw std::runtime_error("Input matrix is not square");
01255     if (!(p < A.rows())) throw std::out_of_range("Row index out of range");
01256     if (!(q < A.cols())) throw std::out_of_range("Column index out of range");
01257     if (A.cols() < 2) throw std::runtime_error("Cofactor calculation requested for matrix with less than
2 rows");
01258
01259     Matrix<T> c(A.rows()-1, A.cols()-1);
01260     unsigned i = 0;
01261     unsigned j = 0;
01262
01263     for (unsigned row = 0; row < A.rows(); row++) {
01264         if (row != p) {
01265             for (unsigned col = 0; col < A.cols(); col++)
01266                 if (col != q) c(i, j++) = A(row, col);
01267             j = 0;
01268             i++;
01269         }
01270     }
01271
01272     return c;
01273 }
01274
01286 template<typename T>
01287 T det_lu(const Matrix<T>& A) {
01288     if (!A.issquare()) throw std::runtime_error("Input matrix is not square");
01289
01290     // LU decomposition with pivoting

```

```

01291     auto res = lup(A);
01292
01293     // Determinants of LU
01294     T detLU = static_cast<T>(1);
01295
01296     for (unsigned i = 0; i < res.L.rows(); i++)
01297         detLU *= res.L(i,i) * res.U(i,i);
01298
01299     // Determinant of P
01300     unsigned len = res.P.size();
01301     T detP = 1;
01302
01303     std::vector<unsigned> p(res.P);
01304     std::vector<unsigned> q;
01305     q.resize(len);
01306
01307     for (unsigned i = 0; i < len; i++)
01308         q[p[i]] = i;
01309
01310     for (unsigned i = 0; i < len; i++) {
01311         unsigned j = p[i];
01312         unsigned k = q[i];
01313         if (j != i) {
01314             p[k] = p[i];
01315             q[j] = q[i];
01316             detP = - detP;
01317         }
01318     }
01319
01320     return detLU * detP;
01321 }
01322
01331 template<typename T>
01332 T det(const Matrix<T>& A) {
01333     if (! A.issquare()) throw std::runtime_error("Input matrix is not square");
01334
01335     if (A.rows() == 1)
01336         return A(0,0);
01337     else if (A.rows() == 2)
01338         return A(0,0)*A(1,1) - A(0,1)*A(1,0);
01339     else if (A.rows() == 3)
01340         return A(0,0)*(A(1,1)*A(2,2) - A(1,2)*A(2,1)) -
01341             A(0,1)*(A(1,0)*A(2,2) - A(1,2)*A(2,0)) +
01342             A(0,2)*(A(1,0)*A(2,1) - A(1,1)*A(2,0));
01343     else
01344         return det_lu(A);
01345 }
01346
01355 template<typename T>
01356 LU_result<T> lu(const Matrix<T>& A) {
01357     const unsigned M = A.rows();
01358     const unsigned N = A.cols();
01359
01360     LU_result<T> res;
01361     res.L = eye<T>(M);
01362     res.U = Matrix<T>(A);
01363
01364     // aliases
01365     auto& L = res.L;
01366     auto& U = res.U;
01367
01368     if (A.numel() == 0)
01369         return res;
01370
01371     for (unsigned k = 0; k < M-1; k++) {
01372         for (unsigned i = k+1; i < M; i++) {
01373             L(i,k) = U(i,k) / U(k,k);
01374             for (unsigned l = k+1; l < N; l++) {
01375                 U(i,l) -= L(i,k) * U(k,l);
01376             }
01377         }
01378     }
01379
01380     for (unsigned col = 0; col < N; col++)
01381         for (unsigned row = col+1; row < M; row++)
01382             U(row,col) = 0;
01383
01384     return res;
01385 }
01386
01400 template<typename T>
01401 LUP_result<T> lup(const Matrix<T>& A) {
01402     const unsigned M = A.rows();
01403     const unsigned N = A.cols();
01404
01405     // Initialize L, U, and PP
01406     LUP_result<T> res;

```

```

01407
01408     if (A.numel() == 0)
01409         return res;
01410
01411     res.L = eye<T>(M);
01412     res.U = Matrix<T>(A);
01413     std::vector<unsigned> PP;
01414
01415     // aliases
01416     auto& L = res.L;
01417     auto& U = res.U;
01418
01419     PP.resize(N);
01420     for (unsigned i = 0; i < N; i++)
01421         PP[i] = i;
01422
01423     for (unsigned k = 0; k < M-1; k++) {
01424         // Find the column with the largest absolute value in the current row
01425         auto max_col_value = std::abs(U(k,k));
01426         unsigned max_col_index = k;
01427         for (unsigned l = k+1; l < N; l++) {
01428             auto val = std::abs(U(k,l));
01429             if (val > max_col_value) {
01430                 max_col_value = val;
01431                 max_col_index = l;
01432             }
01433         }
01434
01435         // Swap columns k and max_col_index in U and update P
01436         if (max_col_index != k) {
01437             U.swap_cols(k, max_col_index); // TODO: This could be reworked to avoid column swap in U during
every iteration by:
01438                                     //      1. using PP[k] for column indexing across iterations
01439                                     //      2. doing just one permutation of U at the end
01440             std::swap(PP[k], PP[max_col_index]);
01441         }
01442
01443         // Update L and U
01444         for (unsigned i = k+1; i < M; i++) {
01445             L(i,k) = U(i,k) / U(k,k);
01446             for (unsigned l = k+1; l < N; l++) {
01447                 U(i,l) -= L(i,k) * U(k,l);
01448             }
01449         }
01450     }
01451
01452     // Set elements in lower triangular part of U to zero
01453     for (unsigned col = 0; col < N; col++)
01454         for (unsigned row = col+1; row < M; row++)
01455             U(row,col) = 0;
01456
01457     // Transpose indices in permutation vector
01458     res.P.resize(N);
01459     for (unsigned i = 0; i < N; i++)
01460         res.P[PP[i]] = i;
01461
01462     return res;
01463 }
01464
01475 template<typename T>
01476 Matrix<T> inv_gauss_jordan(const Matrix<T>& A) {
01477     if (! A.issquare()) throw std::runtime_error("Input matrix is not square");
01478
01479     const unsigned N = A.rows();
01480     Matrix<T> AA(A);
01481     auto IA = eye<T>(N);
01482
01483     bool found_nonzero;
01484     for (unsigned j = 0; j < N; j++) {
01485         found_nonzero = false;
01486         for (unsigned i = j; i < N; i++) {
01487             if (AA(i,j) != static_cast<T>(0)) {
01488                 found_nonzero = true;
01489                 for (unsigned k = 0; k < N; k++) {
01490                     std::swap(AA(j,k), AA(i,k));
01491                     std::swap(IA(j,k), IA(i,k));
01492                 }
01493                 if (AA(j,j) != static_cast<T>(1)) {
01494                     T s = static_cast<T>(1) / AA(j,j);
01495                     for (unsigned k = 0; k < N; k++) {
01496                         AA(j,k) *= s;
01497                         IA(j,k) *= s;
01498                     }
01499                 }
01500                 for (unsigned l = 0; l < N; l++) {
01501                     if (l != j) {
01502                         T s = AA(l,j);

```



```

01503         for (unsigned k = 0; k < N; k++) {
01504             AA(l,k) -= s * AA(j,k);
01505             IA(l,k) -= s * IA(j,k);
01506         }
01507     }
01508 }
01509 }
01510     break;
01511 }
01512 // if a row full of zeros is found, the input matrix was singular
01513 if (!found_nonzero) throw singular_matrix_exception("Singular matrix in inv_gauss_jordan");
01514 }
01515 return IA;
01516 }
01517
01528 template<typename T>
01529 Matrix<T> inv_tril(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 (unsigned i = 0; i < N; i++) {
01537         if (A(i,i) == 0.0) throw singular_matrix_exception("Division by zero in inv_tril");
01538
01539         IA(i,i) = static_cast<T>(1.0) / A(i,i);
01540         for (unsigned j = 0; j < i; j++) {
01541             T s = 0.0;
01542             for (unsigned k = j; k < i; 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
01561 template<typename T>
01562 Matrix<T> inv_triu(const Matrix<T>& A) {
01563     if (!A.issquare()) throw std::runtime_error("Input matrix is not square");
01564
01565     const unsigned N = A.rows();
01566
01567     auto IA = zeros<T>(N);
01568
01569     for (int i = N - 1; i >= 0; i--) {
01570         if (A(i,i) == 0.0) throw singular_matrix_exception("Division by zero in inv_triu");
01571
01572         IA(i,i) = static_cast<T>(1.0) / A(i,i);
01573         for (int j = N - 1; j > i; j--) {
01574             T s = 0.0;
01575             for (int k = i + 1; k <= j; k++)
01576                 s += A(i,k) * IA(k,j);
01577             IA(i,j) = -s * IA(i,i);
01578         }
01579     }
01580
01581     return IA;
01582 }
01583
01596 template<typename T>
01597 Matrix<T> inv_posdef(const Matrix<T>& A) {
01598     auto L = cholinv(A);
01599     return mult<T,true,false>(L,L);
01600 }
01601
01612 template<typename T>
01613 Matrix<T> inv_square(const Matrix<T>& A) {
01614     if (!A.issquare()) throw std::runtime_error("Input matrix is not square");
01615
01616     // LU decomposition with pivoting
01617     auto LU = lup(A);
01618     auto IL = inv_tril(LU.L);
01619     auto IU = inv_triu(LU.U);
01620
01621     return permute_rows(IU * IL, LU.P);
01622 }
01623
01634 template<typename T>
01635 Matrix<T> inv(const Matrix<T>& A) {
01636     if (!A.issquare()) throw std::runtime_error("Input matrix is not square");
01637
01638     if (A.numel() == 0) {
01639         return Matrix<T>();
01640     } else if (A.rows() < 4) {
01641         T d = det(A);

```

```

01642
01643     if (d == 0.0) throw singular_matrix_exception("Singular matrix in inv");
01644
01645     Matrix<T> IA(A.rows(), A.cols());
01646     T invdet = static_cast<T>(1.0) / d;
01647
01648     if (A.rows() == 1) {
01649         IA(0,0) = invdet;
01650     } else if (A.rows() == 2) {
01651         IA(0,0) = A(1,1) * invdet;
01652         IA(0,1) = - A(0,1) * invdet;
01653         IA(1,0) = - A(1,0) * invdet;
01654         IA(1,1) = A(0,0) * invdet;
01655     } else if (A.rows() == 3) {
01656         IA(0,0) = (A(1,1)*A(2,2) - A(2,1)*A(1,2)) * invdet;
01657         IA(0,1) = (A(0,2)*A(2,1) - A(0,1)*A(2,2)) * invdet;
01658         IA(0,2) = (A(0,1)*A(1,2) - A(0,2)*A(1,1)) * invdet;
01659         IA(1,0) = (A(1,2)*A(2,0) - A(1,0)*A(2,2)) * invdet;
01660         IA(1,1) = (A(0,0)*A(2,2) - A(0,2)*A(2,0)) * invdet;
01661         IA(1,2) = (A(1,0)*A(0,2) - A(0,0)*A(1,2)) * invdet;
01662         IA(2,0) = (A(1,0)*A(2,1) - A(2,0)*A(1,1)) * invdet;
01663         IA(2,1) = (A(2,0)*A(0,1) - A(0,0)*A(2,1)) * invdet;
01664         IA(2,2) = (A(0,0)*A(1,1) - A(1,0)*A(0,1)) * invdet;
01665     }
01666
01667     return IA;
01668 } else {
01669     return inv_square(A);
01670 }
01671 }
01672
01680 template<typename T>
01681 Matrix<T> pinv(const Matrix<T>& A) {
01682     if (A.rows() > A.cols()) {
01683         auto AH_A = mult<T,true,false>(A, A);
01684         auto Linv = inv_posdef(AH_A);
01685         return mult<T,false,true>(Linv, A);
01686     } else {
01687         auto AA_H = mult<T,false,true>(A, A);
01688         auto Linv = inv_posdef(AA_H);
01689         return mult<T,true,false>(A, Linv);
01690     }
01691 }
01692
01698 template<typename T>
01699 T trace(const Matrix<T>& A) {
01700     T t = static_cast<T>(0);
01701     for (int i = 0; i < A.rows(); i++)
01702         t += A(i,i);
01703     return t;
01704 }
01705
01713 template<typename T>
01714 double cond(const Matrix<T>& A) {
01715     try {
01716         auto A_inv = inv(A);
01717         return norm_fro(A) * norm_fro(A_inv);
01718     } catch (singular_matrix_exception& e) {
01719         return std::numeric_limits<double>::max();
01720     }
01721 }
01722
01740 template<typename T, bool is_upper = false>
01741 Matrix<T> chol(const Matrix<T>& A) {
01742     if (!A.issquare()) throw std::runtime_error("Input matrix is not square");
01743
01744     const unsigned N = A.rows();
01745
01746     // Calculate lower or upper triangular, depending on template parameter.
01747     // Calculation is the same - the difference is in transposed row and column indexing.
01748     Matrix<T> C = is_upper ? triu(A) : tril(A);
01749
01750     for (unsigned j = 0; j < N; j++) {
01751         if (C(j,j) == 0.0) throw singular_matrix_exception("Singular matrix in chol");
01752
01753         C(j,j) = std::sqrt(C(j,j));
01754
01755         for (unsigned k = j+1; k < N; k++)
01756             if (is_upper)
01757                 C(j,k) /= C(j,j);
01758             else
01759                 C(k,j) /= C(j,j);
01760
01761         for (unsigned k = j+1; k < N; k++)
01762             for (unsigned i = k; i < N; i++)
01763                 if (is_upper)
01764                     C(k,i) -= C(j,i) * cconj(C(j,k));

```

```

01765         else
01766             C(i,k) -= C(i,j) * cconj(C(k,j));
01767     }
01768
01769     return C;
01770 }
01771
01772 template<typename T>
01773 Matrix<T> cholinv(const Matrix<T>& A) {
01774     if (! A.issquare()) throw std::runtime_error("Input matrix is not square");
01775
01776     const unsigned N = A.rows();
01777     Matrix<T> L(A);
01778     auto Linv = eye<T>(N);
01779
01780     for (unsigned j = 0; j < N; j++) {
01781         if (L(j,j) == 0.0) throw singular_matrix_exception("Singular matrix in cholinv");
01782
01783         L(j,j) = 1.0 / std::sqrt(L(j,j));
01784
01785         for (unsigned k = j+1; k < N; k++)
01786             L(k,j) = L(k,j) * L(j,j);
01787
01788         for (unsigned k = j+1; k < N; k++)
01789             for (unsigned i = k; i < N; i++)
01790                 L(i,k) = L(i,k) - L(i,j) * cconj(L(k,j));
01791     }
01792
01793     for (unsigned k = 0; k < N; k++) {
01794         for (unsigned i = k; i < N; i++) {
01795             Linv(i,k) = Linv(i,k) * L(i,i);
01796             for (unsigned j = i+1; j < N; j++)
01797                 Linv(j,k) = Linv(j,k) - L(j,i) * Linv(i,k);
01798         }
01799     }
01800
01801     return Linv;
01802 }
01803
01804 template<typename T>
01805 LDL_result<T> ldl(const Matrix<T>& A) {
01806     if (! A.issquare()) throw std::runtime_error("Input matrix is not square");
01807
01808     const unsigned N = A.rows();
01809
01810     LDL_result<T> res;
01811
01812     // aliases
01813     auto& L = res.L;
01814     auto& d = res.d;
01815
01816     L = eye<T>(N);
01817     d.resize(N);
01818
01819     for (unsigned m = 0; m < N; m++) {
01820         d[m] = A(m,m);
01821
01822         for (unsigned k = 0; k < m; k++)
01823             d[m] -= L(m,k) * cconj(L(m,k)) * d[k];
01824
01825         if (d[m] == 0.0) throw singular_matrix_exception("Singular matrix in ldl");
01826
01827         for (unsigned n = m+1; n < N; n++) {
01828             L(n,m) = A(n,m);
01829             for (unsigned k = 0; k < m; k++)
01830                 L(n,m) -= L(n,k) * cconj(L(m,k)) * d[k];
01831             L(n,m) /= d[m];
01832         }
01833     }
01834
01835     return res;
01836 }
01837
01838 template<typename T>
01839 QR_result<T> qr_red_gs(const Matrix<T>& A) {
01840     const int rows = A.rows();
01841     const int cols = A.cols();
01842
01843     QR_result<T> res;
01844
01845     //aliases
01846     auto& Q = res.Q;
01847     auto& R = res.R;
01848
01849     Q = zeros<T>(rows, cols);
01850     R = zeros<T>(cols, cols);
01851

```

```

01887     for (int c = 0; c < cols; c++) {
01888         Matrix<T> v = A.get_submatrix(0, rows-1, c, c);
01889         for (int r = 0; r < c; r++) {
01890             for (int k = 0; k < rows; k++)
01891                 R(r,c) = R(r,c) + cconj(Q(k,r)) * A(k,c);
01892             for (int k = 0; k < rows; k++)
01893                 v(k) = v(k) - R(r,c) * Q(k,r);
01894         }
01895
01896         R(c,c) = static_cast<T>(norm_fro(v));
01897
01898         if (R(c,c) == 0.0) throw singular_matrix_exception("Division by 0 in QR GS");
01899
01900         for (int k = 0; k < rows; k++)
01901             Q(k,c) = v(k) / R(c,c);
01902     }
01903
01904     return res;
01905 }
01906
01914 template<typename T>
01915 Matrix<T> householder_reflection(const Matrix<T>& a) {
01916     if (a.cols() != 1) throw std::runtime_error("Input not a column vector");
01917
01918     static const T ISQRT2 = static_cast<T>(0.707106781186547);
01919
01920     Matrix<T> v(a);
01921     v(0) += csign(v(0)) * norm_fro(v);
01922     auto vn = norm_fro(v) * ISQRT2;
01923     for (unsigned i = 0; i < v.numel(); i++)
01924         v(i) /= vn;
01925     return v;
01926 }
01927
01939 template<typename T>
01940 QR_result<T> qr_householder(const Matrix<T>& A, bool calculate_Q = true) {
01941     const unsigned rows = A.rows();
01942     const unsigned cols = A.cols();
01943
01944     QR_result<T> res;
01945
01946     //aliases
01947     auto& Q = res.Q;
01948     auto& R = res.R;
01949
01950     R = Matrix<T>(A);
01951
01952     if (calculate_Q)
01953         Q = eye<T>(rows);
01954
01955     const unsigned N = (rows > cols) ? cols : rows;
01956
01957     for (unsigned j = 0; j < N; j++) {
01958         auto v = householder_reflection(R.get_submatrix(j, rows-1, j, j));
01959
01960         auto R1 = R.get_submatrix(j, rows-1, j, cols-1);
01961         auto WR = v * mult<T,true,false>(v, R1);
01962         for (unsigned c = j; c < cols; c++)
01963             for (unsigned r = j; r < rows; r++)
01964                 R(r,c) -= WR(r-j,c-j);
01965
01966         if (calculate_Q) {
01967             auto Q1 = Q.get_submatrix(0, rows-1, j, rows-1);
01968             auto WQ = mult<T,false,true>(Q1 * v, v);
01969             for (unsigned c = j; c < rows; c++)
01970                 for (unsigned r = 0; r < rows; r++)
01971                     Q(r,c) -= WQ(r,c-j);
01972         }
01973     }
01974
01975     for (unsigned col = 0; col < R.cols(); col++)
01976         for (unsigned row = col+1; row < R.rows(); row++)
01977             R(row,col) = 0;
01978
01979     return res;
01980 }
01981
01992 template<typename T>
01993 inline QR_result<T> qr(const Matrix<T>& A, bool calculate_Q = true) {
01994     return qr_householder(A, calculate_Q);
01995 }
01996
02007 template<typename T>
02008 Hessenberg_result<T> hessenberg(const Matrix<T>& A, bool calculate_Q = true) {
02009     if (! A.isquare()) throw std::runtime_error("Input matrix is not square");
02010
02011     Hessenberg_result<T> res;

```

```

02012
02013 // aliases
02014 auto& H = res.H;
02015 auto& Q = res.Q;
02016
02017 const unsigned N = A.rows();
02018 H = Matrix<T>(A);
02019
02020 if (calculate_Q)
02021     Q = eye<T>(N);
02022
02023 for (unsigned k = 1; k < N-1; k++) {
02024     auto v = householder_reflection(H.get_submatrix(k, N-1, k-1, k-1));
02025
02026     auto H1 = H.get_submatrix(k, N-1, 0, N-1);
02027     auto W1 = v * mult<T,true,false>(v, H1);
02028     for (unsigned c = 0; c < N; c++)
02029         for (unsigned r = k; r < N; r++)
02030             H(r,c) -= W1(r-k,c);
02031
02032     auto H2 = H.get_submatrix(0, N-1, k, N-1);
02033     auto W2 = mult<T,false,true>(H2 * v, v);
02034     for (unsigned c = k; c < N; c++)
02035         for (unsigned r = 0; r < N; r++)
02036             H(r,c) -= W2(r,c-k);
02037
02038     if (calculate_Q) {
02039         auto Q1 = Q.get_submatrix(0, N-1, k, N-1);
02040         auto W3 = mult<T,false,true>(Q1 * v, v);
02041         for (unsigned c = k; c < N; c++)
02042             for (unsigned r = 0; r < N; r++)
02043                 Q(r,c) -= W3(r,c-k);
02044     }
02045 }
02046
02047 for (unsigned row = 2; row < N; row++)
02048     for (unsigned col = 0; col < row-2; col++)
02049         H(row,col) = static_cast<T>(0);
02050
02051 return res;
02052 }
02053
02062 template<typename T>
02063 std::complex<T> wilkinson_shift(const Matrix<std::complex<T>& H, T tol = 1e-10) {
02064     if (! H.issquare()) throw std::runtime_error("Input matrix is not square");
02065
02066     const unsigned n = H.rows();
02067     std::complex<T> mu;
02068
02069     if (std::abs(H(n-1,n-2)) < tol) {
02070         mu = H(n-2,n-2);
02071     } else {
02072         auto trA = H(n-2,n-2) + H(n-1,n-1);
02073         auto detA = H(n-2,n-2) * H(n-1,n-1) - H(n-2, n-1) * H(n-1, n-2);
02074         mu = (trA + std::sqrt(trA*trA - 4.0*detA)) / 2.0;
02075     }
02076
02077     return mu;
02078 }
02079
02091 template<typename T>
02092 Eigenvalues_result<T> eigenvalues(const Matrix<std::complex<T>& A, T tol = 1e-12, unsigned max_iter =
100) {
02093     if (! A.issquare()) throw std::runtime_error("Input matrix is not square");
02094
02095     const unsigned N = A.rows();
02096     Matrix<std::complex<T>> H;
02097     bool success = false;
02098
02099     QR_result<std::complex<T>> QR;
02100
02101     // aliases
02102     auto& Q = QR.Q;
02103     auto& R = QR.R;
02104
02105     // Transfer A to Hessenberg form to improve convergence (skip calculation of Q)
02106     H = hessenberg(A, false).H;
02107
02108     for (unsigned iter = 0; iter < max_iter; iter++) {
02109         auto mu = wilkinson_shift(H, tol);
02110
02111         // subtract mu from diagonal
02112         for (unsigned n = 0; n < N; n++)
02113             H(n,n) -= mu;
02114
02115         // QR factorization with shifted H
02116         QR = qr(H);

```

```

02117     H = R * Q;
02118
02119     // add back mu to diagonal
02120     for (unsigned n = 0; n < N; n++)
02121         H(n,n) += mu;
02122
02123     // Check for convergence
02124     if (std::abs(H(N-2,N-1)) <= tol) {
02125         success = true;
02126         break;
02127     }
02128 }
02129
02130 Eigenvalues_result<T> res;
02131 res.eig = diag(H);
02132 res.err = std::abs(H(N-2,N-1));
02133 res.converged = success;
02134
02135 return res;
02136 }
02137
02147 template<typename T>
02148 Eigenvalues_result<T> eigenvalues(const Matrix<T>& A, T tol = 1e-12, unsigned max_iter = 100) {
02149     auto A_cplx = make_complex(A);
02150     return eigenvalues(A_cplx, tol, max_iter);
02151 }
02152
02167 template<typename T>
02168 Matrix<T> solve_triu(const Matrix<T>& U, const Matrix<T>& B) {
02169     if (! U.isquare()) throw std::runtime_error("Input matrix is not square");
02170     if (U.rows() != B.rows()) throw std::runtime_error("Unmatching matrix dimensions for solve");
02171
02172     const unsigned N = U.rows();
02173     const unsigned M = B.cols();
02174
02175     if (U.numel() == 0)
02176         return Matrix<T>();
02177
02178     Matrix<T> X(B);
02179
02180     for (unsigned m = 0; m < M; m++) {
02181         // backwards substitution for each column of B
02182         for (int n = N-1; n >= 0; n--) {
02183             for (unsigned j = n + 1; j < N; j++)
02184                 X(n,m) -= U(n,j) * X(j,m);
02185
02186             if (U(n,n) == 0.0) throw singular_matrix_exception("Singular matrix in solve_triu");
02187
02188             X(n,m) /= U(n,n);
02189         }
02190     }
02191
02192     return X;
02193 }
02194
02209 template<typename T>
02210 Matrix<T> solve_tril(const Matrix<T>& L, const Matrix<T>& B) {
02211     if (! L.isquare()) throw std::runtime_error("Input matrix is not square");
02212     if (L.rows() != B.rows()) throw std::runtime_error("Unmatching matrix dimensions for solve");
02213
02214     const unsigned N = L.rows();
02215     const unsigned M = B.cols();
02216
02217     if (L.numel() == 0)
02218         return Matrix<T>();
02219
02220     Matrix<T> X(B);
02221
02222     for (unsigned m = 0; m < M; m++) {
02223         // forwards substitution for each column of B
02224         for (unsigned n = 0; n < N; n++) {
02225             for (unsigned j = 0; j < n; j++)
02226                 X(n,m) -= L(n,j) * X(j,m);
02227
02228             if (L(n,n) == 0.0) throw singular_matrix_exception("Singular matrix in solve_tril");
02229
02230             X(n,m) /= L(n,n);
02231         }
02232     }
02233
02234     return X;
02235 }
02236
02251 template<typename T>
02252 Matrix<T> solve_square(const Matrix<T>& A, const Matrix<T>& B) {
02253     if (! A.isquare()) throw std::runtime_error("Input matrix is not square");
02254     if (A.rows() != B.rows()) throw std::runtime_error("Unmatching matrix dimensions for solve");

```

```

02255
02256     if (A.numel() == 0)
02257         return Matrix<T>();
02258
02259     Matrix<T> L;
02260     Matrix<T> U;
02261     std::vector<unsigned> P;
02262
02263     // LU decomposition with pivoting
02264     auto lup_res = lup(A);
02265
02266     auto y = solve_tril(lup_res.L, B);
02267     auto x = solve_triu(lup_res.U, y);
02268
02269     return permute_rows(x, lup_res.P);
02270 }
02271
02286 template<typename T>
02287 Matrix<T> solve_posdef(const Matrix<T>& A, const Matrix<T>& B) {
02288     if (!A.issquare()) throw std::runtime_error("Input matrix is not square");
02289     if (A.rows() != B.rows()) throw std::runtime_error("Unmatching matrix dimensions for solve");
02290
02291     if (A.numel() == 0)
02292         return Matrix<T>();
02293
02294     // LU decomposition with pivoting
02295     auto L = chol(A);
02296
02297     auto Y = solve_tril(L, B);
02298     return solve_triu(L.ctranspose(), Y);
02299 }
02300
02305 template<typename T>
02306 class Matrix {
02307     public:
02312         Matrix();
02313
02318         Matrix(unsigned size);
02319
02324         Matrix(unsigned nrows, unsigned ncols);
02325
02330         Matrix(T x, unsigned nrows, unsigned ncols);
02331
02337         Matrix(const T* array, unsigned nrows, unsigned ncols);
02338
02346         Matrix(const std::vector<T>& vec, unsigned nrows, unsigned ncols);
02347
02355         Matrix(std::initializer_list<T> init_list, unsigned nrows, unsigned ncols);
02356
02359         Matrix(const Matrix &);
02360
02363         virtual ~Matrix();
02364
02372         Matrix<T> get_submatrix(unsigned row_first, unsigned row_last, unsigned col_first, unsigned
col_last) const;
02373
02382         void set_submatrix(const Matrix<T>& smtx, unsigned row_first, unsigned col_first);
02383
02388         void clear();
02389
02397         void reshape(unsigned rows, unsigned cols);
02398
02404         void resize(unsigned rows, unsigned cols);
02405
02411         bool exists(unsigned row, unsigned col) const;
02412
02417         T* ptr(unsigned row, unsigned col);
02418
02425         T* ptr();
02426
02430         void fill(T value);
02431
02438         void fill_col(T value, unsigned col);
02439
02446         void fill_row(T value, unsigned row);
02447
02452         bool isempty() const;
02453
02457         bool issquare() const;
02458
02463         bool isequal(const Matrix<T>&) const;
02464
02470         bool isequal(const Matrix<T>&, T) const;
02471
02476         unsigned numel() const;
02477
02482         unsigned rows() const;

```

```

02483
02488     unsigned cols() const;
02489
02494     Matrix<T> transpose() const;
02495
02501     Matrix<T> ctranspose() const;
02502
02510     Matrix<T>& add(const Matrix<T>&);
02511
02519     Matrix<T>& subtract(const Matrix<T>&);
02520
02529     Matrix<T>& mult_hadamard(const Matrix<T>&);
02530
02536     Matrix<T>& add(T);
02537
02543     Matrix<T>& subtract(T);
02544
02550     Matrix<T>& mult(T);
02551
02557     Matrix<T>& div(T);
02558
02563     Matrix<T>& operator=(const Matrix<T>&);
02564
02569     Matrix<T>& operator=(T);
02570
02575     explicit operator std::vector<T>() const;
02576     std::vector<T> to_vector() const;
02577
02584     T& operator()(unsigned nel);
02585     T operator()(unsigned nel) const;
02586     T& at(unsigned nel);
02587     T at(unsigned nel) const;
02588
02595     T& operator()(unsigned row, unsigned col);
02596     T operator()(unsigned row, unsigned col) const;
02597     T& at(unsigned row, unsigned col);
02598     T at(unsigned row, unsigned col) const;
02599
02607     void add_row_to_another(unsigned to, unsigned from);
02608
02616     void add_col_to_another(unsigned to, unsigned from);
02617
02625     void mult_row_by_another(unsigned to, unsigned from);
02626
02634     void mult_col_by_another(unsigned to, unsigned from);
02635
02642     void swap_rows(unsigned i, unsigned j);
02643
02650     void swap_cols(unsigned i, unsigned j);
02651
02658     std::vector<T> col_to_vector(unsigned col) const;
02659
02666     std::vector<T> row_to_vector(unsigned row) const;
02667
02675     void col_from_vector(const std::vector<T>&, unsigned col);
02676
02684     void row_from_vector(const std::vector<T>&, unsigned row);
02685
02686     private:
02687         unsigned nrows;
02688         unsigned ncols;
02689         std::vector<T> data;
02690 };
02691
02692 /*
02693  * Implementation of Matrix class methods
02694  */
02695
02696 template<typename T>
02697 Matrix<T>::Matrix() : nrows(0), ncols(0), data() { }
02698
02699 template<typename T>
02700 Matrix<T>::Matrix(unsigned size) : Matrix(size, size) { }
02701
02702 template<typename T>
02703 Matrix<T>::Matrix(unsigned rows, unsigned cols) : nrows(rows), ncols(cols) {
02704     data.resize(numel());
02705 }
02706
02707 template<typename T>
02708 Matrix<T>::Matrix(T x, unsigned rows, unsigned cols) : Matrix(rows, cols) {
02709     fill(x);
02710 }
02711
02712 template<typename T>
02713 Matrix<T>::Matrix(const T* array, unsigned rows, unsigned cols) : Matrix(rows, cols) {
02714     data.assign(array, array + numel());

```



```

02715 }
02716
02717 template<typename T>
02718 Matrix<T>::Matrix(const std::vector<T>& vec, unsigned rows, unsigned cols) : Matrix(rows, cols) {
02719     if (vec.size() != numel()) throw std::runtime_error("Size of initialization vector not consistent
with matrix dimensions");
02720
02721     data.assign(vec.begin(), vec.end());
02722 }
02723
02724 template<typename T>
02725 Matrix<T>::Matrix(std::initializer_list<T> init_list, unsigned rows, unsigned cols) : Matrix(rows,
cols) {
02726     if (init_list.size() != numel()) throw std::runtime_error("Size of initialization list not
consistent with matrix dimensions");
02727
02728     auto it = init_list.begin();
02729
02730     for (unsigned row = 0; row < this->nrows; row++)
02731         for (unsigned col = 0; col < this->ncols; col++)
02732             this->at(row,col) = *(it++);
02733 }
02734
02735 template<typename T>
02736 Matrix<T>::Matrix(const Matrix & other) : Matrix(other.nrows, other.ncols) {
02737     this->data.assign(other.data.begin(), other.data.end());
02738 }
02739
02740 template<typename T>
02741 Matrix<T>& Matrix<T>::operator=(const Matrix<T>& other) {
02742     this->nrows = other.nrows;
02743     this->ncols = other.ncols;
02744     this->data.assign(other.data.begin(), other.data.end());
02745     return *this;
02746 }
02747
02748 template<typename T>
02749 Matrix<T>& Matrix<T>::operator=(T s) {
02750     fill(s);
02751     return *this;
02752 }
02753
02754 template<typename T>
02755 inline Matrix<T>::operator std::vector<T>() const {
02756     return data;
02757 }
02758
02759 template<typename T>
02760 inline void Matrix<T>::clear() {
02761     this->nrows = 0;
02762     this->ncols = 0;
02763     data.resize(0);
02764 }
02765
02766 template<typename T>
02767 void Matrix<T>::reshape(unsigned rows, unsigned cols) {
02768     if (this->numel() != rows * cols) throw std::runtime_error("Illegal attempt to change number of
elements via reshape");
02769
02770     this->nrows = rows;
02771     this->ncols = cols;
02772 }
02773
02774 template<typename T>
02775 void Matrix<T>::resize(unsigned rows, unsigned cols) {
02776     this->nrows = rows;
02777     this->ncols = cols;
02778     data.resize(nrows*ncols);
02779 }
02780
02781 template<typename T>
02782 Matrix<T> Matrix<T>::get_submatrix(unsigned row_base, unsigned row_lim, unsigned col_base, unsigned
col_lim) const {
02783     if (row_base > row_lim) throw std::out_of_range("Row index of submatrix out of range");
02784     if (col_base > col_lim) throw std::out_of_range("Column index of submatrix out of range");
02785     if (row_lim >= this->rows()) throw std::out_of_range("Row index of submatrix out of range");
02786     if (col_lim >= this->cols()) throw std::out_of_range("Column index of submatrix out of range");
02787
02788     unsigned num_rows = row_lim - row_base + 1;
02789     unsigned num_cols = col_lim - col_base + 1;
02790     Matrix<T> S(num_rows, num_cols);
02791     for (unsigned i = 0; i < num_rows; i++) {
02792         for (unsigned j = 0; j < num_cols; j++) {
02793             S(i,j) = at(row_base + i, col_base + j);
02794         }
02795     }
02796     return S;

```

```

02797 }
02798
02799 template<typename T>
02800 void Matrix<T>::set_submatrix(const Matrix<T>& S, unsigned row_base, unsigned col_base) {
02801     if (this->isempty()) throw std::runtime_error("Invalid attempt to set submatrix in empty matrix");
02802
02803     const unsigned row_lim = row_base + S.rows() - 1;
02804     const unsigned col_lim = col_base + S.cols() - 1;
02805
02806     if (row_base > row_lim) throw std::out_of_range("Row index of submatrix out of range");
02807     if (col_base > col_lim) throw std::out_of_range("Column index of submatrix out of range");
02808     if (row_lim >= this->rows()) throw std::out_of_range("Row index of submatrix out of range");
02809     if (col_lim >= this->cols()) throw std::out_of_range("Column index of submatrix out of range");
02810
02811     unsigned num_rows = row_lim - row_base + 1;
02812     unsigned num_cols = col_lim - col_base + 1;
02813     for (unsigned i = 0; i < num_rows; i++)
02814         for (unsigned j = 0; j < num_cols; j++)
02815             at(row_base + i, col_base + j) = S(i, j);
02816 }
02817
02818 template<typename T>
02819 inline T & Matrix<T>::operator()(unsigned nel) {
02820     return at(nel);
02821 }
02822
02823 template<typename T>
02824 inline T & Matrix<T>::operator()(unsigned row, unsigned col) {
02825     return at(row, col);
02826 }
02827
02828 template<typename T>
02829 inline T Matrix<T>::operator()(unsigned nel) const {
02830     return at(nel);
02831 }
02832
02833 template<typename T>
02834 inline T Matrix<T>::operator()(unsigned row, unsigned col) const {
02835     return at(row, col);
02836 }
02837
02838 template<typename T>
02839 inline T & Matrix<T>::at(unsigned nel) {
02840     if (!(nel < numel())) throw std::out_of_range("Element index out of range");
02841
02842     return data[nel];
02843 }
02844
02845 template<typename T>
02846 inline T & Matrix<T>::at(unsigned row, unsigned col) {
02847     if (!(row < rows() && col < cols())) throw std::out_of_range("Element index out of range");
02848
02849     return data[nrows * col + row];
02850 }
02851
02852 template<typename T>
02853 inline T Matrix<T>::at(unsigned nel) const {
02854     if (!(nel < numel())) throw std::out_of_range("Element index out of range");
02855
02856     return data[nel];
02857 }
02858
02859 template<typename T>
02860 inline T Matrix<T>::at(unsigned row, unsigned col) const {
02861     if (!(row < rows() && col < cols())) throw std::out_of_range("Row index out of range");
02862     if (!(col < cols())) throw std::out_of_range("Column index out of range");
02863
02864     return data[nrows * col + row];
02865 }
02866
02867 template<typename T>
02868 inline void Matrix<T>::fill(T value) {
02869     for (unsigned i = 0; i < numel(); i++)
02870         data[i] = value;
02871 }
02872
02873 template<typename T>
02874 inline void Matrix<T>::fill_col(T value, unsigned col) {
02875     if (!(col < cols())) throw std::out_of_range("Column index out of range");
02876
02877     for (unsigned i = col * nrows; i < (col+1) * nrows; i++)
02878         data[i] = value;
02879 }
02880
02881 template<typename T>
02882 inline void Matrix<T>::fill_row(T value, unsigned row) {
02883     if (!(row < rows())) throw std::out_of_range("Row index out of range");

```

```

02884
02885     for (unsigned i = 0; i < ncols; i++)
02886         data[row + i * nrows] = value;
02887 }
02888
02889 template<typename T>
02890 inline bool Matrix<T>::exists(unsigned row, unsigned col) const {
02891     return (row < nrows && col < ncols);
02892 }
02893
02894 template<typename T>
02895 inline T* Matrix<T>::ptr(unsigned row, unsigned col) {
02896     if (!(row < rows())) throw std::out_of_range("Row index out of range");
02897     if (!(col < cols())) throw std::out_of_range("Column index out of range");
02898
02899     return data.data() + nrows * col + row;
02900 }
02901
02902 template<typename T>
02903 inline T* Matrix<T>::ptr() {
02904     return data.data();
02905 }
02906
02907 template<typename T>
02908 inline bool Matrix<T>::isempty() const {
02909     return (nrows == 0) || (ncols == 0);
02910 }
02911
02912 template<typename T>
02913 inline bool Matrix<T>::issquare() const {
02914     return (nrows == ncols) && !isempty();
02915 }
02916
02917 template<typename T>
02918 bool Matrix<T>::isequal(const Matrix<T>& A) const {
02919     bool ret = true;
02920     if (nrows != A.rows() || ncols != A.cols()) {
02921         ret = false;
02922     } else {
02923         for (unsigned i = 0; i < numel(); i++) {
02924             if (at(i) != A(i)) {
02925                 ret = false;
02926                 break;
02927             }
02928         }
02929     }
02930     return ret;
02931 }
02932
02933 template<typename T>
02934 bool Matrix<T>::isequal(const Matrix<T>& A, T tol) const {
02935     bool ret = true;
02936     if (rows() != A.rows() || cols() != A.cols()) {
02937         ret = false;
02938     } else {
02939         auto abs_tol = std::abs(tol); // workaround for complex
02940         for (unsigned i = 0; i < A.numel(); i++) {
02941             if (abs_tol < std::abs(at(i) - A(i))) {
02942                 ret = false;
02943                 break;
02944             }
02945         }
02946     }
02947     return ret;
02948 }
02949
02950 template<typename T>
02951 inline unsigned Matrix<T>::numel() const {
02952     return nrows * ncols;
02953 }
02954
02955 template<typename T>
02956 inline unsigned Matrix<T>::rows() const {
02957     return nrows;
02958 }
02959
02960 template<typename T>
02961 inline unsigned Matrix<T>::cols() const {
02962     return ncols;
02963 }
02964
02965 template<typename T>
02966 inline Matrix<T> Matrix<T>::transpose() const {
02967     Matrix<T> res(ncols, nrows);
02968     for (unsigned c = 0; c < ncols; c++)
02969         for (unsigned r = 0; r < nrows; r++)
02970             res(c,r) = at(r,c);

```

```

02971     return res;
02972 }
02973
02974 template<typename T>
02975 inline Matrix<T> Matrix<T>::ctranspose() const {
02976     Matrix<T> res(ncols, nrows);
02977     for (unsigned c = 0; c < ncols; c++)
02978         for (unsigned r = 0; r < nrows; r++)
02979             res(c,r) = cconj(at(r,c));
02980     return res;
02981 }
02982
02983 template<typename T>
02984 Matrix<T>& Matrix<T>::add(const Matrix<T>& m) {
02985     if (!(m.rows() == rows() && m.cols() == cols())) throw std::runtime_error("Unmatching matrix
dimensions for iadd");
02986
02987     for (unsigned i = 0; i < numel(); i++)
02988         data[i] += m(i);
02989     return *this;
02990 }
02991
02992 template<typename T>
02993 Matrix<T>& Matrix<T>::subtract(const Matrix<T>& m) {
02994     if (!(m.rows() == rows() && m.cols() == cols())) throw std::runtime_error("Unmatching matrix
dimensions for isubtract");
02995
02996     for (unsigned i = 0; i < numel(); i++)
02997         data[i] -= m(i);
02998     return *this;
02999 }
03000
03001 template<typename T>
03002 Matrix<T>& Matrix<T>::mult_hadamard(const Matrix<T>& m) {
03003     if (!(m.rows() == rows() && m.cols() == cols())) throw std::runtime_error("Unmatching matrix
dimensions for ihprod");
03004
03005     for (unsigned i = 0; i < numel(); i++)
03006         data[i] *= m(i);
03007     return *this;
03008 }
03009
03010 template<typename T>
03011 Matrix<T>& Matrix<T>::add(T s) {
03012     for (auto& x : data)
03013         x += s;
03014     return *this;
03015 }
03016
03017 template<typename T>
03018 Matrix<T>& Matrix<T>::subtract(T s) {
03019     for (auto& x : data)
03020         x -= s;
03021     return *this;
03022 }
03023
03024 template<typename T>
03025 Matrix<T>& Matrix<T>::mult(T s) {
03026     for (auto& x : data)
03027         x *= s;
03028     return *this;
03029 }
03030
03031 template<typename T>
03032 Matrix<T>& Matrix<T>::div(T s) {
03033     for (auto& x : data)
03034         x /= s;
03035     return *this;
03036 }
03037
03038 template<typename T>
03039 void Matrix<T>::add_row_to_another(unsigned to, unsigned from) {
03040     if (!(to < rows() && from < rows())) throw std::out_of_range("Row index out of range");
03041
03042     for (unsigned k = 0; k < cols(); k++)
03043         at(to, k) += at(from, k);
03044 }
03045
03046 template<typename T>
03047 void Matrix<T>::add_col_to_another(unsigned to, unsigned from) {
03048     if (!(to < cols() && from < cols())) throw std::out_of_range("Column index out of range");
03049
03050     for (unsigned k = 0; k < rows(); k++)
03051         at(k, to) += at(k, from);
03052 }
03053
03054 template<typename T>

```

```

03055 void Matrix<T>::mult_row_by_another(unsigned to, unsigned from) {
03056     if (!(to < rows() && from < rows())) throw std::out_of_range("Row index out of range");
03057
03058     for (unsigned k = 0; k < cols(); k++)
03059         at(to, k) *= at(from, k);
03060 }
03061
03062 template<typename T>
03063 void Matrix<T>::mult_col_by_another(unsigned to, unsigned from) {
03064     if (!(to < cols() && from < cols())) throw std::out_of_range("Column index out of range");
03065
03066     for (unsigned k = 0; k < rows(); k++)
03067         at(k, to) *= at(k, from);
03068 }
03069
03070 template<typename T>
03071 void Matrix<T>::swap_rows(unsigned i, unsigned j) {
03072     if (!(i < rows() && j < rows())) throw std::out_of_range("Row index out of range");
03073
03074     for (unsigned k = 0; k < cols(); k++) {
03075         T tmp = at(i, k);
03076         at(i, k) = at(j, k);
03077         at(j, k) = tmp;
03078     }
03079 }
03080
03081 template<typename T>
03082 void Matrix<T>::swap_cols(unsigned i, unsigned j) {
03083     if (!(i < cols() && j < cols())) throw std::out_of_range("Column index out of range");
03084
03085     for (unsigned k = 0; k < rows(); k++) {
03086         T tmp = at(k, i);
03087         at(k, i) = at(k, j);
03088         at(k, j) = tmp;
03089     }
03090 }
03091
03092 template<typename T>
03093 inline std::vector<T> Matrix<T>::to_vector() const {
03094     return data;
03095 }
03096
03097 template<typename T>
03098 inline std::vector<T> Matrix<T>::col_to_vector(unsigned col) const {
03099     std::vector<T> vec(rows());
03100     for (unsigned i = 0; i < rows(); i++)
03101         vec[i] = at(i, col);
03102     return vec;
03103 }
03104
03105 template<typename T>
03106 inline std::vector<T> Matrix<T>::row_to_vector(unsigned row) const {
03107     std::vector<T> vec(cols());
03108     for (unsigned i = 0; i < cols(); i++)
03109         vec[i] = at(row, i);
03110     return vec;
03111 }
03112
03113 template<typename T>
03114 inline void Matrix<T>::col_from_vector(const std::vector<T>& vec, unsigned col) {
03115     if (vec.size() != rows()) throw std::runtime_error("Vector size is not equal to number of rows");
03116     if (col >= cols()) throw std::out_of_range("Column index out of range");
03117
03118     for (unsigned i = 0; i < rows(); i++)
03119         data[col*rows() + i] = vec[i];
03120 }
03121
03122 template<typename T>
03123 inline void Matrix<T>::row_from_vector(const std::vector<T>& vec, unsigned row) {
03124     if (vec.size() != cols()) throw std::runtime_error("Vector size is not equal to number of columns");
03125     if (row >= rows()) throw std::out_of_range("Row index out of range");
03126
03127     for (unsigned i = 0; i < cols(); i++)
03128         data[row + i*rows()] = vec[i];
03129 }
03130
03131 template<typename T>
03132 Matrix<T>::~Matrix() { }
03133
03134 } // namespace Matrix_hpp
03135
03136 #endif // __MATRIX_HPP__

```

