

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 Namespace Index	3
2.1 Namespace List	3
3 Hierarchical Index	5
3.1 Class Hierarchy	5
4 Class Index	7
4.1 Class List	7
5 File Index	9
5.1 File List	9
6 Namespace Documentation	11
6.1 Mtx::Util Namespace Reference	11
6.1.1 Detailed Description	11
6.1.2 Function Documentation	11
6.1.2.1 cconj()	11
6.1.2.2 creal()	12
6.1.2.3 csign()	12
7 Class Documentation	13
7.1 Mtx::Eigenvalues_result< T > Struct Template Reference	13
7.1.1 Detailed Description	13
7.2 Mtx::Hessenberg_result< T > Struct Template Reference	13
7.2.1 Detailed Description	14
7.3 Mtx::Util::is_complex< T > Struct Template Reference	14
7.4 Mtx::Util::is_complex< std::complex< T > > Struct Template Reference	14
7.5 Mtx::LDL_result< T > Struct Template Reference	14
7.5.1 Detailed Description	15
7.6 Mtx::LU_result< T > Struct Template Reference	15
7.6.1 Detailed Description	15
7.7 Mtx::LUP_result< T > Struct Template Reference	16
7.7.1 Detailed Description	16
7.8 Mtx::Matrix< T > Class Template Reference	16
7.8.1 Detailed Description	18
7.8.2 Constructor & Destructor Documentation	19
7.8.2.1 Matrix() [1/8]	19

7.8.2.2 Matrix() [2/8]	19
7.8.2.3 Matrix() [3/8]	19
7.8.2.4 Matrix() [4/8]	19
7.8.2.5 Matrix() [5/8]	20
7.8.2.6 Matrix() [6/8]	20
7.8.2.7 Matrix() [7/8]	20
7.8.2.8 Matrix() [8/8]	21
7.8.2.9 ~Matrix()	21
7.8.3 Member Function Documentation	21
7.8.3.1 add() [1/2]	21
7.8.3.2 add() [2/2]	21
7.8.3.3 add_col_to_another()	22
7.8.3.4 add_row_to_another()	22
7.8.3.5 clear()	22
7.8.3.6 col_from_vector()	23
7.8.3.7 col_to_vector()	23
7.8.3.8 cols()	23
7.8.3.9 ctranspose()	24
7.8.3.10 div()	24
7.8.3.11 exists()	24
7.8.3.12 fill()	25
7.8.3.13 fill_col()	25
7.8.3.14 fill_row()	25
7.8.3.15 get_submatrix()	25
7.8.3.16 isempty()	26
7.8.3.17 isequal() [1/2]	26
7.8.3.18 isequal() [2/2]	26
7.8.3.19 mult()	27
7.8.3.20 mult_col_by_another()	27
7.8.3.21 mult_hadamard()	27
7.8.3.22 mult_row_by_another()	28
7.8.3.23 numel()	28
7.8.3.24 operator std::vector< T >()	28
7.8.3.25 operator>() [1/2]	28
7.8.3.26 operator>() [2/2]	29
7.8.3.27 operator=() [1/2]	29
7.8.3.28 operator=() [2/2]	29
7.8.3.29 ptr() [1/2]	29
7.8.3.30 ptr() [2/2]	30
7.8.3.31 reshape()	30
7.8.3.32 resize()	30
7.8.3.33 row_from_vector()	31

7.8.3.34 row_to_vector()	31
7.8.3.35 rows()	31
7.8.3.36 set_submatrix()	32
7.8.3.37 shape()	32
7.8.3.38 subtract() [1/2]	32
7.8.3.39 subtract() [2/2]	33
7.8.3.40 swap_cols()	33
7.8.3.41 swap_rows()	33
7.8.3.42 transpose()	34
7.9 Mtx::QR_result< T > Struct Template Reference	34
7.9.1 Detailed Description	34
7.10 Mtx::singular_matrix_exception Class Reference	34
8 File Documentation	35
8.1 matrix.hpp File Reference	35
8.1.1 Function Documentation	42
8.1.1.1 add() [1/2]	42
8.1.1.2 add() [2/2]	42
8.1.1.3 adj()	43
8.1.1.4 chol()	43
8.1.1.5 cholinv()	44
8.1.1.6 circshift()	44
8.1.1.7 circulant() [1/2]	45
8.1.1.8 circulant() [2/2]	45
8.1.1.9 cofactor()	46
8.1.1.10 concatenate_horizontal()	46
8.1.1.11 concatenate_vertical()	47
8.1.1.12 cond()	47
8.1.1.13 ctranspose()	47
8.1.1.14 det()	48
8.1.1.15 det_lu()	48
8.1.1.16 diag() [1/3]	48
8.1.1.17 diag() [2/3]	49
8.1.1.18 diag() [3/3]	49
8.1.1.19 div()	50
8.1.1.20 eigenvalues() [1/2]	50
8.1.1.21 eigenvalues() [2/2]	51
8.1.1.22 eye()	51
8.1.1.23 foreach_elem()	52
8.1.1.24 foreach_elem_copy()	52
8.1.1.25 hessenberg()	53
8.1.1.26 householder_reflection()	53

8.1.1.27 <code>imag()</code>	54
8.1.1.28 <code>inv()</code>	54
8.1.1.29 <code>inv_gauss_jordan()</code>	55
8.1.1.30 <code>inv_posdef()</code>	55
8.1.1.31 <code>inv_square()</code>	56
8.1.1.32 <code>inv_tril()</code>	56
8.1.1.33 <code>inv_triu()</code>	56
8.1.1.34 <code>ishess()</code>	57
8.1.1.35 <code>istril()</code>	57
8.1.1.36 <code>istriu()</code>	58
8.1.1.37 <code>kron()</code>	58
8.1.1.38 <code>ldl()</code>	58
8.1.1.39 <code>lu()</code>	59
8.1.1.40 <code>lup()</code>	59
8.1.1.41 <code>make_complex()</code> [1/2]	60
8.1.1.42 <code>make_complex()</code> [2/2]	60
8.1.1.43 <code>mult()</code> [1/4]	61
8.1.1.44 <code>mult()</code> [2/4]	62
8.1.1.45 <code>mult()</code> [3/4]	62
8.1.1.46 <code>mult()</code> [4/4]	63
8.1.1.47 <code>mult_and_add()</code>	64
8.1.1.48 <code>mult_hadamard()</code>	65
8.1.1.49 <code>norm_fro()</code>	65
8.1.1.50 <code>norm_inf()</code>	66
8.1.1.51 <code>norm_p1()</code>	66
8.1.1.52 <code>ones()</code> [1/2]	66
8.1.1.53 <code>ones()</code> [2/2]	67
8.1.1.54 <code>operator"!=()</code>	67
8.1.1.55 <code>operator*()</code> [1/5]	67
8.1.1.56 <code>operator*()</code> [2/5]	68
8.1.1.57 <code>operator*()</code> [3/5]	68
8.1.1.58 <code>operator*()</code> [4/5]	68
8.1.1.59 <code>operator*()</code> [5/5]	68
8.1.1.60 <code>operator*==()</code> [1/2]	69
8.1.1.61 <code>operator*==()</code> [2/2]	69
8.1.1.62 <code>operator+()</code> [1/3]	69
8.1.1.63 <code>operator+()</code> [2/3]	69
8.1.1.64 <code>operator+()</code> [3/3]	70
8.1.1.65 <code>operator+=()</code> [1/2]	70
8.1.1.66 <code>operator+=()</code> [2/2]	70
8.1.1.67 <code>operator-()</code> [1/2]	70
8.1.1.68 <code>operator-()</code> [2/2]	71

8.1.1.69 operator==() [1/2]	71
8.1.1.70 operator==() [2/2]	71
8.1.1.71 operator/()	71
8.1.1.72 operator/=()	72
8.1.1.73 operator<<()	72
8.1.1.74 operator==()	72
8.1.1.75 operator^()	73
8.1.1.76 operator^=()	73
8.1.1.77 permute_cols()	73
8.1.1.78 permute_rows()	74
8.1.1.79 permute_rows_and_cols()	74
8.1.1.80 pinv()	75
8.1.1.81 qr()	76
8.1.1.82 qr_householder()	76
8.1.1.83 qr_red_gs()	77
8.1.1.84 real()	77
8.1.1.85 repmat()	78
8.1.1.86 solve_posdef()	78
8.1.1.87 solve_square()	79
8.1.1.88 solve_tril()	79
8.1.1.89 solve_triu()	80
8.1.1.90 subtract() [1/2]	81
8.1.1.91 subtract() [2/2]	81
8.1.1.92 to_string()	82
8.1.1.93 trace()	82
8.1.1.94 transpose()	82
8.1.1.95 tril()	83
8.1.1.96 triu()	83
8.1.1.97 wilkinson_shift()	83
8.1.1.98 zeros() [1/2]	84
8.1.1.99 zeros() [2/2]	84
8.2 matrix.hpp	84

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

Namespace Index

2.1 Namespace List

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

Mtx::Util	11
-------------------------------------	----

Chapter 3

Hierarchical Index

3.1 Class Hierarchy

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

std::domain_error	
Mtx::singular_matrix_exception	34
Mtx::Eigenvalues_result< T >	13
std::false_type	
Mtx::Util::is_complex< T >	14
Mtx::Hessenberg_result< T >	13
Mtx::LDL_result< T >	14
Mtx::LU_result< T >	15
Mtx::LUP_result< T >	16
Mtx::Matrix< T >	16
Mtx::QR_result< T >	34
std::true_type	
Mtx::Util::is_complex< std::complex< T > >	14

Chapter 4

Class Index

4.1 Class List

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

Mtx::Eigenvalues_result< T >	
Result of eigenvalues	13
Mtx::Hessenberg_result< T >	
Result of Hessenberg decomposition	13
Mtx::Util::is_complex< T >	14
Mtx::Util::is_complex< std::complex< T > >	14
Mtx::LDL_result< T >	
Result of LDL decomposition	14
Mtx::LU_result< T >	
Result of LU decomposition	15
Mtx::LUP_result< T >	
Result of LU decomposition with pivoting	16
Mtx::Matrix< T >	16
Mtx::QR_result< T >	
Result of QR decomposition	34
Mtx::singular_matrix_exception	
Singular matrix exception	34

Chapter 5

File Index

5.1 File List

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

matrix.hpp	35
--------------------------------------	----

Chapter 6

Namespace Documentation

6.1 Mtx::Util Namespace Reference

Classes

- struct [is_complex](#)
- struct [is_complex< std::complex< T > >](#)

Functions

- [template<typename T , typename std::enable_if<!is_complex< T >::value, int >::type = 0> T cconj \(T x\)](#)
Complex conjugate helper.
- [template<typename T , typename std::enable_if<!is_complex< T >::value, int >::type = 0> T csign \(T x\)](#)
Complex sign helper.
- [template<typename T , typename std::enable_if<!is_complex< T >::value, int >::type = 0> T creal \(std::complex< T > x\)](#)
Complex real part helper.
- [template<typename T , typename std::enable_if<!is_complex< T >::value, int >::type = 0> T creal \(T x\)](#)

6.1.1 Detailed Description

Collection of various helper functions that allow for generalization of code for complex and real datatypes.

6.1.2 Function Documentation

6.1.2.1 cconj()

```
template<typename T , typename std::enable_if<!is_complex< T >::value, int >::type = 0>
T Mtx::Util::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`.

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

6.1.2.2 creal()

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

Complex real part 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 returns the real part.

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

6.1.2.3 csign()

```
template<typename T , typename std::enable_if<!is_complex< T >::value, int >::type = 0>
T Mtx::Util::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 $e^{i \cdot \arg(x)}$.

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

Chapter 7

Class Documentation

7.1 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.

7.1.1 Detailed Description

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

Result of eigenvalues.

This structure stores the result of matrix eigenvalue calculation, returned by `Mtx::eigenvalues()` function.

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

- [matrix.hpp](#)

7.2 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.

7.2.1 Detailed Description

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

Result of Hessenberg decomposition.

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

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

- [matrix.hpp](#)

7.3 Mtx::Util::is_complex< T > Struct Template Reference

Inheritance diagram for Mtx::Util::is_complex< T >:

Collaboration diagram for Mtx::Util::is_complex< T >:

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

- [matrix.hpp](#)

7.4 Mtx::Util::is_complex< std::complex< T > > Struct Template Reference

Inheritance diagram for Mtx::Util::is_complex< std::complex< T > >:

Collaboration diagram for Mtx::Util::is_complex< std::complex< T > >:

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

- [matrix.hpp](#)

7.5 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.

7.5.1 Detailed Description

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

Result of LDL decomposition.

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

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

- [matrix.hpp](#)

7.6 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.

7.6.1 Detailed Description

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

Result of LU decomposition.

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

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

- [matrix.hpp](#)

7.7 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.

7.7.1 Detailed Description

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

Result of LU decomposition with pivoting.

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

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

- [matrix.hpp](#)

7.8 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.

- `std::pair< unsigned, unsigned > shape () const`

Matrix shape.

- `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.

7.8.1 Detailed Description

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

`Matrix` class definition.

7.8.2 Constructor & Destructor Documentation

7.8.2.1 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\(\)](#).

7.8.2.2 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.

7.8.2.3 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\(\)](#).

7.8.2.4 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\(\)](#).

7.8.2.5 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\(\)](#).

7.8.2.6 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 given by *nrows* and *ncols*.

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\(\)](#).

7.8.2.7 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`. Number of elements in the list must be equal to the number of matrix elements given by *nrows* and *ncols*.

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\(\)](#).

7.8.2.8 Matrix() [8/8]

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

Copy constructor.

7.8.2.9 ~Matrix()

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

Destructor.

7.8.3 Member Function Documentation**7.8.3.1 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+=\(\)](#).

7.8.3.2 add() [2/2]

```
template<typename T >
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.

7.8.3.3 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\(\)](#).

7.8.3.4 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\(\)](#).

7.8.3.5 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.

7.8.3.6 col_from_vector()

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

Column from vector.

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

Exceptions

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

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

7.8.3.7 col_to_vector()

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

Column to vector.

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

Exceptions

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

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

7.8.3.8 cols()

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

Number of columns.

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

Referenced by [Mtx::Matrix< T >::add\(\)](#), [Mtx::add\(\)](#), [Mtx::add\(\)](#), [Mtx::Matrix< T >::add_col_to_another\(\)](#), [Mtx::Matrix< T >::add_row_to_another\(\)](#), [Mtx::adj\(\)](#), [Mtx::circshift\(\)](#), [Mtx::cofactor\(\)](#), [Mtx::Matrix< T >::col_from_vector\(\)](#), [Mtx::concatenate_horizontal\(\)](#), [Mtx::concatenate_vertical\(\)](#), [Mtx::div\(\)](#), [Mtx::Matrix< T >::fill_col\(\)](#), [Mtx::Matrix< T >::get_submatrix\(\)](#), [Mtx::householder_reflection\(\)](#), [Mtx::imag\(\)](#), [Mtx::Matrix< T >::isequal\(\)](#), [Mtx::Matrix< T >::isequal\(\)](#), [Mtx::istril\(\)](#), [Mtx::istriu\(\)](#), [Mtx::kron\(\)](#), [Mtx::lu\(\)](#), [Mtx::lup\(\)](#), [Mtx::make_complex\(\)](#), [Mtx::make_complex\(\)](#), [Mtx::mult\(\)](#), [Mtx::mult\(\)](#), [Mtx::mult\(\)](#), [Mtx::mult\(\)](#), [Mtx::mult_and_add\(\)](#), [Mtx::Matrix< T >::mult_col_by_another\(\)](#), [Mtx::Matrix< T >::mult_hadamard\(\)](#), [Mtx::mult_hadamard\(\)](#), [Mtx::Matrix< T >::mult_row_by_another\(\)](#), [Mtx::norm_inf\(\)](#), [Mtx::norm_p1\(\)](#), [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::to_string\(\)](#), [Mtx::tril\(\)](#), and [Mtx::triu\(\)](#).

7.8.3.9 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::Util::conj\(\)](#), and [Mtx::Matrix< T >::Matrix\(\)](#).

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

7.8.3.10 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/=\(\)](#).

7.8.3.11 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.

7.8.3.12 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=\(\)](#).

7.8.3.13 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\(\)](#).

7.8.3.14 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\(\)](#).

7.8.3.15 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\(\)](#).

7.8.3.16 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\(\)](#).

7.8.3.17 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==\(\(\)\)](#).

7.8.3.18 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\(\)](#).

7.8.3.19 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*=\(.\)](#).

7.8.3.20 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\(\)](#).

7.8.3.21 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^=\(\)](#).

7.8.3.22 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\(\)](#).

7.8.3.23 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 >::is_equal\(\)](#), [Mtx::Matrix< T >::is_equal\(\)](#), [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::real\(\)](#), [Mtx::Matrix< T >::reshape\(\)](#), [Mtx::solve_posdef\(\)](#), [Mtx::solve_square\(\)](#), [Mtx::solve_tril\(\)](#), [Mtx::solve_triu\(\)](#), [Mtx::Matrix< T >::subtract\(\)](#), and [Mtx::subtract\(\)](#).

7.8.3.24 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.

7.8.3.25 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
--------------------------------	------------------------------------

7.8.3.26 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
--------------------------------	---

7.8.3.27 operator=() [1/2]

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

Matrix assignment.

Performs deep-copy of the matrix.

7.8.3.28 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()`.

7.8.3.29 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
--------------------------------	--

7.8.3.30 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\(\)](#).

7.8.3.31 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\(\)](#).

7.8.3.32 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\(\)](#).

7.8.3.33 row_from_vector()

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

Row from vector.

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

Exceptions

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

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

7.8.3.34 row_to_vector()

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

Row to vector.

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

Exceptions

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

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

7.8.3.35 rows()

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

Number of rows.

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

Referenced by [Mtx::Matrix< T >::add\(\)](#), [Mtx::add\(\)](#), [Mtx::add\(\)](#), [Mtx::Matrix< T >::add_col_to_another\(\)](#), [Mtx::Matrix< T >::add_row_to_another\(\)](#), [Mtx::adj\(\)](#), [Mtx::chol\(\)](#), [Mtx::cholinv\(\)](#), [Mtx::circshift\(\)](#), [Mtx::cofactor\(\)](#), [Mtx::Matrix< T >::col_from_vector\(\)](#), [Mtx::Matrix< T >::col_to_vector\(\)](#), [Mtx::concatenate_horizontal\(\)](#), [Mtx::concatenate_vertical\(\)](#), [Mtx::det\(\)](#), [Mtx::det_lu\(\)](#), [Mtx::diag\(\)](#), [Mtx::div\(\)](#), [Mtx::eigenvalues\(\)](#), [Mtx::Matrix< T >::fill_row\(\)](#), [Mtx::Matrix< T >::get_submatrix\(\)](#), [Mtx::hessenberg\(\)](#), [Mtx::imag\(\)](#), [Mtx::inv\(\)](#), [Mtx::inv_gauss_jordan\(\)](#), [Mtx::inv_tril\(\)](#), [Mtx::inv_triu\(\)](#), [Mtx::Matrix< T >::isequal\(\)](#), [Mtx::Matrix< T >::isequal\(\)](#), [Mtx::ishess\(\)](#), [Mtx::istril\(\)](#), [Mtx::istriu\(\)](#), [Mtx::kron\(\)](#), [Mtx::ldl\(\)](#), [Mtx::lu\(\)](#), [Mtx::lup\(\)](#), [Mtx::make_complex\(\)](#), [Mtx::make_complex\(\)](#), [Mtx::mult\(\)](#), [Mtx::mult\(\)](#), [Mtx::mult\(\)](#), [Mtx::mult\(\)](#), [Mtx::mult_and_add\(\)](#),

Mtx::Matrix< T >::mult_col_by_another(), Mtx::Matrix< T >::mult_hadamard(), Mtx::mult_hadamard(), Mtx::Matrix< T >::mult_row_ Mtx::norm_inf(), Mtx::norm_p1(), 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::to_string(), Mtx::trace(), Mtx::tril(), Mtx::triu(), and Mtx::wilkinson_shift().

7.8.3.36 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\(\)](#).

7.8.3.37 shape()

```
template<typename T >
std::pair< unsigned, unsigned > Mtx::Matrix< T >::shape ( ) const [inline]
```

[Matrix](#) shape.

Returns std::pair with the *first* element providing the number of rows and the *second* element providing the number of columns.

7.8.3.38 subtract() [1/2]

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

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

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

Operation is performed in-place by modifying elements of the matrix.

Exceptions

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

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

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

7.8.3.39 subtract() [2/2]

```
template<typename T >
Mtx::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.

7.8.3.40 swap_cols()

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

Column swap.

Swaps element values between two columns.

Exceptions

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

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

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

7.8.3.41 swap_rows()

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

Row swap.

Swaps element values of two columns.

Exceptions

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

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

7.8.3.42 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](#)

7.9 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.

7.9.1 Detailed Description

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

Result of QR decomposition.

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

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

- [matrix.hpp](#)

7.10 Mtx::singular_matrix_exception Class Reference

Singular matrix exception.

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

Inheritance diagram for Mtx::singular_matrix_exception:

Chapter 8

File Documentation

8.1 matrix.hpp File Reference

Classes

- struct [Mtx::Util::is_complex< T >](#)
- struct [Mtx::Util::is_complex< std::complex< T > >](#)
- 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 >](#)

Namespaces

- namespace [Mtx::Util](#)

Functions

- `template<typename T, typename std::enable_if<!is_complex< T >::value, int >::type = 0> T Mtx::Util::cconj (T x)`
Complex conjugate helper.
- `template<typename T, typename std::enable_if<!is_complex< T >::value, int >::type = 0> T Mtx::Util::csign (T x)`
Complex sign helper.
- `template<typename T, typename std::enable_if<!is_complex< T >::value, int >::type = 0> T Mtx::Util::creal (std::complex< T > x)`
Complex real part helper.
- `template<typename T, typename std::enable_if<!is_complex< T >::value, int >::type = 0> T Mtx::Util::creal (T x)`
- `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_p1 (const Matrix< T > &A)`
Matrix $p = 1$ norm (column norm).
- `template<typename T >`
`double Mtx::norm_inf (const Matrix< T > &A)`
Matrix $p = \infty$ norm (row norm).
- `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_A = false, bool transpose_B = false, bool transpose_C = false>`
`Matrix< T > Mtx::mult_and_add (const Matrix< T > &A, const Matrix< T > &B, const Matrix< T > &C)`
Matrix multiplication with addition.
- `template<typename T , bool transpose_first = false, bool transpose_second = false>`
`Matrix< T > Mtx::mult_hadamard (const Matrix< T > &A, const Matrix< T > &B)`
Matrix Hadamard (element-wise) multiplication.
- `template<typename T , bool transpose_first = false, bool transpose_second = false>`
`Matrix< T > Mtx::add (const Matrix< T > &A, const Matrix< T > &B)`
Matrix addition.
- `template<typename T , bool transpose_first = false, bool transpose_second = false>`
`Matrix< T > Mtx::subtract (const Matrix< T > &A, const Matrix< T > &B)`
Matrix subtraction.
- `template<typename T , bool transpose_matrix = false>`
`std::vector< T > Mtx::mult (const Matrix< T > &A, const std::vector< T > &v)`
Multiplication of matrix by std::vector.
- `template<typename T , bool transpose_matrix = false>`
`std::vector< T > Mtx::mult (const std::vector< T > &v, const Matrix< T > &A)`
Multiplication of std::vector by matrix.
- `template<typename T >`
`Matrix< T > Mtx::add (const Matrix< T > &A, T s)`
Addition of scalar to matrix.
- `template<typename T >`
`Matrix< T > Mtx::subtract (const Matrix< T > &A, T s)`
Subtraction of scalar from matrix.
- `template<typename T >`
`Matrix< T > Mtx::mult (const Matrix< T > &A, T s)`
Multiplication of matrix by scalar.
- `template<typename T >`
`Matrix< T > Mtx::div (const Matrix< T > &A, T s)`
Division of matrix by scalar.
- `template<typename T >`
`std::string Mtx::to_string (const Matrix< T > &A, char col_separator=' ', char row_separator='\n')`
Converts matrix to std::string.
- `template<typename T >`
`std::ostream & Mtx::operator<< (std::ostream &os, const Matrix< T > &A)`
Matrix ostream operator.
- `template<typename T >`
`Matrix< T > Mtx::operator+ (const Matrix< T > &A, const Matrix< T > &B)`
Matrix sum.
- `template<typename T >`
`Matrix< T > Mtx::operator- (const Matrix< T > &A, const Matrix< T > &B)`
Matrix subtraction.

- `template<typename T>`
`Matrix< T> Mtx::operator^ (const Matrix< T> &A, const Matrix< T> &B)`
Matrix Hadamard product.
- `template<typename T>`
`Matrix< T> Mtx::operator* (const Matrix< T> &A, const Matrix< T> &B)`
Matrix product.
- `template<typename T>`
`std::vector< T> Mtx::operator* (const Matrix< T> &A, const std::vector< T> &v)`
Matrix and std::vector product.
- `template<typename T>`
`std::vector< T> Mtx::operator* (const std::vector< T> &v, const Matrix< T> &A)`
std::vector and matrix product.
- `template<typename T>`
`Matrix< T> Mtx::operator+ (const Matrix< T> &A, T s)`
Matrix sum with scalar.
- `template<typename T>`
`Matrix< T> Mtx::operator- (const Matrix< T> &A, T s)`
Matrix subtraction with scalar.
- `template<typename T>`
`Matrix< T> Mtx::operator* (const Matrix< T> &A, T s)`
Matrix product with scalar.
- `template<typename T>`
`Matrix< T> Mtx::operator/ (const Matrix< T> &A, T s)`
Matrix division by scalar.
- `template<typename T>`
`Matrix< T> Mtx::operator+ (T s, const Matrix< T> &A)`
- `template<typename T>`
`Matrix< T> Mtx::operator* (T s, const Matrix< T> &A)`
Matrix product with scalar.
- `template<typename T>`
`Matrix< T> & Mtx::operator+= (Matrix< T> &A, const Matrix< T> &B)`
Matrix sum.
- `template<typename T>`
`Matrix< T> & Mtx::operator-= (Matrix< T> &A, const Matrix< T> &B)`
Matrix subtraction.
- `template<typename T>`
`Matrix< T> & Mtx::operator*= (Matrix< T> &A, const Matrix< T> &B)`
Matrix product.
- `template<typename T>`
`Matrix< T> & Mtx::operator^= (Matrix< T> &A, const Matrix< T> &B)`
Matrix Hadamard product.
- `template<typename T>`
`Matrix< T> & Mtx::operator+= (Matrix< T> &A, T s)`
Matrix sum with scalar.
- `template<typename T>`
`Matrix< T> & Mtx::operator-= (Matrix< T> &A, T s)`
Matrix subtraction with scalar.
- `template<typename T>`
`Matrix< T> & Mtx::operator*= (Matrix< T> &A, T s)`
Matrix product with scalar.
- `template<typename T>`
`Matrix< T> & Mtx::operator/= (Matrix< T> &A, T s)`
Matrix division by scalar.

- `template<typename T >`
`bool Mtx::operator== (const Matrix< T > &A, const Matrix< T > &b)`
Matrix equality check operator.
- `template<typename T >`
`bool Mtx::operator!= (const Matrix< T > &A, const Matrix< T > &b)`
Matrix non-equality check operator.
- `template<typename T >`
`Matrix< T > Mtx::kron (const Matrix< T > &A, const Matrix< T > &B)`
Kronecker product.
- `template<typename T >`
`Matrix< T > Mtx::adj (const Matrix< T > &A)`
Adjugate matrix.
- `template<typename T >`
`Matrix< T > Mtx::cofactor (const Matrix< T > &A, unsigned p, unsigned q)`
Cofactor matrix.
- `template<typename T >`
`T Mtx::det_lu (const Matrix< T > &A)`
Matrix determinant from on LU decomposition.
- `template<typename T >`
`T Mtx::det (const Matrix< T > &A)`
Matrix determinant.
- `template<typename T >`
`LU_result< T > Mtx::lu (const Matrix< T > &A)`
LU decomposition.
- `template<typename T >`
`LUP_result< T > Mtx::lup (const Matrix< T > &A)`
LU decomposition with pivoting.
- `template<typename T >`
`Matrix< T > Mtx::inv_gauss_jordan (const Matrix< T > &A)`
Matrix inverse using Gauss-Jordan elimination.
- `template<typename T >`
`Matrix< T > Mtx::inv_tril (const Matrix< T > &A)`
Matrix inverse for lower triangular matrix.
- `template<typename T >`
`Matrix< T > Mtx::inv_triu (const Matrix< T > &A)`
Matrix inverse for upper triangular matrix.
- `template<typename T >`
`Matrix< T > Mtx::inv_posdef (const Matrix< T > &A)`
Matrix inverse for Hermitian positive-definite matrix.
- `template<typename T >`
`Matrix< T > Mtx::inv_square (const Matrix< T > &A)`
Matrix inverse for general square matrix.
- `template<typename T >`
`Matrix< T > Mtx::inv (const Matrix< T > &A)`
Matrix inverse (universal).
- `template<typename T >`
`Matrix< T > Mtx::pinv (const Matrix< T > &A)`
Moore-Penrose pseudo-inverse.
- `template<typename T >`
`T Mtx::trace (const Matrix< T > &A)`
Matrix trace.
- `template<typename T >`
`double Mtx::cond (const Matrix< T > &A)`

Condition number of a matrix.

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

Cholesky decomposition.

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

Inverse of Cholesky decomposition.

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

LDL decomposition.

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

Reduced QR decomposition based on Gram-Schmidt method.

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

Generate Householder reflection.

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

QR decomposition based on Householder method.

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

QR decomposition.

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

Hessenberg decomposition.

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

Wilkinson's shift for complex eigenvalues.

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

Matrix eigenvalues of complex matrix.

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

Matrix eigenvalues of real matrix.

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

Solves the upper triangular system.

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

Solves the lower triangular system.

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

Solves the square system.

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

Solves the positive definite (Hermitian) system.

8.1.1 Function Documentation

8.1.1.1 `add()` [1/2]

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

Matrix addition.

Performs addition of two matrices.

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

Template Parameters

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

Parameters

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

Returns

output matrix of size $N \times M$

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

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

8.1.1.2 `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\(\)](#).

8.1.1.3 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\(\)](#).

8.1.1.4 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

<code>is_upper</code>	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

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

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

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

8.1.1.5 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 $A = LL^H$.

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

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

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

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

8.1.1.6 circshift()

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

Circular shift.

Returns a matrix that is created by shifting the columns and rows of an input matrix in a circular manner.

If the specified shift factor is a positive value, columns of the matrix are shifted towards right or rows are shifted towards the bottom. A negative value may be used to apply shifts in opposite directions.

Parameters

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

Returns

matrix inverse

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

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

8.1.1.7 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\(\)](#).

8.1.1.8 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\(\)](#).

8.1.1.9 cofactor()

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

Cofactor matrix.

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

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

Parameters

A	input square matrix
p	row to be deleted in the output matrix
q	column to be deleted in the output matrix

Exceptions

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

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

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

8.1.1.10 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\(\)](#).

8.1.1.11 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\(\)](#).

8.1.1.12 cond()

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

Condition number of a matrix.

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

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

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

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

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

8.1.1.13 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 element transposition.

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

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

8.1.1.14 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\(\)](#).

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

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

8.1.1.16 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

<i>A</i>	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\(\)](#).

8.1.1.17 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

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

Returns

diagonal matrix

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

8.1.1.18 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

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

Returns

diagonal matrix

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

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

8.1.1.19 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/\(\)](#).

8.1.1.20 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

<code>std::runtime_error</code>	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\(\)](#).

8.1.1.21 `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\(\)](#).

8.1.1.22 `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\(\)](#).

8.1.1.23 foreach_elem()

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

Applies custom function element-wise in-place.

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

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

Parameters

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

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

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

8.1.1.24 foreach_elem_copy()

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

Applies custom function element-wise with matrix copy.

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

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

Parameters

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

Returns

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

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

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

8.1.1.25 hessenberg()

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

Hessenberg decomposition.

Finds the Hessenberg decomposition of $A = QH Q^*$. Hessenberg matrix H has zero entries below the first subdiagonal.

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

Parameters

A	input matrix to be decomposed
$calculate_Q$	indicates if Q to be calculated

Returns

structure encapsulating calculated H and Q . Q is calculated only when $calculate_Q = \text{True}$.

Exceptions

<code>std::runtime_error</code>	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\(\)](#).

8.1.1.26 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 $N \times 1$
----------	------------------------------------

Returns

column vector with Householder reflection of *a*

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

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

8.1.1.27 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\(\)](#).

8.1.1.28 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 exists, 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\(\)](#).

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

Exceptions

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

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

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

8.1.1.30 inv_posdef()

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

Matrix inverse for Hermitian positive-definite matrix.

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

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

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

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\(\)](#).

8.1.1.31 inv_square()

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

Matrix inverse for general square matrix.

Calculates an inverse of square matrix using matrix.

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

Exceptions

<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\(\)](#).

8.1.1.32 inv_tril()

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

Matrix inverse for lower triangular matrix.

Calculates an inverse of lower triangular matrix.

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

Exceptions

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

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

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

8.1.1.33 inv_triu()

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


Matrix inverse for upper triangular matrix.

Calculates an inverse of upper triangular matrix.

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

Exceptions

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

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

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

8.1.1.34 ishess()

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

Hessenberg matrix check.

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

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

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

8.1.1.35 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\(\)](#).

8.1.1.36 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\(\)](#).

8.1.1.37 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\(\)](#).

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

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

8.1.1.39 lu()

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

LU decomposition.

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

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

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

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\(\)](#).

8.1.1.40 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\(\)](#).

8.1.1.41 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\(\)](#).

8.1.1.42 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. Input matrices 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\(\)](#).

8.1.1.43 mult() [1/4]

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

Matrix multiplication.

Performs multiplication of two matrices.

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

Template Parameters

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

Parameters

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

Returns

output matrix of size $N \times K$

References [Mtx::Util::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*\(\)=\(\)](#).

8.1.1.44 `mult()` [2/4]

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

Multiplication of matrix by `std::vector`.

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

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

Template Parameters

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

Parameters

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

Returns

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

References `Mtx::Util::conj()`, `Mtx::Matrix< T >::cols()`, `Mtx::mult()`, and `Mtx::Matrix< T >::rows()`.

8.1.1.45 `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()`.

8.1.1.46 mult() [4/4]

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

Multiplication of std::vector by matrix.

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

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

Template Parameters

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

Parameters

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

Returns

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

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

8.1.1.47 mult_and_add()

```
template<typename T , bool transpose_A = false, bool transpose_B = false, bool transpose_C =
false>
Matrix< T > Mtx::mult_and_add (
    const Matrix< T > & A,
    const Matrix< T > & B,
    const Matrix< T > & C )
```

Matrix multiplication with addition.

Performs matrix multiplication and addition according to the formula $A \cdot B + C$.

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

Template Parameters

<i>transpose_↵ _A</i>	if set to true, matrix <i>A</i> shall be transposed during operation
<i>transpose_↵ _B</i>	if set to true, matrix <i>B</i> shall be transposed during operation
<i>transpose_↵ _C</i>	if set to true, matrix <i>C</i> shall be transposed during operation

Parameters

<i>A</i>	left-side factor matrix of size <i>N</i> x <i>M</i> (after transposition)
<i>B</i>	right-side factor matrix of size <i>M</i> x <i>K</i> (after transposition)
<i>C</i>	matrix to be added to the result of multiplication of size <i>N</i> x <i>K</i> (after transposition)

Returns

output matrix of size $N \times K$

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

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

8.1.1.48 mult_hadamard()

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

Matrix Hadamard (element-wise) multiplication.

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

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

Template Parameters

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

Parameters

<i>A</i>	left-side matrix of size $N \times M$ (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::Util::conj\(\)](#), [Mtx::Matrix< T >::cols\(\)](#), [Mtx::mult_hadamard\(\)](#), and [Mtx::Matrix< T >::rows\(\)](#).

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

8.1.1.49 norm_fro()

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

Frobenius norm.

Calculates Frobenius norm of a matrix.

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

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

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

8.1.1.50 norm_inf()

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

Matrix $p = \infty$ norm (row norm).

Calculates $p = \infty$ norm $\|A\|_\infty$ of the input matrix. The $p = \infty$ norm is defined as the maximum absolute sum of elements of each row.

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

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

8.1.1.51 norm_p1()

```
template<typename T >
double Mtx::norm_p1 (
    const Matrix< T > & A )
```

Matrix $p = 1$ norm (column norm).

Calculates $p = 1$ norm $\|A\|_1$ of the input matrix. The $p = 1$ norm is defined as the maximum absolute sum of elements of each column.

References [Mtx::Matrix< T >::cols\(\)](#), [Mtx::norm_p1\(\)](#), and [Mtx::Matrix< T >::rows\(\)](#).

Referenced by [Mtx::norm_p1\(\)](#).

8.1.1.52 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\(\)](#).

8.1.1.53 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\(\)](#).

8.1.1.54 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!=\(\)](#).

8.1.1.55 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*\(\)](#).

8.1.1.56 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*\(\)](#).

8.1.1.57 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*\(\)](#).

8.1.1.58 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*\(\)](#).

8.1.1.59 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*\(\)](#).

8.1.1.60 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*=\(\)](#).

8.1.1.61 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*=\(\)](#).

8.1.1.62 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+\(\)](#).

8.1.1.63 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+\(\)](#).

8.1.1.64 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+\(\)](#).

8.1.1.65 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+=\(\)](#).

8.1.1.66 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+=\(\)](#).

8.1.1.67 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-\(\)](#).

8.1.1.68 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\(\)](#).

8.1.1.69 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-=\(\)](#).

8.1.1.70 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\(\)](#).

8.1.1.71 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/\(\)](#).

8.1.1.72 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/=\(\)](#).

8.1.1.73 operator<<()

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

Matrix ostream operator.

Formats a string incorporating elements of the matrix in row-major order. Elements within the same row are separated by the space character. Different lines (rows) are separated by the newline delimiter `std::endl`.

This function does not allow to control the default delimiter characters. Refer to [Mtx::to_string\(\)](#) if control of delimiter characters is needed.

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

8.1.1.74 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==\(\)](#).

8.1.1.75 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^\(\)](#).

8.1.1.76 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^=\(\)](#).

8.1.1.77 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

A	input matrix
$perm$	permutation vector with column indices

Returns

output matrix created by column permutation of *A*

Exceptions

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

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

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

8.1.1.78 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

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

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

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

8.1.1.79 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

<i>std::runtime_error</i>	when any of permutation vectors 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_cols\(\)](#), and [Mtx::Matrix< T >::rows\(\)](#).

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

8.1.1.80 pinv()

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

Moore-Penrose pseudo-inverse.

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

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

More information: https://en.wikipedia.org/wiki/Moore%E2%80%93Penrose_inverse

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

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

8.1.1.81 qr()

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

QR decomposition.

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

Currently, this function is a wrapper around [Mtx::qr_householder\(\)](#). Refer to [qr_red_gs\(\)](#) for alternative implementation.

Parameters

A	input matrix to be decomposed
$calculate_Q$	indicates if Q to be calculated

Returns

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

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

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

8.1.1.82 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 $n \times m$
$calculate_Q$	indicates if Q to be calculated

Returns

structure encapsulating calculated Q of size $n \times n$ and R of size $n \times m$. Q is calculated only when `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\(\)](#).

8.1.1.83 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

A	input matrix to be decomposed, size $n \times m$
-----	--

Returns

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

Exceptions

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

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

8.1.1.84 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\(\)](#).

8.1.1.85 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

A	input matrix to be repeated
m	number of times to repeat matrix A in vertical dimension (rows)
n	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\(\)](#).

8.1.1.86 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\(\)](#).

8.1.1.87 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

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

Returns

solution matrix of size $N \times M$.

Exceptions

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

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

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

8.1.1.88 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\(\)](#).

8.1.1.89 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

<i>U</i>	left side matrix of size $N \times N$. Must be square and upper triangular
<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 >::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\(\)](#).

8.1.1.90 subtract() [1/2]

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

Matrix subtraction.

Performs subtraction of two matrices.

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

Template Parameters

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

Parameters

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

Returns

output matrix of size $N \times M$

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

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

8.1.1.91 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\(\)](#).

8.1.1.92 to_string()

```
template<typename T >
std::string Mtx::to_string (
    const Matrix< T > & A,
    char col_separator = ' ',
    char row_separator = '\n' )
```

Converts matrix to std::string.

This function converts a matrix into a string representation in row-major order. Each element of the matrix is converted to its string equivalent. Elements within the same row are separated by the *col_separator* character. Rows are separated by the *row_separator* character.

Parameters

<i>A</i>	input matrix to be converted
<i>col_separator</i>	character used to separate elements within the same row. The default character is the space
<i>row_separator</i>	character used to separate rows. The default character is the new line '\n'

Returns

std::string representation of the input matrix

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

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

8.1.1.93 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\(\)](#).

8.1.1.94 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\(\)](#).

8.1.1.95 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\(\)](#).

8.1.1.96 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\(\)](#).

8.1.1.97 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 μ 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\(\)](#).

8.1.1.98 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\(\)](#).

8.1.1.99 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\(\)](#).

8.2 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 #include <utility>
00037
00038 namespace Mtx {
00039
00040 template<typename T> class Matrix;
00041
00042 namespace Util {
00043     template<class T> struct is_complex : std::false_type {};
00044     template<class T> struct is_complex<std::complex<T> > : std::true_type {};
00045
00046     template<typename T, typename std::enable_if<!is_complex<T>::value, int>::type = 0>
00047     inline T cconj(T x) {
00048         return x;
00049     }
00050
00051     template<typename T, typename std::enable_if<is_complex<T>::value, int>::type = 0>
00052     inline T cconj(T x) {
00053         return std::conj(x);
00054     }
00055
00056     template<typename T, typename std::enable_if<!is_complex<T>::value, int>::type = 0>
00057     inline T csign(T x) {
00058         return (x > static_cast<T>(0)) ? static_cast<T>(1) : static_cast<T>(-1);
00059     }
00060
00061     template<typename T, typename std::enable_if<is_complex<T>::value, int>::type = 0>
00062     inline T csign(T x) {
00063         auto x_arg = std::arg(x);
00064         T y(0, x_arg);
00065         return std::exp(y);
00066     }
00067
00068     template<typename T, typename std::enable_if<!is_complex<T>::value, int>::type = 0>
00069     inline T creal(std::complex<T> x) {
00070         return std::real(x);
00071     }
00072
00073     template<typename T, typename std::enable_if<is_complex<T>::value, int>::type = 0>
00074     inline T creal(T x) {
00075         return x;
00076     }
00077 } // namespace Util
00078
00079 class singular_matrix_exception : public std::domain_error {
00080 public:
00081     singular_matrix_exception(const std::string& message) : std::domain_error(message) {}
00082 };
00083
00084 template<typename T>
00085 struct LU_result {
00086     Matrix<T> L;
00087     Matrix<T> U;
00088 };

```

```

00131
00136 template<typename T>
00137 struct LUP_result {
00140     Matrix<T> L;
00141
00144     Matrix<T> U;
00145
00148     std::vector<unsigned> P;
00149 };
00150
00156 template<typename T>
00157 struct QR_result {
00160     Matrix<T> Q;
00161
00164     Matrix<T> R;
00165 };
00166
00171 template<typename T>
00172 struct Hessenberg_result {
00175     Matrix<T> H;
00176
00179     Matrix<T> Q;
00180 };
00181
00186 template<typename T>
00187 struct LDL_result {
00190     Matrix<T> L;
00191
00194     std::vector<T> d;
00195 };
00196
00201 template<typename T>
00202 struct Eigenvalues_result {
00205     std::vector<std::complex<T>> eig;
00206
00209     bool converged;
00210
00213     T err;
00214 };
00215
00216
00224 template<typename T>
00225 inline Matrix<T> zeros(unsigned nrows, unsigned ncols) {
00226     return Matrix<T>(static_cast<T>(0), nrows, ncols);
00227 }
00228
00235 template<typename T>
00236 inline Matrix<T> zeros(unsigned n) {
00237     return zeros<T>(n,n);
00238 }
00239
00250 template<typename T>
00251 inline Matrix<T> ones(unsigned nrows, unsigned ncols) {
00252     return Matrix<T>(static_cast<T>(1), nrows, ncols);
00253 }
00254
00264 template<typename T>
00265 inline Matrix<T> ones(unsigned n) {
00266     return ones<T>(n,n);
00267 }
00268
00276 template<typename T>
00277 Matrix<T> eye(unsigned n) {
00278     Matrix<T> A(static_cast<T>(0), n, n);
00279     for (unsigned i = 0; i < n; i++)
00280         A(i,i) = static_cast<T>(1);
00281     return A;
00282 }
00283
00292 template<typename T>
00293 Matrix<T> diag(const T* array, size_t n) {
00294     Matrix<T> A(static_cast<T>(0), n, n);
00295     for (unsigned i = 0; i < n; i++) {
00296         A(i,i) = array[i];
00297     }
00298     return A;
00299 }
00300
00309 template<typename T>
00310 inline Matrix<T> diag(const std::vector<T>& v) {
00311     return diag(v.data(), v.size());
00312 }
00313
00323 template<typename T>
00324 std::vector<T> diag(const Matrix<T>& A) {
00325     if (! A.issquare()) throw std::runtime_error("Input matrix is not square");
00326

```

```

00327     std::vector<T> v;
00328     v.resize(A.rows());
00329
00330     for (unsigned i = 0; i < A.rows(); i++)
00331         v[i] = A(i,i);
00332     return v;
00333 }
00334
00343 template<typename T>
00344 Matrix<T> circulant(const T* array, unsigned n) {
00345     Matrix<T> A(n, n);
00346     for (unsigned j = 0; j < n; j++)
00347         for (unsigned i = 0; i < n; i++)
00348             A((i+j) % n, j) = array[i];
00349     return A;
00350 }
00351
00363 template<typename T>
00364 Matrix<std::complex<T>> make_complex(const Matrix<T>& Re, const Matrix<T>& Im) {
00365     if (Re.rows() != Im.rows() || Re.cols() != Im.cols()) throw std::runtime_error("Size of input
matrices does not match");
00366
00367     Matrix<std::complex<T>> C(Re.rows(), Re.cols());
00368     for (unsigned n = 0; n < Re.numel(); n++) {
00369         C(n).real(Re(n));
00370         C(n).imag(Im(n));
00371     }
00372
00373     return C;
00374 }
00375
00383 template<typename T>
00384 Matrix<std::complex<T>> make_complex(const Matrix<T>& Re) {
00385     Matrix<std::complex<T>> C(Re.rows(), Re.cols());
00386
00387     for (unsigned n = 0; n < Re.numel(); n++) {
00388         C(n).real(Re(n));
00389         C(n).imag(static_cast<T>(0));
00390     }
00391
00392     return C;
00393 }
00394
00399 template<typename T>
00400 Matrix<T> real(const Matrix<std::complex<T>& C) {
00401     Matrix<T> Re(C.rows(), C.cols());
00402
00403     for (unsigned n = 0; n < C.numel(); n++)
00404         Re(n) = C(n).real();
00405
00406     return Re;
00407 }
00408
00413 template<typename T>
00414 Matrix<T> imag(const Matrix<std::complex<T>& C) {
00415     Matrix<T> Re(C.rows(), C.cols());
00416
00417     for (unsigned n = 0; n < C.numel(); n++)
00418         Re(n) = C(n).imag();
00419
00420     return Re;
00421 }
00422
00431 template<typename T>
00432 inline Matrix<T> circshift(const std::vector<T>& v) {
00433     return circulant(v.data(), v.size());
00434 }
00435
00440 template<typename T>
00441 inline Matrix<T> transpose(const Matrix<T>& A) {
00442     return A.transpose();
00443 }
00444
00450 template<typename T>
00451 inline Matrix<T> ctranspose(const Matrix<T>& A) {
00452     return A.ctranspose();
00453 }
00454
00467 template<typename T>
00468 Matrix<T> circshift(const Matrix<T>& A, int row_shift, int col_shift) {
00469     Matrix<T> B(A.rows(), A.cols());
00470     for (int i = 0; i < A.rows(); i++) {
00471         int ii = (i + row_shift) % A.rows();
00472         for (int j = 0; j < A.cols(); j++) {
00473             int jj = (j + col_shift) % A.cols();
00474             B(ii, jj) = A(i, j);
00475         }

```

```

00476     }
00477     return B;
00478 }
00479
00480 template<typename T>
00481 Matrix<T> repmat(const Matrix<T>& A, unsigned m, unsigned n) {
00490     Matrix<T> B(m * A.rows(), n * A.cols());
00491
00492     for (unsigned cb = 0; cb < n; cb++)
00493         for (unsigned rb = 0; rb < m; rb++)
00494             for (unsigned c = 0; c < A.cols(); c++)
00495                 for (unsigned r = 0; r < A.rows(); r++)
00496                     B(rb*A.rows() + r, cb*A.cols() + c) = A(r, c);
00497
00498     return B;
00499 }
00500
00501 template<typename T>
00502 Matrix<T> concatenate_horizontal(const Matrix<T>& A, const Matrix<T>& B) {
00509     if (A.rows() != B.rows()) throw std::runtime_error("Unmatching number of rows for horizontal
concatenation");
00510
00511     Matrix<T> C(A.rows(), A.cols() + B.cols());
00512
00513     for (unsigned c = 0; c < A.cols(); c++)
00514         for (unsigned r = 0; r < A.rows(); r++)
00515             C(r,c) = A(r,c);
00516
00517     for (unsigned c = 0; c < B.cols(); c++)
00518         for (unsigned r = 0; r < B.rows(); r++)
00519             C(r,c+A.cols()) = B(r,c);
00520
00521     return C;
00522 }
00523
00524 template<typename T>
00525 Matrix<T> concatenate_vertical(const Matrix<T>& A, const Matrix<T>& B) {
00532     if (A.cols() != B.cols()) throw std::runtime_error("Unmatching number of columns for vertical
concatenation");
00533
00534     Matrix<T> C(A.rows() + B.rows(), A.cols());
00535
00536     for (unsigned c = 0; c < A.cols(); c++)
00537         for (unsigned r = 0; r < A.rows(); r++)
00538             C(r,c) = A(r,c);
00539
00540     for (unsigned c = 0; c < B.cols(); c++)
00541         for (unsigned r = 0; r < B.rows(); r++)
00542             C(r+A.rows(),c) = B(r,c);
00543
00544     return C;
00545 }
00546
00547 template<typename T>
00548 double norm_fro(const Matrix<T>& A) {
00555     double sum = 0;
00556
00557     for (unsigned i = 0; i < A.numel(); i++)
00558         sum += Util::creal(A(i) * Util::cconj(A(i)));
00559
00560     return std::sqrt(sum);
00561 }
00562
00563 template<typename T>
00564 double norm_p1(const Matrix<T>& A) {
00570     double max_sum = 0.0;
00571
00572     for (unsigned c = 0; c < A.cols(); c++) {
00573         double sum = 0.0;
00574
00575         for (unsigned r = 0; r < A.rows(); r++)
00576             sum += std::abs(A(r,c));
00577
00578         if (sum > max_sum)
00579             max_sum = sum;
00580     }
00581
00582     return max_sum;
00583 }
00584
00585 template<typename T>
00586 double norm_inf(const Matrix<T>& A) {
00592     double max_sum = 0.0;
00593
00594     for (unsigned r = 0; r < A.rows(); r++) {
00595         double sum = 0.0;
00596

```



```

00597     for (unsigned c = 0; c < A.cols(); c++)
00598         sum += std::abs(A(r,c));
00599
00600     if (sum > max_sum)
00601         max_sum = sum;
00602 }
00603
00604 return max_sum;
00605 }
00606
00612 template<typename T>
00613 Matrix<T> tril(const Matrix<T>& A) {
00614     Matrix<T> B(A);
00615
00616     for (unsigned row = 0; row < B.rows(); row++)
00617         for (unsigned col = row+1; col < B.cols(); col++)
00618             B(row,col) = static_cast<T>(0);
00619
00620     return B;
00621 }
00622
00628 template<typename T>
00629 Matrix<T> triu(const Matrix<T>& A) {
00630     Matrix<T> B(A);
00631
00632     for (unsigned col = 0; col < B.cols(); col++)
00633         for (unsigned row = col+1; row < B.rows(); row++)
00634             B(row,col) = static_cast<T>(0);
00635
00636     return B;
00637 }
00638
00644 template<typename T>
00645 bool istril(const Matrix<T>& A) {
00646     for (unsigned row = 0; row < A.rows(); row++)
00647         for (unsigned col = row+1; col < A.cols(); col++)
00648             if (A(row,col) != static_cast<T>(0)) return false;
00649     return true;
00650 }
00651
00657 template<typename T>
00658 bool istriu(const Matrix<T>& A) {
00659     for (unsigned col = 0; col < A.cols(); col++)
00660         for (unsigned row = col+1; row < A.rows(); row++)
00661             if (A(row,col) != static_cast<T>(0)) return false;
00662     return true;
00663 }
00664
00670 template<typename T>
00671 bool ishess(const Matrix<T>& A) {
00672     if (!A.issquare())
00673         return false;
00674     for (unsigned row = 2; row < A.rows(); row++)
00675         for (unsigned col = 0; col < row-2; col++)
00676             if (A(row,col) != static_cast<T>(0)) return false;
00677     return true;
00678 }
00679
00691 template<typename T>
00692 inline void foreach_elem(Matrix<T>& A, std::function<T(T)> func) {
00693     for (unsigned i = 0; i < A.numel(); i++)
00694         A(i) = func(A(i));
00695 }
00696
00709 template<typename T>
00710 inline Matrix<T> foreach_elem_copy(const Matrix<T>& A, std::function<T(T)> func) {
00711     Matrix<T> B(A);
00712     foreach_elem(B, func);
00713     return B;
00714 }
00715
00729 template<typename T>
00730 Matrix<T> permute_rows(const Matrix<T>& A, const std::vector<unsigned> perm) {
00731     if (perm.empty()) throw std::runtime_error("Permutation vector is empty");
00732
00733     for (unsigned p = 0; p < perm.size(); p++)
00734         if (!(perm[p] < A.rows())) throw std::out_of_range("Index in permutation vector out of range");
00735
00736     Matrix<T> B(perm.size(), A.cols());
00737
00738     for (unsigned p = 0; p < perm.size(); p++)
00739         for (unsigned c = 0; c < A.cols(); c++)
00740             B(p,c) = A(perm[p],c);
00741
00742     return B;
00743 }
00744

```

```

00758 template<typename T>
00759 Matrix<T> permute_cols(const Matrix<T>& A, const std::vector<unsigned> perm) {
00760     if (perm.empty()) throw std::runtime_error("Permutation vector is empty");
00761
00762     for (unsigned p = 0; p < perm.size(); p++)
00763         if (!perm[p] < A.cols()) throw std::out_of_range("Index in permutation vector out of range");
00764
00765     Matrix<T> B(A.rows(), perm.size());
00766
00767     for (unsigned p = 0; p < perm.size(); p++)
00768         for (unsigned r = 0; r < A.rows(); r++)
00769             B(r,p) = A(r,perm[p]);
00770
00771     return B;
00772 }
00773
00790 template<typename T>
00791 Matrix<T> permute_rows_and_cols(const Matrix<T>& A, const std::vector<unsigned> perm_rows, const
std::vector<unsigned> perm_cols) {
00792     if (perm_rows.empty()) throw std::runtime_error("Row permutation vector is empty");
00793     if (perm_cols.empty()) throw std::runtime_error("Column permutation vector is empty");
00794
00795     for (unsigned pc = 0; pc < perm_cols.size(); pc++)
00796         if (!perm_cols[pc] < A.cols()) throw std::out_of_range("Column index in permutation vector out
of range");
00797
00798     for (unsigned pr = 0; pr < perm_rows.size(); pr++)
00799         if (!perm_rows[pr] < A.rows()) throw std::out_of_range("Row index in permutation vector out of
range");
00800
00801     Matrix<T> B(perm_rows.size(), perm_cols.size());
00802
00803     for (unsigned pc = 0; pc < perm_cols.size(); pc++)
00804         for (unsigned pr = 0; pr < perm_rows.size(); pr++)
00805             B(pr,pc) = A(perm_rows[pr],perm_cols[pc]);
00806
00807     return B;
00808 }
00809
00825 template<typename T, bool transpose_first = false, bool transpose_second = false>
00826 Matrix<T> mult(const Matrix<T>& A, const Matrix<T>& B) {
00827     // Adjust dimensions based on transpositions
00828     unsigned rows_A = transpose_first ? A.cols() : A.rows();
00829     unsigned cols_A = transpose_first ? A.rows() : A.cols();
00830     unsigned rows_B = transpose_second ? B.cols() : B.rows();
00831     unsigned cols_B = transpose_second ? B.rows() : B.cols();
00832
00833     if (cols_A != rows_B) throw std::runtime_error("Unmatching matrix dimensions for mult");
00834
00835     Matrix<T> C(static_cast<T>(0), rows_A, cols_B);
00836
00837     for (unsigned i = 0; i < rows_A; i++)
00838         for (unsigned j = 0; j < cols_B; j++)
00839             for (unsigned k = 0; k < cols_A; k++)
00840                 C(i,j) += (transpose_first ? Util::cconj(A(k,i)) : A(i,k)) *
00841                     (transpose_second ? Util::cconj(B(j,k)) : B(k,j));
00842
00843     return C;
00844 }
00845
00863 template<typename T, bool transpose_A = false, bool transpose_B = false, bool transpose_C = false>
00864 Matrix<T> mult_and_add(const Matrix<T>& A, const Matrix<T>& B, const Matrix<T>& C) {
00865     // Adjust dimensions based on transpositions
00866     unsigned rows_A = transpose_A ? A.cols() : A.rows();
00867     unsigned cols_A = transpose_A ? A.rows() : A.cols();
00868     unsigned rows_B = transpose_B ? B.cols() : B.rows();
00869     unsigned cols_B = transpose_B ? B.rows() : B.cols();
00870     unsigned rows_C = transpose_C ? C.cols() : C.rows();
00871     unsigned cols_C = transpose_C ? C.rows() : C.cols();
00872
00873     if ((cols_A != rows_B) || (rows_A != rows_C) || (cols_B != cols_C))
00874         throw std::runtime_error("Unmatching matrix dimensions for mult_and_add");
00875
00876     Matrix<T> D(rows_C, cols_C);
00877
00878     for (unsigned i = 0; i < rows_A; i++) {
00879         for (unsigned j = 0; j < cols_B; j++) {
00880             D(i,j) = transpose_C ? Util::cconj(C(j,i)) : C(i,j);
00881             for (unsigned k = 0; k < cols_A; k++) {
00882                 D(i,j) += (transpose_A ? Util::cconj(A(k,i)) : A(i,k)) *
00883                     (transpose_B ? Util::cconj(B(j,k)) : B(k,j));
00884             }
00885         }
00886     }
00887
00888     return D;
00889 }

```

```

00890
00906 template<typename T, bool transpose_first = false, bool transpose_second = false>
00907 Matrix<T> mult_hadamard(const Matrix<T>& A, const Matrix<T>& B) {
00908     // Adjust dimensions based on transpositions
00909     unsigned rows_A = transpose_first ? A.cols() : A.rows();
00910     unsigned cols_A = transpose_first ? A.rows() : A.cols();
00911     unsigned rows_B = transpose_second ? B.cols() : B.rows();
00912     unsigned cols_B = transpose_second ? B.rows() : B.cols();
00913
00914     if ((rows_A != rows_B) || (cols_A != cols_B)) throw std::runtime_error("Unmatching matrix dimensions
for mult_hadamard");
00915
00916     Matrix<T> C(static_cast<T>(0), rows_A, cols_A);
00917
00918     for (unsigned i = 0; i < rows_A; i++)
00919         for (unsigned j = 0; j < cols_A; j++)
00920             C(i,j) += (transpose_first ? Util::cconj(A(j,i)) : A(i,j)) *
00921                 (transpose_second ? Util::cconj(B(j,i)) : B(i,j));
00922
00923     return C;
00924 }
00925
00941 template<typename T, bool transpose_first = false, bool transpose_second = false>
00942 Matrix<T> add(const Matrix<T>& A, const Matrix<T>& B) {
00943     // Adjust dimensions based on transpositions
00944     unsigned rows_A = transpose_first ? A.cols() : A.rows();
00945     unsigned cols_A = transpose_first ? A.rows() : A.cols();
00946     unsigned rows_B = transpose_second ? B.cols() : B.rows();
00947     unsigned cols_B = transpose_second ? B.rows() : B.cols();
00948
00949     if ((rows_A != rows_B) || (cols_A != cols_B)) throw std::runtime_error("Unmatching matrix dimensions
for add");
00950
00951     Matrix<T> C(static_cast<T>(0), rows_A, cols_A);
00952
00953     for (unsigned i = 0; i < rows_A; i++)
00954         for (unsigned j = 0; j < cols_A; j++)
00955             C(i,j) += (transpose_first ? Util::cconj(A(j,i)) : A(i,j)) +
00956                 (transpose_second ? Util::cconj(B(j,i)) : B(i,j));
00957
00958     return C;
00959 }
00960
00976 template<typename T, bool transpose_first = false, bool transpose_second = false>
00977 Matrix<T> subtract(const Matrix<T>& A, const Matrix<T>& B) {
00978     // Adjust dimensions based on transpositions
00979     unsigned rows_A = transpose_first ? A.cols() : A.rows();
00980     unsigned cols_A = transpose_first ? A.rows() : A.cols();
00981     unsigned rows_B = transpose_second ? B.cols() : B.rows();
00982     unsigned cols_B = transpose_second ? B.rows() : B.cols();
00983
00984     if ((rows_A != rows_B) || (cols_A != cols_B)) throw std::runtime_error("Unmatching matrix dimensions
for subtract");
00985
00986     Matrix<T> C(static_cast<T>(0), rows_A, cols_A);
00987
00988     for (unsigned i = 0; i < rows_A; i++)
00989         for (unsigned j = 0; j < cols_A; j++)
00990             C(i,j) += (transpose_first ? Util::cconj(A(j,i)) : A(i,j)) -
00991                 (transpose_second ? Util::cconj(B(j,i)) : B(i,j));
00992
00993     return C;
00994 }
00995
01011 template<typename T, bool transpose_matrix = false>
01012 std::vector<T> mult(const Matrix<T>& A, const std::vector<T>& v) {
01013     // Adjust dimensions based on transpositions
01014     unsigned rows_A = transpose_matrix ? A.cols() : A.rows();
01015     unsigned cols_A = transpose_matrix ? A.rows() : A.cols();
01016
01017     if (cols_A != v.size()) throw std::runtime_error("Unmatching matrix dimensions for mult");
01018
01019     std::vector<T> u(rows_A, static_cast<T>(0));
01020     for (unsigned r = 0; r < rows_A; r++)
01021         for (unsigned c = 0; c < cols_A; c++)
01022             u[r] += v[c] * (transpose_matrix ? Util::cconj(A(c,r)) : A(r,c));
01023
01024     return u;
01025 }
01026
01042 template<typename T, bool transpose_matrix = false>
01043 std::vector<T> mult(const std::vector<T>& v, const Matrix<T>& A) {
01044     // Adjust dimensions based on transpositions
01045     unsigned rows_A = transpose_matrix ? A.cols() : A.rows();
01046     unsigned cols_A = transpose_matrix ? A.rows() : A.cols();
01047
01048     if (rows_A != v.size()) throw std::runtime_error("Unmatching matrix dimensions for mult");

```

```

01049
01050     std::vector<T> u(cols_A, static_cast<T>(0));
01051     for (unsigned c = 0; c < cols_A; c++)
01052         for (unsigned r = 0; r < rows_A; r++)
01053             u[c] += v[r] * (transpose_matrix ? Util::cconj(A(c,r)) : A(r,c));
01054
01055     return u;
01056 }
01057
01063 template<typename T>
01064 Matrix<T> add(const Matrix<T>& A, T s) {
01065     Matrix<T> B(A.rows(), A.cols());
01066     for (unsigned i = 0; i < A.numel(); i++)
01067         B(i) = A(i) + s;
01068     return B;
01069 }
01070
01076 template<typename T>
01077 Matrix<T> subtract(const Matrix<T>& A, T s) {
01078     Matrix<T> B(A.rows(), A.cols());
01079     for (unsigned i = 0; i < A.numel(); i++)
01080         B(i) = A(i) - s;
01081     return B;
01082 }
01083
01089 template<typename T>
01090 Matrix<T> mult(const Matrix<T>& A, T s) {
01091     Matrix<T> B(A.rows(), A.cols());
01092     for (unsigned i = 0; i < A.numel(); i++)
01093         B(i) = A(i) * s;
01094     return B;
01095 }
01096
01102 template<typename T>
01103 Matrix<T> div(const Matrix<T>& A, T s) {
01104     Matrix<T> B(A.rows(), A.cols());
01105     for (unsigned i = 0; i < A.numel(); i++)
01106         B(i) = A(i) / s;
01107     return B;
01108 }
01109
01121 template<typename T>
01122 std::string to_string(const Matrix<T>& A, char col_separator = ' ', char row_separator = '\n') {
01123     std::stringstream ss;
01124     for (unsigned row = 0; row < A.rows(); row++) {
01125         for (unsigned col = 0; col < A.cols(); col++)
01126             ss << A(row,col) << col_separator;
01127         if (row < static_cast<unsigned>(A.rows()-1)) ss << row_separator;
01128     }
01129     return ss.str();
01130 }
01131
01140 template<typename T>
01141 std::ostream& operator<<(std::ostream& os, const Matrix<T>& A) {
01142     for (unsigned row = 0; row < A.rows(); row++) {
01143         for (unsigned col = 0; col < A.cols(); col++)
01144             os << A(row,col) << " ";
01145         if (row < static_cast<unsigned>(A.rows()-1)) os << std::endl;
01146     }
01147     return os;
01148 }
01149
01154 template<typename T>
01155 inline Matrix<T> operator+(const Matrix<T>& A, const Matrix<T>& B) {
01156     return add(A,B);
01157 }
01158
01163 template<typename T>
01164 inline Matrix<T> operator-(const Matrix<T>& A, const Matrix<T>& B) {
01165     return subtract(A,B);
01166 }
01167
01173 template<typename T>
01174 inline Matrix<T> operator^(const Matrix<T>& A, const Matrix<T>& B) {
01175     return mult_hadamard(A,B);
01176 }
01177
01182 template<typename T>
01183 inline Matrix<T> operator*(const Matrix<T>& A, const Matrix<T>& B) {
01184     return mult(A,B);
01185 }
01186
01192 template<typename T>
01193 inline std::vector<T> operator*(const Matrix<T>& A, const std::vector<T>& v) {
01194     return mult(A,v);
01195 }
01196

```

```

01202 template<typename T>
01203 inline std::vector<T> operator*(const std::vector<T>& v, const Matrix<T>& A) {
01204     return mult(v,A);
01205 }
01206
01211 template<typename T>
01212 inline Matrix<T> operator+(const Matrix<T>& A, T s) {
01213     return add(A,s);
01214 }
01215
01220 template<typename T>
01221 inline Matrix<T> operator-(const Matrix<T>& A, T s) {
01222     return subtract(A,s);
01223 }
01224
01229 template<typename T>
01230 inline Matrix<T> operator*(const Matrix<T>& A, T s) {
01231     return mult(A,s);
01232 }
01233
01238 template<typename T>
01239 inline Matrix<T> operator/(const Matrix<T>& A, T s) {
01240     return div(A,s);
01241 }
01242
01246 template<typename T>
01247 inline Matrix<T> operator+(T s, const Matrix<T>& A) {
01248     return add(A,s);
01249 }
01250
01255 template<typename T>
01256 inline Matrix<T> operator*(T s, const Matrix<T>& A) {
01257     return mult(A,s);
01258 }
01259
01264 template<typename T>
01265 inline Matrix<T>& operator+=(Matrix<T>& A, const Matrix<T>& B) {
01266     return A.add(B);
01267 }
01268
01273 template<typename T>
01274 inline Matrix<T>& operator-=(Matrix<T>& A, const Matrix<T>& B) {
01275     return A.subtract(B);
01276 }
01277
01282 template<typename T>
01283 inline Matrix<T>& operator*=(Matrix<T>& A, const Matrix<T>& B) {
01284     A = mult(A,B);
01285     return A;
01286 }
01287
01293 template<typename T>
01294 inline Matrix<T>& operator^=(Matrix<T>& A, const Matrix<T>& B) {
01295     return A.mult_hadamard(B);
01296 }
01297
01302 template<typename T>
01303 inline Matrix<T>& operator+=(Matrix<T>& A, T s) {
01304     return A.add(s);
01305 }
01306
01311 template<typename T>
01312 inline Matrix<T>& operator-=(Matrix<T>& A, T s) {
01313     return A.subtract(s);
01314 }
01315
01320 template<typename T>
01321 inline Matrix<T>& operator*=(Matrix<T>& A, T s) {
01322     return A.mult(s);
01323 }
01324
01329 template<typename T>
01330 inline Matrix<T>& operator/=(Matrix<T>& A, T s) {
01331     return A.div(s);
01332 }
01333
01338 template<typename T>
01339 inline bool operator==(const Matrix<T>& A, const Matrix<T>& b) {
01340     return A.isequal(b);
01341 }
01342
01347 template<typename T>
01348 inline bool operator!=(const Matrix<T>& A, const Matrix<T>& b) {
01349     return !(A.isequal(b));
01350 }
01351
01359 template<typename T>

```

```

01360 Matrix<T> kron(const Matrix<T>& A, const Matrix<T>& B) {
01361     const unsigned rows_A = A.rows();
01362     const unsigned cols_A = A.cols();
01363     const unsigned rows_B = B.rows();
01364     const unsigned cols_B = B.cols();
01365
01366     unsigned rows_C = rows_A * rows_B;
01367     unsigned cols_C = cols_A * cols_B;
01368
01369     Matrix<T> C(rows_C, cols_C);
01370
01371     for (unsigned i = 0; i < rows_A; i++)
01372         for (unsigned j = 0; j < cols_A; j++)
01373             for (unsigned k = 0; k < rows_B; k++)
01374                 for (unsigned l = 0; l < cols_B; l++)
01375                     C(i+rows_B * k, j+cols_B * l) = A(i, j) * B(k, l);
01376
01377     return C;
01378 }
01379
01388 template<typename T>
01389 Matrix<T> adj(const Matrix<T>& A) {
01390     if (! A.issquare()) throw std::runtime_error("Input matrix is not square");
01391
01392     Matrix<T> B(A.rows(), A.cols());
01393     if (A.rows() == 1) {
01394         B(0) = static_cast<T>(1.0);
01395     } else {
01396         for (unsigned i = 0; i < A.rows(); i++) {
01397             for (unsigned j = 0; j < A.cols(); j++) {
01398                 T sgn = static_cast<T>(1.0) (((i + j) % 2 == 0) ? (1.0) : (-1.0));
01399                 B(j, i) = sgn * det(cofactor(A, i, j));
01400             }
01401         }
01402     }
01403     return B;
01404 }
01405
01420 template<typename T>
01421 Matrix<T> cofactor(const Matrix<T>& A, unsigned p, unsigned q) {
01422     if (! A.issquare()) throw std::runtime_error("Input matrix is not square");
01423     if (! (p < A.rows())) throw std::out_of_range("Row index out of range");
01424     if (! (q < A.cols())) throw std::out_of_range("Column index out of range");
01425     if (A.cols() < 2) throw std::runtime_error("Cofactor calculation requested for matrix with less than
2 rows");
01426
01427     Matrix<T> c(A.rows()-1, A.cols()-1);
01428     unsigned i = 0;
01429     unsigned j = 0;
01430
01431     for (unsigned row = 0; row < A.rows(); row++) {
01432         if (row != p) {
01433             for (unsigned col = 0; col < A.cols(); col++)
01434                 if (col != q) c(i, j++) = A(row, col);
01435             j = 0;
01436             i++;
01437         }
01438     }
01439
01440     return c;
01441 }
01442
01456 template<typename T>
01457 T det_lu(const Matrix<T>& A) {
01458     if (! A.issquare()) throw std::runtime_error("Input matrix is not square");
01459
01460     // LU decomposition with pivoting
01461     auto res = lup(A);
01462
01463     // Determinants of LU
01464     T detLU = static_cast<T>(1);
01465
01466     for (unsigned i = 0; i < res.L.rows(); i++)
01467         detLU *= res.L(i, i) * res.U(i, i);
01468
01469     // Determinant of P
01470     unsigned len = res.P.size();
01471     T detP = static_cast<T>(1);
01472
01473     std::vector<unsigned> p(res.P);
01474     std::vector<unsigned> q;
01475     q.resize(len);
01476
01477     for (unsigned i = 0; i < len; i++)
01478         q[p[i]] = i;
01479
01480     for (unsigned i = 0; i < len; i++) {

```

```

01481     unsigned j = p[i];
01482     unsigned k = q[i];
01483     if (j != i) {
01484         p[k] = p[i];
01485         q[j] = q[i];
01486         detP = - detP;
01487     }
01488 }
01489
01490 return detLU * detP;
01491 }
01492
01503 template<typename T>
01504 T det(const Matrix<T>& A) {
01505     if (! A.issquare()) throw std::runtime_error("Input matrix is not square");
01506
01507     if (A.rows() == 1)
01508         return A(0,0);
01509     else if (A.rows() == 2)
01510         return A(0,0)*A(1,1) - A(0,1)*A(1,0);
01511     else if (A.rows() == 3)
01512         return A(0,0)*(A(1,1)*A(2,2) - A(1,2)*A(2,1)) -
01513             A(0,1)*(A(1,0)*A(2,2) - A(1,2)*A(2,0)) +
01514             A(0,2)*(A(1,0)*A(2,1) - A(1,1)*A(2,0));
01515     else
01516         return det_lu(A);
01517 }
01518
01530 template<typename T>
01531 LU_result<T> lu(const Matrix<T>& A) {
01532     const unsigned M = A.rows();
01533     const unsigned N = A.cols();
01534
01535     LU_result<T> res;
01536     res.L = eye<T>(M);
01537     res.U = Matrix<T>(A);
01538
01539     // aliases
01540     auto& L = res.L;
01541     auto& U = res.U;
01542
01543     if (A.numel() == 0)
01544         return res;
01545
01546     for (unsigned k = 0; k < M-1; k++) {
01547         for (unsigned i = k+1; i < M; i++) {
01548             L(i,k) = U(i,k) / U(k,k);
01549             for (unsigned l = k+1; l < N; l++) {
01550                 U(i,l) -= L(i,k) * U(k,l);
01551             }
01552         }
01553     }
01554
01555     for (unsigned col = 0; col < N; col++)
01556         for (unsigned row = col+1; row < M; row++)
01557             U(row,col) = 0;
01558
01559     return res;
01560 }
01561
01576 template<typename T>
01577 LUP_result<T> lup(const Matrix<T>& A) {
01578     const unsigned M = A.rows();
01579     const unsigned N = A.cols();
01580
01581     // Initialize L, U, and PP
01582     LUP_result<T> res;
01583
01584     if (A.numel() == 0)
01585         return res;
01586
01587     res.L = eye<T>(M);
01588     res.U = Matrix<T>(A);
01589     std::vector<unsigned> PP;
01590
01591     // aliases
01592     auto& L = res.L;
01593     auto& U = res.U;
01594
01595     PP.resize(N);
01596     for (unsigned i = 0; i < N; i++)
01597         PP[i] = i;
01598
01599     for (unsigned k = 0; k < M-1; k++) {
01600         // Find the column with the largest absolute value in the current row
01601         auto max_col_value = std::abs(U(k,k));
01602         unsigned max_col_index = k;

```

```

01603     for (unsigned l = k+1; l < N; l++) {
01604         auto val = std::abs(U(k,l));
01605         if (val > max_col_value) {
01606             max_col_value = val;
01607             max_col_index = l;
01608         }
01609     }
01610
01611     // Swap columns k and max_col_index in U and update P
01612     if (max_col_index != k) {
01613         U.swap_cols(k, max_col_index); // TODO: This could be reworked to avoid column swap in U during
every iteration by:
01614                                     // 1. using PP[k] for column indexing across iterations
01615                                     // 2. doing just one permutation of U at the end
01616         std::swap(PP[k], PP[max_col_index]);
01617     }
01618
01619     // Update L and U
01620     for (unsigned i = k+1; i < M; i++) {
01621         L(i,k) = U(i,k) / U(k,k);
01622         for (unsigned l = k+1; l < N; l++) {
01623             U(i,l) -= L(i,k) * U(k,l);
01624         }
01625     }
01626 }
01627
01628 // Set elements in lower triangular part of U to zero
01629 for (unsigned col = 0; col < N; col++)
01630     for (unsigned row = col+1; row < M; row++)
01631         U(row,col) = 0;
01632
01633 // Transpose indices in permutation vector
01634 res.P.resize(N);
01635 for (unsigned i = 0; i < N; i++)
01636     res.P[PP[i]] = i;
01637
01638 return res;
01639 }
01640
01653 template<typename T>
01654 Matrix<T> inv_gauss_jordan(const Matrix<T>& A) {
01655     if (! A.issquare()) throw std::runtime_error("Input matrix is not square");
01656
01657     const unsigned N = A.rows();
01658     Matrix<T> AA(A);
01659     auto IA = eye<T>(N);
01660
01661     bool found_nonzero;
01662     for (unsigned j = 0; j < N; j++) {
01663         found_nonzero = false;
01664         for (unsigned i = j; i < N; i++) {
01665             if (AA(i,j) != static_cast<T>(0)) {
01666                 found_nonzero = true;
01667                 for (unsigned k = 0; k < N; k++) {
01668                     std::swap(AA(j,k), AA(i,k));
01669                     std::swap(IA(j,k), IA(i,k));
01670                 }
01671                 if (AA(j,j) != static_cast<T>(1)) {
01672                     T s = static_cast<T>(1) / AA(j,j);
01673                     for (unsigned k = 0; k < N; k++) {
01674                         AA(j,k) *= s;
01675                         IA(j,k) *= s;
01676                     }
01677                 }
01678                 for (unsigned l = 0; l < N; l++) {
01679                     if (l != j) {
01680                         T s = AA(l,j);
01681                         for (unsigned k = 0; k < N; k++) {
01682                             AA(l,k) -= s * AA(j,k);
01683                             IA(l,k) -= s * IA(j,k);
01684                         }
01685                     }
01686                 }
01687             }
01688             break;
01689         }
01690         // if a row full of zeros is found, the input matrix was singular
01691         if (!found_nonzero) throw singular_matrix_exception("Singular matrix in inv_gauss_jordan");
01692     }
01693     return IA;
01694 }
01695
01707 template<typename T>
01708 Matrix<T> inv_tril(const Matrix<T>& A) {
01709     if (! A.issquare()) throw std::runtime_error("Input matrix is not square");
01710
01711     const unsigned N = A.rows();

```



```

01712
01713     auto IA = zeros<T>(N);
01714
01715     for (unsigned i = 0; i < N; i++) {
01716         if (A(i,i) == static_cast<T>(0.0)) throw singular_matrix_exception("Division by zero in
inv_tril");
01717
01718         IA(i,i) = static_cast<T>(1.0) / A(i,i);
01719         for (unsigned j = 0; j < i; j++) {
01720             T s = 0.0;
01721             for (unsigned k = j; k < i; k++)
01722                 s += A(i,k) * IA(k,j);
01723             IA(i,j) = -s * IA(i,i);
01724         }
01725     }
01726
01727     return IA;
01728 }
01729
01741 template<typename T>
01742 Matrix<T> inv_triu(const Matrix<T>& A) {
01743     if (!A.issquare()) throw std::runtime_error("Input matrix is not square");
01744
01745     const unsigned N = A.rows();
01746
01747     auto IA = zeros<T>(N);
01748
01749     for (int i = N - 1; i >= 0; i--) {
01750         if (A(i,i) == static_cast<T>(0.0)) throw singular_matrix_exception("Division by zero in
inv_triu");
01751
01752         IA(i,i) = static_cast<T>(1.0) / A(i,i);
01753         for (int j = N - 1; j > i; j--) {
01754             T s = static_cast<T>(0.0);
01755             for (int k = i + 1; k <= j; k++)
01756                 s += A(i,k) * IA(k,j);
01757             IA(i,j) = -s * IA(i,i);
01758         }
01759     }
01760
01761     return IA;
01762 }
01763
01778 template<typename T>
01779 Matrix<T> inv_posdef(const Matrix<T>& A) {
01780     auto L = cholinv(A);
01781     return mult<T,true,false>(L,L);
01782 }
01783
01795 template<typename T>
01796 Matrix<T> inv_square(const Matrix<T>& A) {
01797     if (!A.issquare()) throw std::runtime_error("Input matrix is not square");
01798
01799     // LU decomposition with pivoting
01800     auto LU = lup(A);
01801     auto IL = inv_tril(LU.L);
01802     auto IU = inv_triu(LU.U);
01803
01804     return permute_rows(IU * IL, LU.P);
01805 }
01806
01820 template<typename T>
01821 Matrix<T> inv(const Matrix<T>& A) {
01822     if (!A.issquare()) throw std::runtime_error("Input matrix is not square");
01823
01824     if (A.numel() == 0) {
01825         return Matrix<T>();
01826     } else if (A.rows() < 4) {
01827         T d = det(A);
01828
01829         if (d == static_cast<T>(0.0)) throw singular_matrix_exception("Singular matrix in inv");
01830
01831         Matrix<T> IA(A.rows(), A.rows());
01832         T invdet = static_cast<T>(1.0) / d;
01833
01834         if (A.rows() == 1) {
01835             IA(0,0) = invdet;
01836         } else if (A.rows() == 2) {
01837             IA(0,0) = A(1,1) * invdet;
01838             IA(0,1) = -A(0,1) * invdet;
01839             IA(1,0) = -A(1,0) * invdet;
01840             IA(1,1) = A(0,0) * invdet;
01841         } else if (A.rows() == 3) {
01842             IA(0,0) = (A(1,1)*A(2,2) - A(2,1)*A(1,2)) * invdet;
01843             IA(0,1) = (A(0,2)*A(2,1) - A(0,1)*A(2,2)) * invdet;
01844             IA(0,2) = (A(0,1)*A(1,2) - A(0,2)*A(1,1)) * invdet;
01845             IA(1,0) = (A(1,2)*A(2,0) - A(1,0)*A(2,2)) * invdet;

```

```

01846     IA(1,1) = (A(0,0)*A(2,2) - A(0,2)*A(2,0)) * invdet;
01847     IA(1,2) = (A(1,0)*A(0,2) - A(0,0)*A(1,2)) * invdet;
01848     IA(2,0) = (A(1,0)*A(2,1) - A(2,0)*A(1,1)) * invdet;
01849     IA(2,1) = (A(2,0)*A(0,1) - A(0,0)*A(2,1)) * invdet;
01850     IA(2,2) = (A(0,0)*A(1,1) - A(1,0)*A(0,1)) * invdet;
01851 }
01852
01853 return IA;
01854 } else {
01855     return inv_square(A);
01856 }
01857 }
01858
01870 template<typename T>
01871 Matrix<T> pinv(const Matrix<T>& A) {
01872     if (A.rows() > A.cols()) {
01873         auto AH_A = mult<T,true,false>(A, A);
01874         auto Linv = inv_posdef(AH_A);
01875         return mult<T,false,true>(Linv, A);
01876     } else {
01877         auto AA_H = mult<T,false,true>(A, A);
01878         auto Linv = inv_posdef(AA_H);
01879         return mult<T,true,false>(A, Linv);
01880     }
01881 }
01882
01889 template<typename T>
01890 T trace(const Matrix<T>& A) {
01891     T t = static_cast<T>(0);
01892     for (int i = 0; i < A.rows(); i++)
01893         t += A(i,i);
01894     return t;
01895 }
01896
01907 template<typename T>
01908 double cond(const Matrix<T>& A) {
01909     try {
01910         auto A_inv = inv(A);
01911         return norm_fro(A) * norm_fro(A_inv);
01912     } catch (singular_matrix_exception& e) {
01913         return std::numeric_limits<double>::max();
01914     }
01915 }
01916
01938 template<typename T, bool is_upper = false>
01939 Matrix<T> chol(const Matrix<T>& A) {
01940     if (!A.issquare()) throw std::runtime_error("Input matrix is not square");
01941
01942     const unsigned N = A.rows();
01943
01944     // Calculate lower or upper triangular, depending on template parameter.
01945     // Calculation is the same - the difference is in transposed row and column indexing.
01946     Matrix<T> C = is_upper ? triu(A) : tril(A);
01947
01948     for (unsigned j = 0; j < N; j++) {
01949         if (C(j,j) == static_cast<T>(0.0)) throw singular_matrix_exception("Singular matrix in chol");
01950
01951         C(j,j) = std::sqrt(C(j,j));
01952
01953         for (unsigned k = j+1; k < N; k++)
01954             if (is_upper)
01955                 C(j,k) /= C(j,j);
01956             else
01957                 C(k,j) /= C(j,j);
01958
01959         for (unsigned k = j+1; k < N; k++)
01960             for (unsigned i = k; i < N; i++)
01961                 if (is_upper)
01962                     C(k,i) -= C(j,i) * Util::cconj(C(j,k));
01963                 else
01964                     C(i,k) -= C(i,j) * Util::cconj(C(k,j));
01965     }
01966
01967     return C;
01968 }
01969
01984 template<typename T>
01985 Matrix<T> cholinv(const Matrix<T>& A) {
01986     if (!A.issquare()) throw std::runtime_error("Input matrix is not square");
01987
01988     const unsigned N = A.rows();
01989     Matrix<T> L(A);
01990     auto Linv = eye<T>(N);
01991
01992     for (unsigned j = 0; j < N; j++) {
01993         if (L(j,j) == static_cast<T>(0.0)) throw singular_matrix_exception("Singular matrix in cholinv");
01994

```

```

01995     L(j,j) = static_cast<T>(1.0) / std::sqrt(L(j,j));
01996
01997     for (unsigned k = j+1; k < N; k++)
01998         L(k,j) = L(k,j) * L(j,j);
01999
02000     for (unsigned k = j+1; k < N; k++)
02001         for (unsigned i = k; i < N; i++)
02002             L(i,k) = L(i,k) - L(i,j) * Util::cconj(L(k,j));
02003 }
02004
02005 for (unsigned k = 0; k < N; k++) {
02006     for (unsigned i = k; i < N; i++) {
02007         Linv(i,k) = Linv(i,k) * L(i,i);
02008         for (unsigned j = i+1; j < N; j++)
02009             Linv(j,k) = Linv(j,k) - L(j,i) * Linv(i,k);
02010     }
02011 }
02012
02013 return Linv;
02014 }
02015
02036 template<typename T>
02037 LDL_result<T> ldl(const Matrix<T>& A) {
02038     if (! A.issquare()) throw std::runtime_error("Input matrix is not square");
02039
02040     const unsigned N = A.rows();
02041
02042     LDL_result<T> res;
02043
02044     // aliases
02045     auto& L = res.L;
02046     auto& d = res.d;
02047
02048     L = eye<T>(N);
02049     d.resize(N);
02050
02051     for (unsigned m = 0; m < N; m++) {
02052         d[m] = A(m,m);
02053
02054         for (unsigned k = 0; k < m; k++)
02055             d[m] -= L(m,k) * Util::cconj(L(m,k)) * d[k];
02056
02057         if (d[m] == static_cast<T>(0.0)) throw singular_matrix_exception("Singular matrix in ldl");
02058
02059         for (unsigned n = m+1; n < N; n++) {
02060             L(n,m) = A(n,m);
02061             for (unsigned k = 0; k < m; k++)
02062                 L(n,m) -= L(n,k) * Util::cconj(L(m,k)) * d[k];
02063             L(n,m) /= d[m];
02064         }
02065     }
02066
02067     return res;
02068 }
02069
02083 template<typename T>
02084 QR_result<T> qr_red_gs(const Matrix<T>& A) {
02085     const int rows = A.rows();
02086     const int cols = A.cols();
02087
02088     QR_result<T> res;
02089
02090     //aliases
02091     auto& Q = res.Q;
02092     auto& R = res.R;
02093
02094     Q = zeros<T>(rows, cols);
02095     R = zeros<T>(cols, cols);
02096
02097     for (int c = 0; c < cols; c++) {
02098         Matrix<T> v = A.get_submatrix(0, rows-1, c, c);
02099         for (int r = 0; r < c; r++) {
02100             for (int k = 0; k < rows; k++)
02101                 R(r,c) = R(r,c) + Util::cconj(Q(k,r)) * A(k,c);
02102             for (int k = 0; k < rows; k++)
02103                 v(k) = v(k) - R(r,c) * Q(k,r);
02104         }
02105
02106         R(c,c) = static_cast<T>(norm_fro(v));
02107
02108         if (R(c,c) == static_cast<T>(0.0)) throw singular_matrix_exception("Division by 0 in QR GS");
02109
02110         for (int k = 0; k < rows; k++)
02111             Q(k,c) = v(k) / R(c,c);
02112     }
02113
02114     return res;

```

```

02115 }
02116
02124 template<typename T>
02125 Matrix<T> householder_reflection(const Matrix<T>& a) {
02126     if (a.cols() != 1) throw std::runtime_error("Input not a column vector");
02127
02128     static const T ISQRT2 = static_cast<T>(0.707106781186547);
02129
02130     Matrix<T> v(a);
02131     v(0) += Util::csign(v(0)) * norm_fro(v);
02132     auto vn = norm_fro(v) * ISQRT2;
02133     for (unsigned i = 0; i < v.numel(); i++)
02134         v(i) /= vn;
02135     return v;
02136 }
02137
02152 template<typename T>
02153 QR_result<T> qr_householder(const Matrix<T>& A, bool calculate_Q = true) {
02154     const unsigned rows = A.rows();
02155     const unsigned cols = A.cols();
02156
02157     QR_result<T> res;
02158
02159     //aliases
02160     auto& Q = res.Q;
02161     auto& R = res.R;
02162
02163     R = Matrix<T>(A);
02164
02165     if (calculate_Q)
02166         Q = eye<T>(rows);
02167
02168     const unsigned N = (rows > cols) ? cols : rows;
02169
02170     for (unsigned j = 0; j < N; j++) {
02171         auto v = householder_reflection(R.get_submatrix(j, rows-1, j, j));
02172
02173         auto R1 = R.get_submatrix(j, rows-1, j, cols-1);
02174         auto WR = v * mult<T,true,false>(v, R1);
02175         for (unsigned c = j; c < cols; c++)
02176             for (unsigned r = j; r < rows; r++)
02177                 R(r,c) -= WR(r-j,c-j);
02178
02179         if (calculate_Q) {
02180             auto Q1 = Q.get_submatrix(0, rows-1, j, rows-1);
02181             auto WQ = mult<T,false,true>(Q1 * v, v);
02182             for (unsigned c = j; c < rows; c++)
02183                 for (unsigned r = 0; r < rows; r++)
02184                     Q(r,c) -= WQ(r,c-j);
02185         }
02186     }
02187
02188     for (unsigned col = 0; col < R.cols(); col++)
02189         for (unsigned row = col+1; row < R.rows(); row++)
02190             R(row,col) = 0;
02191
02192     return res;
02193 }
02194
02208 template<typename T>
02209 inline QR_result<T> qr(const Matrix<T>& A, bool calculate_Q = true) {
02210     return qr_householder(A, calculate_Q);
02211 }
02212
02225 template<typename T>
02226 Hessenberg_result<T> hessenberg(const Matrix<T>& A, bool calculate_Q = true) {
02227     if (! A.issquare()) throw std::runtime_error("Input matrix is not square");
02228
02229     Hessenberg_result<T> res;
02230
02231     // aliases
02232     auto& H = res.H;
02233     auto& Q = res.Q;
02234
02235     const unsigned N = A.rows();
02236     H = Matrix<T>(A);
02237
02238     if (calculate_Q)
02239         Q = eye<T>(N);
02240
02241     for (unsigned k = 1; k < N-1; k++) {
02242         auto v = householder_reflection(H.get_submatrix(k, N-1, k-1, k-1));
02243
02244         auto H1 = H.get_submatrix(k, N-1, 0, N-1);
02245         auto W1 = v * mult<T,true,false>(v, H1);
02246         for (unsigned c = 0; c < N; c++)
02247             for (unsigned r = k; r < N; r++)

```

```

02248         H(r,c) -= W1(r-k,c);
02249
02250         auto H2 = H.get_submatrix(0, N-1, k, N-1);
02251         auto W2 = mult<T,false,true>(H2 * v, v);
02252         for (unsigned c = k; c < N; c++)
02253             for (unsigned r = 0; r < N; r++)
02254                 H(r,c) -= W2(r,c-k);
02255
02256         if (calculate_Q) {
02257             auto Q1 = Q.get_submatrix(0, N-1, k, N-1);
02258             auto W3 = mult<T,false,true>(Q1 * v, v);
02259             for (unsigned c = k; c < N; c++)
02260                 for (unsigned r = 0; r < N; r++)
02261                     Q(r,c) -= W3(r,c-k);
02262         }
02263     }
02264
02265     for (unsigned row = 2; row < N; row++)
02266         for (unsigned col = 0; col < row-2; col++)
02267             H(row,col) = static_cast<T>(0);
02268
02269     return res;
02270 }
02271
02281 template<typename T>
02282 std::complex<T> wilkinson_shift(const Matrix<std::complex<T>& H, T tol = 1e-10) {
02283     if (! H.issquare()) throw std::runtime_error("Input matrix is not square");
02284
02285     const unsigned n = H.rows();
02286     std::complex<T> mu;
02287
02288     if (std::abs(H(n-1,n-2)) < tol) {
02289         mu = H(n-2,n-2);
02290     } else {
02291         auto trA = H(n-2,n-2) + H(n-1,n-1);
02292         auto detA = H(n-2,n-2) * H(n-1,n-1) - H(n-2, n-1) * H(n-1, n-2);
02293         mu = (trA + std::sqrt(trA*trA - 4.0*detA)) / 2.0;
02294     }
02295
02296     return mu;
02297 }
02298
02310 template<typename T>
02311 Eigenvalues_result<T> eigenvalues(const Matrix<std::complex<T>& A, T tol = 1e-12, unsigned max_iter =
100) {
02312     if (! A.issquare()) throw std::runtime_error("Input matrix is not square");
02313
02314     const unsigned N = A.rows();
02315     Matrix<std::complex<T>> H;
02316     bool success = false;
02317
02318     QR_result<std::complex<T>> QR;
02319
02320     // aliases
02321     auto& Q = QR.Q;
02322     auto& R = QR.R;
02323
02324     // Transfer A to Hessenberg form to improve convergence (skip calculation of Q)
02325     H = hessenberg(A, false).H;
02326
02327     for (unsigned iter = 0; iter < max_iter; iter++) {
02328         auto mu = wilkinson_shift(H, tol);
02329
02330         // subtract mu from diagonal
02331         for (unsigned n = 0; n < N; n++)
02332             H(n,n) -= mu;
02333
02334         // QR factorization with shifted H
02335         QR = qr(H);
02336         H = R * Q;
02337
02338         // add back mu to diagonal
02339         for (unsigned n = 0; n < N; n++)
02340             H(n,n) += mu;
02341
02342         // Check for convergence
02343         if (std::abs(H(N-2,N-1)) <= tol) {
02344             success = true;
02345             break;
02346         }
02347     }
02348
02349     Eigenvalues_result<T> res;
02350     res.eig = diag(H);
02351     res.err = std::abs(H(N-2,N-1));
02352     res.converged = success;
02353 }

```

```

02354     return res;
02355 }
02356
02357 template<typename T>
02358 Eigenvalues_result<T> eigenvalues(const Matrix<T>& A, T tol = 1e-12, unsigned max_iter = 100) {
02359     auto A_cplx = make_complex(A);
02360     return eigenvalues(A_cplx, tol, max_iter);
02361 }
02362
02363 template<typename T>
02364 Matrix<T> solve_triu(const Matrix<T>& U, const Matrix<T>& B) {
02365     if (! U.isquare()) throw std::runtime_error("Input matrix is not square");
02366     if (U.rows() != B.rows()) throw std::runtime_error("Unmatching matrix dimensions for solve");
02367
02368     const unsigned N = U.rows();
02369     const unsigned M = B.cols();
02370
02371     if (U.numel() == 0)
02372         return Matrix<T>();
02373
02374     Matrix<T> X(B);
02375
02376     for (unsigned m = 0; m < M; m++) {
02377         // backwards substitution for each column of B
02378         for (int n = N-1; n >= 0; n--) {
02379             for (unsigned j = n + 1; j < N; j++)
02380                 X(n,m) -= U(n,j) * X(j,m);
02381
02382             if (U(n,n) == static_cast<T>(0.0)) throw singular_matrix_exception("Singular matrix in
solve_triu");
02383
02384             X(n,m) /= U(n,n);
02385         }
02386     }
02387
02388     return X;
02389 }
02390
02391 template<typename T>
02392 Matrix<T> solve_tril(const Matrix<T>& L, const Matrix<T>& B) {
02393     if (! L.isquare()) throw std::runtime_error("Input matrix is not square");
02394     if (L.rows() != B.rows()) throw std::runtime_error("Unmatching matrix dimensions for solve");
02395
02396     const unsigned N = L.rows();
02397     const unsigned M = B.cols();
02398
02399     if (L.numel() == 0)
02400         return Matrix<T>();
02401
02402     Matrix<T> X(B);
02403
02404     for (unsigned m = 0; m < M; m++) {
02405         // forwards substitution for each column of B
02406         for (unsigned n = 0; n < N; n++) {
02407             for (unsigned j = 0; j < n; j++)
02408                 X(n,m) -= L(n,j) * X(j,m);
02409
02410             if (L(n,n) == static_cast<T>(0.0)) throw singular_matrix_exception("Singular matrix in
solve_tril");
02411
02412             X(n,m) /= L(n,n);
02413         }
02414     }
02415
02416     return X;
02417 }
02418
02419 template<typename T>
02420 Matrix<T> solve_square(const Matrix<T>& A, const Matrix<T>& B) {
02421     if (! A.isquare()) throw std::runtime_error("Input matrix is not square");
02422     if (A.rows() != B.rows()) throw std::runtime_error("Unmatching matrix dimensions for solve");
02423
02424     if (A.numel() == 0)
02425         return Matrix<T>();
02426
02427     Matrix<T> L;
02428     Matrix<T> U;
02429     std::vector<unsigned> P;
02430
02431     // LU decomposition with pivoting
02432     auto lup_res = lup(A);
02433
02434     auto y = solve_tril(lup_res.L, B);
02435     auto x = solve_triu(lup_res.U, y);
02436
02437     return permute_rows(x, lup_res.P);
02438 }
02439
02440 }

```

```

02496
02513 template<typename T>
02514 Matrix<T> solve_posdef(const Matrix<T>& A, const Matrix<T>& B) {
02515     if (! A.issquare()) throw std::runtime_error("Input matrix is not square");
02516     if (A.rows() != B.rows()) throw std::runtime_error("Unmatching matrix dimensions for solve");
02517
02518     if (A.numel() == 0)
02519         return Matrix<T>();
02520
02521     // LU decomposition with pivoting
02522     auto L = chol(A);
02523
02524     auto Y = solve_tril(L, B);
02525     return solve_triu(L.ctranspose(), Y);
02526 }
02527
02532 template<typename T>
02533 class Matrix {
02534 public:
02539     Matrix();
02540
02545     Matrix(unsigned size);
02546
02551     Matrix(unsigned nrows, unsigned ncols);
02552
02557     Matrix(T x, unsigned nrows, unsigned ncols);
02558
02563     Matrix(const T* array, unsigned nrows, unsigned ncols);
02566
02571     Matrix(const std::vector<T>& vec, unsigned nrows, unsigned ncols);
02572
02577     Matrix(std::initializer_list<T> init_list, unsigned nrows, unsigned ncols);
02578
02583     Matrix(const Matrix &);
02584
02589     virtual ~Matrix();
02590
02595     Matrix<T> get_submatrix(unsigned row_first, unsigned row_last, unsigned col_first, unsigned
02600 col_last) const;
02601
02606     void set_submatrix(const Matrix<T>& smtx, unsigned row_first, unsigned col_first);
02607
02612     void clear();
02613
02618     void reshape(unsigned rows, unsigned cols);
02619
02624     void resize(unsigned rows, unsigned cols);
02625
02630     bool exists(unsigned row, unsigned col) const;
02631
02636     T* ptr(unsigned row, unsigned col);
02637
02642     T* ptr();
02643
02648     void fill(T value);
02649
02654     void fill_col(T value, unsigned col);
02655
02660     void fill_row(T value, unsigned row);
02661
02666     bool isempty() const;
02667
02672     bool issquare() const;
02673
02678     bool isequal(const Matrix<T>&) const;
02679
02684     bool isequal(const Matrix<T>&, T) const;
02685
02690     unsigned numel() const;
02691
02696     unsigned rows() const;
02697
02702     unsigned cols() const;
02703
02708     std::pair<unsigned,unsigned> shape() const;
02709
02714     Matrix<T> transpose() const;
02715
02720     Matrix<T> ctranspose() const;
02721
02726     Matrix<T>& add(const Matrix<T>&);
02727
02732     Matrix<T>& subtract(const Matrix<T>&);
02733
02738     Matrix<T>& mult_hadamard(const Matrix<T>&);
02739
02744     Matrix<T>& add(T);

```

```

02786
02793     Matrix<T>& subtract(T);
02794
02801     Matrix<T>& mult(T);
02802
02809     Matrix<T>& div(T);
02810
02815     Matrix<T>& operator=(const Matrix<T>&);
02816
02822     Matrix<T>& operator=(T);
02823
02829     explicit operator std::vector<T>() const;
02830     std::vector<T> to_vector() const;
02831
02838     T& operator()(unsigned nel);
02839     T operator()(unsigned nel) const;
02840     T& at(unsigned nel);
02841     T at(unsigned nel) const;
02842
02849     T& operator()(unsigned row, unsigned col);
02850     T operator()(unsigned row, unsigned col) const;
02851     T& at(unsigned row, unsigned col);
02852     T at(unsigned row, unsigned col) const;
02853
02861     void add_row_to_another(unsigned to, unsigned from);
02862
02870     void add_col_to_another(unsigned to, unsigned from);
02871
02879     void mult_row_by_another(unsigned to, unsigned from);
02880
02888     void mult_col_by_another(unsigned to, unsigned from);
02889
02896     void swap_rows(unsigned i, unsigned j);
02897
02904     void swap_cols(unsigned i, unsigned j);
02905
02912     std::vector<T> col_to_vector(unsigned col) const;
02913
02920     std::vector<T> row_to_vector(unsigned row) const;
02921
02930     void col_from_vector(const std::vector<T>&, unsigned col);
02931
02940     void row_from_vector(const std::vector<T>&, unsigned row);
02941
02942 private:
02943     unsigned nrows;
02944     unsigned ncols;
02945     std::vector<T> data;
02946 };
02947
02948 /*
02949  * Implementation of Matrix class methods
02950  */
02951
02952 template<typename T>
02953 Matrix<T>::Matrix() : nrows(0), ncols(0), data() { }
02954
02955 template<typename T>
02956 Matrix<T>::Matrix(unsigned size) : Matrix(size, size) { }
02957
02958 template<typename T>
02959 Matrix<T>::Matrix(unsigned rows, unsigned cols) : nrows(rows), ncols(cols) {
02960     data.resize(numel());
02961 }
02962
02963 template<typename T>
02964 Matrix<T>::Matrix(T x, unsigned rows, unsigned cols) : Matrix(rows, cols) {
02965     fill(x);
02966 }
02967
02968 template<typename T>
02969 Matrix<T>::Matrix(const T* array, unsigned rows, unsigned cols) : Matrix(rows, cols) {
02970     data.assign(array, array + numel());
02971 }
02972
02973 template<typename T>
02974 Matrix<T>::Matrix(const std::vector<T>& vec, unsigned rows, unsigned cols) : Matrix(rows, cols) {
02975     if (vec.size() != numel()) throw std::runtime_error("Size of initialization vector not consistent
02976         with matrix dimensions");
02977     data.assign(vec.begin(), vec.end());
02978 }
02979
02980 template<typename T>
02981 Matrix<T>::Matrix(std::initializer_list<T> init_list, unsigned rows, unsigned cols) : Matrix(rows,
02982     cols) {
02983     if (init_list.size() != numel()) throw std::runtime_error("Size of initialization list not

```



```

        consistent with matrix dimensions");
02983
02984     auto it = init_list.begin();
02985
02986     for (unsigned row = 0; row < this->nrows; row++)
02987         for (unsigned col = 0; col < this->ncols; col++)
02988             this->at(row,col) = *(it++);
02989 }
02990
02991 template<typename T>
02992 Matrix<T>::Matrix(const Matrix & other) : Matrix(other.nrows, other.ncols) {
02993     this->data.assign(other.data.begin(), other.data.end());
02994 }
02995
02996 template<typename T>
02997 Matrix<T> & Matrix<T>::operator=(const Matrix<T> & other) {
02998     this->nrows = other.nrows;
02999     this->ncols = other.ncols;
03000     this->data.assign(other.data.begin(), other.data.end());
03001     return *this;
03002 }
03003
03004 template<typename T>
03005 Matrix<T> & Matrix<T>::operator=(T s) {
03006     fill(s);
03007     return *this;
03008 }
03009
03010 template<typename T>
03011 inline Matrix<T>::operator std::vector<T>() const {
03012     return data;
03013 }
03014
03015 template<typename T>
03016 inline void Matrix<T>::clear() {
03017     this->nrows = 0;
03018     this->ncols = 0;
03019     data.resize(0);
03020 }
03021
03022 template<typename T>
03023 void Matrix<T>::reshape(unsigned rows, unsigned cols) {
03024     if (this->numel() != rows * cols) throw std::runtime_error("Illegal attempt to change number of
elements via reshape");
03025
03026     this->nrows = rows;
03027     this->ncols = cols;
03028 }
03029
03030 template<typename T>
03031 void Matrix<T>::resize(unsigned rows, unsigned cols) {
03032     this->nrows = rows;
03033     this->ncols = cols;
03034     data.resize(nrows*ncols);
03035 }
03036
03037 template<typename T>
03038 Matrix<T> Matrix<T>::get_submatrix(unsigned row_base, unsigned row_lim, unsigned col_base, unsigned
col_lim) const {
03039     if (row_base > row_lim) throw std::out_of_range("Row index of submatrix out of range");
03040     if (col_base > col_lim) throw std::out_of_range("Column index of submatrix out of range");
03041     if (row_lim >= this->rows()) throw std::out_of_range("Row index of submatrix out of range");
03042     if (col_lim >= this->cols()) throw std::out_of_range("Column index of submatrix out of range");
03043
03044     unsigned num_rows = row_lim - row_base + 1;
03045     unsigned num_cols = col_lim - col_base + 1;
03046     Matrix<T> S(num_rows, num_cols);
03047     for (unsigned i = 0; i < num_rows; i++) {
03048         for (unsigned j = 0; j < num_cols; j++) {
03049             S(i,j) = at(row_base + i, col_base + j);
03050         }
03051     }
03052     return S;
03053 }
03054
03055 template<typename T>
03056 void Matrix<T>::set_submatrix(const Matrix<T> & S, unsigned row_base, unsigned col_base) {
03057     if (this->isempty()) throw std::runtime_error("Invalid attempt to set submatrix in empty matrix");
03058
03059     const unsigned row_lim = row_base + S.rows() - 1;
03060     const unsigned col_lim = col_base + S.cols() - 1;
03061
03062     if (row_base > row_lim) throw std::out_of_range("Row index of submatrix out of range");
03063     if (col_base > col_lim) throw std::out_of_range("Column index of submatrix out of range");
03064     if (row_lim >= this->rows()) throw std::out_of_range("Row index of submatrix out of range");
03065     if (col_lim >= this->cols()) throw std::out_of_range("Column index of submatrix out of range");
03066

```

```

03067     unsigned num_rows = row_lim - row_base + 1;
03068     unsigned num_cols = col_lim - col_base + 1;
03069     for (unsigned i = 0; i < num_rows; i++)
03070         for (unsigned j = 0; j < num_cols; j++)
03071             at(row_base + i, col_base + j) = S(i,j);
03072 }
03073
03074 template<typename T>
03075 inline T & Matrix<T>::operator()(unsigned nel) {
03076     return at(nel);
03077 }
03078
03079 template<typename T>
03080 inline T & Matrix<T>::operator()(unsigned row, unsigned col) {
03081     return at(row, col);
03082 }
03083
03084 template<typename T>
03085 inline T Matrix<T>::operator()(unsigned nel) const {
03086     return at(nel);
03087 }
03088
03089 template<typename T>
03090 inline T Matrix<T>::operator()(unsigned row, unsigned col) const {
03091     return at(row, col);
03092 }
03093
03094 template<typename T>
03095 inline T & Matrix<T>::at(unsigned nel) {
03096     if (!(nel < numel())) throw std::out_of_range("Element index out of range");
03097     return data[nel];
03098 }
03099
03100
03101 template<typename T>
03102 inline T & Matrix<T>::at(unsigned row, unsigned col) {
03103     if (!(row < rows() && col < cols())) throw std::out_of_range("Element index out of range");
03104     return data[nrows * col + row];
03105 }
03106
03107
03108 template<typename T>
03109 inline T Matrix<T>::at(unsigned nel) const {
03110     if (!(nel < numel())) throw std::out_of_range("Element index out of range");
03111     return data[nel];
03112 }
03113
03114
03115 template<typename T>
03116 inline T Matrix<T>::at(unsigned row, unsigned col) const {
03117     if (!(row < rows())) throw std::out_of_range("Row index out of range");
03118     if (!(col < cols())) throw std::out_of_range("Column index out of range");
03119     return data[nrows * col + row];
03120 }
03121
03122
03123 template<typename T>
03124 inline void Matrix<T>::fill(T value) {
03125     for (unsigned i = 0; i < numel(); i++)
03126         data[i] = value;
03127 }
03128
03129 template<typename T>
03130 inline void Matrix<T>::fill_col(T value, unsigned col) {
03131     if (!(col < cols())) throw std::out_of_range("Column index out of range");
03132     for (unsigned i = col * nrows; i < (col+1) * nrows; i++)
03133         data[i] = value;
03134 }
03135
03136
03137 template<typename T>
03138 inline void Matrix<T>::fill_row(T value, unsigned row) {
03139     if (!(row < rows())) throw std::out_of_range("Row index out of range");
03140     for (unsigned i = 0; i < ncols; i++)
03141         data[row + i * nrows] = value;
03142 }
03143
03144
03145 template<typename T>
03146 inline bool Matrix<T>::exists(unsigned row, unsigned col) const {
03147     return (row < nrows && col < ncols);
03148 }
03149
03150 template<typename T>
03151 inline T* Matrix<T>::ptr(unsigned row, unsigned col) {
03152     if (!(row < rows())) throw std::out_of_range("Row index out of range");
03153     if (!(col < cols())) throw std::out_of_range("Column index out of range");

```

```

03154
03155     return data.data() + nrows * col + row;
03156 }
03157
03158 template<typename T>
03159 inline T* Matrix<T>::ptr() {
03160     return data.data();
03161 }
03162
03163 template<typename T>
03164 inline bool Matrix<T>::isempty() const {
03165     return (nrows == 0) || (ncols == 0);
03166 }
03167
03168 template<typename T>
03169 inline bool Matrix<T>::issquare() const {
03170     return (nrows == ncols) && !isempty();
03171 }
03172
03173 template<typename T>
03174 bool Matrix<T>::isequal(const Matrix<T>& A) const {
03175     bool ret = true;
03176     if (nrows != A.rows() || ncols != A.cols()) {
03177         ret = false;
03178     } else {
03179         for (unsigned i = 0; i < numel(); i++) {
03180             if (at(i) != A(i)) {
03181                 ret = false;
03182                 break;
03183             }
03184         }
03185     }
03186     return ret;
03187 }
03188
03189 template<typename T>
03190 bool Matrix<T>::isequal(const Matrix<T>& A, T tol) const {
03191     bool ret = true;
03192     if (rows() != A.rows() || cols() != A.cols()) {
03193         ret = false;
03194     } else {
03195         auto abs_tol = std::abs(tol); // workaround for complex
03196         for (unsigned i = 0; i < A.numel(); i++) {
03197             if (abs_tol < std::abs(at(i) - A(i))) {
03198                 ret = false;
03199                 break;
03200             }
03201         }
03202     }
03203     return ret;
03204 }
03205
03206 template<typename T>
03207 inline unsigned Matrix<T>::numel() const {
03208     return nrows * ncols;
03209 }
03210
03211 template<typename T>
03212 inline unsigned Matrix<T>::rows() const {
03213     return nrows;
03214 }
03215
03216 template<typename T>
03217 inline unsigned Matrix<T>::cols() const {
03218     return ncols;
03219 }
03220
03221 template<typename T>
03222 inline std::pair<unsigned,unsigned> Matrix<T>::shape() const {
03223     return std::pair<unsigned,unsigned>(nrows,ncols);
03224 }
03225
03226 template<typename T>
03227 inline Matrix<T> Matrix<T>::transpose() const {
03228     Matrix<T> res(ncols, nrows);
03229     for (unsigned c = 0; c < ncols; c++)
03230         for (unsigned r = 0; r < nrows; r++)
03231             res(c,r) = at(r,c);
03232     return res;
03233 }
03234
03235 template<typename T>
03236 inline Matrix<T> Matrix<T>::ctranspose() const {
03237     Matrix<T> res(ncols, nrows);
03238     for (unsigned c = 0; c < ncols; c++)
03239         for (unsigned r = 0; r < nrows; r++)
03240             res(c,r) = Util::cconj(at(r,c));

```

```

03241     return res;
03242 }
03243
03244 template<typename T>
03245 Matrix<T>& Matrix<T>::add(const Matrix<T>& m) {
03246     if (!(m.rows() == rows() && m.cols() == cols())) throw std::runtime_error("Unmatching matrix
dimensions for iadd");
03247
03248     for (unsigned i = 0; i < numel(); i++)
03249         data[i] += m(i);
03250     return *this;
03251 }
03252
03253 template<typename T>
03254 Matrix<T>& Matrix<T>::subtract(const Matrix<T>& m) {
03255     if (!(m.rows() == rows() && m.cols() == cols())) throw std::runtime_error("Unmatching matrix
dimensions for isubtract");
03256
03257     for (unsigned i = 0; i < numel(); i++)
03258         data[i] -= m(i);
03259     return *this;
03260 }
03261
03262 template<typename T>
03263 Matrix<T>& Matrix<T>::mult_hadamard(const Matrix<T>& m) {
03264     if (!(m.rows() == rows() && m.cols() == cols())) throw std::runtime_error("Unmatching matrix
dimensions for ihprod");
03265
03266     for (unsigned i = 0; i < numel(); i++)
03267         data[i] *= m(i);
03268     return *this;
03269 }
03270
03271 template<typename T>
03272 Matrix<T>& Matrix<T>::add(T s) {
03273     for (auto& x : data)
03274         x += s;
03275     return *this;
03276 }
03277
03278 template<typename T>
03279 Matrix<T>& Matrix<T>::subtract(T s) {
03280     for (auto& x : data)
03281         x -= s;
03282     return *this;
03283 }
03284
03285 template<typename T>
03286 Matrix<T>& Matrix<T>::mult(T s) {
03287     for (auto& x : data)
03288         x *= s;
03289     return *this;
03290 }
03291
03292 template<typename T>
03293 Matrix<T>& Matrix<T>::div(T s) {
03294     for (auto& x : data)
03295         x /= s;
03296     return *this;
03297 }
03298
03299 template<typename T>
03300 void Matrix<T>::add_row_to_another(unsigned to, unsigned from) {
03301     if (!(to < rows() && from < rows())) throw std::out_of_range("Row index out of range");
03302
03303     for (unsigned k = 0; k < cols(); k++)
03304         at(to, k) += at(from, k);
03305 }
03306
03307 template<typename T>
03308 void Matrix<T>::add_col_to_another(unsigned to, unsigned from) {
03309     if (!(to < cols() && from < cols())) throw std::out_of_range("Column index out of range");
03310
03311     for (unsigned k = 0; k < rows(); k++)
03312         at(k, to) += at(k, from);
03313 }
03314
03315 template<typename T>
03316 void Matrix<T>::mult_row_by_another(unsigned to, unsigned from) {
03317     if (!(to < rows() && from < rows())) throw std::out_of_range("Row index out of range");
03318
03319     for (unsigned k = 0; k < cols(); k++)
03320         at(to, k) *= at(from, k);
03321 }
03322
03323 template<typename T>
03324 void Matrix<T>::mult_col_by_another(unsigned to, unsigned from) {

```

```

03325     if (!(to < cols() && from < cols())) throw std::out_of_range("Column index out of range");
03326
03327     for (unsigned k = 0; k < rows(); k++)
03328         at(k, to) *= at(k, from);
03329 }
03330
03331 template<typename T>
03332 void Matrix<T>::swap_rows(unsigned i, unsigned j) {
03333     if (!(i < rows() && j < rows())) throw std::out_of_range("Row index out of range");
03334
03335     for (unsigned k = 0; k < cols(); k++) {
03336         T tmp = at(i,k);
03337         at(i,k) = at(j,k);
03338         at(j,k) = tmp;
03339     }
03340 }
03341
03342 template<typename T>
03343 void Matrix<T>::swap_cols(unsigned i, unsigned j) {
03344     if (!(i < cols() && j < cols())) throw std::out_of_range("Column index out of range");
03345
03346     for (unsigned k = 0; k < rows(); k++) {
03347         T tmp = at(k,i);
03348         at(k,i) = at(k,j);
03349         at(k,j) = tmp;
03350     }
03351 }
03352
03353 template<typename T>
03354 inline std::vector<T> Matrix<T>::to_vector() const {
03355     return data;
03356 }
03357
03358 template<typename T>
03359 inline std::vector<T> Matrix<T>::col_to_vector(unsigned col) const {
03360     std::vector<T> vec(rows());
03361     for (unsigned i = 0; i < rows(); i++)
03362         vec[i] = at(i,col);
03363     return vec;
03364 }
03365
03366 template<typename T>
03367 inline std::vector<T> Matrix<T>::row_to_vector(unsigned row) const {
03368     std::vector<T> vec(cols());
03369     for (unsigned i = 0; i < cols(); i++)
03370         vec[i] = at(row,i);
03371     return vec;
03372 }
03373
03374 template<typename T>
03375 inline void Matrix<T>::col_from_vector(const std::vector<T>& vec, unsigned col) {
03376     if (vec.size() != rows()) throw std::runtime_error("Vector size is not equal to number of rows");
03377     if (col >= cols()) throw std::out_of_range("Column index out of range");
03378
03379     for (unsigned i = 0; i < rows(); i++)
03380         data[col*rows() + i] = vec[i];
03381 }
03382
03383 template<typename T>
03384 inline void Matrix<T>::row_from_vector(const std::vector<T>& vec, unsigned row) {
03385     if (vec.size() != cols()) throw std::runtime_error("Vector size is not equal to number of columns");
03386     if (row >= rows()) throw std::out_of_range("Row index out of range");
03387
03388     for (unsigned i = 0; i < cols(); i++)
03389         data[row + i*rows()] = vec[i];
03390 }
03391
03392 template<typename T>
03393 Matrix<T>::~Matrix() { }
03394
03395 } // namespace Matrix_hpp
03396
03397 #endif // __MATRIX_HPP__

```

