

Matrix HPP

Generated by Doxygen 1.9.8

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

5.6.3.5 clear()	19
5.6.3.6 col_from_vector()	19
5.6.3.7 col_to_vector()	19
5.6.3.8 cols()	20
5.6.3.9 ctranspose()	20
5.6.3.10 div()	20
5.6.3.11 exists()	21
5.6.3.12 fill()	21
5.6.3.13 fill_col()	21
5.6.3.14 fill_row()	21
5.6.3.15 get_submatrix()	22
5.6.3.16 isempty()	22
5.6.3.17 isequal() [1/2]	22
5.6.3.18 isequal() [2/2]	23
5.6.3.19 mult()	23
5.6.3.20 mult_col_by_another()	23
5.6.3.21 mult_hadamard()	23
5.6.3.22 mult_row_by_another()	24
5.6.3.23 numel()	24
5.6.3.24 operator std::vector< T >()	24
5.6.3.25 operator>() [1/2]	25
5.6.3.26 operator>() [2/2]	25
5.6.3.27 operator=() [1/2]	25
5.6.3.28 operator=() [2/2]	25
5.6.3.29 ptr() [1/2]	26
5.6.3.30 ptr() [2/2]	26
5.6.3.31 reshape()	26
5.6.3.32 resize()	26
5.6.3.33 row_from_vector()	27
5.6.3.34 row_to_vector()	27
5.6.3.35 rows()	27
5.6.3.36 set_submatrix()	28
5.6.3.37 shape()	28
5.6.3.38 subtract() [1/2]	29
5.6.3.39 subtract() [2/2]	29
5.6.3.40 swap_cols()	29
5.6.3.41 swap_rows()	30
5.6.3.42 transpose()	30
5.7 Mtx::QR_result< T > Struct Template Reference	30
5.7.1 Detailed Description	31
5.8 Mtx::singular_matrix_exception Class Reference	31

6 File Documentation	33
6.1 matrix.hpp File Reference	33
6.1.1 Function Documentation	39
6.1.1.1 add() [1/2]	39
6.1.1.2 add() [2/2]	40
6.1.1.3 adj()	40
6.1.1.4 cconj()	40
6.1.1.5 chol()	41
6.1.1.6 cholinv()	41
6.1.1.7 circshift()	42
6.1.1.8 circulant() [1/2]	42
6.1.1.9 circulant() [2/2]	43
6.1.1.10 cofactor()	43
6.1.1.11 concatenate_horizontal()	44
6.1.1.12 concatenate_vertical()	44
6.1.1.13 cond()	44
6.1.1.14 csign()	45
6.1.1.15 ctranspose()	45
6.1.1.16 det()	45
6.1.1.17 det_lu()	46
6.1.1.18 diag() [1/3]	46
6.1.1.19 diag() [2/3]	47
6.1.1.20 diag() [3/3]	47
6.1.1.21 div()	48
6.1.1.22 eigenvalues() [1/2]	48
6.1.1.23 eigenvalues() [2/2]	48
6.1.1.24 eye()	49
6.1.1.25 foreach_elem()	49
6.1.1.26 foreach_elem_copy()	50
6.1.1.27 hessenberg()	50
6.1.1.28 householder_reflection()	51
6.1.1.29 imag()	51
6.1.1.30 inv()	52
6.1.1.31 inv_gauss_jordan()	52
6.1.1.32 inv_posdef()	52
6.1.1.33 inv_square()	53
6.1.1.34 inv_tril()	53
6.1.1.35 inv_triu()	54
6.1.1.36 ishess()	54
6.1.1.37 istril()	55
6.1.1.38 istriu()	55
6.1.1.39 kron()	55

6.1.1.40 <code>ldl()</code>	55
6.1.1.41 <code>lu()</code>	56
6.1.1.42 <code>lup()</code>	56
6.1.1.43 <code>make_complex()</code> [1/2]	57
6.1.1.44 <code>make_complex()</code> [2/2]	57
6.1.1.45 <code>mult()</code> [1/4]	58
6.1.1.46 <code>mult()</code> [2/4]	59
6.1.1.47 <code>mult()</code> [3/4]	59
6.1.1.48 <code>mult()</code> [4/4]	60
6.1.1.49 <code>mult_hadamard()</code>	61
6.1.1.50 <code>norm_fro()</code> [1/2]	62
6.1.1.51 <code>norm_fro()</code> [2/2]	62
6.1.1.52 <code>ones()</code> [1/2]	62
6.1.1.53 <code>ones()</code> [2/2]	63
6.1.1.54 <code>operator"!=()</code>	63
6.1.1.55 <code>operator*()</code> [1/5]	63
6.1.1.56 <code>operator*()</code> [2/5]	64
6.1.1.57 <code>operator*()</code> [3/5]	64
6.1.1.58 <code>operator*()</code> [4/5]	64
6.1.1.59 <code>operator*()</code> [5/5]	64
6.1.1.60 <code>operator*==()</code> [1/2]	65
6.1.1.61 <code>operator*==()</code> [2/2]	65
6.1.1.62 <code>operator+()</code> [1/3]	65
6.1.1.63 <code>operator+()</code> [2/3]	65
6.1.1.64 <code>operator+()</code> [3/3]	66
6.1.1.65 <code>operator+=()</code> [1/2]	66
6.1.1.66 <code>operator+=()</code> [2/2]	66
6.1.1.67 <code>operator-()</code> [1/2]	66
6.1.1.68 <code>operator-()</code> [2/2]	67
6.1.1.69 <code>operator-=()</code> [1/2]	67
6.1.1.70 <code>operator-=()</code> [2/2]	67
6.1.1.71 <code>operator/()</code>	67
6.1.1.72 <code>operator/=()</code>	68
6.1.1.73 <code>operator<<()</code>	68
6.1.1.74 <code>operator==()</code>	68
6.1.1.75 <code>operator^()</code>	68
6.1.1.76 <code>operator^=()</code>	69
6.1.1.77 <code>permute_cols()</code>	69
6.1.1.78 <code>permute_rows()</code>	69
6.1.1.79 <code>permute_rows_and_cols()</code>	70
6.1.1.80 <code>pinv()</code>	71
6.1.1.81 <code>qr()</code>	71

6.1.1.82 qr_householder()	72
6.1.1.83 qr_red_gs()	72
6.1.1.84 real()	73
6.1.1.85 repmat()	73
6.1.1.86 solve_posdef()	74
6.1.1.87 solve_square()	74
6.1.1.88 solve_tril()	75
6.1.1.89 solve_triu()	76
6.1.1.90 subtract() [1/2]	76
6.1.1.91 subtract() [2/2]	77
6.1.1.92 trace()	77
6.1.1.93 transpose()	78
6.1.1.94 tril()	78
6.1.1.95 triu()	78
6.1.1.96 wilkinson_shift()	78
6.1.1.97 zeros() [1/2]	79
6.1.1.98 zeros() [2/2]	79
6.2 matrix.hpp	80

Chapter 1

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

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

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

1.1 Installation

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

1.2 Functionality

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

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

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

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

1.3 Hello world example

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

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

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

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

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

    auto C = A + B;

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

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

1.4 Tests

Unit tests are compiled with `make tests`.

1.5 License

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

Chapter 2

Hierarchical Index

2.1 Class Hierarchy

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

std::domain_error	
Mtx::singular_matrix_exception	31
Mtx::Eigenvalues_result< T >	9
Mtx::Hessenberg_result< T >	9
Mtx::LDL_result< T >	10
Mtx::LU_result< T >	11
Mtx::LUP_result< T >	11
Mtx::Matrix< T >	12
Mtx::QR_result< T >	30

Chapter 3

Class Index

3.1 Class List

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

Mtx::Eigenvalues_result< T >	
Result of eigenvalues	9
Mtx::Hessenberg_result< T >	
Result of Hessenberg decomposition	9
Mtx::LDL_result< T >	
Result of LDL decomposition	10
Mtx::LU_result< T >	
Result of LU decomposition	11
Mtx::LUP_result< T >	
Result of LU decomposition with pivoting	11
Mtx::Matrix< T >	12
Mtx::QR_result< T >	
Result of QR decomposition	30
Mtx::singular_matrix_exception	
Singular matrix exception	31

Chapter 4

File Index

4.1 File List

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

matrix.hpp	33
--------------------------------------	----

Chapter 5

Class Documentation

5.1 Mtx::Eigenvalues_result< T > Struct Template Reference

Result of eigenvalues.

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

Public Attributes

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

5.1.1 Detailed Description

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

Result of eigenvalues.

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

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

- [matrix.hpp](#)

5.2 Mtx::Hessenberg_result< T > Struct Template Reference

Result of Hessenberg decomposition.

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

Public Attributes

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

5.2.1 Detailed Description

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

Result of Hessenberg decomposition.

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

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

- [matrix.hpp](#)

5.3 Mtx::LDL_result< T > Struct Template Reference

Result of LDL decomposition.

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

Public Attributes

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

5.3.1 Detailed Description

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

Result of LDL decomposition.

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

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

- [matrix.hpp](#)

5.4 Mtx::LU_result< T > Struct Template Reference

Result of LU decomposition.

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

Public Attributes

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

5.4.1 Detailed Description

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

Result of LU decomposition.

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

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

- [matrix.hpp](#)

5.5 Mtx::LUP_result< T > Struct Template Reference

Result of LU decomposition with pivoting.

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

Public Attributes

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

5.5.1 Detailed Description

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

Result of LU decomposition with pivoting.

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

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

- [matrix.hpp](#)

5.6 Mtx::Matrix< T > Class Template Reference

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

Public Member Functions

- [Matrix \(\)](#)
Default constructor.
- [Matrix \(unsigned size\)](#)
Square matrix constructor.
- [Matrix \(unsigned nrows, unsigned ncols\)](#)
Rectangular matrix constructor.
- [Matrix \(T x, unsigned nrows, unsigned ncols\)](#)
Rectangular matrix constructor with fill.
- [Matrix \(const T *array, unsigned nrows, unsigned ncols\)](#)
Rectangular matrix constructor with initialization.
- [Matrix \(const std::vector< T > &vec, unsigned nrows, unsigned ncols\)](#)
Rectangular matrix constructor with initialization.
- [Matrix \(std::initializer_list< T > init_list, unsigned nrows, unsigned ncols\)](#)
Rectangular matrix constructor with initialization.
- [Matrix \(const Matrix &\)](#)
- [virtual ~Matrix \(\)](#)
- [Matrix< T > get_submatrix \(unsigned row_first, unsigned row_last, unsigned col_first, unsigned col_last\) const](#)
Extract a submatrix.
- [void set_submatrix \(const Matrix< T > &smtx, unsigned row_first, unsigned col_first\)](#)
Embed a submatrix.
- [void clear \(\)](#)
Clears the matrix.
- [void reshape \(unsigned rows, unsigned cols\)](#)
Matrix dimension reshape.
- [void resize \(unsigned rows, unsigned cols\)](#)
Resize the matrix.
- [bool exists \(unsigned row, unsigned col\) const](#)
Element exist check.
- [T * ptr \(unsigned row, unsigned col\)](#)

- *Memory pointer.*
- `T * ptr ()`
- *Memory pointer.*
- `void fill (T value)`
- `void fill_col (T value, unsigned col)`
- *Fill column with a scalar.*
- `void fill_row (T value, unsigned row)`
- *Fill row with a scalar.*
- `bool isempty () const`
- *Emptiness check.*
- `bool issquare () const`
- *Squareness check. Check if the matrix is square, i.e., the width of the first and the second dimensions are equal.*
- `bool isequal (const Matrix< T > &) const`
- *Matrix equality check.*
- `bool isequal (const Matrix< T > &, T) const`
- *Matrix equality check with tolerance.*
- `unsigned numel () const`
- *Matrix capacity.*
- `unsigned rows () const`
- *Number of rows.*
- `unsigned cols () const`
- *Number of columns.*
- `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.

5.6.1 Detailed Description

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

[Matrix](#) class definition.

5.6.2 Constructor & Destructor Documentation

5.6.2.1 Matrix() [1/8]

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

Default constructor.

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

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

5.6.2.2 Matrix() [2/8]

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

Square matrix constructor.

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

5.6.2.3 Matrix() [3/8]

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

Rectangular matrix constructor.

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

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

5.6.2.4 Matrix() [4/8]

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

Rectangular matrix constructor with fill.

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

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

5.6.2.5 Matrix() [5/8]

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

Rectangular matrix constructor with initialization.

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

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

5.6.2.6 Matrix() [6/8]

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

Rectangular matrix constructor with initialization.

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

Exceptions

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

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

5.6.2.7 Matrix() [7/8]

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

Rectangular matrix constructor with initialization.

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

Exceptions

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

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

5.6.2.8 Matrix() [8/8]

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

Copy constructor.

5.6.2.9 ~Matrix()

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

Destructor.

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

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

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

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

Exceptions

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

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

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

5.6.3.2 add() [2/2]

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

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

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

5.6.3.3 add_col_to_another()

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

Column addition.

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

Exceptions

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

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

5.6.3.4 add_row_to_another()

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

Row addition.

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

Exceptions

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

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

5.6.3.5 clear()

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

Clears the matrix.

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

5.6.3.6 col_from_vector()

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

Column from vector.

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

Exceptions

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

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

5.6.3.7 col_to_vector()

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

Column to vector.

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

Exceptions

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

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

5.6.3.8 cols()

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

Number of columns.

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

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

5.6.3.9 ctranspose()

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

Transpose a complex matrix.

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

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

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

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

5.6.3.10 div()

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

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

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

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

5.6.3.11 exists()

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

Element exist check.

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

5.6.3.12 fill()

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

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

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

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

5.6.3.13 fill_col()

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

Fill column with a scalar.

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

Exceptions

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

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

5.6.3.14 fill_row()

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

Fill row with a scalar.

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

Exceptions

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

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

5.6.3.15 `get_submatrix()`

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

Extract a submatrix.

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

Exceptions

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

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

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

5.6.3.16 `isempty()`

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

Emptiness check.

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

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

5.6.3.17 `isequal()` [1/2]

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

[Matrix](#) equality check.

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

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

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

5.6.3.18 isequal() [2/2]

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

Matrix equality check with tolerance.

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

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

5.6.3.19 mult()

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

Matrix product with scalar (in-place).

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

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

5.6.3.20 mult_col_by_another()

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

Column multiplication.

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

Exceptions

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

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

5.6.3.21 mult_hadamard()

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

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

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

Exceptions

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

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

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

5.6.3.22 mult_row_by_another()

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

Row multiplication.

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

Exceptions

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

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

5.6.3.23 numel()

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

[Matrix](#) capacity.

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

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

5.6.3.24 operator std::vector< T >()

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

Vector cast operator.

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

5.6.3.25 operator>() [1/2]

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

Element access operator (1D)

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

Exceptions

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

5.6.3.26 operator>() [2/2]

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

Element access operator (2D)

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

Exceptions

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

5.6.3.27 operator=() [1/2]

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

Matrix assignment.

Performs deep-copy of another matrix.

5.6.3.28 operator=() [2/2]

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

Matrix fill operator.

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

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

5.6.3.29 ptr() [1/2]

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

Memory pointer.

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

Exceptions

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

5.6.3.30 ptr() [2/2]

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

Memory pointer.

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

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

5.6.3.31 reshape()

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

[Matrix](#) dimension reshape.

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

Exceptions

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

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

5.6.3.32 resize()

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

```
    unsigned rows,
    unsigned cols )
```

Resize the matrix.

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

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

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

5.6.3.33 row_from_vector()

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

Row from vector.

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

Exceptions

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

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

5.6.3.34 row_to_vector()

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

Row to vector.

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

Exceptions

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

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

5.6.3.35 rows()

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

Number of rows.

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

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

5.6.3.36 set_submatrix()

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

Embed a submatrix.

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

Exceptions

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

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

5.6.3.37 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.

5.6.3.38 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-=\(\)](#).

5.6.3.39 subtract() [2/2]

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

Matrix subtraction with scalar (in-place).

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

5.6.3.40 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\(\)](#).

5.6.3.41 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\(\)](#).

5.6.3.42 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](#)

5.7 Mtx::QR_result< T > Struct Template Reference

Result of QR decomposition.

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

Public Attributes

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

5.7.1 Detailed Description

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

Result of QR decomposition.

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

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

- [matrix.hpp](#)

5.8 Mtx::singular_matrix_exception Class Reference

Singular matrix exception.

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

Inheritance diagram for Mtx::singular_matrix_exception:

Chapter 6

File Documentation

6.1 matrix.hpp File Reference

Classes

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

Functions

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

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

- Vertical matrix concatenation.*

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

Multiplication of matrix by std::vector.

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

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

- Matrix inverse using Gauss-Jordan elimination.*

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

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

6.1.1 Function Documentation

6.1.1.1 add() [1/2]

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

Matrix addition.

Performs addition of two matrices.

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

Template Parameters

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

Parameters

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

Returns

output matrix of size $N \times M$

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

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

6.1.1.2 add() [2/2]

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

Addition of scalar to matrix.

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

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

6.1.1.3 adj()

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

Adjugate matrix.

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

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

Exceptions

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

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

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

6.1.1.4 cconj()

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

Complex conjugate helper.

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

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

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

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

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

6.1.1.5 chol()

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

Cholesky decomposition.

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

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

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

Template Parameters

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

Exceptions

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

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

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

6.1.1.6 cholinv()

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

Inverse of Cholesky decomposition.

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

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

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

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

Exceptions

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

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

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

6.1.1.7 circshift()

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

Circular shift.

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

Parameters

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

Returns

matrix inverse

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

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

6.1.1.8 circulant() [1/2]

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

Circulant matrix from std::vector.

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

Parameters

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

Returns

circulant matrix

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

6.1.1.9 `circulant()` [2/2]

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

Circulant matrix from array.

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

Parameters

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

Returns

circulant matrix

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

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

6.1.1.10 `cofactor()`

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

Cofactor matrix.

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

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

Parameters

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

Exceptions

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

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

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

6.1.1.11 concatenate_horizontal()

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

Horizontal matrix concatenation.

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

Exceptions

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

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

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

6.1.1.12 concatenate_vertical()

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

Vertical matrix concatenation.

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

Exceptions

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

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

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

6.1.1.13 cond()

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

Condition number of a matrix.

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

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

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

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

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

6.1.1.14 csign()

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

Complex sign helper.

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

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

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

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

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

6.1.1.15 ctranspose()

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

Transpose a complex matrix.

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

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

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

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

6.1.1.16 det()

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

Matrix determinant.

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

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

Exceptions

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

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

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

6.1.1.17 det_lu()

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

Matrix determinant from on LU decomposition.

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

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

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

Exceptions

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

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

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

6.1.1.18 diag() [1/3]

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

Diagonal extraction.

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

Parameters

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

Returns

vector of diagonal elements

Exceptions

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

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

6.1.1.19 diag() [2/3]

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

Diagonal matrix from `std::vector`.

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

Parameters

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

Returns

diagonal matrix

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

6.1.1.20 diag() [3/3]

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

Diagonal matrix from array.

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

Parameters

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

Returns

diagonal matrix

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

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

6.1.1.21 div()

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

Division of matrix by scalar.

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

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

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

6.1.1.22 eigenvalues() [1/2]

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

Matrix eigenvalues of complex matrix.

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

Parameters

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

Returns

structure containing the result and status of eigenvalue calculation

Exceptions

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

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

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

6.1.1.23 eigenvalues() [2/2]

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



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

Matrix eigenvalues of real matrix.

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

Parameters

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

Returns

structure containing the result and status of eigenvalue calculation

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

6.1.1.24 eye()

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

Identity matrix.

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

Parameters

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

Returns

zeros matrix

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

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

6.1.1.25 foreach_elem()

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

Applies custom function element-wise in-place.

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

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

Parameters

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

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

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

6.1.1.26 foreach_elem_copy()

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

Applies custom function element-wise with matrix copy.

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

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

Parameters

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

Returns

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

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

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

6.1.1.27 hessenberg()

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

Hessenberg decomposition.

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

Parameters

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

Returns

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

Exceptions

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

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

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

6.1.1.28 householder_reflection()

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

Generate Householder reflection.

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

Parameters

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

Returns

column vector with Householder reflection of *a*

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

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

6.1.1.29 imag()

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

Get imaginary part of complex matrix.

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

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

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

6.1.1.30 inv()

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

Matrix inverse (universal).

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

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

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

Exceptions

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

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

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

6.1.1.31 inv_gauss_jordan()

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

Matrix inverse using Gauss-Jordan elimination.

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

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

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

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

Exceptions

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

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

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

6.1.1.32 inv_posdef()

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

Matrix inverse for Hermitian positive-definite matrix.

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

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

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

Exceptions

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

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

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

6.1.1.33 inv_square()

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

Matrix inverse for general square matrix.

Calculates an inverse of square matrix using matrix.

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

Exceptions

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

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

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

6.1.1.34 inv_tril()

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

Matrix inverse for lower triangular matrix.

Calculates an inverse of lower triangular matrix.

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

Exceptions

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

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

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

6.1.1.35 inv_triu()

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

Matrix inverse for upper triangular matrix.

Calculates an inverse of upper triangular matrix.

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

Exceptions

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

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

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

6.1.1.36 ishess()

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

Hessenberg matrix check.

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

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

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

6.1.1.37 istril()

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

Lower triangular matrix check.

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

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

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

6.1.1.38 istriu()

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

Lower triangular matrix check.

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

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

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

6.1.1.39 kron()

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

Kronecker product.

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

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

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

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

6.1.1.40 ldl()

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

LDL decomposition.

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

$$A = LDL^H$$

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

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

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

Parameters

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

Returns

structure encapsulating calculated L and D

Exceptions

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

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

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

6.1.1.41 lu()

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

LU decomposition.

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

This function implements LU factorization without pivoting. Use [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\(\)](#).

6.1.1.42 lup()

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

LU decomposition with pivoting.

Performs LU factorization with partial pivoting, employing column permutations.

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

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

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

Parameters

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

Returns

structure containing L , U and P .

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

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

6.1.1.43 make_complex() [1/2]

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

Create complex matrix from real matrix.

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

Parameters

<i>Re</i>	real part matrix
-----------	------------------

Returns

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

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

6.1.1.44 make_complex() [2/2]

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

Create complex matrix from real and imaginary matrices.

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

Parameters

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

Returns

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

Exceptions

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

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

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

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

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

Matrix multiplication.

Performs multiplication of two matrices.

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

Template Parameters

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

Parameters

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

Returns

output matrix of size $N \times K$

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

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

6.1.1.46 mult() [2/4]

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

Multiplication of matrix by std::vector.

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

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

Template Parameters

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

Parameters

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

Returns

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

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

6.1.1.47 mult() [3/4]

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

Multiplication of matrix by scalar.

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

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

6.1.1.48 mult() [4/4]

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

Multiplication of std::vector by matrix.

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

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

Template Parameters

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

Parameters

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

Returns

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

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

6.1.1.49 mult_hadamard()

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

Matrix Hadamard (element-wise) multiplication.

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

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

Template Parameters

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

Parameters

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

Returns

output matrix of size *N* x *M*

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

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

6.1.1.50 norm_fro() [1/2]

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

Frobenius norm of a complex matrix.

Calculates Frobenius norm of complex matrix.

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

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

6.1.1.51 norm_fro() [2/2]

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

Frobenius norm.

Calculates Frobenius norm of a real matrix.

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

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

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

6.1.1.52 ones() [1/2]

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

Square matrix of ones.

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

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

Parameters

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

Returns

zeros matrix

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

6.1.1.53 ones() [2/2]

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

Matrix of ones.

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

Parameters

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

Returns

ones matrix

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

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

6.1.1.54 operator!=(())

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

Matrix non-equality check operator.

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

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

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

6.1.1.55 operator*() [1/5]

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

Matrix product.

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

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

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

6.1.1.56 operator*() [2/5]

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

Matrix and std::vector product.

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

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

6.1.1.57 operator*() [3/5]

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

Matrix product with scalar.

Multiplies each element of the matrix by a scalar s .

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

6.1.1.58 operator*() [4/5]

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

std::vector and matrix product.

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

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

6.1.1.59 operator*() [5/5]

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

Matrix product with scalar.

Multiplies each element of the matrix by a scalar s .

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

6.1.1.60 operator*=() [1/2]

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

Matrix product.

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

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

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

6.1.1.61 operator*=() [2/2]

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

Matrix product with scalar.

Multiplies each element of the matrix by a scalar s .

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

6.1.1.62 operator+() [1/3]

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

Matrix sum.

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

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

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

6.1.1.63 operator+() [2/3]

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

Matrix sum with scalar.

Adds a scalar s to each element of the matrix.

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

6.1.1.64 operator+() [3/3]

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

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

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

6.1.1.65 operator+=() [1/2]

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

Matrix sum.

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

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

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

6.1.1.66 operator+=() [2/2]

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

Matrix sum with scalar.

Adds a scalar s to each element of the matrix.

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

6.1.1.67 operator-() [1/2]

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

Matrix subtraction.

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

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

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

6.1.1.68 operator-() [2/2]

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

Matrix subtraction with scalar.

Subtracts a scalar s from each element of the matrix.

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

6.1.1.69 operator-=() [1/2]

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

Matrix subtraction.

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

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

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

6.1.1.70 operator-=() [2/2]

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

Matrix subtraction with scalar.

Subtracts a scalar s from each element of the matrix.

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

6.1.1.71 operator/()

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

Matrix division by scalar.

Divides each element of the matrix by a scalar s .

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

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

6.1.1.72 operator/=()

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

Matrix division by scalar.

Divides each element of the matrix by a scalar s .

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

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

6.1.1.73 operator<<()

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

Matrix ostream operator.

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

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

6.1.1.74 operator==()

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

Matrix equality check operator.

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

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

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

6.1.1.75 operator^()

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

Matrix Hadamard product.

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

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

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

6.1.1.76 operator^=()

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

Matrix Hadamard product.

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

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

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

6.1.1.77 permute_cols()

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

Permute columns of the matrix.

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

Parameters

A	input matrix
$perm$	permutation vector with column indices

Returns

output matrix created by column permutation of A

Exceptions

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

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

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

6.1.1.78 permute_rows()

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

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

Permute rows of the matrix.

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

Parameters

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

Returns

output matrix created by row permutation of *A*

Exceptions

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

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

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

6.1.1.79 permute_rows_and_cols()

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

Permute both rows and columns of the matrix.

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

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

Parameters

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

Returns

output matrix created by row and column permutation of A

Exceptions

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

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

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

6.1.1.80 pinv()

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

Moore-Penrose pseudo-inverse.

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

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

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

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

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

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

6.1.1.81 qr()

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

QR decomposition.

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

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

Parameters

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

Returns

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

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

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

6.1.1.82 qr_householder()

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

QR decomposition based on Householder method.

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

This function implements QR decomposition based on Householder reflections method.

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

Parameters

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

Returns

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

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

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

6.1.1.83 qr_red_gs()

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

Reduced QR decomposition based on Gram-Schmidt method.

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

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

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

Parameters

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

Returns

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

Exceptions

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

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

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

6.1.1.84 real()

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

Get real part of complex matrix.

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

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

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

6.1.1.85 repmat()

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

Repeat matrix.

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

Parameters

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

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

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

6.1.1.86 solve_posdef()

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

Solves the positive definite (Hermitian) system.

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

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

Parameters

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

Returns

solution matrix of size $N \times M$.

Exceptions

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

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

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

6.1.1.87 solve_square()

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

Solves the square system.

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

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

Parameters

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

Returns

solution matrix of size $N \times M$.

Exceptions

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

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

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

6.1.1.88 solve_tril()

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

Solves the lower triangular system.

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

Parameters

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

Returns

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

Exceptions

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

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

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

6.1.1.89 solve_triu()

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

Solves the upper triangular system.

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

Parameters

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

Returns

solution matrix of size $N \times M$.

Exceptions

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

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

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

6.1.1.90 subtract() [1/2]

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

Matrix subtraction.

Performs subtraction of two matrices.

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

Template Parameters

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

Parameters

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

Returns

output matrix of size $N \times M$

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

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

6.1.1.91 subtract() [2/2]

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

Subtraction of scalar from matrix.

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

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

6.1.1.92 trace()

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

Matrix trace.

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

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

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

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

6.1.1.93 transpose()

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

Transpose a matrix.

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

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

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

6.1.1.94 tril()

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

Extract triangular lower part.

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

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

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

6.1.1.95 triu()

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

Extract triangular upper part.

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

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

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

6.1.1.96 wilkinson_shift()

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

Wilkinson's shift for complex eigenvalues.

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

Exceptions

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

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

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

6.1.1.97 zeros() [1/2]

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

Square matrix of zeros.

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

Parameters

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

Returns

zeros matrix

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

6.1.1.98 zeros() [2/2]

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

Matrix of zeros.

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

Parameters

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

Returns

zeros matrix

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

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

6.2 matrix.hpp

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

```

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



```

00120     Matrix<T> U;
00121
00124     std::vector<unsigned> P;
00125 };
00126
00132 template<typename T>
00133 struct QR_result {
00134     Matrix<T> Q;
00135
00137     Matrix<T> R;
00141 };
00142
00147 template<typename T>
00148 struct Hessenberg_result {
00149     Matrix<T> H;
00150
00152     Matrix<T> Q;
00156 };
00157
00162 template<typename T>
00163 struct LDL_result {
00164     Matrix<T> L;
00165
00170     std::vector<T> d;
00172 };
00177 template<typename T>
00178 struct Eigenvalues_result {
00179     std::vector<std::complex<T>> eig;
00182
00185     bool converged;
00186
00189     T err;
00190 };
00191
00192
00200 template<typename T>
00201 inline Matrix<T> zeros(unsigned nrows, unsigned ncols) {
00202     return Matrix<T>(static_cast<T>(0), nrows, ncols);
00203 }
00204
00211 template<typename T>
00212 inline Matrix<T> zeros(unsigned n) {
00213     return zeros<T>(n,n);
00214 }
00215
00224 template<typename T>
00225 inline Matrix<T> ones(unsigned nrows, unsigned ncols) {
00226     return Matrix<T>(static_cast<T>(1), nrows, ncols);
00227 }
00228
00236 template<typename T>
00237 inline Matrix<T> ones(unsigned n) {
00238     return ones<T>(n,n);
00239 }
00240
00248 template<typename T>
00249 Matrix<T> eye(unsigned n) {
00250     Matrix<T> A(static_cast<T>(0), n, n);
00251     for (unsigned i = 0; i < n; i++)
00252         A(i,i) = static_cast<T>(1);
00253     return A;
00254 }
00255
00263 template<typename T>
00264 Matrix<T> diag(const T* array, size_t n) {
00265     Matrix<T> A(static_cast<T>(0), n, n);
00266     for (unsigned i = 0; i < n; i++) {
00267         A(i,i) = array[i];
00268     }
00269     return A;
00270 }
00271
00279 template<typename T>
00280 inline Matrix<T> diag(const std::vector<T>& v) {
00281     return diag(v.data(), v.size());
00282 }
00283
00292 template<typename T>
00293 std::vector<T> diag(const Matrix<T>& A) {
00294     if (! A.issquare()) throw std::runtime_error("Input matrix is not square");
00295
00296     std::vector<T> v;
00297     v.resize(A.rows());
00298
00299     for (unsigned i = 0; i < A.rows(); i++)
00300         v[i] = A(i,i);

```

```

00301     return v;
00302 }
00303
00311 template<typename T>
00312 Matrix<T> circulant(const T* array, unsigned n) {
00313     Matrix<T> A(n, n);
00314     for (unsigned j = 0; j < n; j++)
00315         for (unsigned i = 0; i < n; i++)
00316             A((i+j) % n, j) = array[i];
00317     return A;
00318 }
00319
00330 template<typename T>
00331 Matrix<std::complex<T>> make_complex(const Matrix<T>& Re, const Matrix<T>& Im) {
00332     if (Re.rows() != Im.rows() || Re.cols() != Im.cols()) throw std::runtime_error("Size of input
matrices does not match");
00333
00334     Matrix<std::complex<T> > C(Re.rows(), Re.cols());
00335     for (unsigned n = 0; n < Re.numel(); n++) {
00336         C(n).real(Re(n));
00337         C(n).imag(Im(n));
00338     }
00339
00340     return C;
00341 }
00342
00349 template<typename T>
00350 Matrix<std::complex<T>> make_complex(const Matrix<T>& Re) {
00351     Matrix<std::complex<T>> C(Re.rows(), Re.cols());
00352
00353     for (unsigned n = 0; n < Re.numel(); n++) {
00354         C(n).real(Re(n));
00355         C(n).imag(static_cast<T>(0));
00356     }
00357
00358     return C;
00359 }
00360
00365 template<typename T>
00366 Matrix<T> real(const Matrix<std::complex<T>& C) {
00367     Matrix<T> Re(C.rows(), C.cols());
00368
00369     for (unsigned n = 0; n < C.numel(); n++)
00370         Re(n) = C(n).real();
00371
00372     return Re;
00373 }
00374
00379 template<typename T>
00380 Matrix<T> imag(const Matrix<std::complex<T>& C) {
00381     Matrix<T> Re(C.rows(), C.cols());
00382
00383     for (unsigned n = 0; n < C.numel(); n++)
00384         Re(n) = C(n).imag();
00385
00386     return Re;
00387 }
00388
00396 template<typename T>
00397 inline Matrix<T> circulant(const std::vector<T>& v) {
00398     return circulant(v.data(), v.size());
00399 }
00400
00405 template<typename T>
00406 inline Matrix<T> transpose(const Matrix<T>& A) {
00407     return A.transpose();
00408 }
00409
00415 template<typename T>
00416 inline Matrix<T> ctranspose(const Matrix<T>& A) {
00417     return A.ctranspose();
00418 }
00419
00430 template<typename T>
00431 Matrix<T> circshift(const Matrix<T>& A, int row_shift, int col_shift) {
00432     Matrix<T> B(A.rows(), A.cols());
00433     for (int i = 0; i < A.rows(); i++) {
00434         int ii = (i + row_shift) % A.rows();
00435         for (int j = 0; j < A.cols(); j++) {
00436             int jj = (j + col_shift) % A.cols();
00437             B(ii, jj) = A(i, j);
00438         }
00439     }
00440     return B;
00441 }
00442
00450 template<typename T>

```

```

00451 Matrix<T> repmat(const Matrix<T>& A, unsigned m, unsigned n) {
00452     Matrix<T> B(m * A.rows(), n * A.cols());
00453
00454     for (unsigned cb = 0; cb < n; cb++)
00455         for (unsigned rb = 0; rb < m; rb++)
00456             for (unsigned c = 0; c < A.cols(); c++)
00457                 for (unsigned r = 0; r < A.rows(); r++)
00458                     B(rb*A.rows() + r, cb*A.cols() + c) = A(r, c);
00459
00460     return B;
00461 }
00462
00463 template<typename T>
00464 Matrix<T> concatenate_horizontal(const Matrix<T>& A, const Matrix<T>& B) {
00465     if (A.rows() != B.rows()) throw std::runtime_error("Unmatching number of rows for horizontal
concatenation");
00466
00467     Matrix<T> C(A.rows(), A.cols() + B.cols());
00468
00469     for (unsigned c = 0; c < A.cols(); c++)
00470         for (unsigned r = 0; r < A.rows(); r++)
00471             C(r,c) = A(r,c);
00472
00473     for (unsigned c = 0; c < B.cols(); c++)
00474         for (unsigned r = 0; r < B.rows(); r++)
00475             C(r,c+A.cols()) = B(r,c);
00476
00477     return C;
00478 }
00479
00480 template<typename T>
00481 Matrix<T> concatenate_vertical(const Matrix<T>& A, const Matrix<T>& B) {
00482     if (A.cols() != B.cols()) throw std::runtime_error("Unmatching number of columns for vertical
concatenation");
00483
00484     Matrix<T> C(A.rows() + B.rows(), A.cols());
00485
00486     for (unsigned c = 0; c < A.cols(); c++)
00487         for (unsigned r = 0; r < A.rows(); r++)
00488             C(r,c) = A(r,c);
00489
00490     for (unsigned c = 0; c < B.cols(); c++)
00491         for (unsigned r = 0; r < B.rows(); r++)
00492             C(r+A.rows(),c) = B(r,c);
00493
00494     return C;
00495 }
00496
00497 template<typename T>
00498 double norm_fro(const Matrix<T>& A) {
00499     double sum = 0;
00500
00501     for (unsigned i = 0; i < A.numel(); i++)
00502         sum += A(i) * A(i);
00503
00504     return std::sqrt(sum);
00505 }
00506
00507 template<typename T>
00508 double norm_fro(const Matrix<std::complex<T>>& A) {
00509     double sum = 0;
00510
00511     for (unsigned i = 0; i < A.numel(); i++) {
00512         T x = std::abs(A(i));
00513         sum += x * x;
00514     }
00515
00516     return std::sqrt(sum);
00517 }
00518
00519 template<typename T>
00520 Matrix<T> tril(const Matrix<T>& A) {
00521     Matrix<T> B(A);
00522
00523     for (unsigned row = 0; row < B.rows(); row++)
00524         for (unsigned col = row+1; col < B.cols(); col++)
00525             B(row,col) = 0;
00526
00527     return B;
00528 }
00529
00530 template<typename T>
00531 Matrix<T> triu(const Matrix<T>& A) {
00532     Matrix<T> B(A);
00533
00534     for (unsigned col = 0; col < B.cols(); col++)
00535         for (unsigned row = col+1; row < B.rows(); row++)

```

```

00568         B(row,col) = 0;
00569
00570     return B;
00571 }
00572
00573 template<typename T>
00574 bool istril(const Matrix<T>& A) {
00575     for (unsigned row = 0; row < A.rows(); row++)
00576         for (unsigned col = row+1; col < A.cols(); col++)
00577             if (A(row,col) != static_cast<T>(0)) return false;
00578     return true;
00579 }
00580
00581 template<typename T>
00582 bool istriu(const Matrix<T>& A) {
00583     for (unsigned col = 0; col < A.cols(); col++)
00584         for (unsigned row = col+1; row < A.rows(); row++)
00585             if (A(row,col) != static_cast<T>(0)) return false;
00586     return true;
00587 }
00588
00589 template<typename T>
00590 bool ishess(const Matrix<T>& A) {
00591     if (!A.issquare())
00592         return false;
00593     for (unsigned row = 2; row < A.rows(); row++)
00594         for (unsigned col = 0; col < row-2; col++)
00595             if (A(row,col) != static_cast<T>(0)) return false;
00596     return true;
00597 }
00598
00599 template<typename T>
00600 inline void foreach_elem(Matrix<T>& A, std::function<T(T)> func) {
00601     for (unsigned i = 0; i < A.numel(); i++)
00602         A(i) = func(A(i));
00603 }
00604
00605 template<typename T>
00606 inline Matrix<T> foreach_elem_copy(const Matrix<T>& A, std::function<T(T)> func) {
00607     Matrix<T> B(A);
00608     foreach_elem(B, func);
00609     return B;
00610 }
00611
00612 template<typename T>
00613 Matrix<T> permute_rows(const Matrix<T>& A, const std::vector<unsigned> perm) {
00614     if (perm.empty()) throw std::runtime_error("Permutation vector is empty");
00615     for (unsigned p = 0; p < perm.size(); p++)
00616         if (!perm[p] < A.rows()) throw std::out_of_range("Index in permutation vector out of range");
00617     Matrix<T> B(perm.size(), A.cols());
00618     for (unsigned p = 0; p < perm.size(); p++)
00619         for (unsigned c = 0; c < A.cols(); c++)
00620             B(p,c) = A(perm[p],c);
00621     return B;
00622 }
00623
00624 template<typename T>
00625 Matrix<T> permute_cols(const Matrix<T>& A, const std::vector<unsigned> perm) {
00626     if (perm.empty()) throw std::runtime_error("Permutation vector is empty");
00627     for (unsigned p = 0; p < perm.size(); p++)
00628         if (!perm[p] < A.cols()) throw std::out_of_range("Index in permutation vector out of range");
00629     Matrix<T> B(A.rows(), perm.size());
00630     for (unsigned p = 0; p < perm.size(); p++)
00631         for (unsigned r = 0; r < A.rows(); r++)
00632             B(r,p) = A(r,perm[p]);
00633     return B;
00634 }
00635
00636 template<typename T>
00637 Matrix<T> permute_rows_and_cols(const Matrix<T>& A, const std::vector<unsigned> perm_rows, const
std::vector<unsigned> perm_cols) {
00638     if (perm_rows.empty()) throw std::runtime_error("Row permutation vector is empty");
00639     if (perm_cols.empty()) throw std::runtime_error("Column permutation vector is empty");
00640     for (unsigned pc = 0; pc < perm_cols.size(); pc++)
00641         if (!perm_cols[pc] < A.cols()) throw std::out_of_range("Column index in permutation vector out
of range");
00642     for (unsigned pr = 0; pr < perm_rows.size(); pr++)

```

```

00725     if (!(perm_rows[pr] < A.rows())) throw std::out_of_range("Row index in permutation vector out of
range");
00726
00727     Matrix<T> B(perm_rows.size(), perm_cols.size());
00728
00729     for (unsigned pc = 0; pc < perm_cols.size(); pc++)
00730         for (unsigned pr = 0; pr < perm_rows.size(); pr++)
00731             B(pr,pc) = A(perm_rows[pr],perm_cols[pc]);
00732
00733     return B;
00734 }
00735
00751 template<typename T, bool transpose_first = false, bool transpose_second = false>
00752 Matrix<T> mult(const Matrix<T>& A, const Matrix<T>& B) {
00753     // Adjust dimensions based on transpositions
00754     unsigned rows_A = transpose_first ? A.cols() : A.rows();
00755     unsigned cols_A = transpose_first ? A.rows() : A.cols();
00756     unsigned rows_B = transpose_second ? B.cols() : B.rows();
00757     unsigned cols_B = transpose_second ? B.rows() : B.cols();
00758
00759     if (cols_A != rows_B) throw std::runtime_error("Unmatching matrix dimensions for mult");
00760
00761     Matrix<T> C(static_cast<T>(0), rows_A, cols_B);
00762
00763     for (unsigned i = 0; i < rows_A; i++)
00764         for (unsigned j = 0; j < cols_B; j++)
00765             for (unsigned k = 0; k < cols_A; k++)
00766                 C(i,j) += (transpose_first ? cconj(A(k,i)) : A(i,k)) *
00767                     (transpose_second ? cconj(B(j,k)) : B(k,j));
00768
00769     return C;
00770 }
00771
00787 template<typename T, bool transpose_first = false, bool transpose_second = false>
00788 Matrix<T> mult_hadamard(const Matrix<T>& A, const Matrix<T>& B) {
00789     // Adjust dimensions based on transpositions
00790     unsigned rows_A = transpose_first ? A.cols() : A.rows();
00791     unsigned cols_A = transpose_first ? A.rows() : A.cols();
00792     unsigned rows_B = transpose_second ? B.cols() : B.rows();
00793     unsigned cols_B = transpose_second ? B.rows() : B.cols();
00794
00795     if ((rows_A != rows_B) || (cols_A != cols_B)) throw std::runtime_error("Unmatching matrix dimensions
for mult_hadamard");
00796
00797     Matrix<T> C(static_cast<T>(0), rows_A, cols_A);
00798
00799     for (unsigned i = 0; i < rows_A; i++)
00800         for (unsigned j = 0; j < cols_A; j++)
00801             C(i,j) += (transpose_first ? cconj(A(j,i)) : A(i,j)) *
00802                 (transpose_second ? cconj(B(j,i)) : B(i,j));
00803
00804     return C;
00805 }
00806
00822 template<typename T, bool transpose_first = false, bool transpose_second = false>
00823 Matrix<T> add(const Matrix<T>& A, const Matrix<T>& B) {
00824     // Adjust dimensions based on transpositions
00825     unsigned rows_A = transpose_first ? A.cols() : A.rows();
00826     unsigned cols_A = transpose_first ? A.rows() : A.cols();
00827     unsigned rows_B = transpose_second ? B.cols() : B.rows();
00828     unsigned cols_B = transpose_second ? B.rows() : B.cols();
00829
00830     if ((rows_A != rows_B) || (cols_A != cols_B)) throw std::runtime_error("Unmatching matrix dimensions
for add");
00831
00832     Matrix<T> C(static_cast<T>(0), rows_A, cols_A);
00833
00834     for (unsigned i = 0; i < rows_A; i++)
00835         for (unsigned j = 0; j < cols_A; j++)
00836             C(i,j) += (transpose_first ? cconj(A(j,i)) : A(i,j)) +
00837                 (transpose_second ? cconj(B(j,i)) : B(i,j));
00838
00839     return C;
00840 }
00841
00857 template<typename T, bool transpose_first = false, bool transpose_second = false>
00858 Matrix<T> subtract(const Matrix<T>& A, const Matrix<T>& B) {
00859     // Adjust dimensions based on transpositions
00860     unsigned rows_A = transpose_first ? A.cols() : A.rows();
00861     unsigned cols_A = transpose_first ? A.rows() : A.cols();
00862     unsigned rows_B = transpose_second ? B.cols() : B.rows();
00863     unsigned cols_B = transpose_second ? B.rows() : B.cols();
00864
00865     if ((rows_A != rows_B) || (cols_A != cols_B)) throw std::runtime_error("Unmatching matrix dimensions
for subtract");
00866
00867     Matrix<T> C(static_cast<T>(0), rows_A, cols_A);

```

```

00868
00869     for (unsigned i = 0; i < rows_A; i++)
00870         for (unsigned j = 0; j < cols_A; j++)
00871             C(i,j) += (transpose_first ? cconj(A(j,i)) : A(i,j)) -
00872                        (transpose_second ? cconj(B(j,i)) : B(i,j));
00873
00874     return C;
00875 }
00876
00892 template<typename T, bool transpose_matrix = false>
00893 std::vector<T> mult(const Matrix<T>& A, const std::vector<T>& v) {
00894     // Adjust dimensions based on transpositions
00895     unsigned rows_A = transpose_matrix ? A.cols() : A.rows();
00896     unsigned cols_A = transpose_matrix ? A.rows() : A.cols();
00897
00898     if (cols_A != v.size()) throw std::runtime_error("Unmatching matrix dimensions for mult");
00899
00900     std::vector<T> u(rows_A, static_cast<T>(0));
00901     for (unsigned r = 0; r < rows_A; r++)
00902         for (unsigned c = 0; c < cols_A; c++)
00903             u[r] += v[c] * (transpose_matrix ? cconj(A(c,r)) : A(r,c));
00904
00905     return u;
00906 }
00907
00923 template<typename T, bool transpose_matrix = false>
00924 std::vector<T> mult(const std::vector<T>& v, const Matrix<T>& A) {
00925     // Adjust dimensions based on transpositions
00926     unsigned rows_A = transpose_matrix ? A.cols() : A.rows();
00927     unsigned cols_A = transpose_matrix ? A.rows() : A.cols();
00928
00929     if (rows_A != v.size()) throw std::runtime_error("Unmatching matrix dimensions for mult");
00930
00931     std::vector<T> u(cols_A, static_cast<T>(0));
00932     for (unsigned c = 0; c < cols_A; c++)
00933         for (unsigned r = 0; r < rows_A; r++)
00934             u[c] += v[r] * (transpose_matrix ? cconj(A(c,r)) : A(r,c));
00935
00936     return u;
00937 }
00938
00944 template<typename T>
00945 Matrix<T> add(const Matrix<T>& A, T s) {
00946     Matrix<T> B(A.rows(), A.cols());
00947     for (unsigned i = 0; i < A.numel(); i++)
00948         B(i) = A(i) + s;
00949     return B;
00950 }
00951
00957 template<typename T>
00958 Matrix<T> subtract(const Matrix<T>& A, T s) {
00959     Matrix<T> B(A.rows(), A.cols());
00960     for (unsigned i = 0; i < A.numel(); i++)
00961         B(i) = A(i) - s;
00962     return B;
00963 }
00964
00970 template<typename T>
00971 Matrix<T> mult(const Matrix<T>& A, T s) {
00972     Matrix<T> B(A.rows(), A.cols());
00973     for (unsigned i = 0; i < A.numel(); i++)
00974         B(i) = A(i) * s;
00975     return B;
00976 }
00977
00983 template<typename T>
00984 Matrix<T> div(const Matrix<T>& A, T s) {
00985     Matrix<T> B(A.rows(), A.cols());
00986     for (unsigned i = 0; i < A.numel(); i++)
00987         B(i) = A(i) / s;
00988     return B;
00989 }
00990
00996 template<typename T>
00997 std::ostream& operator<<(std::ostream& os, const Matrix<T>& A) {
00998     for (unsigned row = 0; row < A.rows(); row++) {
00999         for (unsigned col = 0; col < A.cols(); col++)
01000             os << A(row,col) << " ";
01001         if (row < static_cast<unsigned>(A.rows()-1)) os << std::endl;
01002     }
01003     return os;
01004 }
01005
01010 template<typename T>
01011 inline Matrix<T> operator+(const Matrix<T>& A, const Matrix<T>& B) {
01012     return add(A,B);
01013 }

```

```

01014
01019 template<typename T>
01020 inline Matrix<T> operator-(const Matrix<T>& A, const Matrix<T>& B) {
01021     return subtract(A,B);
01022 }
01023
01029 template<typename T>
01030 inline Matrix<T> operator^(const Matrix<T>& A, const Matrix<T>& B) {
01031     return mult_hadamard(A,B);
01032 }
01033
01038 template<typename T>
01039 inline Matrix<T> operator*(const Matrix<T>& A, const Matrix<T>& B) {
01040     return mult(A,B);
01041 }
01042
01047 template<typename T>
01048 inline std::vector<T> operator*(const Matrix<T>& A, const std::vector<T>& v) {
01049     return mult(A,v);
01050 }
01051
01056 template<typename T>
01057 inline std::vector<T> operator*(const std::vector<T>& v, const Matrix<T>& A) {
01058     return mult(v,A);
01059 }
01060
01065 template<typename T>
01066 inline Matrix<T> operator+(const Matrix<T>& A, T s) {
01067     return add(A,s);
01068 }
01069
01074 template<typename T>
01075 inline Matrix<T> operator-(const Matrix<T>& A, T s) {
01076     return subtract(A,s);
01077 }
01078
01083 template<typename T>
01084 inline Matrix<T> operator*(const Matrix<T>& A, T s) {
01085     return mult(A,s);
01086 }
01087
01092 template<typename T>
01093 inline Matrix<T> operator/(const Matrix<T>& A, T s) {
01094     return div(A,s);
01095 }
01096
01100 template<typename T>
01101 inline Matrix<T> operator+(T s, const Matrix<T>& A) {
01102     return add(A,s);
01103 }
01104
01109 template<typename T>
01110 inline Matrix<T> operator*(T s, const Matrix<T>& A) {
01111     return mult(A,s);
01112 }
01113
01118 template<typename T>
01119 inline Matrix<T>& operator+=(Matrix<T>& A, const Matrix<T>& B) {
01120     return A.add(B);
01121 }
01122
01127 template<typename T>
01128 inline Matrix<T>& operator-=(Matrix<T>& A, const Matrix<T>& B) {
01129     return A.subtract(B);
01130 }
01131
01136 template<typename T>
01137 inline Matrix<T>& operator*=(Matrix<T>& A, const Matrix<T>& B) {
01138     A = mult(A,B);
01139     return A;
01140 }
01141
01147 template<typename T>
01148 inline Matrix<T>& operator^=(Matrix<T>& A, const Matrix<T>& B) {
01149     return A.mult_hadamard(B);
01150 }
01151
01156 template<typename T>
01157 inline Matrix<T>& operator+=(Matrix<T>& A, T s) {
01158     return A.add(s);
01159 }
01160
01165 template<typename T>
01166 inline Matrix<T>& operator-=(Matrix<T>& A, T s) {
01167     return A.subtract(s);
01168 }
01169

```

```

01174 template<typename T>
01175 inline Matrix<T>& operator*=(Matrix<T>& A, T s) {
01176     return A.mult(s);
01177 }
01178
01183 template<typename T>
01184 inline Matrix<T>& operator/=(Matrix<T>& A, T s) {
01185     return A.div(s);
01186 }
01187
01192 template<typename T>
01193 inline bool operator==(const Matrix<T>& A, const Matrix<T>& b) {
01194     return A.isequal(b);
01195 }
01196
01201 template<typename T>
01202 inline bool operator!=(const Matrix<T>& A, const Matrix<T>& b) {
01203     return !(A.isequal(b));
01204 }
01205
01212 template<typename T>
01213 Matrix<T> kron(const Matrix<T>& A, const Matrix<T>& B) {
01214     const unsigned rows_A = A.rows();
01215     const unsigned cols_A = A.cols();
01216     const unsigned rows_B = B.rows();
01217     const unsigned cols_B = B.cols();
01218
01219     unsigned rows_C = rows_A * rows_B;
01220     unsigned cols_C = cols_A * cols_B;
01221
01222     Matrix<T> C(rows_C, cols_C);
01223
01224     for (unsigned i = 0; i < rows_A; i++)
01225         for (unsigned j = 0; j < cols_A; j++)
01226             for (unsigned k = 0; k < rows_B; k++)
01227                 for (unsigned l = 0; l < cols_B; l++)
01228                     C(i*rows_B + k, j*cols_B + l) = A(i, j) * B(k, l);
01229
01230     return C;
01231 }
01232
01240 template<typename T>
01241 Matrix<T> adj(const Matrix<T>& A) {
01242     if (!A.issquare()) throw std::runtime_error("Input matrix is not square");
01243
01244     Matrix<T> B(A.rows(), A.cols());
01245     if (A.rows() == 1) {
01246         B(0) = 1.0;
01247     } else {
01248         for (unsigned i = 0; i < A.rows(); i++) {
01249             for (unsigned j = 0; j < A.cols(); j++) {
01250                 T sgn = ((i + j) % 2 == 0) ? 1.0 : -1.0;
01251                 B(j, i) = sgn * det(cofactor(A, i, j));
01252             }
01253         }
01254     }
01255     return B;
01256 }
01257
01271 template<typename T>
01272 Matrix<T> cofactor(const Matrix<T>& A, unsigned p, unsigned q) {
01273     if (!A.issquare()) throw std::runtime_error("Input matrix is not square");
01274     if (!(p < A.rows())) throw std::out_of_range("Row index out of range");
01275     if (!(q < A.cols())) throw std::out_of_range("Column index out of range");
01276     if (A.cols() < 2) throw std::runtime_error("Cofactor calculation requested for matrix with less than
2 rows");
01277
01278     Matrix<T> c(A.rows()-1, A.cols()-1);
01279     unsigned i = 0;
01280     unsigned j = 0;
01281
01282     for (unsigned row = 0; row < A.rows(); row++) {
01283         if (row != p) {
01284             for (unsigned col = 0; col < A.cols(); col++)
01285                 if (col != q) c(i, j++) = A(row, col);
01286             j = 0;
01287             i++;
01288         }
01289     }
01290
01291     return c;
01292 }
01293
01305 template<typename T>
01306 T det_lu(const Matrix<T>& A) {
01307     if (!A.issquare()) throw std::runtime_error("Input matrix is not square");
01308

```



```

01309 // LU decomposition with pivoting
01310 auto res = lup(A);
01311
01312 // Determinants of LU
01313 T detLU = static_cast<T>(1);
01314
01315 for (unsigned i = 0; i < res.L.rows(); i++)
01316     detLU *= res.L(i,i) * res.U(i,i);
01317
01318 // Determinant of P
01319 unsigned len = res.P.size();
01320 T detP = 1;
01321
01322 std::vector<unsigned> p(res.P);
01323 std::vector<unsigned> q;
01324 q.resize(len);
01325
01326 for (unsigned i = 0; i < len; i++)
01327     q[p[i]] = i;
01328
01329 for (unsigned i = 0; i < len; i++) {
01330     unsigned j = p[i];
01331     unsigned k = q[i];
01332     if (j != i) {
01333         p[k] = p[i];
01334         q[j] = q[i];
01335         detP = - detP;
01336     }
01337 }
01338
01339 return detLU * detP;
01340 }
01341
01351 template<typename T>
01352 T det(const Matrix<T>& A) {
01353     if (! A.issquare()) throw std::runtime_error("Input matrix is not square");
01354
01355     if (A.rows() == 1)
01356         return A(0,0);
01357     else if (A.rows() == 2)
01358         return A(0,0)*A(1,1) - A(0,1)*A(1,0);
01359     else if (A.rows() == 3)
01360         return A(0,0)*(A(1,1)*A(2,2) - A(1,2)*A(2,1)) -
01361             A(0,1)*(A(1,0)*A(2,2) - A(1,2)*A(2,0)) +
01362             A(0,2)*(A(1,0)*A(2,1) - A(1,1)*A(2,0));
01363     else
01364         return det_lu(A);
01365 }
01366
01376 template<typename T>
01377 LU_result<T> lu(const Matrix<T>& A) {
01378     const unsigned M = A.rows();
01379     const unsigned N = A.cols();
01380
01381     LU_result<T> res;
01382     res.L = eye<T>(M);
01383     res.U = Matrix<T>(A);
01384
01385     // aliases
01386     auto& L = res.L;
01387     auto& U = res.U;
01388
01389     if (A.numel() == 0)
01390         return res;
01391
01392     for (unsigned k = 0; k < M-1; k++) {
01393         for (unsigned i = k+1; i < M; i++) {
01394             L(i,k) = U(i,k) / U(k,k);
01395             for (unsigned l = k+1; l < N; l++) {
01396                 U(i,l) -= L(i,k) * U(k,l);
01397             }
01398         }
01399     }
01400
01401     for (unsigned col = 0; col < N; col++)
01402         for (unsigned row = col+1; row < M; row++)
01403             U(row,col) = 0;
01404
01405     return res;
01406 }
01407
01421 template<typename T>
01422 LUP_result<T> lup(const Matrix<T>& A) {
01423     const unsigned M = A.rows();
01424     const unsigned N = A.cols();
01425
01426     // Initialize L, U, and PP

```

```

01427     LUP_result<T> res;
01428
01429     if (A.numel() == 0)
01430         return res;
01431
01432     res.L = eye<T>(M);
01433     res.U = Matrix<T>(A);
01434     std::vector<unsigned> PP;
01435
01436     // aliases
01437     auto& L = res.L;
01438     auto& U = res.U;
01439
01440     PP.resize(N);
01441     for (unsigned i = 0; i < N; i++)
01442         PP[i] = i;
01443
01444     for (unsigned k = 0; k < M-1; k++) {
01445         // Find the column with the largest absolute value in the current row
01446         auto max_col_value = std::abs(U(k,k));
01447         unsigned max_col_index = k;
01448         for (unsigned l = k+1; l < N; l++) {
01449             auto val = std::abs(U(k,l));
01450             if (val > max_col_value) {
01451                 max_col_value = val;
01452                 max_col_index = l;
01453             }
01454         }
01455
01456         // Swap columns k and max_col_index in U and update P
01457         if (max_col_index != k) {
01458             U.swap_cols(k, max_col_index); // TODO: This could be reworked to avoid column swap in U during
every iteration by:
01459                                     //      1. using PP[k] for column indexing across iterations
01460                                     //      2. doing just one permutation of U at the end
01461             std::swap(PP[k], PP[max_col_index]);
01462         }
01463
01464         // Update L and U
01465         for (unsigned i = k+1; i < M; i++) {
01466             L(i,k) = U(i,k) / U(k,k);
01467             for (unsigned l = k+1; l < N; l++) {
01468                 U(i,l) -= L(i,k) * U(k,l);
01469             }
01470         }
01471     }
01472
01473     // Set elements in lower triangular part of U to zero
01474     for (unsigned col = 0; col < N; col++)
01475         for (unsigned row = col+1; row < M; row++)
01476             U(row,col) = 0;
01477
01478     // Transpose indices in permutation vector
01479     res.P.resize(N);
01480     for (unsigned i = 0; i < N; i++)
01481         res.P[PP[i]] = i;
01482
01483     return res;
01484 }
01485
01496 template<typename T>
01497 Matrix<T> inv_gauss_jordan(const Matrix<T>& A) {
01498     if (! A.issquare()) throw std::runtime_error("Input matrix is not square");
01499
01500     const unsigned N = A.rows();
01501     Matrix<T> AA(A);
01502     auto IA = eye<T>(N);
01503
01504     bool found_nonzero;
01505     for (unsigned j = 0; j < N; j++) {
01506         found_nonzero = false;
01507         for (unsigned i = j; i < N; i++) {
01508             if (AA(i,j) != static_cast<T>(0)) {
01509                 found_nonzero = true;
01510                 for (unsigned k = 0; k < N; k++) {
01511                     std::swap(AA(j,k), AA(i,k));
01512                     std::swap(IA(j,k), IA(i,k));
01513                 }
01514                 if (AA(j,j) != static_cast<T>(1)) {
01515                     T s = static_cast<T>(1) / AA(j,j);
01516                     for (unsigned k = 0; k < N; k++) {
01517                         AA(j,k) *= s;
01518                         IA(j,k) *= s;
01519                     }
01520                 }
01521                 for (unsigned l = 0; l < N; l++) {
01522                     if (l != j) {

```

```

01523         T s = AA(l,j);
01524         for (unsigned k = 0; k < N; k++) {
01525             AA(l,k) -= s * AA(j,k);
01526             IA(l,k) -= s * IA(j,k);
01527         }
01528     }
01529 }
01530 }
01531 break;
01532 }
01533 // if a row full of zeros is found, the input matrix was singular
01534 if (!found_nonzero) throw singular_matrix_exception("Singular matrix in inv_gauss_jordan");
01535 }
01536 return IA;
01537 }
01538
01539 template<typename T>
01540 Matrix<T> inv_tril(const Matrix<T>& A) {
01541     if (!A.issquare()) throw std::runtime_error("Input matrix is not square");
01542
01543     const unsigned N = A.rows();
01544     auto IA = zeros<T>(N);
01545
01546     for (unsigned i = 0; i < N; i++) {
01547         if (A(i,i) == 0.0) throw singular_matrix_exception("Division by zero in inv_tril");
01548
01549         IA(i,i) = static_cast<T>(1.0) / A(i,i);
01550         for (unsigned j = 0; j < i; j++) {
01551             T s = 0.0;
01552             for (unsigned k = j; k < i; k++)
01553                 s += A(i,k) * IA(k,j);
01554             IA(i,j) = -s * IA(i,i);
01555         }
01556     }
01557
01558     return IA;
01559 }
01560
01561 template<typename T>
01562 Matrix<T> inv_triu(const Matrix<T>& A) {
01563     if (!A.issquare()) throw std::runtime_error("Input matrix is not square");
01564
01565     const unsigned N = A.rows();
01566     auto IA = zeros<T>(N);
01567
01568     for (int i = N - 1; i >= 0; i--) {
01569         if (A(i,i) == 0.0) throw singular_matrix_exception("Division by zero in inv_triu");
01570
01571         IA(i,i) = static_cast<T>(1.0) / A(i,i);
01572         for (int j = N - 1; j > i; j--) {
01573             T s = 0.0;
01574             for (int k = i + 1; k <= j; k++)
01575                 s += A(i,k) * IA(k,j);
01576             IA(i,j) = -s * IA(i,i);
01577         }
01578     }
01579
01580     return IA;
01581 }
01582
01583 template<typename T>
01584 Matrix<T> inv_posdef(const Matrix<T>& A) {
01585     auto L = cholinv(A);
01586     return mult<T,true,false>(L,L);
01587 }
01588
01589 template<typename T>
01590 Matrix<T> inv_square(const Matrix<T>& A) {
01591     if (!A.issquare()) throw std::runtime_error("Input matrix is not square");
01592
01593     // LU decomposition with pivoting
01594     auto LU = lup(A);
01595     auto IL = inv_tril(LU.L);
01596     auto IU = inv_triu(LU.U);
01597
01598     return permute_rows(IU * IL, LU.P);
01599 }
01600
01601 template<typename T>
01602 Matrix<T> inv(const Matrix<T>& A) {
01603     if (!A.issquare()) throw std::runtime_error("Input matrix is not square");
01604
01605     if (A.numel() == 0) {
01606         return Matrix<T>();
01607     } else if (A.rows() < 4) {

```

```

01663     T d = det(A);
01664
01665     if (d == 0.0) throw singular_matrix_exception("Singular matrix in inv");
01666
01667     Matrix<T> IA(A.rows(), A.rows());
01668     T invdet = static_cast<T>(1.0) / d;
01669
01670     if (A.rows() == 1) {
01671         IA(0,0) = invdet;
01672     } else if (A.rows() == 2) {
01673         IA(0,0) = A(1,1) * invdet;
01674         IA(0,1) = - A(0,1) * invdet;
01675         IA(1,0) = - A(1,0) * invdet;
01676         IA(1,1) = A(0,0) * invdet;
01677     } else if (A.rows() == 3) {
01678         IA(0,0) = (A(1,1)*A(2,2) - A(2,1)*A(1,2)) * invdet;
01679         IA(0,1) = (A(0,2)*A(2,1) - A(0,1)*A(2,2)) * invdet;
01680         IA(0,2) = (A(0,1)*A(1,2) - A(0,2)*A(1,1)) * invdet;
01681         IA(1,0) = (A(1,2)*A(2,0) - A(1,0)*A(2,2)) * invdet;
01682         IA(1,1) = (A(0,0)*A(2,2) - A(0,2)*A(2,0)) * invdet;
01683         IA(1,2) = (A(1,0)*A(0,2) - A(0,0)*A(1,2)) * invdet;
01684         IA(2,0) = (A(1,0)*A(2,1) - A(2,0)*A(1,1)) * invdet;
01685         IA(2,1) = (A(2,0)*A(0,1) - A(0,0)*A(2,1)) * invdet;
01686         IA(2,2) = (A(0,0)*A(1,1) - A(1,0)*A(0,1)) * invdet;
01687     }
01688
01689     return IA;
01690 } else {
01691     return inv_square(A);
01692 }
01693 }
01694
01704 template<typename T>
01705 Matrix<T> pinv(const Matrix<T>& A) {
01706     if (A.rows() > A.cols()) {
01707         auto AH_A = mult<T,true,false>(A, A);
01708         auto Linv = inv_posdef(AH_A);
01709         return mult<T,false,true>(Linv, A);
01710     } else {
01711         auto AA_H = mult<T,false,true>(A, A);
01712         auto Linv = inv_posdef(AA_H);
01713         return mult<T,true,false>(A, Linv);
01714     }
01715 }
01716
01722 template<typename T>
01723 T trace(const Matrix<T>& A) {
01724     T t = static_cast<T>(0);
01725     for (int i = 0; i < A.rows(); i++)
01726         t += A(i,i);
01727     return t;
01728 }
01729
01738 template<typename T>
01739 double cond(const Matrix<T>& A) {
01740     try {
01741         auto A_inv = inv(A);
01742         return norm_fro(A) * norm_fro(A_inv);
01743     } catch (singular_matrix_exception& e) {
01744         return std::numeric_limits<double>::max();
01745     }
01746 }
01747
01766 template<typename T, bool is_upper = false>
01767 Matrix<T> chol(const Matrix<T>& A) {
01768     if (! A.issquare()) throw std::runtime_error("Input matrix is not square");
01769
01770     const unsigned N = A.rows();
01771
01772     // Calculate lower or upper triangular, depending on template parameter.
01773     // Calculation is the same - the difference is in transposed row and column indexing.
01774     Matrix<T> C = is_upper ? triu(A) : tril(A);
01775
01776     for (unsigned j = 0; j < N; j++) {
01777         if (C(j,j) == 0.0) throw singular_matrix_exception("Singular matrix in chol");
01778
01779         C(j,j) = std::sqrt(C(j,j));
01780
01781         for (unsigned k = j+1; k < N; k++)
01782             if (is_upper)
01783                 C(j,k) /= C(j,j);
01784             else
01785                 C(k,j) /= C(j,j);
01786
01787         for (unsigned k = j+1; k < N; k++)
01788             for (unsigned i = k; i < N; i++)
01789                 if (is_upper)

```

```

01790         C(k,i) -= C(j,i) * cconj(C(j,k));
01791     else
01792         C(i,k) -= C(i,j) * cconj(C(k,j));
01793     }
01794 }
01795 return C;
01796 }
01797
01809 template<typename T>
01810 Matrix<T> cholinv(const Matrix<T>& A) {
01811     if (! A.issquare()) throw std::runtime_error("Input matrix is not square");
01812
01813     const unsigned N = A.rows();
01814     Matrix<T> L(A);
01815     auto Linv = eye<T>(N);
01816
01817     for (unsigned j = 0; j < N; j++) {
01818         if (L(j,j) == 0.0) throw singular_matrix_exception("Singular matrix in cholinv");
01819
01820         L(j,j) = 1.0 / std::sqrt(L(j,j));
01821
01822         for (unsigned k = j+1; k < N; k++)
01823             L(k,j) = L(k,j) * L(j,j);
01824
01825         for (unsigned k = j+1; k < N; k++)
01826             for (unsigned i = k; i < N; i++)
01827                 L(i,k) = L(i,k) - L(i,j) * cconj(L(k,j));
01828     }
01829
01830     for (unsigned k = 0; k < N; k++) {
01831         for (unsigned i = k; i < N; i++) {
01832             Linv(i,k) = Linv(i,k) * L(i,i);
01833             for (unsigned j = i+1; j < N; j++)
01834                 Linv(j,k) = Linv(j,k) - L(j,i) * Linv(i,k);
01835         }
01836     }
01837
01838     return Linv;
01839 }
01840
01856 template<typename T>
01857 LDL_result<T> ldl(const Matrix<T>& A) {
01858     if (! A.issquare()) throw std::runtime_error("Input matrix is not square");
01859
01860     const unsigned N = A.rows();
01861
01862     LDL_result<T> res;
01863
01864     // aliases
01865     auto& L = res.L;
01866     auto& d = res.d;
01867
01868     L = eye<T>(N);
01869     d.resize(N);
01870
01871     for (unsigned m = 0; m < N; m++) {
01872         d[m] = A(m,m);
01873
01874         for (unsigned k = 0; k < m; k++)
01875             d[m] -= L(m,k) * cconj(L(m,k)) * d[k];
01876
01877         if (d[m] == 0.0) throw singular_matrix_exception("Singular matrix in ldl");
01878
01879         for (unsigned n = m+1; n < N; n++) {
01880             L(n,m) = A(n,m);
01881             for (unsigned k = 0; k < m; k++)
01882                 L(n,m) -= L(n,k) * cconj(L(m,k)) * d[k];
01883             L(n,m) /= d[m];
01884         }
01885     }
01886
01887     return res;
01888 }
01889
01901 template<typename T>
01902 QR_result<T> qr_red_gs(const Matrix<T>& A) {
01903     const int rows = A.rows();
01904     const int cols = A.cols();
01905
01906     QR_result<T> res;
01907
01908     //aliases
01909     auto& Q = res.Q;
01910     auto& R = res.R;
01911
01912     Q = zeros<T>(rows, cols);
01913     R = zeros<T>(cols, cols);

```

```

01914
01915     for (int c = 0; c < cols; c++) {
01916         Matrix<T> v = A.get_submatrix(0, rows-1, c, c);
01917         for (int r = 0; r < c; r++) {
01918             for (int k = 0; k < rows; k++)
01919                 R(r,c) = R(r,c) + cconj(Q(k,r)) * A(k,c);
01920             for (int k = 0; k < rows; k++)
01921                 v(k) = v(k) - R(r,c) * Q(k,r);
01922         }
01923         R(c,c) = static_cast<T>(norm_fro(v));
01924
01925         if (R(c,c) == 0.0) throw singular_matrix_exception("Division by 0 in QR GS");
01926         for (int k = 0; k < rows; k++)
01927             Q(k,c) = v(k) / R(c,c);
01928     }
01929     return res;
01930 }
01931
01932 template<typename T>
01933 Matrix<T> householder_reflection(const Matrix<T>& a) {
01934     if (a.cols() != 1) throw std::runtime_error("Input not a column vector");
01935     static const T ISQRT2 = static_cast<T>(0.707106781186547);
01936     Matrix<T> v(a);
01937     v(0) += csign(v(0)) * norm_fro(v);
01938     auto vn = norm_fro(v) * ISQRT2;
01939     for (unsigned i = 0; i < v.numel(); i++)
01940         v(i) /= vn;
01941     return v;
01942 }
01943
01944 template<typename T>
01945 QR_result<T> qr_householder(const Matrix<T>& A, bool calculate_Q = true) {
01946     const unsigned rows = A.rows();
01947     const unsigned cols = A.cols();
01948     QR_result<T> res;
01949     //aliases
01950     auto& Q = res.Q;
01951     auto& R = res.R;
01952     R = Matrix<T>(A);
01953     if (calculate_Q)
01954         Q = eye<T>(rows);
01955     const unsigned N = (rows > cols) ? cols : rows;
01956     for (unsigned j = 0; j < N; j++) {
01957         auto v = householder_reflection(R.get_submatrix(j, rows-1, j, j));
01958         auto R1 = R.get_submatrix(j, rows-1, j, cols-1);
01959         auto WR = v * mult<T,true,false>(v, R1);
01960         for (unsigned c = j; c < cols; c++)
01961             for (unsigned r = j; r < rows; r++)
01962                 R(r,c) -= WR(r-j,c-j);
01963         if (calculate_Q) {
01964             auto Q1 = Q.get_submatrix(0, rows-1, j, rows-1);
01965             auto WQ = mult<T,false,true>(Q1 * v, v);
01966             for (unsigned c = j; c < rows; c++)
01967                 for (unsigned r = 0; r < rows; r++)
01968                     Q(r,c) -= WQ(r,c-j);
01969         }
01970     }
01971     for (unsigned col = 0; col < R.cols(); col++)
01972         for (unsigned row = col+1; row < R.rows(); row++)
01973             R(row,col) = 0;
01974     return res;
01975 }
01976
01977 template<typename T>
01978 inline QR_result<T> qr(const Matrix<T>& A, bool calculate_Q = true) {
01979     return qr_householder(A, calculate_Q);
01980 }
01981
01982 template<typename T>
01983 Hessenberg_result<T> hessenberg(const Matrix<T>& A, bool calculate_Q = true) {
01984     if (! A.isquare()) throw std::runtime_error("Input matrix is not square");
01985 }

```

```

02040     Hessenberg_result<T> res;
02041
02042     // aliases
02043     auto& H = res.H;
02044     auto& Q = res.Q;
02045
02046     const unsigned N = A.rows();
02047     H = Matrix<T>(A);
02048
02049     if (calculate_Q)
02050         Q = eye<T>(N);
02051
02052     for (unsigned k = 1; k < N-1; k++) {
02053         auto v = householder_reflection(H.get_submatrix(k, N-1, k-1, k-1));
02054
02055         auto H1 = H.get_submatrix(k, N-1, 0, N-1);
02056         auto W1 = v * mult<T,true,false>(v, H1);
02057         for (unsigned c = 0; c < N; c++)
02058             for (unsigned r = k; r < N; r++)
02059                 H(r,c) -= W1(r-k,c);
02060
02061         auto H2 = H.get_submatrix(0, N-1, k, N-1);
02062         auto W2 = mult<T,false,true>(H2 * v, v);
02063         for (unsigned c = k; c < N; c++)
02064             for (unsigned r = 0; r < N; r++)
02065                 H(r,c) -= W2(r,c-k);
02066
02067         if (calculate_Q) {
02068             auto Q1 = Q.get_submatrix(0, N-1, k, N-1);
02069             auto W3 = mult<T,false,true>(Q1 * v, v);
02070             for (unsigned c = k; c < N; c++)
02071                 for (unsigned r = 0; r < N; r++)
02072                     Q(r,c) -= W3(r,c-k);
02073         }
02074     }
02075
02076     for (unsigned row = 2; row < N; row++)
02077         for (unsigned col = 0; col < row-2; col++)
02078             H(row,col) = static_cast<T>(0);
02079
02080     return res;
02081 }
02082
02091 template<typename T>
02092 std::complex<T> wilkinson_shift(const Matrix<std::complex<T>& H, T tol = 1e-10) {
02093     if (! H.issquare()) throw std::runtime_error("Input matrix is not square");
02094
02095     const unsigned n = H.rows();
02096     std::complex<T> mu;
02097
02098     if (std::abs(H(n-1,n-2)) < tol) {
02099         mu = H(n-2,n-2);
02100     } else {
02101         auto trA = H(n-2,n-2) + H(n-1,n-1);
02102         auto detA = H(n-2,n-2) * H(n-1,n-1) - H(n-2, n-1) * H(n-1, n-2);
02103         mu = (trA + std::sqrt(trA*trA - 4.0*detA)) / 2.0;
02104     }
02105
02106     return mu;
02107 }
02108
02120 template<typename T>
02121 Eigenvalues_result<T> eigenvalues(const Matrix<std::complex<T>& A, T tol = 1e-12, unsigned max_iter =
02122 100) {
02123     if (! A.issquare()) throw std::runtime_error("Input matrix is not square");
02124
02125     const unsigned N = A.rows();
02126     Matrix<std::complex<T>> H;
02127     bool success = false;
02128
02129     QR_result<std::complex<T>> QR;
02130
02131     // aliases
02132     auto& Q = QR.Q;
02133     auto& R = QR.R;
02134
02135     // Transfer A to Hessenberg form to improve convergence (skip calculation of Q)
02136     H = hessenberg(A, false).H;
02137
02138     for (unsigned iter = 0; iter < max_iter; iter++) {
02139         auto mu = wilkinson_shift(H, tol);
02140
02141         // subtract mu from diagonal
02142         for (unsigned n = 0; n < N; n++)
02143             H(n,n) -= mu;
02144
02145         // QR factorization with shifted H

```

```

02145     QR = qr(H);
02146     H = R * Q;
02147
02148     // add back mu to diagonal
02149     for (unsigned n = 0; n < N; n++)
02150         H(n,n) += mu;
02151
02152     // Check for convergence
02153     if (std::abs(H(N-2,N-1)) <= tol) {
02154         success = true;
02155         break;
02156     }
02157 }
02158
02159 Eigenvalues_result<T> res;
02160 res.eig = diag(H);
02161 res.err = std::abs(H(N-2,N-1));
02162 res.converged = success;
02163
02164 return res;
02165 }
02166
02176 template<typename T>
02177 Eigenvalues_result<T> eigenvalues(const Matrix<T>& A, T tol = 1e-12, unsigned max_iter = 100) {
02178     auto A_cplx = make_complex(A);
02179     return eigenvalues(A_cplx, tol, max_iter);
02180 }
02181
02197 template<typename T>
02198 Matrix<T> solve_triu(const Matrix<T>& U, const Matrix<T>& B) {
02199     if (! U.issquare()) throw std::runtime_error("Input matrix is not square");
02200     if (U.rows() != B.rows()) throw std::runtime_error("Unmatching matrix dimensions for solve");
02201
02202     const unsigned N = U.rows();
02203     const unsigned M = B.cols();
02204
02205     if (U.numel() == 0)
02206         return Matrix<T>();
02207
02208     Matrix<T> X(B);
02209
02210     for (unsigned m = 0; m < M; m++) {
02211         // backwards substitution for each column of B
02212         for (int n = N-1; n >= 0; n--) {
02213             for (unsigned j = n + 1; j < N; j++)
02214                 X(n,m) -= U(n,j) * X(j,m);
02215
02216             if (U(n,n) == 0.0) throw singular_matrix_exception("Singular matrix in solve_triu");
02217
02218             X(n,m) /= U(n,n);
02219         }
02220     }
02221
02222     return X;
02223 }
02224
02240 template<typename T>
02241 Matrix<T> solve_tril(const Matrix<T>& L, const Matrix<T>& B) {
02242     if (! L.issquare()) throw std::runtime_error("Input matrix is not square");
02243     if (L.rows() != B.rows()) throw std::runtime_error("Unmatching matrix dimensions for solve");
02244
02245     const unsigned N = L.rows();
02246     const unsigned M = B.cols();
02247
02248     if (L.numel() == 0)
02249         return Matrix<T>();
02250
02251     Matrix<T> X(B);
02252
02253     for (unsigned m = 0; m < M; m++) {
02254         // forwards substitution for each column of B
02255         for (unsigned n = 0; n < N; n++) {
02256             for (unsigned j = 0; j < n; j++)
02257                 X(n,m) -= L(n,j) * X(j,m);
02258
02259             if (L(n,n) == 0.0) throw singular_matrix_exception("Singular matrix in solve_tril");
02260
02261             X(n,m) /= L(n,n);
02262         }
02263     }
02264
02265     return X;
02266 }
02267
02283 template<typename T>
02284 Matrix<T> solve_square(const Matrix<T>& A, const Matrix<T>& B) {
02285     if (! A.issquare()) throw std::runtime_error("Input matrix is not square");

```



```

02286     if (A.rows() != B.rows()) throw std::runtime_error("Unmatching matrix dimensions for solve");
02287
02288     if (A.numel() == 0)
02289         return Matrix<T>();
02290
02291     Matrix<T> L;
02292     Matrix<T> U;
02293     std::vector<unsigned> P;
02294
02295     // LU decomposition with pivoting
02296     auto lup_res = lup(A);
02297
02298     auto y = solve_tril(lup_res.L, B);
02299     auto x = solve_triu(lup_res.U, y);
02300
02301     return permute_rows(x, lup_res.P);
02302 }
02303
02319 template<typename T>
02320 Matrix<T> solve_posdef(const Matrix<T>& A, const Matrix<T>& B) {
02321     if (! A.issquare()) throw std::runtime_error("Input matrix is not square");
02322     if (A.rows() != B.rows()) throw std::runtime_error("Unmatching matrix dimensions for solve");
02323
02324     if (A.numel() == 0)
02325         return Matrix<T>();
02326
02327     // LU decomposition with pivoting
02328     auto L = chol(A);
02329
02330     auto Y = solve_tril(L, B);
02331     return solve_triu(L.ctranspose(), Y);
02332 }
02333
02338 template<typename T>
02339 class Matrix {
02340 public:
02341     Matrix();
02342
02343     Matrix(unsigned size);
02344
02345     Matrix(unsigned nrow, unsigned ncol);
02346
02347     Matrix(T x, unsigned nrow, unsigned ncol);
02348
02349     Matrix(const T* array, unsigned nrow, unsigned ncol);
02350
02351     Matrix(const std::vector<T>& vec, unsigned nrow, unsigned ncol);
02352
02353     Matrix(std::initializer_list<T> init_list, unsigned nrow, unsigned ncol);
02354
02355     Matrix(const Matrix &);
02356
02357     virtual ~Matrix();
02358
02359     Matrix<T> get_submatrix(unsigned row_first, unsigned row_last, unsigned col_first, unsigned
02360 col_last) const;
02361
02362     void set_submatrix(const Matrix<T>& smtx, unsigned row_first, unsigned col_first);
02363
02364     void clear();
02365
02366     void reshape(unsigned rows, unsigned cols);
02367
02368     void resize(unsigned rows, unsigned cols);
02369
02370     bool exists(unsigned row, unsigned col) const;
02371
02372     T* ptr(unsigned row, unsigned col);
02373
02374     T* ptr();
02375
02376     void fill(T value);
02377
02378     void fill_col(T value, unsigned col);
02379
02380     void fill_row(T value, unsigned row);
02381
02382     bool isempty() const;
02383
02384     bool issquare() const;
02385
02386     bool isequal(const Matrix<T>&) const;
02387
02388     bool isequal(const Matrix<T>&, T) const;
02389
02390     unsigned numel() const;
02391
02392

```

```

02523     unsigned rows() const;
02524
02529     unsigned cols() const;
02530
02536     std::pair<unsigned,unsigned> shape() const;
02537
02542     Matrix<T> transpose() const;
02543
02549     Matrix<T> ctranspose() const;
02550
02558     Matrix<T>& add(const Matrix<T>&);
02559
02567     Matrix<T>& subtract(const Matrix<T>&);
02568
02577     Matrix<T>& mult_hadamard(const Matrix<T>&);
02578
02584     Matrix<T>& add(T);
02585
02591     Matrix<T>& subtract(T);
02592
02598     Matrix<T>& mult(T);
02599
02605     Matrix<T>& div(T);
02606
02611     Matrix<T>& operator=(const Matrix<T>&);
02612
02618     Matrix<T>& operator=(T);
02619
02625     explicit operator std::vector<T>() const;
02626     std::vector<T> to_vector() const;
02627
02634     T& operator()(unsigned nel);
02635     T operator()(unsigned nel) const;
02636     T& at(unsigned nel);
02637     T at(unsigned nel) const;
02638
02645     T& operator()(unsigned row, unsigned col);
02646     T operator()(unsigned row, unsigned col) const;
02647     T& at(unsigned row, unsigned col);
02648     T at(unsigned row, unsigned col) const;
02649
02657     void add_row_to_another(unsigned to, unsigned from);
02658
02666     void add_col_to_another(unsigned to, unsigned from);
02667
02675     void mult_row_by_another(unsigned to, unsigned from);
02676
02684     void mult_col_by_another(unsigned to, unsigned from);
02685
02692     void swap_rows(unsigned i, unsigned j);
02693
02700     void swap_cols(unsigned i, unsigned j);
02701
02708     std::vector<T> col_to_vector(unsigned col) const;
02709
02716     std::vector<T> row_to_vector(unsigned row) const;
02717
02726     void col_from_vector(const std::vector<T>&, unsigned col);
02727
02736     void row_from_vector(const std::vector<T>&, unsigned row);
02737
02738 private:
02739     unsigned nrows;
02740     unsigned ncols;
02741     std::vector<T> data;
02742 };
02743
02744 /*
02745  * Implementation of Matrix class methods
02746  */
02747
02748 template<typename T>
02749 Matrix<T>::Matrix() : nrows(0), ncols(0), data() { }
02750
02751 template<typename T>
02752 Matrix<T>::Matrix(unsigned size) : Matrix(size, size) { }
02753
02754 template<typename T>
02755 Matrix<T>::Matrix(unsigned rows, unsigned cols) : nrows(rows), ncols(cols) {
02756     data.resize(numel());
02757 }
02758
02759 template<typename T>
02760 Matrix<T>::Matrix(T x, unsigned rows, unsigned cols) : Matrix(rows, cols) {
02761     fill(x);
02762 }
02763

```

```

02764 template<typename T>
02765 Matrix<T>::Matrix(const T* array, unsigned rows, unsigned cols) : Matrix(rows, cols) {
02766     data.assign(array, array + numel());
02767 }
02768
02769 template<typename T>
02770 Matrix<T>::Matrix(const std::vector<T>& vec, unsigned rows, unsigned cols) : Matrix(rows, cols) {
02771     if (vec.size() != numel()) throw std::runtime_error("Size of initialization vector not consistent
    with matrix dimensions");
02772     data.assign(vec.begin(), vec.end());
02773 }
02774
02775 template<typename T>
02776 Matrix<T>::Matrix(std::initializer_list<T> init_list, unsigned rows, unsigned cols) : Matrix(rows,
    cols) {
02777     if (init_list.size() != numel()) throw std::runtime_error("Size of initialization list not
    consistent with matrix dimensions");
02778     auto it = init_list.begin();
02779     for (unsigned row = 0; row < this->nrows; row++)
02780         for (unsigned col = 0; col < this->ncols; col++)
02781             this->at(row, col) = *(it++);
02782 }
02783
02784 template<typename T>
02785 Matrix<T>::Matrix(const Matrix & other) : Matrix(other.nrows, other.ncols) {
02786     this->data.assign(other.data.begin(), other.data.end());
02787 }
02788
02789 template<typename T>
02790 Matrix<T>& Matrix<T>::operator=(const Matrix<T>& other) {
02791     this->nrows = other.nrows;
02792     this->ncols = other.ncols;
02793     this->data.assign(other.data.begin(), other.data.end());
02794     return *this;
02795 }
02796
02797 template<typename T>
02798 Matrix<T>& Matrix<T>::operator=(T s) {
02799     fill(s);
02800     return *this;
02801 }
02802
02803 template<typename T>
02804 inline Matrix<T>::operator std::vector<T>() const {
02805     return data;
02806 }
02807
02808 template<typename T>
02809 inline void Matrix<T>::clear() {
02810     this->nrows = 0;
02811     this->ncols = 0;
02812     data.resize(0);
02813 }
02814
02815 template<typename T>
02816 void Matrix<T>::reshape(unsigned rows, unsigned cols) {
02817     if (this->numel() != rows * cols) throw std::runtime_error("Illegal attempt to change number of
    elements via reshape");
02818     this->nrows = rows;
02819     this->ncols = cols;
02820 }
02821
02822 template<typename T>
02823 void Matrix<T>::resize(unsigned rows, unsigned cols) {
02824     this->nrows = rows;
02825     this->ncols = cols;
02826     data.resize(nrows*ncols);
02827 }
02828
02829 template<typename T>
02830 Matrix<T> Matrix<T>::get_submatrix(unsigned row_base, unsigned row_lim, unsigned col_base, unsigned
    col_lim) const {
02831     if (row_base > row_lim) throw std::out_of_range("Row index of submatrix out of range");
02832     if (col_base > col_lim) throw std::out_of_range("Column index of submatrix out of range");
02833     if (row_lim >= this->rows()) throw std::out_of_range("Row index of submatrix out of range");
02834     if (col_lim >= this->cols()) throw std::out_of_range("Column index of submatrix out of range");
02835     unsigned num_rows = row_lim - row_base + 1;
02836     unsigned num_cols = col_lim - col_base + 1;
02837     Matrix<T> S(num_rows, num_cols);
02838     for (unsigned i = 0; i < num_rows; i++) {
02839         for (unsigned j = 0; j < num_cols; j++) {
02840             S(i, j) = at(row_base + i, col_base + j);
02841         }
02842     }
02843 }

```

```

02846     }
02847 }
02848 return S;
02849 }
02850
02851 template<typename T>
02852 void Matrix<T>::set_submatrix(const Matrix<T>& S, unsigned row_base, unsigned col_base) {
02853     if (this->isempty()) throw std::runtime_error("Invalid attempt to set submatrix in empty matrix");
02854
02855     const unsigned row_lim = row_base + S.rows() - 1;
02856     const unsigned col_lim = col_base + S.cols() - 1;
02857
02858     if (row_base > row_lim) throw std::out_of_range("Row index of submatrix out of range");
02859     if (col_base > col_lim) throw std::out_of_range("Column index of submatrix out of range");
02860     if (row_lim >= this->rows()) throw std::out_of_range("Row index of submatrix out of range");
02861     if (col_lim >= this->cols()) throw std::out_of_range("Column index of submatrix out of range");
02862
02863     unsigned num_rows = row_lim - row_base + 1;
02864     unsigned num_cols = col_lim - col_base + 1;
02865     for (unsigned i = 0; i < num_rows; i++)
02866         for (unsigned j = 0; j < num_cols; j++)
02867             at(row_base + i, col_base + j) = S(i, j);
02868 }
02869
02870 template<typename T>
02871 inline T & Matrix<T>::operator()(unsigned nel) {
02872     return at(nel);
02873 }
02874
02875 template<typename T>
02876 inline T & Matrix<T>::operator()(unsigned row, unsigned col) {
02877     return at(row, col);
02878 }
02879
02880 template<typename T>
02881 inline T Matrix<T>::operator()(unsigned nel) const {
02882     return at(nel);
02883 }
02884
02885 template<typename T>
02886 inline T Matrix<T>::operator()(unsigned row, unsigned col) const {
02887     return at(row, col);
02888 }
02889
02890 template<typename T>
02891 inline T & Matrix<T>::at(unsigned nel) {
02892     if (!(nel < numel())) throw std::out_of_range("Element index out of range");
02893
02894     return data[nel];
02895 }
02896
02897 template<typename T>
02898 inline T & Matrix<T>::at(unsigned row, unsigned col) {
02899     if (!(row < rows() && col < cols())) throw std::out_of_range("Element index out of range");
02900
02901     return data[nrows * col + row];
02902 }
02903
02904 template<typename T>
02905 inline T Matrix<T>::at(unsigned nel) const {
02906     if (!(nel < numel())) throw std::out_of_range("Element index out of range");
02907
02908     return data[nel];
02909 }
02910
02911 template<typename T>
02912 inline T Matrix<T>::at(unsigned row, unsigned col) const {
02913     if (!(row < rows())) throw std::out_of_range("Row index out of range");
02914     if (!(col < cols())) throw std::out_of_range("Column index out of range");
02915
02916     return data[nrows * col + row];
02917 }
02918
02919 template<typename T>
02920 inline void Matrix<T>::fill(T value) {
02921     for (unsigned i = 0; i < numel(); i++)
02922         data[i] = value;
02923 }
02924
02925 template<typename T>
02926 inline void Matrix<T>::fill_col(T value, unsigned col) {
02927     if (!(col < cols())) throw std::out_of_range("Column index out of range");
02928
02929     for (unsigned i = col * nrows; i < (col+1) * nrows; i++)
02930         data[i] = value;
02931 }
02932

```

```

02933 template<typename T>
02934 inline void Matrix<T>::fill_row(T value, unsigned row) {
02935     if (!(row < rows())) throw std::out_of_range("Row index out of range");
02936     for (unsigned i = 0; i < ncols; i++)
02937         data[row + i * nrows] = value;
02938 }
02939 }
02940
02941 template<typename T>
02942 inline bool Matrix<T>::exists(unsigned row, unsigned col) const {
02943     return (row < nrows && col < ncols);
02944 }
02945
02946 template<typename T>
02947 inline T* Matrix<T>::ptr(unsigned row, unsigned col) {
02948     if (!(row < rows())) throw std::out_of_range("Row index out of range");
02949     if (!(col < cols())) throw std::out_of_range("Column index out of range");
02950     return data.data() + nrows * col + row;
02951 }
02952 }
02953
02954 template<typename T>
02955 inline T* Matrix<T>::ptr() {
02956     return data.data();
02957 }
02958
02959 template<typename T>
02960 inline bool Matrix<T>::isempty() const {
02961     return (nrows == 0) || (ncols == 0);
02962 }
02963
02964 template<typename T>
02965 inline bool Matrix<T>::issquare() const {
02966     return (nrows == ncols) && !isempty();
02967 }
02968
02969 template<typename T>
02970 bool Matrix<T>::isequal(const Matrix<T>& A) const {
02971     bool ret = true;
02972     if (nrows != A.rows() || ncols != A.cols()) {
02973         ret = false;
02974     } else {
02975         for (unsigned i = 0; i < numel(); i++) {
02976             if (at(i) != A(i)) {
02977                 ret = false;
02978                 break;
02979             }
02980         }
02981     }
02982     return ret;
02983 }
02984
02985 template<typename T>
02986 bool Matrix<T>::isequal(const Matrix<T>& A, T tol) const {
02987     bool ret = true;
02988     if (rows() != A.rows() || cols() != A.cols()) {
02989         ret = false;
02990     } else {
02991         auto abs_tol = std::abs(tol); // workaround for complex
02992         for (unsigned i = 0; i < A.numel(); i++) {
02993             if (abs_tol < std::abs(at(i) - A(i))) {
02994                 ret = false;
02995                 break;
02996             }
02997         }
02998     }
02999     return ret;
03000 }
03001
03002 template<typename T>
03003 inline unsigned Matrix<T>::numel() const {
03004     return nrows * ncols;
03005 }
03006
03007 template<typename T>
03008 inline unsigned Matrix<T>::rows() const {
03009     return nrows;
03010 }
03011
03012 template<typename T>
03013 inline unsigned Matrix<T>::cols() const {
03014     return ncols;
03015 }
03016
03017 template<typename T>
03018 inline std::pair<unsigned, unsigned> Matrix<T>::shape() const {
03019     return std::pair<unsigned, unsigned>(nrows, ncols);

```

```

03020 }
03021
03022 template<typename T>
03023 inline Matrix<T> Matrix<T>::transpose() const {
03024     Matrix<T> res(ncols, nrows);
03025     for (unsigned c = 0; c < ncols; c++)
03026         for (unsigned r = 0; r < nrows; r++)
03027             res(c,r) = at(r,c);
03028     return res;
03029 }
03030
03031 template<typename T>
03032 inline Matrix<T> Matrix<T>::ctranspose() const {
03033     Matrix<T> res(ncols, nrows);
03034     for (unsigned c = 0; c < ncols; c++)
03035         for (unsigned r = 0; r < nrows; r++)
03036             res(c,r) = cconj(at(r,c));
03037     return res;
03038 }
03039
03040 template<typename T>
03041 Matrix<T>& Matrix<T>::add(const Matrix<T>& m) {
03042     if (! (m.rows() == rows() && m.cols() == cols())) throw std::runtime_error("Unmatching matrix
dimensions for iadd");
03043     for (unsigned i = 0; i < numel(); i++)
03044         data[i] += m(i);
03045     return *this;
03046 }
03047
03048 template<typename T>
03049 Matrix<T>& Matrix<T>::subtract(const Matrix<T>& m) {
03050     if (! (m.rows() == rows() && m.cols() == cols())) throw std::runtime_error("Unmatching matrix
dimensions for isubtract");
03051     for (unsigned i = 0; i < numel(); i++)
03052         data[i] -= m(i);
03053     return *this;
03054 }
03055
03056 template<typename T>
03057 Matrix<T>& Matrix<T>::mult_hadamard(const Matrix<T>& m) {
03058     if (! (m.rows() == rows() && m.cols() == cols())) throw std::runtime_error("Unmatching matrix
dimensions for ihprod");
03059     for (unsigned i = 0; i < numel(); i++)
03060         data[i] *= m(i);
03061     return *this;
03062 }
03063
03064 template<typename T>
03065 Matrix<T>& Matrix<T>::add(T s) {
03066     for (auto& x : data)
03067         x += s;
03068     return *this;
03069 }
03070
03071 template<typename T>
03072 Matrix<T>& Matrix<T>::subtract(T s) {
03073     for (auto& x : data)
03074         x -= s;
03075     return *this;
03076 }
03077
03078 template<typename T>
03079 Matrix<T>& Matrix<T>::mult(T s) {
03080     for (auto& x : data)
03081         x *= s;
03082     return *this;
03083 }
03084
03085 template<typename T>
03086 Matrix<T>& Matrix<T>::div(T s) {
03087     for (auto& x : data)
03088         x /= s;
03089     return *this;
03090 }
03091
03092 template<typename T>
03093 void Matrix<T>::add_row_to_another(unsigned to, unsigned from) {
03094     if (! (to < rows() && from < rows())) throw std::out_of_range("Row index out of range");
03095     for (unsigned k = 0; k < cols(); k++)
03096         at(to, k) += at(from, k);
03097 }
03098
03099 template<typename T>

```

```

03104 void Matrix<T>::add_col_to_another(unsigned to, unsigned from) {
03105     if (!(to < cols() && from < cols())) throw std::out_of_range("Column index out of range");
03106
03107     for (unsigned k = 0; k < rows(); k++)
03108         at(k, to) += at(k, from);
03109 }
03110
03111 template<typename T>
03112 void Matrix<T>::mult_row_by_another(unsigned to, unsigned from) {
03113     if (!(to < rows() && from < rows())) throw std::out_of_range("Row index out of range");
03114
03115     for (unsigned k = 0; k < cols(); k++)
03116         at(to, k) *= at(from, k);
03117 }
03118
03119 template<typename T>
03120 void Matrix<T>::mult_col_by_another(unsigned to, unsigned from) {
03121     if (!(to < cols() && from < cols())) throw std::out_of_range("Column index out of range");
03122
03123     for (unsigned k = 0; k < rows(); k++)
03124         at(k, to) *= at(k, from);
03125 }
03126
03127 template<typename T>
03128 void Matrix<T>::swap_rows(unsigned i, unsigned j) {
03129     if (!(i < rows() && j < rows())) throw std::out_of_range("Row index out of range");
03130
03131     for (unsigned k = 0; k < cols(); k++) {
03132         T tmp = at(i, k);
03133         at(i, k) = at(j, k);
03134         at(j, k) = tmp;
03135     }
03136 }
03137
03138 template<typename T>
03139 void Matrix<T>::swap_cols(unsigned i, unsigned j) {
03140     if (!(i < cols() && j < cols())) throw std::out_of_range("Column index out of range");
03141
03142     for (unsigned k = 0; k < rows(); k++) {
03143         T tmp = at(k, i);
03144         at(k, i) = at(k, j);
03145         at(k, j) = tmp;
03146     }
03147 }
03148
03149 template<typename T>
03150 inline std::vector<T> Matrix<T>::to_vector() const {
03151     return data;
03152 }
03153
03154 template<typename T>
03155 inline std::vector<T> Matrix<T>::col_to_vector(unsigned col) const {
03156     std::vector<T> vec(rows());
03157     for (unsigned i = 0; i < rows(); i++)
03158         vec[i] = at(i, col);
03159     return vec;
03160 }
03161
03162 template<typename T>
03163 inline std::vector<T> Matrix<T>::row_to_vector(unsigned row) const {
03164     std::vector<T> vec(cols());
03165     for (unsigned i = 0; i < cols(); i++)
03166         vec[i] = at(row, i);
03167     return vec;
03168 }
03169
03170 template<typename T>
03171 inline void Matrix<T>::col_from_vector(const std::vector<T>& vec, unsigned col) {
03172     if (vec.size() != rows()) throw std::runtime_error("Vector size is not equal to number of rows");
03173     if (col >= cols()) throw std::out_of_range("Column index out of range");
03174
03175     for (unsigned i = 0; i < rows(); i++)
03176         data[col*rows() + i] = vec[i];
03177 }
03178
03179 template<typename T>
03180 inline void Matrix<T>::row_from_vector(const std::vector<T>& vec, unsigned row) {
03181     if (vec.size() != cols()) throw std::runtime_error("Vector size is not equal to number of columns");
03182     if (row >= rows()) throw std::out_of_range("Row index out of range");
03183
03184     for (unsigned i = 0; i < cols(); i++)
03185         data[row + i*rows()] = vec[i];
03186 }
03187
03188 template<typename T>
03189 Matrix<T>::~Matrix() { }
03190

```

```
03191 } // namespace Matrix_hpp
03192
03193 #endif // __MATRIX_HPP__
```