

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

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

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

6.2.1.81 repmat()	68
6.2.1.82 solve_posdef()	68
6.2.1.83 solve_square()	69
6.2.1.84 solve_tril()	70
6.2.1.85 solve_triu()	70
6.2.1.86 subtract() [1/2]	71
6.2.1.87 subtract() [2/2]	72
6.2.1.88 trace()	72
6.2.1.89 transpose()	72
6.2.1.90 tril()	73
6.2.1.91 triu()	73
6.2.1.92 wilkinson_shift()	73
6.2.1.93 zeros() [1/2]	74
6.2.1.94 zeros() [2/2]	74
6.3 matrix.hpp	74

Chapter 1

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

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

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

1.1 Installation

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

1.2 Functionality

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

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

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

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

1.3 Hello world example

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

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

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

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

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

    auto C = A + B;

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

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

1.4 Tests

Unit tests are compiled with `make tests`.

1.5 License

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

Chapter 2

Hierarchical Index

2.1 Class Hierarchy

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

std::domain_error	
Mtx::singular_matrix_exception	30
Mtx::Eigenvalues_result< T >	9
Mtx::Hessenberg_result< T >	9
Mtx::LDL_result< T >	10
Mtx::LU_result< T >	11
Mtx::LUP_result< T >	11
Mtx::Matrix< T >	12
Mtx::QR_result< T >	29

Chapter 3

Class Index

3.1 Class List

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

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	29
Mtx::singular_matrix_exception	
Singular matrix exception	30

Chapter 4

File Index

4.1 File List

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

examples.cpp	31
matrix.hpp	31

Chapter 5

Class Documentation

5.1 Mtx::Eigenvalues_result< T > Struct Template Reference

Result of eigenvalues.

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

Public Attributes

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

5.1.1 Detailed Description

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

Result of eigenvalues.

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

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

- [matrix.hpp](#)

5.2 Mtx::Hessenberg_result< T > Struct Template Reference

Result of Hessenberg decomposition.

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

Public Attributes

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

5.2.1 Detailed Description

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

Result of Hessenberg decomposition.

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

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

- [matrix.hpp](#)

5.3 Mtx::LDL_result< T > Struct Template Reference

Result of LDL decomposition.

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

Public Attributes

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

5.3.1 Detailed Description

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

Result of LDL decomposition.

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

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

- [matrix.hpp](#)

5.4 Mtx::LU_result< T > Struct Template Reference

Result of LU decomposition.

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

Public Attributes

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

5.4.1 Detailed Description

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

Result of LU decomposition.

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

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

- [matrix.hpp](#)

5.5 Mtx::LUP_result< T > Struct Template Reference

Result of LU decomposition with pivoting.

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

Public Attributes

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

5.5.1 Detailed Description

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

Result of LU decomposition with pivoting.

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

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

- [matrix.hpp](#)

5.6 Mtx::Matrix< T > Class Template Reference

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

Public Member Functions

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

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

- [T at \(unsigned nel\) const](#)
- [T & operator\(\) \(unsigned row, unsigned col\)](#)
Element access operator (2D)
- [T operator\(\) \(unsigned row, unsigned col\) const](#)
- [T & at \(unsigned row, unsigned col\)](#)
- [T at \(unsigned row, unsigned col\) const](#)
- [void add_row_to_another \(unsigned to, unsigned from\)](#)
Row addition.
- [void add_col_to_another \(unsigned to, unsigned from\)](#)
Column addition.
- [void mult_row_by_another \(unsigned to, unsigned from\)](#)
Row multiplication.
- [void mult_col_by_another \(unsigned to, unsigned from\)](#)
Column multiplication.
- [void swap_rows \(unsigned i, unsigned j\)](#)
Row swap.
- [void swap_cols \(unsigned i, unsigned j\)](#)
Column swap.
- [std::vector< T > col_to_vector \(unsigned col\) const](#)
Column to vector.
- [std::vector< T > row_to_vector \(unsigned row\) const](#)
Row to vector.
- [void col_from_vector \(const std::vector< T > &, unsigned col\)](#)
Column from vector.
- [void row_from_vector \(const std::vector< T > &, unsigned row\)](#)
Row from vector.

5.6.1 Detailed Description

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

[Matrix](#) class definition.

5.6.2 Constructor & Destructor Documentation

5.6.2.1 Matrix() [1/8]

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

Default constructor.

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

5.6.2.2 Matrix() [2/8]

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

Square matrix constructor.

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

5.6.2.3 Matrix() [3/8]

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

Rectangular matrix constructor.

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

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

5.6.2.4 Matrix() [4/8]

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

Rectangular matrix constructor with fill.

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

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

5.6.2.5 Matrix() [5/8]

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

Rectangular matrix constructor with initialization.

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

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

5.6.2.6 Matrix() [6/8]

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

Rectangular matrix constructor with initialization.

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

Exceptions

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

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

5.6.2.7 Matrix() [7/8]

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

Rectangular matrix constructor with initialization.

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

Exceptions

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

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

5.6.2.8 Matrix() [8/8]

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

Copy constructor.

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

5.6.2.9 ~Matrix()

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

Destructor.

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

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

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

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

Exceptions

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

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

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

5.6.3.2 add() [2/2]

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

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

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

5.6.3.3 add_col_to_another()

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

Column addition.

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

Exceptions

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

5.6.3.4 add_row_to_another()

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

Row addition.

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

Exceptions

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

5.6.3.5 clear()

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

Clears the matrix.

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

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

5.6.3.6 col_from_vector()

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

Column from vector.

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

Exceptions

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

5.6.3.7 col_to_vector()

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

Column to vector.

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

Exceptions

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

5.6.3.8 cols()

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

Number of columns.

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

Referenced by [Mtx::Matrix< T >::add\(\)](#), [Mtx::add\(\)](#), [Mtx::add\(\)](#), [Mtx::adj\(\)](#), [Mtx::circshift\(\)](#), [Mtx::cofactor\(\)](#), [Mtx::div\(\)](#), [Mtx::householder_reflection\(\)](#), [Mtx::Matrix< T >::isequal\(\)](#), [Mtx::Matrix< T >::isequal\(\)](#), [Mtx::istril\(\)](#), [Mtx::istriu\(\)](#), [Mtx::kron\(\)](#), [Mtx::lu\(\)](#), [Mtx::lup\(\)](#), [Mtx::make_complex\(\)](#), [Mtx::make_complex\(\)](#), [Mtx::mult\(\)](#), [Mtx::mult\(\)](#), [Mtx::mult\(\)](#), [Mtx::Matrix< T >::mult_hadamard\(\)](#), [Mtx::mult_hadamard\(\)](#), [Mtx::operator<<\(\)](#), [Mtx::permute_cols\(\)](#), [Mtx::permute_rows\(\)](#), [Mtx::pinv\(\)](#), [Mtx::qr_householder\(\)](#), [Mtx::qr_red_gs\(\)](#), [Mtx::repmat\(\)](#), [Mtx::Matrix< T >::set_submatrix\(\)](#), [Mtx::solve_tril\(\)](#), [Mtx::solve_triu\(\)](#), [Mtx::Matrix< T >::subtract\(\)](#), [Mtx::subtract\(\)](#), [Mtx::subtract\(\)](#), [Mtx::tril\(\)](#), and [Mtx::triu\(\)](#).

5.6.3.9 ctranspose()

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

Transpose a complex matrix.

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

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

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

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

5.6.3.10 div()

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

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

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

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

5.6.3.11 exists()

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

Element exist check.

Returns true if the element with specified coordinates exists within the matrix dimension range.

For example, calling *exist(4,0)* on a matrix with dimensions 2 x 2 shall yield false.

5.6.3.12 fill()

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

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

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

5.6.3.13 fill_col()

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

Fill column with a scalar.

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

Exceptions

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

5.6.3.14 fill_row()

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

Fill row with a scalar.

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

Exceptions

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

5.6.3.15 get_submatrix()

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

Extract a submatrix.

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

Exceptions

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

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

5.6.3.16 isempty()

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

Emptiness check.

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

5.6.3.17 isequal() [1/2]

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

[Matrix](#) equality check.

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

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

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

5.6.3.18 isequal() [2/2]

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

[Matrix](#) equality check with tolerance.

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

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

5.6.3.19 mult()

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

Matrix product with scalar (in-place).

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

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

5.6.3.20 mult_col_by_another()

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

Column multiplication.

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

Exceptions

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

5.6.3.21 mult_hadamard()

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

Matrix Hadamard product (in-place).

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

Exceptions

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

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

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

5.6.3.22 mult_row_by_another()

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

Row multiplication.

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

Exceptions

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

5.6.3.23 numel()

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

Matrix capacity.

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

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

5.6.3.24 operator std::vector< T >()

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

Vector cast operator.

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

5.6.3.25 operator()() [1/2]

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

Element access operator (1D)

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

Exceptions

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

5.6.3.26 operator() [2/2]

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

Element access operator (2D)

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

Exceptions

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

5.6.3.27 operator=() [1/2]

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

Matrix assignment.

Performs deep-copy of another matrix.

5.6.3.28 operator=() [2/2]

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

Matrix fill operator.

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

5.6.3.29 ptr() [1/2]

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

Memory pointer.

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

Exceptions

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

5.6.3.30 ptr() [2/2]

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

Memory pointer.

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

5.6.3.31 reshape()

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

[Matrix](#) dimension reshape.

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

Exceptions

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

5.6.3.32 resize()

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

Resize the matrix.

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

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

5.6.3.33 row_from_vector()

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

Row from vector.

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

Exceptions

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

5.6.3.34 row_to_vector()

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

Row to vector.

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

Exceptions

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

5.6.3.35 rows()

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

Number of rows.

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

Referenced by `Mtx::Matrix< T >::add()`, `Mtx::add()`, `Mtx::add()`, `Mtx::adj()`, `Mtx::chol()`, `Mtx::cholinv()`, `Mtx::circshift()`, `Mtx::cofactor()`, `Mtx::det()`, `Mtx::diag()`, `Mtx::div()`, `Mtx::eigenvalues()`, `Mtx::hessenberg()`, `Mtx::inv()`, `Mtx::inv_gauss_jordan()`, `Mtx::inv_tril()`, `Mtx::inv_triu()`, `Mtx::Matrix< T >::isequal()`, `Mtx::Matrix< T >::isequal()`, `Mtx::ishess()`, `Mtx::istril()`, `Mtx::istriu()`, `Mtx::kron()`, `Mtx::ldl()`, `Mtx::lu()`, `Mtx::lup()`, `Mtx::make_complex()`, `Mtx::make_complex()`, `Mtx::mult()`, `Mtx::mult()`, `Mtx::mult()`, `Mtx::Matrix< T >::mult_hadamard()`, `Mtx::mult_hadamard()`, `Mtx::operator<<()`, `Mtx::permute_cols()`, `Mtx::permute_rows()`, `Mtx::pinv()`, `Mtx::qr_householder()`, `Mtx::qr_red_gs()`, `Mtx::repmat()`, `Mtx::Matrix< T >::set_submatrix()`, `Mtx::solve_posdef()`, `Mtx::solve_square()`, `Mtx::solve_tril()`, `Mtx::solve_triu()`, `Mtx::Matrix< T >::subtract()`, `Mtx::subtract()`, `Mtx::subtract()`, `Mtx::trace()`, `Mtx::tril()`, and `Mtx::triu()`.

5.6.3.36 set_submatrix()

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

Embed a submatrix.

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

Exceptions

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

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

5.6.3.37 subtract() [1/2]

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

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

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

Exceptions

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

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

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

5.6.3.38 subtract() [2/2]

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

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

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

5.6.3.39 swap_cols()

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

Column swap.

Swaps element values between two columns.

Exceptions

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

5.6.3.40 swap_rows()

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

Row swap.

Swaps element values of two columns.

Exceptions

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

5.6.3.41 transpose()

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

Transpose a matrix.

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

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

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

- [matrix.hpp](#)

5.7 Mtx::QR_result< T > Struct Template Reference

Result of QR decomposition.

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

Public Attributes

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

5.7.1 Detailed Description

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

Result of QR decomposition.

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

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

- [matrix.hpp](#)

5.8 Mtx::singular_matrix_exception Class Reference

Singular matrix exception.

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

Inheritance diagram for Mtx::singular_matrix_exception:

Chapter 6

File Documentation

6.1 examples.cpp File Reference

6.1.1 Detailed Description

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

6.2 matrix.hpp File Reference

Classes

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

Functions

- `template<typename T, typename std::enable_if<!is_complex< T >::value, int >::type = 0> T Mtx::cconj (T x)`
Complex conjugate helper.
- `template<typename T, typename std::enable_if<!is_complex< T >::value, int >::type = 0> T Mtx::csign (T x)`
Complex sign helper.
- `template<typename T > Matrix< T > Mtx::zeros (unsigned nrows, unsigned ncols)`
Matrix of zeros.
- `template<typename T > Matrix< T > Mtx::zeros (unsigned n)`
Square matrix of zeros.
- `template<typename T > Matrix< T > Mtx::ones (unsigned nrows, unsigned ncols)`
Matrix of ones.
- `template<typename T > Matrix< T > Mtx::ones (unsigned n)`
Square matrix of ones.
- `template<typename T > Matrix< T > Mtx::eye (unsigned n)`
Identity matrix.
- `template<typename T > Matrix< T > Mtx::diag (const T *array, size_t n)`
Diagonal matrix from array.
- `template<typename T > Matrix< T > Mtx::diag (const std::vector< T > &v)`
Diagonal matrix from std::vector.
- `template<typename T > std::vector< T > Mtx::diag (const Matrix< T > &A)`
Diagonal extraction.
- `template<typename T > Matrix< T > Mtx::circulant (const T *array, unsigned n)`
Circulant matrix from array.
- `template<typename T > Matrix< std::complex< T > > Mtx::make_complex (const Matrix< T > &Re, const Matrix< T > &Im)`
Create complex matrix from real and imaginary matrices.
- `template<typename T > Matrix< std::complex< T > > Mtx::make_complex (const Matrix< T > &Re)`
Create complex matrix from real matrix.
- `template<typename T > Matrix< T > Mtx::real (const Matrix< std::complex< T > > &C)`
Get real part of complex matrix.
- `template<typename T > Matrix< T > Mtx::imag (const Matrix< std::complex< T > > &C)`
Get imaginary part of complex matrix.
- `template<typename T > Matrix< T > Mtx::circulant (const std::vector< T > &v)`
Circulant matrix from std::vector.
- `template<typename T > Matrix< T > Mtx::transpose (const Matrix< T > &A)`
Transpose a matrix.

- `template<typename T >`
`Matrix< T > Mtx::ctranspose (const Matrix< T > &A)`
Transpose a complex matrix.
- `template<typename T >`
`Matrix< T > Mtx::circshift (const Matrix< T > &A, int row_shift, int col_shift)`
Circular shift.
- `template<typename T >`
`Matrix< T > Mtx::repmat (const Matrix< T > &A, unsigned m, unsigned n)`
Repeat matrix.
- `template<typename T >`
`double Mtx::norm_fro (const Matrix< T > &A)`
Frobenius norm.
- `template<typename T >`
`double Mtx::norm_fro (const Matrix< std::complex< T > > &A)`
Frobenius norm of complex matrix.
- `template<typename T >`
`Matrix< T > Mtx::tril (const Matrix< T > &A)`
Extract triangular lower part.
- `template<typename T >`
`Matrix< T > Mtx::triu (const Matrix< T > &A)`
Extract triangular upper part.
- `template<typename T >`
`bool Mtx::istril (const Matrix< T > &A)`
Lower triangular matrix check.
- `template<typename T >`
`bool Mtx::istriu (const Matrix< T > &A)`
Lower triangular matrix check.
- `template<typename T >`
`bool Mtx::ishess (const Matrix< T > &A)`
Hessenberg matrix check.
- `template<typename T >`
`void Mtx::foreach_elem (Matrix< T > &A, std::function< T(T)> func)`
Applies custom function element-wise in-place.
- `template<typename T >`
`Matrix< T > Mtx::foreach_elem_copy (const Matrix< T > &A, std::function< T(T)> func)`
Applies custom function element-wise with matrix copy.
- `template<typename T >`
`Matrix< T > Mtx::permute_rows (const Matrix< T > &A, const std::vector< unsigned > perm)`
Permute rows of the matrix.
- `template<typename T >`
`Matrix< T > Mtx::permute_cols (const Matrix< T > &A, const std::vector< unsigned > perm)`
Permute columns of the matrix.
- `template<typename T, bool transpose_first = false, bool transpose_second = false>`
`Matrix< T > Mtx::mult (const Matrix< T > &A, const Matrix< T > &B)`
Matrix multiplication.
- `template<typename T, bool transpose_first = false, bool transpose_second = false>`
`Matrix< T > Mtx::mult_hadamard (const Matrix< T > &A, const Matrix< T > &B)`
Matrix Hadamard (elementwise) multiplication.
- `template<typename T, bool transpose_first = false, bool transpose_second = false>`
`Matrix< T > Mtx::add (const Matrix< T > &A, const Matrix< T > &B)`
Matrix addition.
- `template<typename T, bool transpose_first = false, bool transpose_second = false>`
`Matrix< T > Mtx::subtract (const Matrix< T > &A, const Matrix< T > &B)`

- Matrix subtraction.*

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

- Matrix product with scalar.*

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

Matrix sum.

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

Matrix subtraction.

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

Matrix product.

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

Matrix Hadamard product.

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

Matrix sum with scalar.

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

Matrix subtraction with scalar.

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

Matrix product with scalar.

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

Matrix division by scalar.

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

Matrix equality check operator.

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

Matrix non-equality check operator.

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

Kronecker product.

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

Adjugate matrix.

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

Cofactor matrix.

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

Matrix determinant from on LU decomposition.

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

Matrix determinant.

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

LU decomposition.

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

LU decomposition with pivoting.

- `template<typename T >`
`Matrix< T > Mtx::inv_gauss_jordan (const Matrix< T > &A)`
Matrix inverse using Gauss-Jordan elimination.
- `template<typename T >`
`Matrix< T > Mtx::inv_tril (const Matrix< T > &A)`
Matrix inverse for lower triangular matrix.
- `template<typename T >`
`Matrix< T > Mtx::inv_triu (const Matrix< T > &A)`
Matrix inverse for upper triangular matrix.
- `template<typename T >`
`Matrix< T > Mtx::inv_posdef (const Matrix< T > &A)`
Matrix inverse for Hermitian positive-definite matrix.
- `template<typename T >`
`Matrix< T > Mtx::inv_square (const Matrix< T > &A)`
Matrix inverse for general square matrix.
- `template<typename T >`
`Matrix< T > Mtx::inv (const Matrix< T > &A)`
Matrix inverse (universal).
- `template<typename T >`
`Matrix< T > Mtx::pinv (const Matrix< T > &A)`
Moore-Penrose pseudoinverse.
- `template<typename T >`
`T Mtx::trace (const Matrix< T > &A)`
Matrix trace.
- `template<typename T >`
`double Mtx::cond (const Matrix< T > &A)`
Condition number of a matrix.
- `template<typename T >`
`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.

6.2.1 Function Documentation

6.2.1.1 add() [1/2]

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

Matrix addition.

Performs addition of two matrices.

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

Template Parameters

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

Parameters

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

Returns

output matrix of size $N \times M$

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

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

6.2.1.2 add() [2/2]

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

Addition of scalar to matrix.

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

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

6.2.1.3 adj()

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

Adjugate matrix.

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

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

Exceptions

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

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

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

6.2.1.4 cconj()

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

Complex conjugate helper.

Helper function to allow for generalization of code for complex and real types.
 For real numbers, this function returns the input argument unchanged.
 For complex numbers, this function calls `std::conj`.

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

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

6.2.1.5 chol()

```
template<typename T >
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 L^H denotes the conjugate transpose of L .

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

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

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

6.2.1.6 cholinv()

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

Inverse of Cholesky decomposition.

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

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

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

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

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

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

6.2.1.7 circshift()

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

Circular shift.

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

Parameters

<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\(\)](#).

6.2.1.8 circulant() [1/2]

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

Circulant matrix from std::vector.

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

Parameters

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

Returns

circulant matrix

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

6.2.1.9 `circulant()` [2/2]

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

Circulant matrix from array.

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

Parameters

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

Returns

circulant matrix

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

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

6.2.1.10 `cofactor()`

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

Cofactor matrix.

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

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

Parameters

<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\(\)](#).

6.2.1.11 cond()

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

Condition number of a matrix.

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

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

Frobenius norm is used for the sake of calculations.

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

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

6.2.1.12 csign()

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

Complex sign helper.

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

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

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

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

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

6.2.1.13 ctranspose()

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

Transpose a complex matrix.

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

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

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

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

6.2.1.14 det()

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

Matrix determinant.

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

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

Exceptions

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

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

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

6.2.1.15 det_lu()

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

Matrix determinant from on LU decomposition.

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

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

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

Exceptions

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

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

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

6.2.1.16 diag() [1/3]

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

Diagonal extraction.

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

Parameters

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

Returns

vector of diagonal elements

Exceptions

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

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

6.2.1.17 diag() [2/3]

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

Diagonal matrix from `std::vector`.

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

Parameters

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

Returns

diagonal matrix

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

6.2.1.18 diag() [3/3]

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

Diagonal matrix from array.

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

Parameters

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

Returns

diagonal matrix

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

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

6.2.1.19 div()

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

Division of matrix by scalar.

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

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

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

6.2.1.20 eigenvalues() [1/2]

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

Matrix eigenvalues of complex matrix.

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

Parameters

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

Returns

structure containing the result and status of eigenvalue calculation

Exceptions

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

References [Mtx::Eigenvalues_result< T >::converged](#), [Mtx::diag\(\)](#), [Mtx::Eigenvalues_result< T >::eig](#), [Mtx::eigenvalues\(\)](#), [Mtx::Eigenvalues_result< T >::err](#), [Mtx::hessenberg\(\)](#), [Mtx::Matrix< T >::issquare\(\)](#), [Mtx::QR_result< T >::Q](#), [Mtx::qr\(\)](#), [Mtx::QR_result< T >::R](#), [Mtx::Matrix< T >::rows\(\)](#), and [Mtx::wilkinson_shift\(\)](#).

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

6.2.1.21 eigenvalues() [2/2]

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

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

Matrix eigenvalues of real matrix.

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

Parameters

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

Returns

structure containing the result and status of eigenvalue calculation

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

6.2.1.22 eye()

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

Identity matrix.

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

Parameters

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

Returns

zeros matrix

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

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

6.2.1.23 foreach_elem()

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

Applies custom function element-wise in-place.

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

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

Parameters

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

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

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

6.2.1.24 foreach_elem_copy()

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

Applies custom function element-wise with matrix copy.

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

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

Parameters

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

Returns

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

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

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

6.2.1.25 hessenberg()

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

Hessenberg decomposition.

Finds the Hessenberg decomposition of $A = QHQ^*$. 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::Hessenberg_result< T >::H](#), [Mtx::hessenberg\(\)](#), [Mtx::householder_reflection\(\)](#), [Mtx::Matrix< T >::issquare\(\)](#), [Mtx::Hessenberg_result< T >::Q](#), and [Mtx::Matrix< T >::rows\(\)](#).

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

6.2.1.26 householder_reflection()

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

Generate Householder reflection.

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

Parameters

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

Returns

column vector with Householder reflection of *a*

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

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

6.2.1.27 imag()

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

Get imaginary part of complex matrix.

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

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

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

6.2.1.28 inv()

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

Matrix inverse (universal).

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

If the inverse doesn't 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\(\)](#).

6.2.1.29 inv_gauss_jordan()

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

Matrix inverse using Gauss-Jordan elimination.

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

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

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

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

Exceptions

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

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

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

6.2.1.30 inv_posdef()

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

Matrix inverse for Hermitian positive-definite matrix.

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

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

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

Exceptions

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

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

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

6.2.1.31 inv_square()

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

Matrix inverse for general square matrix.

Calculates an inverse of square matrix using matrix.

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

Exceptions

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

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

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

6.2.1.32 inv_tril()

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

Matrix inverse for lower triangular matrix.

Calculates an inverse of lower triangular matrix.

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

Exceptions

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

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

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

6.2.1.33 inv_triu()

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

Matrix inverse for upper triangular matrix.

Calculates an inverse of upper triangular matrix.

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

Exceptions

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

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

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

6.2.1.34 ishess()

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

Hessenberg matrix check.

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

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

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

6.2.1.35 istril()

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

Lower triangular matrix check.

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

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

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

6.2.1.36 istriu()

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

Lower triangular matrix check.

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

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

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

6.2.1.37 kron()

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

Kronecker product.

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

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

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

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

6.2.1.38 ldl()

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

LDL decomposition.

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

$$A = LDL^H$$

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

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

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

Parameters

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

Returns

structure encapsulating calculated L and D

Exceptions

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

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

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

6.2.1.39 lu()

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

LU decomposition.

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

This function implements LU factorization without pivoting. Use `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_result< T >::L](#), [Mtx::lu\(\)](#), [Mtx::Matrix< T >::numel\(\)](#), [Mtx::Matrix< T >::rows\(\)](#), and [Mtx::LU_result< T >::U](#).

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

6.2.1.40 lup()

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

LU decomposition with pivoting.

Performs LU factorization with partial pivoting, employing column permutations.

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

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

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

Parameters

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

Returns

structure containing L , U and P .

References [Mtx::Matrix< T >::cols\(\)](#), [Mtx::LUP_result< T >::L](#), [Mtx::lup\(\)](#), [Mtx::Matrix< T >::numel\(\)](#), [Mtx::LUP_result< T >::P](#), [Mtx::Matrix< T >::rows\(\)](#), and [Mtx::LUP_result< T >::U](#).

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

6.2.1.41 make_complex() [1/2]

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

Create complex matrix from real matrix.

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

Parameters

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

Returns

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

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

6.2.1.42 make_complex() [2/2]

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

Create complex matrix from real and imaginary matrices.

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

Parameters

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

Returns

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

Exceptions

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

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

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

6.2.1.43 mult() [1/4]

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

Matrix multiplication.

Performs multiplication of two matrices.

This function supports template parameterization of input matrix transposition, providing better efficiency than in case of using `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*\(\)](#), and [Mtx::operator*\(\)=\(\)](#).

6.2.1.44 mult() [2/4]

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

Multiplication of matrix by std::vector.

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

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

6.2.1.45 mult() [3/4]

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

Multiplication of matrix by scalar.

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

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

6.2.1.46 mult() [4/4]

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

Multiplication of std::vector by matrix.

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

Parameters

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

Returns

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

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

6.2.1.47 mult_hadamard()

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

Matrix Hadamard (elementwise) multiplication.

Performs Hadamard (elementwise) multiplication of two matrices.

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

Template Parameters

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

Parameters

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

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

6.2.1.48 norm_fro() [1/2]

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

Frobenius norm of complex matrix.

Calculates Frobenius norm of complex matrix.

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

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

6.2.1.49 norm_fro() [2/2]

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

Frobenius norm.

Calculates Frobenius norm of real matrix.

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

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

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

6.2.1.50 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

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

Returns

zeros matrix

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

6.2.1.51 ones() [2/2]

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

Matrix of ones.

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

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

Parameters

<i>n_rows</i>	number of rows (the first dimension)
<i>n_cols</i>	number of columns (the second dimension)

Returns

ones matrix

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

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

6.2.1.52 operator!=(())

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

Matrix non-equality check operator.

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

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

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

6.2.1.53 operator*() [1/4]

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

6.2.1.54 operator*() [2/4]

```
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 a matrix and a std::vector $A \cdot v$.

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

6.2.1.55 operator*() [3/4]

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

Matrix product with scalar.

Multiplies each element of the matrix by a scalar s .

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

6.2.1.56 operator*() [4/4]

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

Matrix product with scalar.

Multiplies each element of the matrix by a scalar s .

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

6.2.1.57 operator*=() [1/2]

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

Matrix product.

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

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

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

6.2.1.58 operator*=() [2/2]

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

Matrix product with scalar.

Multiplies each element of the matrix by a scalar s .

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

6.2.1.59 operator+() [1/3]

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

Matrix sum.

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

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

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

6.2.1.60 operator+() [2/3]

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

Matrix sum with scalar.

Adds a scalar s to each element of the matrix.

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

6.2.1.61 operator+() [3/3]

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

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

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

6.2.1.62 operator+=() [1/2]

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

Matrix sum.

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

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

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

6.2.1.63 operator+=() [2/2]

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

Matrix sum with scalar.

Adds a scalar s to each element of the matrix.

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

6.2.1.64 operator-() [1/2]

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

Matrix subtraction.

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

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

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

6.2.1.65 operator-() [2/2]

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

Matrix subtraction with scalar.

Subtracts a scalar s from each element of the matrix.

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

6.2.1.66 operator-=() [1/2]

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

Matrix subtraction.

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

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

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

6.2.1.67 operator-=() [2/2]

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

Matrix subtraction with scalar.

Subtracts a scalar s from each element of the matrix.

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

6.2.1.68 operator/()

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

Matrix division by scalar.

Divides each element of the matrix by a scalar s .

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

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

6.2.1.69 operator/=()

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

Matrix division by scalar.

Divides each element of the matrix by a scalar s .

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

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

6.2.1.70 operator<<()

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

Matrix ostream operator.

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

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

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

6.2.1.71 operator==()

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

Matrix equality check operator.

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

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

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

6.2.1.72 operator^()

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

Matrix Hadamard product.

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

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

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

6.2.1.73 operator^=()

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

Matrix Hadamard product.

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

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

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

6.2.1.74 permute_cols()

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

Permute columns of the matrix.

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

Parameters

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

Returns

output matrix created by column permutation of *A*

Exceptions

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

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

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

6.2.1.75 permute_rows()

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

Permute rows of the matrix.

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

Parameters

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

Returns

output matrix created by row permutation of *A*

Exceptions

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

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

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

6.2.1.76 pinv()

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

Moore-Penrose pseudoinverse.

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

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

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

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

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

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

6.2.1.77 qr()

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

QR decomposition.

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

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

Parameters

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

Returns

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

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

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

6.2.1.78 qr_householder()

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

QR decomposition based on Householder method.

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

This function implements QR decomposition based on Householder reflections method.

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

Parameters

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

Returns

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

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

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

6.2.1.79 qr_red_gs()

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

Reduced QR decomposition based on Gram-Schmidt method.

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

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

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

Parameters

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

Returns

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

Exceptions

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

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

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

6.2.1.80 real()

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

Get real part of complex matrix.

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

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

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

6.2.1.81 repmat()

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

Repeat matrix.

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

Parameters

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

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

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

6.2.1.82 solve_posdef()

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

Solves the positive definite (Hermitian) system.

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

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

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

Parameters

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

Returns

solution matrix of size $N \times M$.

Exceptions

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

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

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

6.2.1.83 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\(\)](#).

6.2.1.84 solve_tril()

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

Solves the lower triangular system.

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

Parameters

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

Returns

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

Exceptions

<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\(\)](#).

6.2.1.85 solve_triu()

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

Solves the upper triangular system.

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

Parameters

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

Returns

solution matrix of size $N \times M$.

Exceptions

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

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

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

6.2.1.86 subtract() [1/2]

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

Matrix subtraction.

Performs subtraction of two matrices.

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

Template Parameters

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

Parameters

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

Returns

output matrix of size $N \times M$

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

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

6.2.1.87 subtract() [2/2]

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

Subtraction of scalar from matrix.

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

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

6.2.1.88 trace()

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

Matrix trace.

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

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

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

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

6.2.1.89 transpose()

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

Transpose a matrix.

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

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

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

6.2.1.90 tril()

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

Extract triangular lower part.

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

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

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

6.2.1.91 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::triu\(\)](#).

6.2.1.92 wilkinson_shift()

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

Wilkinson's shift for complex eigenvalues.

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

Exceptions

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

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

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

6.2.1.93 zeros() [1/2]

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

Square matrix of zeros.

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

Parameters

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

Returns

zeros matrix

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

6.2.1.94 zeros() [2/2]

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

Matrix of zeros.

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

Parameters

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

Returns

zeros matrix

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

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

6.3 matrix.hpp

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

00001
00002

```

00003  /* MIT License
00004  *
00005  * Copyright (c) 2024 gc1905
00006  *
00007  * Permission is hereby granted, free of charge, to any person obtaining a copy
00008  * of this software and associated documentation files (the "Software"), to deal
00009  * in the Software without restriction, including without limitation the rights
00010  * to use, copy, modify, merge, publish, distribute, sublicense, and/or sell
00011  * copies of the Software, and to permit persons to whom the Software is
00012  * furnished to do so, subject to the following conditions:
00013  *
00014  * The above copyright notice and this permission notice shall be included in all
00015  * copies or substantial portions of the Software.
00016  *
00017  * THE SOFTWARE IS PROVIDED "AS IS", WITHOUT WARRANTY OF ANY KIND, EXPRESS OR
00018  * IMPLIED, INCLUDING BUT NOT LIMITED TO THE WARRANTIES OF MERCHANTABILITY,
00019  * FITNESS FOR A PARTICULAR PURPOSE AND NONINFRINGEMENT. IN NO EVENT SHALL THE
00020  * AUTHORS OR COPYRIGHT HOLDERS BE LIABLE FOR ANY CLAIM, DAMAGES OR OTHER
00021  * LIABILITY, WHETHER IN AN ACTION OF CONTRACT, TORT OR OTHERWISE, ARISING FROM,
00022  * OUT OF OR IN CONNECTION WITH THE SOFTWARE OR THE USE OR OTHER DEALINGS IN THE
00023  * SOFTWARE.
00024  */
00025
00026 #ifndef __MATRIX_HPP__
00027 #define __MATRIX_HPP__
00028
00029 #include <ostream>
00030 #include <complex>
00031 #include <vector>
00032 #include <initializer_list>
00033 #include <limits>
00034 #include <functional>
00035 #include <algorithm>
00036
00037 namespace Mtx {
00038
00039 template<typename T> class Matrix;
00040
00041 template<class T> struct is_complex : std::false_type {};
00042 template<class T> struct is_complex<std::complex<T> > : std::true_type {};
00043
00044 template<typename T, typename std::enable_if<!is_complex<T>::value, int>::type = 0>
00045 inline T cconj(T x) {
00046     return x;
00047 }
00048
00049 template<typename T, typename std::enable_if<is_complex<T>::value, int>::type = 0>
00050 inline T cconj(T x) {
00051     return std::conj(x);
00052 }
00053
00054 template<typename T, typename std::enable_if<!is_complex<T>::value, int>::type = 0>
00055 inline T csign(T x) {
00056     return (x > static_cast<T>(0)) ? static_cast<T>(1) : static_cast<T>(-1);
00057 }
00058
00059 template<typename T, typename std::enable_if<is_complex<T>::value, int>::type = 0>
00060 inline T csign(T x) {
00061     auto x_arg = std::arg(x);
00062     T y(0, x_arg);
00063     return std::exp(y);
00064 }
00065
00066 class singular_matrix_exception : public std::domain_error {
00067 public:
00068     singular_matrix_exception(const std::string& message) : std::domain_error(message) {}
00069 };
00070
00071 template<typename T>
00072 struct LU_result {
00073     Matrix<T> L;
00074     Matrix<T> U;
00075 };
00076
00077 template<typename T>
00078 struct LUP_result {
00079     Matrix<T> L;
00080     Matrix<T> U;
00081     std::vector<unsigned> P;
00082 };
00083
00084 template<typename T>
00085 struct QR_result {
00086     Matrix<T> Q;

```

```

00134
00137     Matrix<T> R;
00138 };
00139
00144 template<typename T>
00145 struct Hessenberg_result {
00148     Matrix<T> H;
00149
00152     Matrix<T> Q;
00153 };
00154
00159 template<typename T>
00160 struct LDL_result {
00163     Matrix<T> L;
00164
00167     std::vector<T> d;
00168 };
00169
00174 template<typename T>
00175 struct Eigenvalues_result {
00178     std::vector<std::complex<T>> eig;
00179
00182     bool converged;
00183
00186     T err;
00187 };
00188
00189
00197 template<typename T>
00198 inline Matrix<T> zeros(unsigned nrows, unsigned ncols) {
00199     return Matrix<T>(static_cast<T>(0), nrows, ncols);
00200 }
00201
00208 template<typename T>
00209 inline Matrix<T> zeros(unsigned n) {
00210     return zeros<T>(n,n);
00211 }
00212
00221 template<typename T>
00222 inline Matrix<T> ones(unsigned nrows, unsigned ncols) {
00223     return Matrix<T>(static_cast<T>(1), nrows, ncols);
00224 }
00225
00233 template<typename T>
00234 inline Matrix<T> ones(unsigned n) {
00235     return ones<T>(n,n);
00236 }
00237
00245 template<typename T>
00246 Matrix<T> eye(unsigned n) {
00247     Matrix<T> A(static_cast<T>(0), n, n);
00248     for (unsigned i = 0; i < n; i++)
00249         A(i,i) = static_cast<T>(1);
00250     return A;
00251 }
00252
00260 template<typename T>
00261 Matrix<T> diag(const T* array, size_t n) {
00262     Matrix<T> A(static_cast<T>(0), n, n);
00263     for (unsigned i = 0; i < n; i++) {
00264         A(i,i) = array[i];
00265     }
00266     return A;
00267 }
00268
00276 template<typename T>
00277 inline Matrix<T> diag(const std::vector<T>& v) {
00278     return diag(v.data(), v.size());
00279 }
00280
00289 template<typename T>
00290 std::vector<T> diag(const Matrix<T>& A) {
00291     if (! A.issquare()) throw std::runtime_error("Input matrix is not square");
00292
00293     std::vector<T> v;
00294     v.resize(A.rows());
00295
00296     for (unsigned i = 0; i < A.rows(); i++)
00297         v[i] = A(i,i);
00298     return v;
00299 }
00300
00308 template<typename T>
00309 Matrix<T> circulant(const T* array, unsigned n) {
00310     Matrix<T> A(n, n);
00311     for (unsigned j = 0; j < n; j++)
00312         for (unsigned i = 0; i < n; i++)

```

```

00313     A((i+j) % n, j) = array[i];
00314     return A;
00315 }
00316
00327 template<typename T>
00328 Matrix<std::complex<T>> make_complex(const Matrix<T>& Re, const Matrix<T>& Im) {
00329     if (Re.rows() != Im.rows() || Re.cols() != Im.cols()) throw std::runtime_error("Size of input
matrices does not match");
00330
00331     Matrix<std::complex<T> > C(Re.rows(), Re.cols());
00332     for (unsigned n = 0; n < Re.numel(); n++) {
00333         C(n).real(Re(n));
00334         C(n).imag(Im(n));
00335     }
00336
00337     return C;
00338 }
00339
00346 template<typename T>
00347 Matrix<std::complex<T>> make_complex(const Matrix<T>& Re) {
00348     Matrix<std::complex<T>> C(Re.rows(), Re.cols());
00349
00350     for (unsigned n = 0; n < Re.numel(); n++) {
00351         C(n).real(Re(n));
00352         C(n).imag(static_cast<T>(0));
00353     }
00354
00355     return C;
00356 }
00357
00362 template<typename T>
00363 Matrix<T> real(const Matrix<std::complex<T>& C) {
00364     Matrix<T> Re(C.rows(), C.cols());
00365
00366     for (unsigned n = 0; n < C.numel(); n++)
00367         Re(n) = C(n).real();
00368
00369     return Re;
00370 }
00371
00376 template<typename T>
00377 Matrix<T> imag(const Matrix<std::complex<T>& C) {
00378     Matrix<T> Re(C.rows(), C.cols());
00379
00380     for (unsigned n = 0; n < C.numel(); n++)
00381         Re(n) = C(n).imag();
00382
00383     return Re;
00384 }
00385
00393 template<typename T>
00394 inline Matrix<T> circulant(const std::vector<T>& v) {
00395     return circulant(v.data(), v.size());
00396 }
00397
00402 template<typename T>
00403 inline Matrix<T> transpose(const Matrix<T>& A) {
00404     return A.transpose();
00405 }
00406
00412 template<typename T>
00413 inline Matrix<T> ctranspose(const Matrix<T>& A) {
00414     return A.ctranspose();
00415 }
00416
00427 template<typename T>
00428 Matrix<T> circshift(const Matrix<T>& A, int row_shift, int col_shift) {
00429     Matrix<T> B(A.rows(), A.cols());
00430     for (int i = 0; i < A.rows(); i++) {
00431         int ii = (i + row_shift) % A.rows();
00432         for (int j = 0; j < A.cols(); j++) {
00433             int jj = (j + col_shift) % A.cols();
00434             B(ii, jj) = A(i, j);
00435         }
00436     }
00437     return B;
00438 }
00439
00447 template<typename T>
00448 Matrix<T> repmat(const Matrix<T>& A, unsigned m, unsigned n) {
00449     Matrix<T> B(m * A.rows(), n * A.cols());
00450
00451     for (unsigned cb = 0; cb < n; cb++)
00452         for (unsigned rb = 0; rb < m; rb++)
00453             for (unsigned c = 0; c < A.cols(); c++)
00454                 for (unsigned r = 0; r < A.rows(); r++)
00455                     B(rb*A.rows() + r, cb*A.cols() + c) = A(r, c);

```

```

00456
00457     return B;
00458 }
00459
00460 template<typename T>
00461 double norm_fro(const Matrix<T>& A) {
00462     double sum = 0;
00463
00464     for (unsigned i = 0; i < A.numel(); i++)
00465         sum += A(i) * A(i);
00466
00467     return std::sqrt(sum);
00468 }
00469
00470 template<typename T>
00471 double norm_fro(const Matrix<std::complex<T> >& A) {
00472     double sum = 0;
00473
00474     for (unsigned i = 0; i < A.numel(); i++) {
00475         T x = std::abs(A(i));
00476         sum += x * x;
00477     }
00478
00479     return std::sqrt(sum);
00480 }
00481
00482 template<typename T>
00483 Matrix<T> tril(const Matrix<T>& A) {
00484     Matrix<T> B(A);
00485
00486     for (unsigned row = 0; row < B.rows(); row++)
00487         for (unsigned col = row+1; col < B.cols(); col++)
00488             B(row,col) = 0;
00489
00490     return B;
00491 }
00492
00493 template<typename T>
00494 Matrix<T> triu(const Matrix<T>& A) {
00495     Matrix<T> B(A);
00496
00497     for (unsigned col = 0; col < B.cols(); col++)
00498         for (unsigned row = col+1; row < B.rows(); row++)
00499             B(row,col) = 0;
00500
00501     return B;
00502 }
00503
00504 template<typename T>
00505 bool istril(const Matrix<T>& A) {
00506     for (unsigned row = 0; row < A.rows(); row++)
00507         for (unsigned col = row+1; col < A.cols(); col++)
00508             if (A(row,col) != static_cast<T>(0)) return false;
00509     return true;
00510 }
00511
00512 template<typename T>
00513 bool istriu(const Matrix<T>& A) {
00514     for (unsigned col = 0; col < A.cols(); col++)
00515         for (unsigned row = col+1; row < A.rows(); row++)
00516             if (A(row,col) != static_cast<T>(0)) return false;
00517     return true;
00518 }
00519
00520 template<typename T>
00521 bool ishess(const Matrix<T>& A) {
00522     if (!A.issquare())
00523         return false;
00524     for (unsigned row = 2; row < A.rows(); row++)
00525         for (unsigned col = 0; col < row-2; col++)
00526             if (A(row,col) != static_cast<T>(0)) return false;
00527     return true;
00528 }
00529
00530 template<typename T>
00531 inline void foreach_elem(Matrix<T>& A, std::function<T(T)> func) {
00532     for (unsigned i = 0; i < A.numel(); i++)
00533         A(i) = func(A(i));
00534 }
00535
00536 template<typename T>
00537 inline Matrix<T> foreach_elem_copy(const Matrix<T>& A, std::function<T(T)> func) {
00538     Matrix<T> B(A);
00539     foreach_elem(B, func);
00540     return B;
00541 }
00542
00543
00544
00545
00546
00547
00548
00549
00550
00551
00552
00553
00554
00555
00556
00557
00558
00559
00560
00561
00562
00563
00564
00565
00566
00567
00568
00569
00570
00571
00572
00573
00574
00575
00576
00577
00578
00579
00580
00581
00582
00583
00584
00585
00586
00587
00588
00589
00590
00591

```

```

00604 template<typename T>
00605 Matrix<T> permute_rows(const Matrix<T>& A, const std::vector<unsigned> perm) {
00606     if (perm.empty()) throw std::runtime_error("Permutation vector is empty");
00607
00608     Matrix<T> B(perm.size(), A.cols());
00609
00610     for (unsigned p = 0; p < perm.size(); p++) {
00611         if (!perm[p] < A.rows()) throw std::out_of_range("Index in permutation vector out of range");
00612
00613         for (unsigned c = 0; c < A.cols(); c++)
00614             B(p,c) = A(perm[p],c);
00615     }
00616     return B;
00617 }
00618
00619 template<typename T>
00620 Matrix<T> permute_cols(const Matrix<T>& A, const std::vector<unsigned> perm) {
00621     if (perm.empty()) throw std::runtime_error("Permutation vector is empty");
00622
00623     Matrix<T> B(A.rows(), perm.size());
00624
00625     for (unsigned p = 0; p < perm.size(); p++) {
00626         if (!perm[p] < A.cols()) throw std::out_of_range("Index in permutation vector out of range");
00627
00628         for (unsigned r = 0; r < A.rows(); r++)
00629             B(r,p) = A(r,perm[p]);
00630     }
00631     return B;
00632 }
00633
00634 template<typename T, bool transpose_first = false, bool transpose_second = false>
00635 Matrix<T> mult(const Matrix<T>& A, const Matrix<T>& B) {
00636     // Adjust dimensions based on transpositions
00637     unsigned rows_A = transpose_first ? A.cols() : A.rows();
00638     unsigned cols_A = transpose_first ? A.rows() : A.cols();
00639     unsigned rows_B = transpose_second ? B.cols() : B.rows();
00640     unsigned cols_B = transpose_second ? B.rows() : B.cols();
00641
00642     if (cols_A != rows_B) throw std::runtime_error("Unmatching matrix dimensions for mult");
00643
00644     Matrix<T> C(static_cast<T>(0), rows_A, cols_B);
00645
00646     for (unsigned i = 0; i < rows_A; i++)
00647         for (unsigned j = 0; j < cols_B; j++)
00648             for (unsigned k = 0; k < cols_A; k++)
00649                 C(i,j) += (transpose_first ? cconj(A(k,i)) : A(i,k)) *
00650                     (transpose_second ? cconj(B(j,k)) : B(k,j));
00651
00652     return C;
00653 }
00654
00655 template<typename T, bool transpose_first = false, bool transpose_second = false>
00656 Matrix<T> mult_hadamard(const Matrix<T>& A, const Matrix<T>& B) {
00657     // Adjust dimensions based on transpositions
00658     unsigned rows_A = transpose_first ? A.cols() : A.rows();
00659     unsigned cols_A = transpose_first ? A.rows() : A.cols();
00660     unsigned rows_B = transpose_second ? B.cols() : B.rows();
00661     unsigned cols_B = transpose_second ? B.rows() : B.cols();
00662
00663     if ((rows_A != rows_B) || (cols_A != cols_B)) throw std::runtime_error("Unmatching matrix dimensions for mult_hadamard");
00664
00665     Matrix<T> C(static_cast<T>(0), rows_A, cols_A);
00666
00667     for (unsigned i = 0; i < rows_A; i++)
00668         for (unsigned j = 0; j < cols_A; j++)
00669             C(i,j) += (transpose_first ? cconj(A(j,i)) : A(i,j)) *
00670                 (transpose_second ? cconj(B(j,i)) : B(i,j));
00671
00672     return C;
00673 }
00674
00675 template<typename T, bool transpose_first = false, bool transpose_second = false>
00676 Matrix<T> add(const Matrix<T>& A, const Matrix<T>& B) {
00677     // Adjust dimensions based on transpositions
00678     unsigned rows_A = transpose_first ? A.cols() : A.rows();
00679     unsigned cols_A = transpose_first ? A.rows() : A.cols();
00680     unsigned rows_B = transpose_second ? B.cols() : B.rows();
00681     unsigned cols_B = transpose_second ? B.rows() : B.cols();
00682
00683     if ((rows_A != rows_B) || (cols_A != cols_B)) throw std::runtime_error("Unmatching matrix dimensions for add");
00684
00685     Matrix<T> C(static_cast<T>(0), rows_A, cols_A);
00686
00687     for (unsigned i = 0; i < rows_A; i++)
00688         for (unsigned j = 0; j < cols_A; j++)
00689             C(i,j) = A(i,j) + B(i,j);
00690
00691     return C;
00692 }

```

```

00743     for (unsigned i = 0; i < rows_A; i++)
00744         for (unsigned j = 0; j < cols_A; j++)
00745             C(i,j) += (transpose_first ? cconj(A(j,i)) : A(i,j)) +
00746                       (transpose_second ? cconj(B(j,i)) : B(i,j));
00747
00748     return C;
00749 }
00750
00765 template<typename T, bool transpose_first = false, bool transpose_second = false>
00766 Matrix<T> subtract(const Matrix<T>& A, const Matrix<T>& B) {
00767     // Adjust dimensions based on transpositions
00768     unsigned rows_A = transpose_first ? A.cols() : A.rows();
00769     unsigned cols_A = transpose_first ? A.rows() : A.cols();
00770     unsigned rows_B = transpose_second ? B.cols() : B.rows();
00771     unsigned cols_B = transpose_second ? B.rows() : B.cols();
00772
00773     if ((rows_A != rows_B) || (cols_A != cols_B)) throw std::runtime_error("Unmatching matrix dimensions
for subtract");
00774
00775     Matrix<T> C(static_cast<T>(0), rows_A, cols_A);
00776
00777     for (unsigned i = 0; i < rows_A; i++)
00778         for (unsigned j = 0; j < cols_A; j++)
00779             C(i,j) += (transpose_first ? cconj(A(j,i)) : A(i,j)) -
00780                       (transpose_second ? cconj(B(j,i)) : B(i,j));
00781
00782     return C;
00783 }
00784
00793 template<typename T>
00794 std::vector<T> mult(const Matrix<T>& A, const std::vector<T>& v) {
00795     if (A.cols() != v.size()) throw std::runtime_error("Unmatching matrix dimensions for mult");
00796
00797     std::vector<T> u(A.rows(), static_cast<T>(0));
00798     for (unsigned r = 0; r < A.rows(); r++)
00799         for (unsigned c = 0; c < A.cols(); c++)
00800             u[r] += v[c] * A(r,c);
00801     return u;
00802 }
00803
00812 template<typename T>
00813 std::vector<T> mult(const std::vector<T>& v, const Matrix<T>& A) {
00814     if (A.rows() != v.size()) throw std::runtime_error("Unmatching matrix dimensions for mult");
00815
00816     std::vector<T> u(A.rows(), static_cast<T>(0));
00817     for (unsigned c = 0; c < A.cols(); c++)
00818         for (unsigned r = 0; r < A.rows(); r++)
00819             u[c] += v[r] * A(r,c);
00820     return u;
00821 }
00822
00828 template<typename T>
00829 Matrix<T> add(const Matrix<T>& A, T s) {
00830     Matrix<T> B(A.rows(), A.cols());
00831     for (unsigned i = 0; i < A.numel(); i++)
00832         B(i) = A(i) + s;
00833     return B;
00834 }
00835
00841 template<typename T>
00842 Matrix<T> subtract(const Matrix<T>& A, T s) {
00843     Matrix<T> B(A.rows(), A.cols());
00844     for (unsigned i = 0; i < A.numel(); i++)
00845         B(i) = A(i) - s;
00846     return B;
00847 }
00848
00854 template<typename T>
00855 Matrix<T> mult(const Matrix<T>& A, T s) {
00856     Matrix<T> B(A.rows(), A.cols());
00857     for (unsigned i = 0; i < A.numel(); i++)
00858         B(i) = A(i) * s;
00859     return B;
00860 }
00861
00867 template<typename T>
00868 Matrix<T> div(const Matrix<T>& A, T s) {
00869     Matrix<T> B(A.rows(), A.cols());
00870     for (unsigned i = 0; i < A.numel(); i++)
00871         B(i) = A(i) / s;
00872     return B;
00873 }
00874
00880 template<typename T>
00881 std::ostream& operator<<(std::ostream& os, const Matrix<T>& A) {
00882     for (unsigned row = 0; row < A.rows(); row++) {
00883         for (unsigned col = 0; col < A.cols(); col++)

```



```

00884         os << A(row,col) << " ";
00885         if (row < static_cast<unsigned>(A.rows()-1)) os << std::endl;
00886     }
00887     return os;
00888 }
00889
00894 template<typename T>
00895 inline Matrix<T> operator+(const Matrix<T>& A, const Matrix<T>& B) {
00896     return add(A,B);
00897 }
00898
00903 template<typename T>
00904 inline Matrix<T> operator-(const Matrix<T>& A, const Matrix<T>& B) {
00905     return subtract(A,B);
00906 }
00907
00913 template<typename T>
00914 inline Matrix<T> operator^(const Matrix<T>& A, const Matrix<T>& B) {
00915     return mult_hadamard(A,B);
00916 }
00917
00922 template<typename T>
00923 inline Matrix<T> operator*(const Matrix<T>& A, const Matrix<T>& B) {
00924     return mult(A,B);
00925 }
00926
00931 template<typename T>
00932 inline std::vector<T> operator*(const Matrix<T>& A, const std::vector<T>& v) {
00933     return mult(A,v);
00934 }
00935
00940 template<typename T>
00941 inline Matrix<T> operator+(const Matrix<T>& A, T s) {
00942     return add(A,s);
00943 }
00944
00949 template<typename T>
00950 inline Matrix<T> operator-(const Matrix<T>& A, T s) {
00951     return subtract(A,s);
00952 }
00953
00958 template<typename T>
00959 inline Matrix<T> operator*(const Matrix<T>& A, T s) {
00960     return mult(A,s);
00961 }
00962
00967 template<typename T>
00968 inline Matrix<T> operator/(const Matrix<T>& A, T s) {
00969     return div(A,s);
00970 }
00971
00975 template<typename T>
00976 inline Matrix<T> operator+(T s, const Matrix<T>& A) {
00977     return add(A,s);
00978 }
00979
00984 template<typename T>
00985 inline Matrix<T> operator*(T s, const Matrix<T>& A) {
00986     return mult(A,s);
00987 }
00988
00993 template<typename T>
00994 inline Matrix<T>& operator+=(Matrix<T>& A, const Matrix<T>& B) {
00995     return A.add(B);
00996 }
00997
01002 template<typename T>
01003 inline Matrix<T>& operator-=(Matrix<T>& A, const Matrix<T>& B) {
01004     return A.subtract(B);
01005 }
01006
01011 template<typename T>
01012 inline Matrix<T>& operator*=(Matrix<T>& A, const Matrix<T>& B) {
01013     A = mult(A,B);
01014     return A;
01015 }
01016
01022 template<typename T>
01023 inline Matrix<T>& operator^=(Matrix<T>& A, const Matrix<T>& B) {
01024     return A.mult_hadamard(B);
01025 }
01026
01031 template<typename T>
01032 inline Matrix<T>& operator+=(Matrix<T>& A, T s) {
01033     return A.add(s);
01034 }
01035

```

```

01040 template<typename T>
01041 inline Matrix<T>& operator--(Matrix<T>& A, T s) {
01042     return A.subtract(s);
01043 }
01044
01049 template<typename T>
01050 inline Matrix<T>& operator*=(Matrix<T>& A, T s) {
01051     return A.mult(s);
01052 }
01053
01058 template<typename T>
01059 inline Matrix<T>& operator/=(Matrix<T>& A, T s) {
01060     return A.div(s);
01061 }
01062
01067 template<typename T>
01068 inline bool operator==(const Matrix<T>& A, const Matrix<T>& b) {
01069     return A.isequal(b);
01070 }
01071
01076 template<typename T>
01077 inline bool operator!=(const Matrix<T>& A, const Matrix<T>& b) {
01078     return !(A.isequal(b));
01079 }
01080
01086 template<typename T>
01087 Matrix<T> kron(const Matrix<T>& A, const Matrix<T>& B) {
01088     const unsigned rows_A = A.rows();
01089     const unsigned cols_A = A.cols();
01090     const unsigned rows_B = B.rows();
01091     const unsigned cols_B = B.cols();
01092
01093     unsigned rows_C = rows_A * rows_B;
01094     unsigned cols_C = cols_A * cols_B;
01095
01096     Matrix<T> C(rows_C, cols_C);
01097
01098     for (unsigned i = 0; i < rows_A; i++)
01099         for (unsigned j = 0; j < cols_A; j++)
01100             for (unsigned k = 0; k < rows_B; k++)
01101                 for (unsigned l = 0; l < cols_B; l++)
01102                     C(i+rows_B * k, j+cols_B * l) = A(i, j) * B(k, l);
01103
01104     return C;
01105 }
01106
01114 template<typename T>
01115 Matrix<T> adj(const Matrix<T>& A) {
01116     if (!A.issquare()) throw std::runtime_error("Input matrix is not square");
01117
01118     Matrix<T> B(A.rows(), A.cols());
01119     if (A.rows() == 1) {
01120         B(0) = 1.0;
01121     } else {
01122         for (unsigned i = 0; i < A.rows(); i++) {
01123             for (unsigned j = 0; j < A.cols(); j++) {
01124                 T sgn = ((i + j) % 2 == 0) ? 1.0 : -1.0;
01125                 B(j, i) = sgn * det(cofactor(A, i, j));
01126             }
01127         }
01128     }
01129     return B;
01130 }
01131
01144 template<typename T>
01145 Matrix<T> cofactor(const Matrix<T>& A, unsigned p, unsigned q) {
01146     if (!A.issquare()) throw std::runtime_error("Input matrix is not square");
01147     if (!(p < A.rows())) throw std::out_of_range("Row index out of range");
01148     if (!(q < A.cols())) throw std::out_of_range("Column index out of range");
01149     if (A.cols() < 2) throw std::runtime_error("Cofactor calculation requested for matrix with less than
2 rows");
01150
01151     Matrix<T> c(A.rows()-1, A.cols()-1);
01152     unsigned i = 0;
01153     unsigned j = 0;
01154
01155     for (unsigned row = 0; row < A.rows(); row++) {
01156         if (row != p) {
01157             for (unsigned col = 0; col < A.cols(); col++)
01158                 if (col != q) c(i, j++) = A(row, col);
01159             j = 0;
01160             i++;
01161         }
01162     }
01163
01164     return c;
01165 }

```

```

01166
01178 template<typename T>
01179 T det_lu(const Matrix<T>& A) {
01180     if (! A.issquare()) throw std::runtime_error("Input matrix is not square");
01181
01182     // LU decomposition with pivoting
01183     auto res = lup(A);
01184
01185     // Determinants of LU
01186     T detLU = static_cast<T>(1);
01187
01188     for (unsigned i = 0; i < res.L.rows(); i++)
01189         detLU *= res.L(i,i) * res.U(i,i);
01190
01191     // Determinant of P
01192     unsigned len = res.P.size();
01193     T detP = 1;
01194
01195     std::vector<unsigned> p(res.P);
01196     std::vector<unsigned> q;
01197     q.resize(len);
01198
01199     for (unsigned i = 0; i < len; i++)
01200         q[p[i]] = i;
01201
01202     for (unsigned i = 0; i < len; i++) {
01203         unsigned j = p[i];
01204         unsigned k = q[i];
01205         if (j != i) {
01206             p[k] = p[i];
01207             q[j] = q[i];
01208             detP = - detP;
01209         }
01210     }
01211
01212     return detLU * detP;
01213 }
01214
01223 template<typename T>
01224 T det(const Matrix<T>& A) {
01225     if (! A.issquare()) throw std::runtime_error("Input matrix is not square");
01226
01227     if (A.rows() == 1)
01228         return A(0,0);
01229     else if (A.rows() == 2)
01230         return A(0,0)*A(1,1) - A(0,1)*A(1,0);
01231     else if (A.rows() == 3)
01232         return A(0,0)*(A(1,1)*A(2,2) - A(1,2)*A(2,1)) -
01233             A(0,1)*(A(1,0)*A(2,2) - A(1,2)*A(2,0)) +
01234             A(0,2)*(A(1,0)*A(2,1) - A(1,1)*A(2,0));
01235     else
01236         return det_lu(A);
01237 }
01238
01247 template<typename T>
01248 LU_result<T> lu(const Matrix<T>& A) {
01249     const unsigned M = A.rows();
01250     const unsigned N = A.cols();
01251
01252     LU_result<T> res;
01253     res.L = eye<T>(M);
01254     res.U = Matrix<T>(A);
01255
01256     // aliases
01257     auto& L = res.L;
01258     auto& U = res.U;
01259
01260     if (A.numel() == 0)
01261         return res;
01262
01263     for (unsigned k = 0; k < M-1; k++) {
01264         for (unsigned i = k+1; i < M; i++) {
01265             L(i,k) = U(i,k) / U(k,k);
01266             for (unsigned l = k+1; l < N; l++) {
01267                 U(i,l) -= L(i,k) * U(k,l);
01268             }
01269         }
01270     }
01271
01272     for (unsigned col = 0; col < N; col++)
01273         for (unsigned row = col+1; row < M; row++)
01274             U(row,col) = 0;
01275
01276     return res;
01277 }
01278
01292 template<typename T>

```

```

01293 LUP_result<T> lup(const Matrix<T>& A) {
01294     const unsigned M = A.rows();
01295     const unsigned N = A.cols();
01296
01297     // Initialize L, U, and PP
01298     LUP_result<T> res;
01299
01300     if (A.numel() == 0)
01301         return res;
01302
01303     res.L = eye<T>(M);
01304     res.U = Matrix<T>(A);
01305     std::vector<unsigned> PP;
01306
01307     // aliases
01308     auto& L = res.L;
01309     auto& U = res.U;
01310
01311     PP.resize(N);
01312     for (unsigned i = 0; i < N; i++)
01313         PP[i] = i;
01314
01315     for (unsigned k = 0; k < M-1; k++) {
01316         // Find the column with the largest absolute value in the current row
01317         auto max_col_value = std::abs(U(k,k));
01318         unsigned max_col_index = k;
01319         for (unsigned l = k+1; l < N; l++) {
01320             auto val = std::abs(U(k,l));
01321             if (val > max_col_value) {
01322                 max_col_value = val;
01323                 max_col_index = l;
01324             }
01325         }
01326
01327         // Swap columns k and max_col_index in U and update P
01328         if (max_col_index != k) {
01329             U.swap_cols(k, max_col_index); // TODO: This could be reworked to avoid column swap in U during
every iteration by:
01330                                     //      1. using PP[k] for column indexing across iterations
01331                                     //      2. doing just one permutation of U at the end
01332             std::swap(PP[k], PP[max_col_index]);
01333         }
01334
01335         // Update L and U
01336         for (unsigned i = k+1; i < M; i++) {
01337             L(i,k) = U(i,k) / U(k,k);
01338             for (unsigned l = k+1; l < N; l++) {
01339                 U(i,l) -= L(i,k) * U(k,l);
01340             }
01341         }
01342     }
01343
01344     // Set elements in lower triangular part of U to zero
01345     for (unsigned col = 0; col < N; col++)
01346         for (unsigned row = col+1; row < M; row++)
01347             U(row,col) = 0;
01348
01349     // Transpose indices in permutation vector
01350     res.P.resize(N);
01351     for (unsigned i = 0; i < N; i++)
01352         res.P[PP[i]] = i;
01353
01354     return res;
01355 }
01356
01367 template<typename T>
01368 Matrix<T> inv_gauss_jordan(const Matrix<T>& A) {
01369     if (! A.issquare()) throw std::runtime_error("Input matrix is not square");
01370
01371     const unsigned N = A.rows();
01372     Matrix<T> AA(A);
01373     auto IA = eye<T>(N);
01374
01375     bool found_nonzero;
01376     for (unsigned j = 0; j < N; j++) {
01377         found_nonzero = false;
01378         for (unsigned i = j; i < N; i++) {
01379             if (AA(i,j) != static_cast<T>(0)) {
01380                 found_nonzero = true;
01381                 for (unsigned k = 0; k < N; k++) {
01382                     std::swap(AA(j,k), AA(i,k));
01383                     std::swap(IA(j,k), IA(i,k));
01384                 }
01385                 if (AA(j,j) != static_cast<T>(1)) {
01386                     T s = static_cast<T>(1) / AA(j,j);
01387                     for (unsigned k = 0; k < N; k++) {
01388                         AA(j,k) *= s;

```

```

01389         IA(j,k) *= s;
01390     }
01391 }
01392 for (unsigned l = 0; l < N; l++) {
01393     if (l != j) {
01394         T s = AA(l,j);
01395         for (unsigned k = 0; k < N; k++) {
01396             AA(l,k) -= s * AA(j,k);
01397             IA(l,k) -= s * IA(j,k);
01398         }
01399     }
01400 }
01401 }
01402 break;
01403 }
01404 // if a row full of zeros is found, the input matrix was singular
01405 if (!found_nonzero) throw singular_matrix_exception("Singular matrix in inv_gauss_jordan");
01406 }
01407 return IA;
01408 }
01409
01420 template<typename T>
01421 Matrix<T> inv_tril(const Matrix<T>& A) {
01422     if (!A.issquare()) throw std::runtime_error("Input matrix is not square");
01423
01424     const unsigned N = A.rows();
01425
01426     auto IA = zeros<T>(N);
01427
01428     for (unsigned i = 0; i < N; i++) {
01429         if (A(i,i) == 0.0) throw singular_matrix_exception("Division by zero in inv_tril");
01430
01431         IA(i,i) = static_cast<T>(1.0) / A(i,i);
01432         for (unsigned j = 0; j < i; j++) {
01433             T s = 0.0;
01434             for (unsigned k = j; k < i; k++)
01435                 s += A(i,k) * IA(k,j);
01436             IA(i,j) = -s * IA(i,i);
01437         }
01438     }
01439
01440     return IA;
01441 }
01442
01453 template<typename T>
01454 Matrix<T> inv_triu(const Matrix<T>& A) {
01455     if (!A.issquare()) throw std::runtime_error("Input matrix is not square");
01456
01457     const unsigned N = A.rows();
01458
01459     auto IA = zeros<T>(N);
01460
01461     for (int i = N - 1; i >= 0; i--) {
01462         if (A(i,i) == 0.0) throw singular_matrix_exception("Division by zero in inv_triu");
01463
01464         IA(i,i) = static_cast<T>(1.0) / A(i,i);
01465         for (int j = N - 1; j > i; j--) {
01466             T s = 0.0;
01467             for (int k = i + 1; k <= j; k++)
01468                 s += A(i,k) * IA(k,j);
01469             IA(i,j) = -s * IA(i,i);
01470         }
01471     }
01472
01473     return IA;
01474 }
01475
01488 template<typename T>
01489 Matrix<T> inv_posdef(const Matrix<T>& A) {
01490     auto L = cholinv(A);
01491     return mult<T,true,false>(L,L);
01492 }
01493
01504 template<typename T>
01505 Matrix<T> inv_square(const Matrix<T>& A) {
01506     if (!A.issquare()) throw std::runtime_error("Input matrix is not square");
01507
01508     // LU decomposition with pivoting
01509     auto LU = lup(A);
01510     auto IL = inv_tril(LU.L);
01511     auto IU = inv_triu(LU.U);
01512
01513     return permute_rows(IU * IL, LU.P);
01514 }
01515
01526 template<typename T>
01527 Matrix<T> inv(const Matrix<T>& A) {

```

```

01528     if (! A.issquare()) throw std::runtime_error("Input matrix is not square");
01529
01530     if (A.numel() == 0) {
01531         return Matrix<T>();
01532     } else if (A.rows() < 4) {
01533         T d = det(A);
01534
01535         if (d == 0.0) throw singular_matrix_exception("Singular matrix in inv");
01536
01537         Matrix<T> IA(A.rows(), A.rows());
01538         T invdet = static_cast<T>(1.0) / d;
01539
01540         if (A.rows() == 1) {
01541             IA(0,0) = invdet;
01542         } else if (A.rows() == 2) {
01543             IA(0,0) = A(1,1) * invdet;
01544             IA(0,1) = - A(0,1) * invdet;
01545             IA(1,0) = - A(1,0) * invdet;
01546             IA(1,1) = A(0,0) * invdet;
01547         } else if (A.rows() == 3) {
01548             IA(0,0) = (A(1,1)*A(2,2) - A(2,1)*A(1,2)) * invdet;
01549             IA(0,1) = (A(0,2)*A(2,1) - A(0,1)*A(2,2)) * invdet;
01550             IA(0,2) = (A(0,1)*A(1,2) - A(0,2)*A(1,1)) * invdet;
01551             IA(1,0) = (A(1,2)*A(2,0) - A(1,0)*A(2,2)) * invdet;
01552             IA(1,1) = (A(0,0)*A(2,2) - A(0,2)*A(2,0)) * invdet;
01553             IA(1,2) = (A(1,0)*A(0,2) - A(0,0)*A(1,2)) * invdet;
01554             IA(2,0) = (A(1,0)*A(2,1) - A(2,0)*A(1,1)) * invdet;
01555             IA(2,1) = (A(2,0)*A(0,1) - A(0,0)*A(2,1)) * invdet;
01556             IA(2,2) = (A(0,0)*A(1,1) - A(1,0)*A(0,1)) * invdet;
01557         }
01558
01559         return IA;
01560     } else {
01561         return inv_square(A);
01562     }
01563 }
01564
01572 template<typename T>
01573 Matrix<T> pinv(const Matrix<T>& A) {
01574     if (A.rows() > A.cols()) {
01575         auto AH_A = mult<T,true,false>(A, A);
01576         auto Linv = inv_posdef(AH_A);
01577         return mult<T,false,true>(Linv, A);
01578     } else {
01579         auto AA_H = mult<T,false,true>(A, A);
01580         auto Linv = inv_posdef(AA_H);
01581         return mult<T,true,false>(A, Linv);
01582     }
01583 }
01584
01590 template<typename T>
01591 T trace(const Matrix<T>& A) {
01592     T t = static_cast<T>(0);
01593     for (int i = 0; i < A.rows(); i++)
01594         t += A(i,i);
01595     return t;
01596 }
01597
01605 template<typename T>
01606 double cond(const Matrix<T>& A) {
01607     try {
01608         auto A_inv = inv(A);
01609         return norm_fro(A) * norm_fro(A_inv);
01610     } catch (singular_matrix_exception& e) {
01611         return std::numeric_limits<double>::max();
01612     }
01613 }
01614
01626 template<typename T>
01627 Matrix<T> chol(const Matrix<T>& A) {
01628     if (! A.issquare()) throw std::runtime_error("Input matrix is not square");
01629
01630     const unsigned N = A.rows();
01631     Matrix<T> L = tril(A);
01632
01633     for (unsigned j = 0; j < N; j++) {
01634         if (L(j,j) == 0.0) throw singular_matrix_exception("Singular matrix in chol");
01635
01636         L(j,j) = std::sqrt(L(j,j));
01637
01638         for (unsigned k = j+1; k < N; k++)
01639             L(k,j) = L(k,j) / L(j,j);
01640
01641         for (unsigned k = j+1; k < N; k++)
01642             for (unsigned i = k; i < N; i++)
01643                 L(i,k) = L(i,k) - L(i,j) * cconj(L(k,j));
01644     }

```

```

01645
01646     return L;
01647 }
01648
01649 template<typename T>
01650 Matrix<T> cholinv(const Matrix<T>& A) {
01651     if (! A.issquare()) throw std::runtime_error("Input matrix is not square");
01652
01653     const unsigned N = A.rows();
01654     Matrix<T> L(A);
01655     auto Linv = eye<T>(N);
01656
01657     for (unsigned j = 0; j < N; j++) {
01658         if (L(j,j) == 0.0) throw singular_matrix_exception("Singular matrix in cholinv");
01659
01660         L(j,j) = 1.0 / std::sqrt(L(j,j));
01661
01662         for (unsigned k = j+1; k < N; k++)
01663             L(k,j) = L(k,j) * L(j,j);
01664
01665         for (unsigned k = j+1; k < N; k++)
01666             for (unsigned i = k; i < N; i++)
01667                 L(i,k) = L(i,k) - L(i,j) * cconj(L(k,j));
01668     }
01669
01670     for (unsigned k = 0; k < N; k++) {
01671         for (unsigned i = k; i < N; i++) {
01672             Linv(i,k) = Linv(i,k) * L(i,i);
01673             for (unsigned j = i+1; j < N; j++)
01674                 Linv(j,k) = Linv(j,k) - L(j,i) * Linv(i,k);
01675         }
01676     }
01677
01678     return Linv;
01679 }
01680
01681 template<typename T>
01682 LDL_result<T> ldl(const Matrix<T>& A) {
01683     if (! A.issquare()) throw std::runtime_error("Input matrix is not square");
01684
01685     const unsigned N = A.rows();
01686
01687     LDL_result<T> res;
01688
01689     // aliases
01690     auto& L = res.L;
01691     auto& d = res.d;
01692
01693     L = eye<T>(N);
01694     d.resize(N);
01695
01696     for (unsigned m = 0; m < N; m++) {
01697         d[m] = A(m,m);
01698
01699         for (unsigned k = 0; k < m; k++)
01700             d[m] -= L(m,k) * cconj(L(m,k)) * d[k];
01701
01702         if (d[m] == 0.0) throw singular_matrix_exception("Singular matrix in ldl");
01703
01704         for (unsigned n = m+1; n < N; n++) {
01705             L(n,m) = A(n,m);
01706             for (unsigned k = 0; k < m; k++)
01707                 L(n,m) -= L(n,k) * cconj(L(m,k)) * d[k];
01708             L(n,m) /= d[m];
01709         }
01710     }
01711
01712     return res;
01713 }
01714
01715 template<typename T>
01716 QR_result<T> qr_red_gs(const Matrix<T>& A) {
01717     const int rows = A.rows();
01718     const int cols = A.cols();
01719
01720     QR_result<T> res;
01721
01722     //aliases
01723     auto& Q = res.Q;
01724     auto& R = res.R;
01725
01726     Q = zeros<T>(rows, cols);
01727     R = zeros<T>(cols, cols);
01728
01729     for (int c = 0; c < cols; c++) {
01730         Matrix<T> v = A.get_submatrix(0, rows-1, c, c);
01731         for (int r = 0; r < c; r++) {

```

```

01767         for (int k = 0; k < rows; k++)
01768             R(r,c) = R(r,c) + cconj(Q(k,r)) * A(k,c);
01769         for (int k = 0; k < rows; k++)
01770             v(k) = v(k) - R(r,c) * Q(k,r);
01771     }
01772
01773     R(c,c) = static_cast<T>(norm_fro(v));
01774
01775     if (R(c,c) == 0.0) throw singular_matrix_exception("Division by 0 in QR GS");
01776
01777     for (int k = 0; k < rows; k++)
01778         Q(k,c) = v(k) / R(c,c);
01779 }
01780
01781 return res;
01782 }
01783
01791 template<typename T>
01792 Matrix<T> householder_reflection(const Matrix<T>& a) {
01793     if (a.cols() != 1) throw std::runtime_error("Input not a column vector");
01794
01795     static const T ISQRT2 = static_cast<T>(0.707106781186547);
01796
01797     Matrix<T> v(a);
01798     v(0) += csign(v(0)) * norm_fro(v);
01799     auto vn = norm_fro(v) * ISQRT2;
01800     for (unsigned i = 0; i < v.numel(); i++)
01801         v(i) /= vn;
01802     return v;
01803 }
01804
01816 template<typename T>
01817 QR_result<T> qr_householder(const Matrix<T>& A, bool calculate_Q = true) {
01818     const unsigned rows = A.rows();
01819     const unsigned cols = A.cols();
01820
01821     QR_result<T> res;
01822
01823     //aliases
01824     auto& Q = res.Q;
01825     auto& R = res.R;
01826
01827     R = Matrix<T>(A);
01828
01829     if (calculate_Q)
01830         Q = eye<T>(rows);
01831
01832     const unsigned N = (rows > cols) ? cols : rows;
01833
01834     for (unsigned j = 0; j < N; j++) {
01835         auto v = householder_reflection(R.get_submatrix(j, rows-1, j, j));
01836
01837         auto R1 = R.get_submatrix(j, rows-1, j, cols-1);
01838         auto WR = v * mult<T,true,false>(v, R1);
01839         for (unsigned c = j; c < cols; c++)
01840             for (unsigned r = j; r < rows; r++)
01841                 R(r,c) -= WR(r-j,c-j);
01842
01843         if (calculate_Q) {
01844             auto Q1 = Q.get_submatrix(0, rows-1, j, rows-1);
01845             auto WQ = mult<T,false,true>(Q1 * v, v);
01846             for (unsigned c = j; c < rows; c++)
01847                 for (unsigned r = 0; r < rows; r++)
01848                     Q(r,c) -= WQ(r,c-j);
01849         }
01850     }
01851
01852     for (unsigned col = 0; col < R.cols(); col++)
01853         for (unsigned row = col+1; row < R.rows(); row++)
01854             R(row,col) = 0;
01855
01856     return res;
01857 }
01858
01869 template<typename T>
01870 inline QR_result<T> qr(const Matrix<T>& A, bool calculate_Q = true) {
01871     return qr_householder(A, calculate_Q);
01872 }
01873
01884 template<typename T>
01885 Hessenberg_result<T> hessenberg(const Matrix<T>& A, bool calculate_Q = true) {
01886     if (! A.issquare()) throw std::runtime_error("Input matrix is not square");
01887
01888     Hessenberg_result<T> res;
01889
01890     // aliases
01891     auto& H = res.H;

```



```

01892     auto& Q = res.Q;
01893
01894     const unsigned N = A.rows();
01895     H = Matrix<T>(A);
01896
01897     if (calculate_Q)
01898         Q = eye<T>(N);
01899
01900     for (unsigned k = 1; k < N-1; k++) {
01901         auto v = householder_reflection(H.get_submatrix(k, N-1, k-1, k-1));
01902
01903         auto H1 = H.get_submatrix(k, N-1, 0, N-1);
01904         auto W1 = v * mult<T,true,false>(v, H1);
01905         for (unsigned c = 0; c < N; c++)
01906             for (unsigned r = k; r < N; r++)
01907                 H(r,c) -= W1(r-k,c);
01908
01909         auto H2 = H.get_submatrix(0, N-1, k, N-1);
01910         auto W2 = mult<T,false,true>(H2 * v, v);
01911         for (unsigned c = k; c < N; c++)
01912             for (unsigned r = 0; r < N; r++)
01913                 H(r,c) -= W2(r,c-k);
01914
01915         if (calculate_Q) {
01916             auto Q1 = Q.get_submatrix(0, N-1, k, N-1);
01917             auto W3 = mult<T,false,true>(Q1 * v, v);
01918             for (unsigned c = k; c < N; c++)
01919                 for (unsigned r = 0; r < N; r++)
01920                     Q(r,c) -= W3(r,c-k);
01921         }
01922     }
01923
01924     for (unsigned row = 2; row < N; row++)
01925         for (unsigned col = 0; col < row-2; col++)
01926             H(row,col) = static_cast<T>(0);
01927
01928     return res;
01929 }
01930
01931 template<typename T>
01932 std::complex<T> wilkinson_shift(const Matrix<std::complex<T>& H, T tol = 1e-10) {
01933     if (! H.isquare()) throw std::runtime_error("Input matrix is not square");
01934
01935     const unsigned n = H.rows();
01936     std::complex<T> mu;
01937
01938     if (std::abs(H(n-1,n-2)) < tol) {
01939         mu = H(n-2,n-2);
01940     } else {
01941         auto trA = H(n-2,n-2) + H(n-1,n-1);
01942         auto detA = H(n-2,n-2) * H(n-1,n-1) - H(n-2, n-1) * H(n-1, n-2);
01943         mu = (trA + std::sqrt(trA*trA - 4.0*detA)) / 2.0;
01944     }
01945
01946     return mu;
01947 }
01948
01949 template<typename T>
01950 Eigenvalues_result<T> eigenvalues(const Matrix<std::complex<T>& A, T tol = 1e-12, unsigned max_iter =
100) {
01951     if (! A.isquare()) throw std::runtime_error("Input matrix is not square");
01952
01953     const unsigned N = A.rows();
01954     Matrix<std::complex<T>> H;
01955     bool success = false;
01956
01957     QR_result<std::complex<T>> QR;
01958
01959     // aliases
01960     auto& Q = QR.Q;
01961     auto& R = QR.R;
01962
01963     // Transfer A to Hessenberg form to improve convergence (skip calculation of Q)
01964     H = hessenberg(A, false).H;
01965
01966     for (unsigned iter = 0; iter < max_iter; iter++) {
01967         auto mu = wilkinson_shift(H, tol);
01968
01969         // subtract mu from diagonal
01970         for (unsigned n = 0; n < N; n++)
01971             H(n,n) -= mu;
01972
01973         // QR factorization with shifted H
01974         QR = qr(H);
01975         H = R * Q;
01976
01977         // add back mu to diagonal

```

```

01997     for (unsigned n = 0; n < N; n++)
01998         H(n,n) += mu;
01999
02000     // Check for convergence
02001     if (std::abs(H(N-2,N-1)) <= tol) {
02002         success = true;
02003         break;
02004     }
02005 }
02006
02007 Eigenvalues_result<T> res;
02008 res.eig = diag(H);
02009 res.err = std::abs(H(N-2,N-1));
02010 res.converged = success;
02011
02012 return res;
02013 }
02014
02024 template<typename T>
02025 Eigenvalues_result<T> eigenvalues(const Matrix<T>& A, T tol = 1e-12, unsigned max_iter = 100) {
02026     auto A_cplx = make_complex(A);
02027     return eigenvalues(A_cplx, tol, max_iter);
02028 }
02029
02044 template<typename T>
02045 Matrix<T> solve_triu(const Matrix<T>& U, const Matrix<T>& B) {
02046     if (! U.isquare()) throw std::runtime_error("Input matrix is not square");
02047     if (U.rows() != B.rows()) throw std::runtime_error("Unmatching matrix dimensions for solve");
02048
02049     const unsigned N = U.rows();
02050     const unsigned M = B.cols();
02051
02052     if (U.numel() == 0)
02053         return Matrix<T>();
02054
02055     Matrix<T> X(B);
02056
02057     for (unsigned m = 0; m < M; m++) {
02058         // backwards substitution for each column of B
02059         for (int n = N-1; n >= 0; n--) {
02060             for (unsigned j = n + 1; j < N; j++)
02061                 X(n,m) -= U(n,j) * X(j,m);
02062
02063             if (U(n,n) == 0.0) throw singular_matrix_exception("Singular matrix in solve_triu");
02064
02065             X(n,m) /= U(n,n);
02066         }
02067     }
02068
02069     return X;
02070 }
02071
02086 template<typename T>
02087 Matrix<T> solve_tril(const Matrix<T>& L, const Matrix<T>& B) {
02088     if (! L.isquare()) throw std::runtime_error("Input matrix is not square");
02089     if (L.rows() != B.rows()) throw std::runtime_error("Unmatching matrix dimensions for solve");
02090
02091     const unsigned N = L.rows();
02092     const unsigned M = B.cols();
02093
02094     if (L.numel() == 0)
02095         return Matrix<T>();
02096
02097     Matrix<T> X(B);
02098
02099     for (unsigned m = 0; m < M; m++) {
02100         // forwards substitution for each column of B
02101         for (unsigned n = 0; n < N; n++) {
02102             for (unsigned j = 0; j < n; j++)
02103                 X(n,m) -= L(n,j) * X(j,m);
02104
02105             if (L(n,n) == 0.0) throw singular_matrix_exception("Singular matrix in solve_tril");
02106
02107             X(n,m) /= L(n,n);
02108         }
02109     }
02110
02111     return X;
02112 }
02113
02128 template<typename T>
02129 Matrix<T> solve_square(const Matrix<T>& A, const Matrix<T>& B) {
02130     if (! A.isquare()) throw std::runtime_error("Input matrix is not square");
02131     if (A.rows() != B.rows()) throw std::runtime_error("Unmatching matrix dimensions for solve");
02132
02133     if (A.numel() == 0)
02134         return Matrix<T>();

```

```

02135
02136 Matrix<T> L;
02137 Matrix<T> U;
02138 std::vector<unsigned> P;
02139
02140 // LU decomposition with pivoting
02141 auto lup_res = lup(A);
02142
02143 auto y = solve_tril(lup_res.L, B);
02144 auto x = solve_triu(lup_res.U, y);
02145
02146 return permute_rows(x, lup_res.P);
02147 }
02148
02163 template<typename T>
02164 Matrix<T> solve_posdef(const Matrix<T>& A, const Matrix<T>& B) {
02165     if (! A.issquare()) throw std::runtime_error("Input matrix is not square");
02166     if (A.rows() != B.rows()) throw std::runtime_error("Unmatching matrix dimensions for solve");
02167
02168     if (A.numel() == 0)
02169         return Matrix<T>();
02170
02171     // LU decomposition with pivoting
02172     auto L = chol(A);
02173
02174     auto Y = solve_tril(L, B);
02175     return solve_triu(L.ctranspose(), Y);
02176 }
02177
02182 template<typename T>
02183 class Matrix {
02184     public:
02185         Matrix();
02186
02190         Matrix(unsigned size);
02196
02201         Matrix(unsigned nrows, unsigned ncols);
02202
02207         Matrix(T x, unsigned nrows, unsigned ncols);
02208
02214         Matrix(const T* array, unsigned nrows, unsigned ncols);
02215
02223         Matrix(const std::vector<T>& vec, unsigned nrows, unsigned ncols);
02224
02232         Matrix(std::initializer_list<T> init_list, unsigned nrows, unsigned ncols);
02233
02236         Matrix(const Matrix &);
02237
02240         virtual ~Matrix();
02241
02249         Matrix<T> get_submatrix(unsigned row_first, unsigned row_last, unsigned col_first, unsigned
col_last) const;
02250
02259         void set_submatrix(const Matrix<T>& smtx, unsigned row_first, unsigned col_first);
02260
02265         void clear();
02266
02274         void reshape(unsigned rows, unsigned cols);
02275
02281         void resize(unsigned rows, unsigned cols);
02282
02288         bool exists(unsigned row, unsigned col) const;
02289
02294         T* ptr(unsigned row, unsigned col);
02295
02302         T* ptr();
02303
02307         void fill(T value);
02308
02315         void fill_col(T value, unsigned col);
02316
02323         void fill_row(T value, unsigned row);
02324
02329         bool isempty() const;
02330
02334         bool issquare() const;
02335
02340         bool isequal(const Matrix<T>&) const;
02341
02347         bool isequal(const Matrix<T>&, T) const;
02348
02353         unsigned numel() const;
02354
02359         unsigned rows() const;
02360
02365         unsigned cols() const;
02366

```

```

02371     Matrix<T> transpose() const;
02372
02378     Matrix<T> ctranspose() const;
02379
02387     Matrix<T>& add(const Matrix<T>&);
02388
02396     Matrix<T>& subtract(const Matrix<T>&);
02397
02406     Matrix<T>& mult_hadamard(const Matrix<T>&);
02407
02413     Matrix<T>& add(T);
02414
02420     Matrix<T>& subtract(T);
02421
02427     Matrix<T>& mult(T);
02428
02434     Matrix<T>& div(T);
02435
02440     Matrix<T>& operator=(const Matrix<T>&);
02441
02446     Matrix<T>& operator=(T);
02447
02452     explicit operator std::vector<T>() const;
02453     std::vector<T> to_vector() const;
02454
02461     T& operator()(unsigned nel);
02462     T operator()(unsigned nel) const;
02463     T& at(unsigned nel);
02464     T at(unsigned nel) const;
02465
02472     T& operator()(unsigned row, unsigned col);
02473     T operator()(unsigned row, unsigned col) const;
02474     T& at(unsigned row, unsigned col);
02475     T at(unsigned row, unsigned col) const;
02476
02484     void add_row_to_another(unsigned to, unsigned from);
02485
02493     void add_col_to_another(unsigned to, unsigned from);
02494
02502     void mult_row_by_another(unsigned to, unsigned from);
02503
02511     void mult_col_by_another(unsigned to, unsigned from);
02512
02519     void swap_rows(unsigned i, unsigned j);
02520
02527     void swap_cols(unsigned i, unsigned j);
02528
02535     std::vector<T> col_to_vector(unsigned col) const;
02536
02543     std::vector<T> row_to_vector(unsigned row) const;
02544
02552     void col_from_vector(const std::vector<T>&, unsigned col);
02553
02561     void row_from_vector(const std::vector<T>&, unsigned row);
02562
02563 private:
02564     unsigned nrows;
02565     unsigned ncols;
02566     std::vector<T> data;
02567 };
02568
02569 /*
02570  * Implementation of Matrix class methods
02571  */
02572
02573 template<typename T>
02574 Matrix<T>::Matrix() : nrows(0), ncols(0), data() { }
02575
02576 template<typename T>
02577 Matrix<T>::Matrix(unsigned size) : Matrix(size, size) { }
02578
02579 template<typename T>
02580 Matrix<T>::Matrix(unsigned rows, unsigned cols) : nrows(rows), ncols(cols) {
02581     data.resize(numel());
02582 }
02583
02584 template<typename T>
02585 Matrix<T>::Matrix(T x, unsigned rows, unsigned cols) : Matrix(rows, cols) {
02586     fill(x);
02587 }
02588
02589 template<typename T>
02590 Matrix<T>::Matrix(const T* array, unsigned rows, unsigned cols) : Matrix(rows, cols) {
02591     data.assign(array, array + numel());
02592 }
02593
02594 template<typename T>

```

```

02595 Matrix<T>::Matrix(const std::vector<T>& vec, unsigned rows, unsigned cols) : Matrix(rows, cols) {
02596     if (vec.size() != numel()) throw std::runtime_error("Size of initialization vector not consistent
        with matrix dimensions");
02597
02598     data.assign(vec.begin(), vec.end());
02599 }
02600
02601 template<typename T>
02602 Matrix<T>::Matrix(std::initializer_list<T> init_list, unsigned rows, unsigned cols) : Matrix(rows,
    cols) {
02603     if (init_list.size() != numel()) throw std::runtime_error("Size of initialization list not
        consistent with matrix dimensions");
02604
02605     auto it = init_list.begin();
02606
02607     for (unsigned row = 0; row < this->nrows; row++)
02608         for (unsigned col = 0; col < this->ncols; col++)
02609             this->at(row,col) = *(it++);
02610 }
02611
02612 template<typename T>
02613 Matrix<T>::Matrix(const Matrix & other) : Matrix(other.nrows, other.ncols) {
02614     this->data.assign(other.data.begin(), other.data.end());
02615 }
02616
02617 template<typename T>
02618 Matrix<T>& Matrix<T>::operator=(const Matrix<T>& other) {
02619     this->nrows = other.nrows;
02620     this->ncols = other.ncols;
02621     this->data.assign(other.data.begin(), other.data.end());
02622     return *this;
02623 }
02624
02625 template<typename T>
02626 Matrix<T>& Matrix<T>::operator=(T s) {
02627     fill(s);
02628     return *this;
02629 }
02630
02631 template<typename T>
02632 inline Matrix<T>::operator std::vector<T>() const {
02633     return data;
02634 }
02635
02636 template<typename T>
02637 inline void Matrix<T>::clear() {
02638     this->nrows = 0;
02639     this->ncols = 0;
02640     data.resize(0);
02641 }
02642
02643 template<typename T>
02644 void Matrix<T>::reshape(unsigned rows, unsigned cols) {
02645     if (this->numel() != rows * cols) throw std::runtime_error("Illegal attempt to change number of
        elements via reshape");
02646
02647     this->nrows = rows;
02648     this->ncols = cols;
02649 }
02650
02651 template<typename T>
02652 void Matrix<T>::resize(unsigned rows, unsigned cols) {
02653     this->nrows = rows;
02654     this->ncols = cols;
02655     data.resize(nrows*ncols);
02656 }
02657
02658 template<typename T>
02659 Matrix<T> Matrix<T>::get_submatrix(unsigned row_base, unsigned row_lim, unsigned col_base, unsigned
    col_lim) const {
02660     if (row_base > row_lim) throw std::out_of_range("Row index of submatrix out of range");
02661     if (col_base > col_lim) throw std::out_of_range("Column index of submatrix out of range");
02662     if (row_lim >= this->rows()) throw std::out_of_range("Row index of submatrix out of range");
02663     if (col_lim >= this->cols()) throw std::out_of_range("Column index of submatrix out of range");
02664
02665     unsigned num_rows = row_lim - row_base + 1;
02666     unsigned num_cols = col_lim - col_base + 1;
02667     Matrix<T> S(num_rows, num_cols);
02668     for (unsigned i = 0; i < num_rows; i++) {
02669         for (unsigned j = 0; j < num_cols; j++) {
02670             S(i,j) = at(row_base + i, col_base + j);
02671         }
02672     }
02673     return S;
02674 }
02675
02676 template<typename T>

```

```

02677 void Matrix<T>::set_submatrix(const Matrix<T>& S, unsigned row_base, unsigned col_base) {
02678     if (this->isempty()) throw std::runtime_error("Invalid attempt to set submatrix in empty matrix");
02679
02680     const unsigned row_lim = row_base + S.rows() - 1;
02681     const unsigned col_lim = col_base + S.cols() - 1;
02682
02683     if (row_base > row_lim) throw std::out_of_range("Row index of submatrix out of range");
02684     if (col_base > col_lim) throw std::out_of_range("Column index of submatrix out of range");
02685     if (row_lim >= this->rows()) throw std::out_of_range("Row index of submatrix out of range");
02686     if (col_lim >= this->cols()) throw std::out_of_range("Column index of submatrix out of range");
02687
02688     unsigned num_rows = row_lim - row_base + 1;
02689     unsigned num_cols = col_lim - col_base + 1;
02690     for (unsigned i = 0; i < num_rows; i++)
02691         for (unsigned j = 0; j < num_cols; j++)
02692             at(row_base + i, col_base + j) = S(i,j);
02693 }
02694
02695 template<typename T>
02696 inline T & Matrix<T>::operator()(unsigned nel) {
02697     return at(nel);
02698 }
02699
02700 template<typename T>
02701 inline T & Matrix<T>::operator()(unsigned row, unsigned col) {
02702     return at(row, col);
02703 }
02704
02705 template<typename T>
02706 inline T Matrix<T>::operator()(unsigned nel) const {
02707     return at(nel);
02708 }
02709
02710 template<typename T>
02711 inline T Matrix<T>::operator()(unsigned row, unsigned col) const {
02712     return at(row, col);
02713 }
02714
02715 template<typename T>
02716 inline T & Matrix<T>::at(unsigned nel) {
02717     if (!(nel < numel())) throw std::out_of_range("Element index out of range");
02718
02719     return data[nel];
02720 }
02721
02722 template<typename T>
02723 inline T & Matrix<T>::at(unsigned row, unsigned col) {
02724     if (!(row < rows() && col < cols())) std::cout << "at() failed at " << row << ", " << col << std::endl;
02725
02726     return data[nrows * col + row];
02727 }
02728
02729 template<typename T>
02730 inline T Matrix<T>::at(unsigned nel) const {
02731     if (!(nel < numel())) throw std::out_of_range("Element index out of range");
02732
02733     return data[nel];
02734 }
02735
02736 template<typename T>
02737 inline T Matrix<T>::at(unsigned row, unsigned col) const {
02738     if (!(row < rows())) throw std::out_of_range("Row index out of range");
02739     if (!(col < cols())) throw std::out_of_range("Column index out of range");
02740
02741     return data[nrows * col + row];
02742 }
02743
02744 template<typename T>
02745 inline void Matrix<T>::fill(T value) {
02746     for (unsigned i = 0; i < numel(); i++)
02747         data[i] = value;
02748 }
02749
02750 template<typename T>
02751 inline void Matrix<T>::fill_col(T value, unsigned col) {
02752     if (!(col < cols())) throw std::out_of_range("Column index out of range");
02753
02754     for (unsigned i = col * nrows; i < (col+1) * nrows; i++)
02755         data[i] = value;
02756 }
02757
02758 template<typename T>
02759 inline void Matrix<T>::fill_row(T value, unsigned row) {
02760     if (!(row < rows())) throw std::out_of_range("Row index out of range");
02761
02762     for (unsigned i = 0; i < ncols; i++)
02763         data[row + i * nrows] = value;

```

```

02764 }
02765
02766 template<typename T>
02767 inline bool Matrix<T>::exists(unsigned row, unsigned col) const {
02768     return (row < nrows && col < ncols);
02769 }
02770
02771 template<typename T>
02772 inline T* Matrix<T>::ptr(unsigned row, unsigned col) {
02773     if (!(row < rows())) throw std::out_of_range("Row index out of range");
02774     if (!(col < cols())) throw std::out_of_range("Column index out of range");
02775
02776     return data.data() + nrows * col + row;
02777 }
02778
02779 template<typename T>
02780 inline T* Matrix<T>::ptr() {
02781     return data.data();
02782 }
02783
02784 template<typename T>
02785 inline bool Matrix<T>::isempty() const {
02786     return (nrows == 0) || (ncols == 0);
02787 }
02788
02789 template<typename T>
02790 inline bool Matrix<T>::issquare() const {
02791     return (nrows == ncols) && !isempty();
02792 }
02793
02794 template<typename T>
02795 bool Matrix<T>::isequal(const Matrix<T>& A) const {
02796     bool ret = true;
02797     if (nrows != A.rows() || ncols != A.cols()) {
02798         ret = false;
02799     } else {
02800         for (unsigned i = 0; i < numel(); i++) {
02801             if (at(i) != A(i)) {
02802                 ret = false;
02803                 break;
02804             }
02805         }
02806     }
02807     return ret;
02808 }
02809
02810 template<typename T>
02811 bool Matrix<T>::isequal(const Matrix<T>& A, T tol) const {
02812     bool ret = true;
02813     if (rows() != A.rows() || cols() != A.cols()) {
02814         ret = false;
02815     } else {
02816         auto abs_tol = std::abs(tol); // workaround for complex
02817         for (unsigned i = 0; i < A.numel(); i++) {
02818             if (abs_tol < std::abs(at(i) - A(i))) {
02819                 ret = false;
02820                 break;
02821             }
02822         }
02823     }
02824     return ret;
02825 }
02826
02827 template<typename T>
02828 inline unsigned Matrix<T>::numel() const {
02829     return nrows * ncols;
02830 }
02831
02832 template<typename T>
02833 inline unsigned Matrix<T>::rows() const {
02834     return nrows;
02835 }
02836
02837 template<typename T>
02838 inline unsigned Matrix<T>::cols() const {
02839     return ncols;
02840 }
02841
02842 template<typename T>
02843 inline Matrix<T> Matrix<T>::transpose() const {
02844     Matrix<T> res(ncols, nrows);
02845     for (unsigned c = 0; c < ncols; c++)
02846         for (unsigned r = 0; r < nrows; r++)
02847             res(c,r) = at(r,c);
02848     return res;
02849 }
02850

```

```

02851 template<typename T>
02852 inline Matrix<T> Matrix<T>::ctranspose() const {
02853     Matrix<T> res(ncols, nrows);
02854     for (unsigned c = 0; c < ncols; c++)
02855         for (unsigned r = 0; r < nrows; r++)
02856             res(c,r) = cconj(at(r,c));
02857     return res;
02858 }
02859
02860 template<typename T>
02861 Matrix<T>& Matrix<T>::add(const Matrix<T>& m) {
02862     if (!(m.rows() == rows() && m.cols() == cols())) throw std::runtime_error("Unmatching matrix
dimensions for iadd");
02863
02864     for (unsigned i = 0; i < numel(); i++)
02865         data[i] += m(i);
02866     return *this;
02867 }
02868
02869 template<typename T>
02870 Matrix<T>& Matrix<T>::subtract(const Matrix<T>& m) {
02871     if (!(m.rows() == rows() && m.cols() == cols())) throw std::runtime_error("Unmatching matrix
dimensions for isubtract");
02872
02873     for (unsigned i = 0; i < numel(); i++)
02874         data[i] -= m(i);
02875     return *this;
02876 }
02877
02878 template<typename T>
02879 Matrix<T>& Matrix<T>::mult_hadamard(const Matrix<T>& m) {
02880     if (!(m.rows() == rows() && m.cols() == cols())) throw std::runtime_error("Unmatching matrix
dimensions for ihprod");
02881
02882     for (unsigned i = 0; i < numel(); i++)
02883         data[i] *= m(i);
02884     return *this;
02885 }
02886
02887 template<typename T>
02888 Matrix<T>& Matrix<T>::add(T s) {
02889     for (auto& x : data)
02890         x += s;
02891     return *this;
02892 }
02893
02894 template<typename T>
02895 Matrix<T>& Matrix<T>::subtract(T s) {
02896     for (auto& x : data)
02897         x -= s;
02898     return *this;
02899 }
02900
02901 template<typename T>
02902 Matrix<T>& Matrix<T>::mult(T s) {
02903     for (auto& x : data)
02904         x *= s;
02905     return *this;
02906 }
02907
02908 template<typename T>
02909 Matrix<T>& Matrix<T>::div(T s) {
02910     for (auto& x : data)
02911         x /= s;
02912     return *this;
02913 }
02914
02915 template<typename T>
02916 void Matrix<T>::add_row_to_another(unsigned to, unsigned from) {
02917     if (!(to < rows() && from < rows())) throw std::out_of_range("Row index out of range");
02918
02919     for (unsigned k = 0; k < cols(); k++)
02920         at(to, k) += at(from, k);
02921 }
02922
02923 template<typename T>
02924 void Matrix<T>::add_col_to_another(unsigned to, unsigned from) {
02925     if (!(to < cols() && from < cols())) throw std::out_of_range("Column index out of range");
02926
02927     for (unsigned k = 0; k < rows(); k++)
02928         at(k, to) += at(k, from);
02929 }
02930
02931 template<typename T>
02932 void Matrix<T>::mult_row_by_another(unsigned to, unsigned from) {
02933     if (!(to < rows() && from < rows())) throw std::out_of_range("Row index out of range");
02934

```



```

02935     for (unsigned k = 0; k < cols(); k++)
02936         at(to, k) *= at(from, k);
02937 }
02938
02939 template<typename T>
02940 void Matrix<T>::mult_col_by_another(unsigned to, unsigned from) {
02941     if (!(to < cols() && from < cols())) throw std::out_of_range("Column index out of range");
02942
02943     for (unsigned k = 0; k < rows(); k++)
02944         at(k, to) *= at(k, from);
02945 }
02946
02947 template<typename T>
02948 void Matrix<T>::swap_rows(unsigned i, unsigned j) {
02949     if (!(i < rows() && j < rows())) throw std::out_of_range("Row index out of range");
02950
02951     for (unsigned k = 0; k < cols(); k++) {
02952         T tmp = at(i, k);
02953         at(i, k) = at(j, k);
02954         at(j, k) = tmp;
02955     }
02956 }
02957
02958 template<typename T>
02959 void Matrix<T>::swap_cols(unsigned i, unsigned j) {
02960     if (!(i < cols() && j < cols())) throw std::out_of_range("Column index out of range");
02961
02962     for (unsigned k = 0; k < rows(); k++) {
02963         T tmp = at(k, i);
02964         at(k, i) = at(k, j);
02965         at(k, j) = tmp;
02966     }
02967 }
02968
02969 template<typename T>
02970 inline std::vector<T> Matrix<T>::to_vector() const {
02971     return data;
02972 }
02973
02974 template<typename T>
02975 inline std::vector<T> Matrix<T>::col_to_vector(unsigned col) const {
02976     std::vector<T> vec(rows());
02977     for (unsigned i = 0; i < rows(); i++)
02978         vec[i] = at(i, col);
02979     return vec;
02980 }
02981
02982 template<typename T>
02983 inline std::vector<T> Matrix<T>::row_to_vector(unsigned row) const {
02984     std::vector<T> vec(cols());
02985     for (unsigned i = 0; i < cols(); i++)
02986         vec[i] = at(row, i);
02987     return vec;
02988 }
02989
02990 template<typename T>
02991 inline void Matrix<T>::col_from_vector(const std::vector<T>& vec, unsigned col) {
02992     if (vec.size() != rows()) throw std::runtime_error("Vector size is not equal to number of rows");
02993     if (col >= cols()) throw std::out_of_range("Column index out of range");
02994
02995     for (unsigned i = 0; i < rows(); i++)
02996         data[col*rows() + i] = vec[i];
02997 }
02998
02999 template<typename T>
03000 inline void Matrix<T>::row_from_vector(const std::vector<T>& vec, unsigned row) {
03001     if (vec.size() != rows()) throw std::runtime_error("Vector size is not equal to number of columns");
03002     if (row >= rows()) throw std::out_of_range("Row index out of range");
03003
03004     for (unsigned i = 0; i < cols(); i++)
03005         data[row + i*rows()] = vec[i];
03006 }
03007
03008 template<typename T>
03009 Matrix<T>::~Matrix() { }
03010
03011 } // namespace Matrix_hpp
03012
03013 #endif // __MATRIX_HPP__

```

