

Matrix HPP

Generated by Doxygen 1.9.8

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

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

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

6.1.1.40 kron()	56
6.1.1.41 ldl()	56
6.1.1.42 lu()	57
6.1.1.43 lup()	57
6.1.1.44 make_complex() [1/2]	58
6.1.1.45 make_complex() [2/2]	58
6.1.1.46 mult() [1/4]	59
6.1.1.47 mult() [2/4]	59
6.1.1.48 mult() [3/4]	60
6.1.1.49 mult() [4/4]	60
6.1.1.50 mult_hadamard()	61
6.1.1.51 norm_fro()	61
6.1.1.52 norm_inf()	62
6.1.1.53 norm_p1()	62
6.1.1.54 ones() [1/2]	62
6.1.1.55 ones() [2/2]	63
6.1.1.56 operator!=()	63
6.1.1.57 operator*() [1/5]	64
6.1.1.58 operator*() [2/5]	64
6.1.1.59 operator*() [3/5]	64
6.1.1.60 operator*() [4/5]	64
6.1.1.61 operator*() [5/5]	65
6.1.1.62 operator*=() [1/2]	65
6.1.1.63 operator*=() [2/2]	65
6.1.1.64 operator+() [1/3]	65
6.1.1.65 operator+() [2/3]	66
6.1.1.66 operator+() [3/3]	66
6.1.1.67 operator+=() [1/2]	66
6.1.1.68 operator+=() [2/2]	66
6.1.1.69 operator-() [1/2]	67
6.1.1.70 operator-() [2/2]	67
6.1.1.71 operator-=() [1/2]	67
6.1.1.72 operator-=() [2/2]	67
6.1.1.73 operator/()	68
6.1.1.74 operator/=()	68
6.1.1.75 operator<<()	68
6.1.1.76 operator==()	68
6.1.1.77 operator^()	69
6.1.1.78 operator^=()	69
6.1.1.79 permute_cols()	69
6.1.1.80 permute_rows()	70
6.1.1.81 permute_rows_and_cols()	70

6.1.1.82 <code>pinv()</code>	71
6.1.1.83 <code>qr()</code>	72
6.1.1.84 <code>qr_householder()</code>	72
6.1.1.85 <code>qr_red_gs()</code>	73
6.1.1.86 <code>real()</code>	73
6.1.1.87 <code>repmat()</code>	74
6.1.1.88 <code>solve_posdef()</code>	74
6.1.1.89 <code>solve_square()</code>	75
6.1.1.90 <code>solve_tril()</code>	75
6.1.1.91 <code>solve_triu()</code>	76
6.1.1.92 <code>subtract()</code> ^[1/2]	77
6.1.1.93 <code>subtract()</code> ^[2/2]	77
6.1.1.94 <code>trace()</code>	78
6.1.1.95 <code>transpose()</code>	78
6.1.1.96 <code>tril()</code>	78
6.1.1.97 <code>triu()</code>	78
6.1.1.98 <code>wilkinson_shift()</code>	79
6.1.1.99 <code>zeros()</code> ^[1/2]	79
6.1.1.100 <code>zeros()</code> ^[2/2]	79
6.2 <code>matrix.hpp</code>	80

Chapter 1

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

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

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

1.1 Installation

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

1.2 Functionality

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

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

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

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

1.3 Hello world example

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

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

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

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

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

    auto C = A + B;

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

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

1.4 Tests

Unit tests are compiled with `make tests`.

1.5 License

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

Chapter 2

Hierarchical Index

2.1 Class Hierarchy

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

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

Chapter 3

Class Index

3.1 Class List

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

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

Chapter 4

File Index

4.1 File List

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

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

Chapter 5

Class Documentation

5.1 Mtx::Eigenvalues_result< T > Struct Template Reference

Result of eigenvalues.

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

Public Attributes

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

5.1.1 Detailed Description

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

Result of eigenvalues.

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

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

- [matrix.hpp](#)

5.2 Mtx::Hessenberg_result< T > Struct Template Reference

Result of Hessenberg decomposition.

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

Public Attributes

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

5.2.1 Detailed Description

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

Result of Hessenberg decomposition.

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

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

- [matrix.hpp](#)

5.3 Mtx::LDL_result< T > Struct Template Reference

Result of LDL decomposition.

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

Public Attributes

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

5.3.1 Detailed Description

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

Result of LDL decomposition.

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

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

- [matrix.hpp](#)

5.4 Mtx::LU_result< T > Struct Template Reference

Result of LU decomposition.

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

Public Attributes

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

5.4.1 Detailed Description

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

Result of LU decomposition.

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

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

- [matrix.hpp](#)

5.5 Mtx::LUP_result< T > Struct Template Reference

Result of LU decomposition with pivoting.

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

Public Attributes

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

5.5.1 Detailed Description

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

Result of LU decomposition with pivoting.

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

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

- [matrix.hpp](#)

5.6 Mtx::Matrix< T > Class Template Reference

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

Public Member Functions

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

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

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

5.6.1 Detailed Description

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

[Matrix](#) class definition.

5.6.2 Constructor & Destructor Documentation

5.6.2.1 Matrix() [1/8]

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

Default constructor.

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

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

5.6.2.2 Matrix() [2/8]

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

Square matrix constructor.

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

5.6.2.3 Matrix() [3/8]

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

Rectangular matrix constructor.

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

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

5.6.2.4 Matrix() [4/8]

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

Rectangular matrix constructor with fill.

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

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

5.6.2.5 Matrix() [5/8]

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

Rectangular matrix constructor with initialization.

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

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

5.6.2.6 Matrix() [6/8]

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

Rectangular matrix constructor with initialization.

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

Exceptions

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

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

5.6.2.7 Matrix() [7/8]

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

Rectangular matrix constructor with initialization.

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

Exceptions

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

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

5.6.2.8 Matrix() [8/8]

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

Copy constructor.

5.6.2.9 ~Matrix()

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

Destructor.

5.6.3 Member Function Documentation

5.6.3.1 add() [1/2]

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

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

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

Exceptions

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

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

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

5.6.3.2 add() [2/2]

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

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

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

5.6.3.3 add_col_to_another()

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

Column addition.

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

Exceptions

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

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

5.6.3.4 add_row_to_another()

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

Row addition.

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

Exceptions

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

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

5.6.3.5 clear()

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

Clears the matrix.

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

5.6.3.6 col_from_vector()

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

Column from vector.

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

Exceptions

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

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

5.6.3.7 col_to_vector()

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

Column to vector.

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

Exceptions

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

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

5.6.3.8 cols()

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

Number of columns.

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

Referenced by [Mtx::Matrix< T >::add\(\)](#), [Mtx::add\(\)](#), [Mtx::add\(\)](#), [Mtx::Matrix< T >::add_col_to_another\(\)](#), [Mtx::Matrix< T >::add_row_to_another\(\)](#), [Mtx::adj\(\)](#), [Mtx::circshift\(\)](#), [Mtx::cofactor\(\)](#), [Mtx::Matrix< T >::col_from_vector\(\)](#), [Mtx::concatenate_horizontal\(\)](#), [Mtx::concatenate_vertical\(\)](#), [Mtx::div\(\)](#), [Mtx::Matrix< T >::fill_col\(\)](#), [Mtx::Matrix< T >::get_submatrix\(\)](#), [Mtx::householder_reflection\(\)](#), [Mtx::imag\(\)](#), [Mtx::Matrix< T >::isequal\(\)](#), [Mtx::Matrix< T >::isequal\(\)](#), [Mtx::istril\(\)](#), [Mtx::istriu\(\)](#), [Mtx::kron\(\)](#), [Mtx::lu\(\)](#), [Mtx::lup\(\)](#), [Mtx::make_complex\(\)](#), [Mtx::make_complex\(\)](#), [Mtx::mult\(\)](#), [Mtx::mult\(\)](#), [Mtx::mult\(\)](#), [Mtx::mult\(\)](#), [Mtx::Matrix< T >::mult_col_by_another\(\)](#), [Mtx::Matrix< T >::mult_hadamard\(\)](#), [Mtx::mult_hadamard\(\)](#), [Mtx::Matrix< T >::mult_row_by_another\(\)](#), [Mtx::norm_inf\(\)](#), [Mtx::norm_p1\(\)](#), [Mtx::operator<<\(\)](#), [Mtx::permute_cols\(\)](#), [Mtx::permute_rows\(\)](#), [Mtx::permute_rows_and_cols\(\)](#), [Mtx::pinv\(\)](#), [Mtx::Matrix< T >::ptr\(\)](#), [Mtx::qr_householder\(\)](#), [Mtx::qr_red_gs\(\)](#), [Mtx::real\(\)](#), [Mtx::repmat\(\)](#), [Mtx::Matrix< T >::reshape\(\)](#), [Mtx::Matrix< T >::resize\(\)](#), [Mtx::Matrix< T >::row_from_vector\(\)](#), [Mtx::Matrix< T >::row_to_vector\(\)](#), [Mtx::Matrix< T >::set_submatrix\(\)](#), [Mtx::solve_tril\(\)](#), [Mtx::solve_triu\(\)](#), [Mtx::Matrix< T >::subtract\(\)](#), [Mtx::subtract\(\)](#), [Mtx::subtract\(\)](#), [Mtx::Matrix< T >::swap_cols\(\)](#), [Mtx::Matrix< T >::swap_rows\(\)](#), [Mtx::tril\(\)](#), and [Mtx::triu\(\)](#).

5.6.3.9 ctranspose()

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

Transpose a complex matrix.

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

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

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

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

5.6.3.10 div()

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

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

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

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

5.6.3.11 exists()

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

Element exist check.

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

5.6.3.12 fill()

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

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

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

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

5.6.3.13 fill_col()

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

Fill column with a scalar.

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

Exceptions

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

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

5.6.3.14 fill_row()

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

Fill row with a scalar.

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

Exceptions

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

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

5.6.3.15 `get_submatrix()`

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

Extract a submatrix.

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

Exceptions

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

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

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

5.6.3.16 `isempty()`

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

Emptiness check.

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

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

5.6.3.17 `isequal()` [1/2]

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

[Matrix](#) equality check.

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

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

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

5.6.3.18 isequal() [2/2]

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

Matrix equality check with tolerance.

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

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

5.6.3.19 mult()

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

Matrix product with scalar (in-place).

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

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

5.6.3.20 mult_col_by_another()

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

Column multiplication.

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

Exceptions

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

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

5.6.3.21 mult_hadamard()

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

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

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

Exceptions

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

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

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

5.6.3.22 mult_row_by_another()

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

Row multiplication.

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

Exceptions

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

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

5.6.3.23 numel()

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

[Matrix](#) capacity.

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

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

5.6.3.24 operator std::vector< T >()

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

Vector cast operator.

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

5.6.3.25 operator>() [1/2]

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

Element access operator (1D)

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

Exceptions

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

5.6.3.26 operator>() [2/2]

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

Element access operator (2D)

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

Exceptions

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

5.6.3.27 operator=() [1/2]

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

Matrix assignment.

Performs deep-copy of another matrix.

5.6.3.28 operator=() [2/2]

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

Matrix fill operator.

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

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

5.6.3.29 ptr() [1/2]

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

Memory pointer.

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

Exceptions

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

5.6.3.30 ptr() [2/2]

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

Memory pointer.

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

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

5.6.3.31 reshape()

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

[Matrix](#) dimension reshape.

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

Exceptions

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

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

5.6.3.32 resize()

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

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

Resize the matrix.

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

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

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

5.6.3.33 row_from_vector()

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

Row from vector.

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

Exceptions

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

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

5.6.3.34 row_to_vector()

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

Row to vector.

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

Exceptions

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

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

5.6.3.35 rows()

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

Number of rows.

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

Referenced by [Mtx::Matrix< T >::add\(\)](#), [Mtx::add\(\)](#), [Mtx::add\(\)](#), [Mtx::Matrix< T >::add_col_to_another\(\)](#), [Mtx::Matrix< T >::add_row_to_another\(\)](#), [Mtx::adj\(\)](#), [Mtx::chol\(\)](#), [Mtx::cholinv\(\)](#), [Mtx::circshift\(\)](#), [Mtx::cofactor\(\)](#), [Mtx::Matrix< T >::col_from_vector\(\)](#), [Mtx::Matrix< T >::col_to_vector\(\)](#), [Mtx::concatenate_horizontal\(\)](#), [Mtx::concatenate_vertical\(\)](#), [Mtx::det\(\)](#), [Mtx::det_lu\(\)](#), [Mtx::diag\(\)](#), [Mtx::div\(\)](#), [Mtx::eigenvalues\(\)](#), [Mtx::Matrix< T >::fill_row\(\)](#), [Mtx::Matrix< T >::get_submatrix\(\)](#), [Mtx::hessenberg\(\)](#), [Mtx::imag\(\)](#), [Mtx::inv\(\)](#), [Mtx::inv_gauss_jordan\(\)](#), [Mtx::inv_tril\(\)](#), [Mtx::inv_triu\(\)](#), [Mtx::Matrix< T >::is_equal\(\)](#), [Mtx::Matrix< T >::is_equal\(\)](#), [Mtx::ishess\(\)](#), [Mtx::istril\(\)](#), [Mtx::istriu\(\)](#), [Mtx::kron\(\)](#), [Mtx::ldl\(\)](#), [Mtx::lu\(\)](#), [Mtx::lup\(\)](#), [Mtx::make_complex\(\)](#), [Mtx::make_complex\(\)](#), [Mtx::mult\(\)](#), [Mtx::mult\(\)](#), [Mtx::mult\(\)](#), [Mtx::mult\(\)](#), [Mtx::Matrix< T >::mult_col_by_another\(\)](#), [Mtx::Matrix< T >::mult_hadamard\(\)](#), [Mtx::mult_hadamard\(\)](#), [Mtx::Matrix< T >::mult_row_by_another\(\)](#), [Mtx::norm_inf\(\)](#), [Mtx::norm_p1\(\)](#), [Mtx::operator<<\(\)](#), [Mtx::permute_cols\(\)](#), [Mtx::permute_rows\(\)](#), [Mtx::permute_rows_and_cols\(\)](#), [Mtx::pinv\(\)](#), [Mtx::Matrix< T >::ptr\(\)](#), [Mtx::qr_householder\(\)](#), [Mtx::qr_red_gs\(\)](#), [Mtx::real\(\)](#), [Mtx::repmat\(\)](#), [Mtx::Matrix< T >::reshape\(\)](#), [Mtx::Matrix< T >::resize\(\)](#), [Mtx::Matrix< T >::row_from_vector\(\)](#), [Mtx::Matrix< T >::set_submatrix\(\)](#), [Mtx::solve_posdef\(\)](#), [Mtx::solve_square\(\)](#), [Mtx::solve_tril\(\)](#), [Mtx::solve_triu\(\)](#), [Mtx::Matrix< T >::subtract\(\)](#), [Mtx::subtract\(\)](#), [Mtx::subtract\(\)](#), [Mtx::Matrix< T >::swap_cols\(\)](#), [Mtx::Matrix< T >::swap_rows\(\)](#), [Mtx::trace\(\)](#), [Mtx::tril\(\)](#), [Mtx::triu\(\)](#), and [Mtx::wilkinson_shift\(\)](#).

5.6.3.36 set_submatrix()

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

Embed a submatrix.

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

Exceptions

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

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

5.6.3.37 shape()

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

Matrix shape.

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

5.6.3.38 subtract() [1/2]

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

Matrix subtraction (in-place).

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

Exceptions

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

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

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

5.6.3.39 subtract() [2/2]

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

Matrix subtraction with scalar (in-place).

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

5.6.3.40 swap_cols()

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

Column swap.

Swaps element values between two columns.

Exceptions

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

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

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

5.6.3.41 swap_rows()

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

Row swap.

Swaps element values of two columns.

Exceptions

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

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

5.6.3.42 transpose()

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

Transpose a matrix.

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

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

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

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

- [matrix.hpp](#)

5.7 Mtx::QR_result< T > Struct Template Reference

Result of QR decomposition.

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

Public Attributes

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

5.7.1 Detailed Description

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

Result of QR decomposition.

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

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

- [matrix.hpp](#)

5.8 Mtx::singular_matrix_exception Class Reference

Singular matrix exception.

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

Inheritance diagram for Mtx::singular_matrix_exception:

Chapter 6

File Documentation

6.1 matrix.hpp File Reference

Classes

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

Functions

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

- Matrix of zeros.*

 - `template<typename T >`
`Matrix< T > Mtx::zeros (unsigned n)`
Square matrix of zeros.
- `template<typename T >`
`Matrix< T > Mtx::ones (unsigned nrows, unsigned ncols)`
Matrix of ones.

 - `template<typename T >`
`Matrix< T > Mtx::ones (unsigned n)`
Square matrix of ones.
- `template<typename T >`
`Matrix< T > Mtx::eye (unsigned n)`
Identity matrix.

 - `template<typename T >`
`Matrix< T > Mtx::diag (const T *array, size_t n)`
Diagonal matrix from array.
- `template<typename T >`
`Matrix< T > Mtx::diag (const std::vector< T > &v)`
Diagonal matrix from std::vector.

 - `template<typename T >`
`std::vector< T > Mtx::diag (const Matrix< T > &A)`
Diagonal extraction.
- `template<typename T >`
`Matrix< T > Mtx::circulant (const T *array, unsigned n)`
Circulant matrix from array.

 - `template<typename T >`
`Matrix< std::complex< T > > Mtx::make_complex (const Matrix< T > &Re, const Matrix< T > &Im)`
Create complex matrix from real and imaginary matrices.
- `template<typename T >`
`Matrix< std::complex< T > > Mtx::make_complex (const Matrix< T > &Re)`
Create complex matrix from real matrix.

 - `template<typename T >`
`Matrix< T > Mtx::real (const Matrix< std::complex< T > > &C)`
Get real part of complex matrix.
- `template<typename T >`
`Matrix< T > Mtx::imag (const Matrix< std::complex< T > > &C)`
Get imaginary part of complex matrix.

 - `template<typename T >`
`Matrix< T > Mtx::circulant (const std::vector< T > &v)`
Circulant matrix from std::vector.
- `template<typename T >`
`Matrix< T > Mtx::transpose (const Matrix< T > &A)`
Transpose a matrix.

 - `template<typename T >`
`Matrix< T > Mtx::ctranspose (const Matrix< T > &A)`
Transpose a complex matrix.
- `template<typename T >`
`Matrix< T > Mtx::circshift (const Matrix< T > &A, int row_shift, int col_shift)`
Circular shift.

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

- `template<typename T >`
`Matrix< T > Mtx::concatenate_horizontal (const Matrix< T > &A, const Matrix< T > &B)`
Horizontal matrix concatenation.
- `template<typename T >`
`Matrix< T > Mtx::concatenate_vertical (const Matrix< T > &A, const Matrix< T > &B)`
Vertical matrix concatenation.
- `template<typename T >`
`double Mtx::norm_fro (const Matrix< T > &A)`
Frobenius norm.
- `template<typename T >`
`double Mtx::norm_p1 (const Matrix< T > &A)`
Matrix $p = 1$ norm (column norm).
- `template<typename T >`
`double Mtx::norm_inf (const Matrix< T > &A)`
Matrix $p = \infty$ norm (row norm).
- `template<typename T >`
`Matrix< T > Mtx::tril (const Matrix< T > &A)`
Extract triangular lower part.
- `template<typename T >`
`Matrix< T > Mtx::triu (const Matrix< T > &A)`
Extract triangular upper part.
- `template<typename T >`
`bool Mtx::istril (const Matrix< T > &A)`
Lower triangular matrix check.
- `template<typename T >`
`bool Mtx::istriu (const Matrix< T > &A)`
Lower triangular matrix check.
- `template<typename T >`
`bool Mtx::ishess (const Matrix< T > &A)`
Hessenberg matrix check.
- `template<typename T >`
`void Mtx::foreach_elem (Matrix< T > &A, std::function< T(T)> func)`
Applies custom function element-wise in-place.
- `template<typename T >`
`Matrix< T > Mtx::foreach_elem_copy (const Matrix< T > &A, std::function< T(T)> func)`
Applies custom function element-wise with matrix copy.
- `template<typename T >`
`Matrix< T > Mtx::permute_rows (const Matrix< T > &A, const std::vector< unsigned > perm)`
Permute rows of the matrix.
- `template<typename T >`
`Matrix< T > Mtx::permute_cols (const Matrix< T > &A, const std::vector< unsigned > perm)`
Permute columns of the matrix.
- `template<typename T >`
`Matrix< T > Mtx::permute_rows_and_cols (const Matrix< T > &A, const std::vector< unsigned > perm_rows, const std::vector< unsigned > perm_cols)`
Permute both rows and columns of the matrix.
- `template<typename T , bool transpose_first = false, bool transpose_second = false>`
`Matrix< T > Mtx::mult (const Matrix< T > &A, const Matrix< T > &B)`
Matrix multiplication.
- `template<typename T , bool transpose_first = false, bool transpose_second = false>`
`Matrix< T > Mtx::mult_hadamard (const Matrix< T > &A, const Matrix< T > &B)`
Matrix Hadamard (element-wise) multiplication.

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

- Matrix product with scalar.*

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

Matrix division by scalar.

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

Matrix product with scalar.

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

Matrix sum.

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

Matrix subtraction.

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

Matrix product.

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

Matrix Hadamard product.

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

Matrix sum with scalar.

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

Matrix subtraction with scalar.

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

Matrix product with scalar.

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

Matrix division by scalar.

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

Matrix equality check operator.

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

Matrix non-equality check operator.

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

Kronecker product.

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

Adjugate matrix.

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

Cofactor matrix.

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

Matrix determinant from on LU decomposition.

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

- Matrix determinant.*

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

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

LU decomposition with pivoting.
- `template<typename T >`
`Matrix< T > Mtx::inv_gauss_jordan (const Matrix< T > &A)`

Matrix inverse using Gauss-Jordan elimination.
- `template<typename T >`
`Matrix< T > Mtx::inv_tril (const Matrix< T > &A)`

Matrix inverse for lower triangular matrix.
- `template<typename T >`
`Matrix< T > Mtx::inv_triu (const Matrix< T > &A)`

Matrix inverse for upper triangular matrix.
- `template<typename T >`
`Matrix< T > Mtx::inv_posdef (const Matrix< T > &A)`

Matrix inverse for Hermitian positive-definite matrix.
- `template<typename T >`
`Matrix< T > Mtx::inv_square (const Matrix< T > &A)`

Matrix inverse for general square matrix.
- `template<typename T >`
`Matrix< T > Mtx::inv (const Matrix< T > &A)`

Matrix inverse (universal).
- `template<typename T >`
`Matrix< T > Mtx::pinv (const Matrix< T > &A)`

Moore-Penrose pseudo-inverse.
- `template<typename T >`
`T Mtx::trace (const Matrix< T > &A)`

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

Condition number of a matrix.
- `template<typename T , bool is_upper = false>`
`Matrix< T > Mtx::chol (const Matrix< T > &A)`

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

Inverse of Cholesky decomposition.
- `template<typename T >`
`LDL_result< T > Mtx::ldl (const Matrix< T > &A)`

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

Reduced QR decomposition based on Gram-Schmidt method.
- `template<typename T >`
`Matrix< T > Mtx::householder_reflection (const Matrix< T > &a)`

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

QR decomposition based on Householder method.

- `template<typename T>`
`QR_result< T> Mtx::qr (const Matrix< T> &A, bool calculate_Q=true)`
QR decomposition.
- `template<typename T>`
`Hessenberg_result< T> Mtx::hessenberg (const Matrix< T> &A, bool calculate_Q=true)`
Hessenberg decomposition.
- `template<typename T>`
`std::complex< T> Mtx::wilkinson_shift (const Matrix< std::complex< T> > &H, T tol=1e-10)`
Wilkinson's shift for complex eigenvalues.
- `template<typename T>`
`Eigenvalues_result< T> Mtx::eigenvalues (const Matrix< std::complex< T> > &A, T tol=1e-12, unsigned max_iter=100)`
Matrix eigenvalues of complex matrix.
- `template<typename T>`
`Eigenvalues_result< T> Mtx::eigenvalues (const Matrix< T> &A, T tol=1e-12, unsigned max_iter=100)`
Matrix eigenvalues of real matrix.
- `template<typename T>`
`Matrix< T> Mtx::solve_triu (const Matrix< T> &U, const Matrix< T> &B)`
Solves the upper triangular system.
- `template<typename T>`
`Matrix< T> Mtx::solve_tril (const Matrix< T> &L, const Matrix< T> &B)`
Solves the lower triangular system.
- `template<typename T>`
`Matrix< T> Mtx::solve_square (const Matrix< T> &A, const Matrix< T> &B)`
Solves the square system.
- `template<typename T>`
`Matrix< T> Mtx::solve_posdef (const Matrix< T> &A, const Matrix< T> &B)`
Solves the positive definite (Hermitian) system.

6.1.1 Function Documentation

6.1.1.1 add() [1/2]

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

Matrix addition.

Performs addition of two matrices.

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

Template Parameters

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

Parameters

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

Returns

output matrix of size $N \times M$

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

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

6.1.1.2 add() [2/2]

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

Addition of scalar to matrix.

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

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

6.1.1.3 adj()

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

Adjugate matrix.

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

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

Exceptions

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

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

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

6.1.1.4 cconj()

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



```
T Mtx::cconj (
    T x ) [inline]
```

Complex conjugate helper.

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

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

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

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

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

6.1.1.5 chol()

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

Cholesky decomposition.

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

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

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

Template Parameters

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

Exceptions

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

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

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

6.1.1.6 cholinv()

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

Inverse of Cholesky decomposition.

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

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

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

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

Exceptions

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

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

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

6.1.1.7 circshift()

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

Circular shift.

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

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

Parameters

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

Returns

matrix inverse

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

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

6.1.1.8 circulant() [1/2]

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

Circulant matrix from std::vector.

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

Parameters

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

Returns

circulant matrix

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

6.1.1.9 circulant() [2/2]

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

Circulant matrix from array.

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

Parameters

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

Returns

circulant matrix

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

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

6.1.1.10 cofactor()

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

Cofactor matrix.

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

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

Parameters

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

Exceptions

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

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

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

6.1.1.11 concatenate_horizontal()

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

Horizontal matrix concatenation.

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

Exceptions

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

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

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

6.1.1.12 concatenate_vertical()

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

Vertical matrix concatenation.

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

Exceptions

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

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

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

6.1.1.13 cond()

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

Condition number of a matrix.

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

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

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

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

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

6.1.1.14 creal()

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

Complex real part helper.

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

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

For complex numbers, this function returns the real part.

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

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

6.1.1.15 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 $e^{i \cdot \text{arg}(x)}$.

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

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

6.1.1.16 ctranspose()

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

Transpose a complex matrix.

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

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

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

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

6.1.1.17 det()

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

Matrix determinant.

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

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

Exceptions

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

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

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

6.1.1.18 det_lu()

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

Matrix determinant from on LU decomposition.

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

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

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

Exceptions

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

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

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

6.1.1.19 diag() [1/3]

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

Diagonal extraction.

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

Parameters

<i>A</i>	square matrix
----------	---------------

Returns

vector of diagonal elements

Exceptions

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

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

6.1.1.20 diag() [2/3]

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

Diagonal matrix from `std::vector`.

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

Parameters

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

Returns

diagonal matrix

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

6.1.1.21 diag() [3/3]

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

Diagonal matrix from array.

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

Parameters

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

Returns

diagonal matrix

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

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

6.1.1.22 div()

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

Division of matrix by scalar.

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

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

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

6.1.1.23 eigenvalues() [1/2]

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

Matrix eigenvalues of complex matrix.

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

Parameters

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

Returns

structure containing the result and status of eigenvalue calculation

Exceptions

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

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

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

6.1.1.24 eigenvalues() [2/2]

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

Matrix eigenvalues of real matrix.

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

Parameters

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

Returns

structure containing the result and status of eigenvalue calculation

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

6.1.1.25 eye()

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

Identity matrix.

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

Parameters

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

Returns

zeros matrix

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

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

6.1.1.26 foreach_elem()

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

Applies custom function element-wise in-place.

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

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

Parameters

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

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

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

6.1.1.27 foreach_elem_copy()

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

Applies custom function element-wise with matrix copy.

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

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

Parameters

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

Returns

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

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

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

6.1.1.28 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

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

Returns

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

Exceptions

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

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

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

6.1.1.29 householder_reflection()

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

Generate Householder reflection.

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

Parameters

<i>a</i>	column vector of size $N \times 1$
----------	------------------------------------

Returns

column vector with Householder reflection of *a*

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

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

6.1.1.30 imag()

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

Get imaginary part of complex matrix.

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

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

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

6.1.1.31 inv()

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

Matrix inverse (universal).

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

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

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

Exceptions

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

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

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

6.1.1.32 inv_gauss_jordan()

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

Matrix inverse using Gauss-Jordan elimination.

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

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

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

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

Exceptions

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

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

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

6.1.1.33 inv_posdef()

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

Matrix inverse for Hermitian positive-definite matrix.

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

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

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

Exceptions

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

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

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

6.1.1.34 inv_square()

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

Matrix inverse for general square matrix.

Calculates an inverse of square matrix using matrix.

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

Exceptions

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

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

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

6.1.1.35 inv_tril()

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

Matrix inverse for lower triangular matrix.

Calculates an inverse of lower triangular matrix.

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

Exceptions

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

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

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

6.1.1.36 inv_triu()

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

Matrix inverse for upper triangular matrix.

Calculates an inverse of upper triangular matrix.

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

Exceptions

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

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

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

6.1.1.37 ishess()

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

Hessenberg matrix check.

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

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

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

6.1.1.38 istril()

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

Lower triangular matrix check.

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

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

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

6.1.1.39 istriu()

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

Lower triangular matrix check.

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

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

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

6.1.1.40 kron()

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

Kronecker product.

Form the Kronecker product of two matrices. Kronecker product is defined block by block as $C = [A(i, j) \cdot B]$.
More information: https://en.wikipedia.org/wiki/Kronecker_product

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

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

6.1.1.41 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

A	input positive-definite matrix to be decomposed
-----	---

Returns

structure encapsulating calculated L and D

Exceptions

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

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

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

6.1.1.42 lu()

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

LU decomposition.

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

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

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

Parameters

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

Returns

structure containing calculated L and U matrices

References `Mtx::Matrix< T >::cols()`, `Mtx::lu()`, `Mtx::Matrix< T >::numel()`, and `Mtx::Matrix< T >::rows()`.

Referenced by `Mtx::lu()`.

6.1.1.43 lup()

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

LU decomposition with pivoting.

Performs LU factorization with partial pivoting, employing column permutations.

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

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

More information: [https://en.wikipedia.org/wiki/LU_decomposition#LU_factorization↵_with_partial_pivoting](https://en.wikipedia.org/wiki/LU_decomposition#LU_factorization_with_partial_pivoting)

Parameters

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

Returns

structure containing L , U and P .

References `Mtx::Matrix< T >::cols()`, `Mtx::lup()`, `Mtx::Matrix< T >::numel()`, `Mtx::Matrix< T >::resize()`, `Mtx::Matrix< T >::rows()`, and `Mtx::Matrix< T >::swap_cols()`.

Referenced by `Mtx::det_lu()`, `Mtx::inv_square()`, `Mtx::lup()`, and `Mtx::solve_square()`.

6.1.1.44 make_complex() [1/2]

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

Create complex matrix from real matrix.

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

Parameters

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

Returns

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

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

6.1.1.45 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

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

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

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

6.1.1.46 mult() [1/4]

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

Matrix multiplication.

Performs multiplication of two matrices.

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

Template Parameters

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

Parameters

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

Returns

output matrix of size $N \times K$

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

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

6.1.1.47 mult() [2/4]

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

Multiplication of matrix by `std::vector`.

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

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

Template Parameters

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

Parameters

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

Returns

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

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

6.1.1.48 mult() [3/4]

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

Multiplication of matrix by scalar.

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

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

6.1.1.49 mult() [4/4]

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

Multiplication of std::vector by matrix.

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

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

Template Parameters

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

Parameters

v	std::vector of size N
A	input matrix of size $N \times M$

Returns

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

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

6.1.1.50 mult_hadamard()

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

Matrix Hadamard (element-wise) multiplication.

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

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

Template Parameters

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

Parameters

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

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

6.1.1.51 norm_fro()

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

Frobenius norm.

Calculates Frobenius norm of a matrix.

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

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

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

6.1.1.52 norm_inf()

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

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

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

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

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

6.1.1.53 norm_p1()

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

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

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

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

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

6.1.1.54 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\(\)](#).

6.1.1.55 ones() [2/2]

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

Matrix of ones.

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

Parameters

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

Returns

ones matrix

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

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

6.1.1.56 operator!=(())

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

Matrix non-equality check operator.

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

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

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

6.1.1.57 operator*() [1/5]

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

Matrix product.

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

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

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

6.1.1.58 operator*() [2/5]

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

Matrix and std::vector product.

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

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

6.1.1.59 operator*() [3/5]

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

Matrix product with scalar.

Multiplies each element of the matrix by a scalar s .

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

6.1.1.60 operator*() [4/5]

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

std::vector and matrix product.

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

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

6.1.1.61 operator*() [5/5]

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

Matrix product with scalar.

Multiplies each element of the matrix by a scalar s .

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

6.1.1.62 operator*=() [1/2]

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

Matrix product.

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

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

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

6.1.1.63 operator*=() [2/2]

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

Matrix product with scalar.

Multiplies each element of the matrix by a scalar s .

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

6.1.1.64 operator+() [1/3]

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

Matrix sum.

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

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

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

6.1.1.65 operator+() [2/3]

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

Matrix sum with scalar.

Adds a scalar s to each element of the matrix.

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

6.1.1.66 operator+() [3/3]

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

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

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

6.1.1.67 operator+=() [1/2]

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

Matrix sum.

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

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

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

6.1.1.68 operator+=() [2/2]

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

Matrix sum with scalar.

Adds a scalar s to each element of the matrix.

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

6.1.1.69 operator-() [1/2]

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

Matrix subtraction.

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

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

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

6.1.1.70 operator-() [2/2]

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

Matrix subtraction with scalar.

Subtracts a scalar s from each element of the matrix.

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

6.1.1.71 operator-=() [1/2]

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

Matrix subtraction.

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

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

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

6.1.1.72 operator-=() [2/2]

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

Matrix subtraction with scalar.

Subtracts a scalar s from each element of the matrix.

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

6.1.1.73 operator/()

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

Matrix division by scalar.

Divides each element of the matrix by a scalar *s*.

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

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

6.1.1.74 operator/=()

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

Matrix division by scalar.

Divides each element of the matrix by a scalar *s*.

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

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

6.1.1.75 operator<<()

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

Matrix ostream operator.

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

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

6.1.1.76 operator==()

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

Matrix equality check operator.

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

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

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

6.1.1.77 operator^()

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

Matrix Hadamard product.

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

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

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

6.1.1.78 operator^=()

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

Matrix Hadamard product.

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

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

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

6.1.1.79 permute_cols()

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

Permute columns of the matrix.

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

Parameters

A	input matrix
$perm$	permutation vector with column indices

Returns

output matrix created by column permutation of *A*

Exceptions

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

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

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

6.1.1.80 permute_rows()

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

Permute rows of the matrix.

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

Parameters

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

Returns

output matrix created by row permutation of *A*

Exceptions

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

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

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

6.1.1.81 permute_rows_and_cols()

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

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

Permute both rows and columns of the matrix.

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

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

Parameters

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

Returns

output matrix created by row and column permutation of *A*

Exceptions

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

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

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

6.1.1.82 pinv()

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

Moore-Penrose pseudo-inverse.

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

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

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

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

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

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

6.1.1.83 qr()

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

QR decomposition.

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

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

Parameters

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

Returns

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

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

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

6.1.1.84 qr_householder()

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

QR decomposition based on Householder method.

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

This function implements QR decomposition based on Householder reflections method.

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

Parameters

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

Returns

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

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

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

6.1.1.85 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

<code>singular_matrix_exception</code>	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_red_gs\(\)](#), and [Mtx::Matrix< T >::rows\(\)](#).

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

6.1.1.86 real()

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

Get real part of complex matrix.

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

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

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

6.1.1.87 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\(\)](#).

6.1.1.88 solve_posdef()

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

Solves the positive definite (Hermitian) system.

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

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

Parameters

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

Returns

solution matrix of size $N \times M$.

Exceptions

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

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

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

6.1.1.89 solve_square()

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

Solves the square system.

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

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

Parameters

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

Returns

solution matrix of size $N \times M$.

Exceptions

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

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

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

6.1.1.90 solve_tril()

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

Solves the lower triangular system.

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

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

Parameters

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

Returns

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

Exceptions

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

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

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

6.1.1.91 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\(\)](#).

6.1.1.92 subtract() [1/2]

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

Matrix subtraction.

Performs subtraction of two matrices.

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

Template Parameters

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

Parameters

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

Returns

output matrix of size $N \times M$

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

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

6.1.1.93 subtract() [2/2]

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

Subtraction of scalar from matrix.

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

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

6.1.1.94 trace()

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

Matrix trace.

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

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

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

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

6.1.1.95 transpose()

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

Transpose a matrix.

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

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

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

6.1.1.96 tril()

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

Extract triangular lower part.

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

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

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

6.1.1.97 triu()

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

Extract triangular upper part.

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

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

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

6.1.1.98 wilkinson_shift()

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

Wilkinson's shift for complex eigenvalues.

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

Exceptions

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

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

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

6.1.1.99 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

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

Returns

zeros matrix

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

6.1.1.100 zeros() [2/2]

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

Matrix of zeros.

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

Parameters

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

Returns

zeros matrix

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

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

6.2 matrix.hpp

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

```

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

```



```

00070 }
00071
00072 template<typename T, typename std::enable_if<is_complex<T>::value,int>::type = 0>
00073 inline T csign(T x) {
00074     auto x_arg = std::arg(x);
00075     T y(0, x_arg);
00076     return std::exp(y);
00077 }
00078
00079 template<typename T, typename std::enable_if<!is_complex<T>::value,int>::type = 0>
00080 inline T creal(std::complex<T> x) {
00081     return std::real(x);
00082 }
00083
00084 template<typename T, typename std::enable_if<!is_complex<T>::value,int>::type = 0>
00085 inline T creal(T x) {
00086     return x;
00087 }
00088
00089 class singular_matrix_exception : public std::domain_error {
00090 public:
00091     singular_matrix_exception(const std::string& message) : std::domain_error(message) {}
00092 };
00093
00094 template<typename T>
00095 struct LU_result {
00096     Matrix<T> L;
00097     Matrix<T> U;
00098 };
00099
00100 template<typename T>
00101 struct LUP_result {
00102     Matrix<T> L;
00103     Matrix<T> U;
00104     std::vector<unsigned> P;
00105 };
00106
00107 template<typename T>
00108 struct QR_result {
00109     Matrix<T> Q;
00110     Matrix<T> R;
00111 };
00112
00113 template<typename T>
00114 struct Hessenberg_result {
00115     Matrix<T> H;
00116     Matrix<T> Q;
00117 };
00118
00119 template<typename T>
00120 struct LDL_result {
00121     Matrix<T> L;
00122     std::vector<T> d;
00123 };
00124
00125 template<typename T>
00126 struct Eigenvalues_result {
00127     std::vector<std::complex<T>> eig;
00128     bool converged;
00129     T err;
00130 };
00131
00132 template<typename T>
00133 inline Matrix<T> zeros(unsigned nrows, unsigned ncols) {
00134     return Matrix<T>(static_cast<T>(0), nrows, ncols);
00135 }
00136
00137 template<typename T>
00138 inline Matrix<T> zeros(unsigned n) {
00139     return zeros<T>(n,n);
00140 }
00141
00142 template<typename T>
00143 inline Matrix<T> ones(unsigned nrows, unsigned ncols) {
00144     return Matrix<T>(static_cast<T>(1), nrows, ncols);
00145 }
00146
00147 template<typename T>

```

```

00253 inline Matrix<T> ones(unsigned n) {
00254     return ones<T>(n,n);
00255 }
00256
00264 template<typename T>
00265 Matrix<T> eye(unsigned n) {
00266     Matrix<T> A(static_cast<T>(0), n, n);
00267     for (unsigned i = 0; i < n; i++)
00268         A(i,i) = static_cast<T>(1);
00269     return A;
00270 }
00271
00279 template<typename T>
00280 Matrix<T> diag(const T* array, size_t n) {
00281     Matrix<T> A(static_cast<T>(0), n, n);
00282     for (unsigned i = 0; i < n; i++) {
00283         A(i,i) = array[i];
00284     }
00285     return A;
00286 }
00287
00295 template<typename T>
00296 inline Matrix<T> diag(const std::vector<T>& v) {
00297     return diag(v.data(), v.size());
00298 }
00299
00308 template<typename T>
00309 std::vector<T> diag(const Matrix<T>& A) {
00310     if (!A.issquare()) throw std::runtime_error("Input matrix is not square");
00311
00312     std::vector<T> v;
00313     v.resize(A.rows());
00314
00315     for (unsigned i = 0; i < A.rows(); i++)
00316         v[i] = A(i,i);
00317     return v;
00318 }
00319
00327 template<typename T>
00328 Matrix<T> circulant(const T* array, unsigned n) {
00329     Matrix<T> A(n, n);
00330     for (unsigned j = 0; j < n; j++)
00331         for (unsigned i = 0; i < n; i++)
00332             A((i+j) % n, j) = array[i];
00333     return A;
00334 }
00335
00346 template<typename T>
00347 Matrix<std::complex<T>> make_complex(const Matrix<T>& Re, const Matrix<T>& Im) {
00348     if (Re.rows() != Im.rows() || Re.cols() != Im.cols()) throw std::runtime_error("Size of input
matrices does not match");
00349
00350     Matrix<std::complex<T>> C(Re.rows(), Re.cols());
00351     for (unsigned n = 0; n < Re.numel(); n++) {
00352         C(n).real(Re(n));
00353         C(n).imag(Im(n));
00354     }
00355
00356     return C;
00357 }
00358
00365 template<typename T>
00366 Matrix<std::complex<T>> make_complex(const Matrix<T>& Re) {
00367     Matrix<std::complex<T>> C(Re.rows(), Re.cols());
00368
00369     for (unsigned n = 0; n < Re.numel(); n++) {
00370         C(n).real(Re(n));
00371         C(n).imag(static_cast<T>(0));
00372     }
00373
00374     return C;
00375 }
00376
00381 template<typename T>
00382 Matrix<T> real(const Matrix<std::complex<T>& C) {
00383     Matrix<T> Re(C.rows(), C.cols());
00384
00385     for (unsigned n = 0; n < C.numel(); n++)
00386         Re(n) = C(n).real();
00387
00388     return Re;
00389 }
00390
00395 template<typename T>
00396 Matrix<T> imag(const Matrix<std::complex<T>& C) {
00397     Matrix<T> Re(C.rows(), C.cols());
00398

```

```

00399     for (unsigned n = 0; n < C.numel(); n++)
00400         Re(n) = C(n).imag();
00401
00402     return Re;
00403 }
00404
00412 template<typename T>
00413 inline Matrix<T> circulant(const std::vector<T>& v) {
00414     return circulant(v.data(), v.size());
00415 }
00416
00421 template<typename T>
00422 inline Matrix<T> transpose(const Matrix<T>& A) {
00423     return A.transpose();
00424 }
00425
00431 template<typename T>
00432 inline Matrix<T> ctranspose(const Matrix<T>& A) {
00433     return A.ctranspose();
00434 }
00435
00446 template<typename T>
00447 Matrix<T> circshift(const Matrix<T>& A, int row_shift, int col_shift) {
00448     Matrix<T> B(A.rows(), A.cols());
00449     for (int i = 0; i < A.rows(); i++) {
00450         int ii = (i + row_shift) % A.rows();
00451         for (int j = 0; j < A.cols(); j++) {
00452             int jj = (j + col_shift) % A.cols();
00453             B(ii, jj) = A(i, j);
00454         }
00455     }
00456     return B;
00457 }
00458
00466 template<typename T>
00467 Matrix<T> repmat(const Matrix<T>& A, unsigned m, unsigned n) {
00468     Matrix<T> B(m * A.rows(), n * A.cols());
00469
00470     for (unsigned cb = 0; cb < n; cb++)
00471         for (unsigned rb = 0; rb < m; rb++)
00472             for (unsigned c = 0; c < A.cols(); c++)
00473                 for (unsigned r = 0; r < A.rows(); r++)
00474                     B(rb*A.rows() + r, cb*A.cols() + c) = A(r, c);
00475
00476     return B;
00477 }
00478
00485 template<typename T>
00486 Matrix<T> concatenate_horizontal(const Matrix<T>& A, const Matrix<T>& B) {
00487     if (A.rows() != B.rows()) throw std::runtime_error("Unmatching number of rows for horizontal
concatenation");
00488
00489     Matrix<T> C(A.rows(), A.cols() + B.cols());
00490
00491     for (unsigned c = 0; c < A.cols(); c++)
00492         for (unsigned r = 0; r < A.rows(); r++)
00493             C(r, c) = A(r, c);
00494
00495     for (unsigned c = 0; c < B.cols(); c++)
00496         for (unsigned r = 0; r < B.rows(); r++)
00497             C(r, c+A.cols()) = B(r, c);
00498
00499     return C;
00500 }
00501
00508 template<typename T>
00509 Matrix<T> concatenate_vertical(const Matrix<T>& A, const Matrix<T>& B) {
00510     if (A.cols() != B.cols()) throw std::runtime_error("Unmatching number of columns for vertical
concatenation");
00511
00512     Matrix<T> C(A.rows() + B.rows(), A.cols());
00513
00514     for (unsigned c = 0; c < A.cols(); c++)
00515         for (unsigned r = 0; r < A.rows(); r++)
00516             C(r, c) = A(r, c);
00517
00518     for (unsigned c = 0; c < B.cols(); c++)
00519         for (unsigned r = 0; r < B.rows(); r++)
00520             C(r+A.rows(), c) = B(r, c);
00521
00522     return C;
00523 }
00524
00530 template<typename T>
00531 double norm_fro(const Matrix<T>& A) {
00532     double sum = 0;
00533

```

```

00534     for (unsigned i = 0; i < A.numel(); i++)
00535         sum += creal(A(i) * cconj(A(i)));
00536
00537     return std::sqrt(sum);
00538 }
00539
00540 template<typename T>
00541 double norm_p1(const Matrix<T>& A) {
00542     double max_sum = 0.0;
00543
00544     for (unsigned c = 0; c < A.cols(); c++) {
00545         double sum = 0.0;
00546
00547         for (unsigned r = 0; r < A.rows(); r++)
00548             sum += std::abs(A(r,c));
00549
00550         if (sum > max_sum)
00551             max_sum = sum;
00552     }
00553
00554     return max_sum;
00555 }
00556
00557 template<typename T>
00558 double norm_inf(const Matrix<T>& A) {
00559     double max_sum = 0.0;
00560
00561     for (unsigned r = 0; r < A.rows(); r++) {
00562         double sum = 0.0;
00563
00564         for (unsigned c = 0; c < A.cols(); c++)
00565             sum += std::abs(A(r,c));
00566
00567         if (sum > max_sum)
00568             max_sum = sum;
00569     }
00570
00571     return max_sum;
00572 }
00573
00574 template<typename T>
00575 Matrix<T> tril(const Matrix<T>& A) {
00576     Matrix<T> B(A);
00577
00578     for (unsigned row = 0; row < B.rows(); row++)
00579         for (unsigned col = row+1; col < B.cols(); col++)
00580             B(row,col) = static_cast<T>(0);
00581
00582     return B;
00583 }
00584
00585 template<typename T>
00586 Matrix<T> triu(const Matrix<T>& A) {
00587     Matrix<T> B(A);
00588
00589     for (unsigned col = 0; col < B.cols(); col++)
00590         for (unsigned row = col+1; row < B.rows(); row++)
00591             B(row,col) = static_cast<T>(0);
00592
00593     return B;
00594 }
00595
00596 template<typename T>
00597 bool istril(const Matrix<T>& A) {
00598     for (unsigned row = 0; row < A.rows(); row++)
00599         for (unsigned col = row+1; col < A.cols(); col++)
00600             if (A(row,col) != static_cast<T>(0)) return false;
00601     return true;
00602 }
00603
00604 template<typename T>
00605 bool istriu(const Matrix<T>& A) {
00606     for (unsigned col = 0; col < A.cols(); col++)
00607         for (unsigned row = col+1; row < A.rows(); row++)
00608             if (A(row,col) != static_cast<T>(0)) return false;
00609     return true;
00610 }
00611
00612 template<typename T>
00613 bool ishess(const Matrix<T>& A) {
00614     if (!A.issquare())
00615         return false;
00616     for (unsigned row = 2; row < A.rows(); row++)
00617         for (unsigned col = 0; col < row-2; col++)
00618             if (A(row,col) != static_cast<T>(0)) return false;
00619     return true;
00620 }
00621 }

```

```

00656
00666 template<typename T>
00667 inline void foreach_elem(Matrix<T>& A, std::function<T(T)> func) {
00668     for (unsigned i = 0; i < A.numel(); i++)
00669         A(i) = func(A(i));
00670 }
00671
00682 template<typename T>
00683 inline Matrix<T> foreach_elem_copy(const Matrix<T>& A, std::function<T(T)> func) {
00684     Matrix<T> B(A);
00685     foreach_elem(B, func);
00686     return B;
00687 }
00688
00701 template<typename T>
00702 Matrix<T> permute_rows(const Matrix<T>& A, const std::vector<unsigned> perm) {
00703     if (perm.empty()) throw std::runtime_error("Permutation vector is empty");
00704
00705     for (unsigned p = 0; p < perm.size(); p++)
00706         if (!perm[p] < A.rows()) throw std::out_of_range("Index in permutation vector out of range");
00707
00708     Matrix<T> B(perm.size(), A.cols());
00709
00710     for (unsigned p = 0; p < perm.size(); p++)
00711         for (unsigned c = 0; c < A.cols(); c++)
00712             B(p,c) = A(perm[p],c);
00713
00714     return B;
00715 }
00716
00729 template<typename T>
00730 Matrix<T> permute_cols(const Matrix<T>& A, const std::vector<unsigned> perm) {
00731     if (perm.empty()) throw std::runtime_error("Permutation vector is empty");
00732
00733     for (unsigned p = 0; p < perm.size(); p++)
00734         if (!perm[p] < A.cols()) throw std::out_of_range("Index in permutation vector out of range");
00735
00736     Matrix<T> B(A.rows(), perm.size());
00737
00738     for (unsigned p = 0; p < perm.size(); p++)
00739         for (unsigned r = 0; r < A.rows(); r++)
00740             B(r,perm[p]) = A(r,p);
00741
00742     return B;
00743 }
00744
00759 template<typename T>
00760 Matrix<T> permute_rows_and_cols(const Matrix<T>& A, const std::vector<unsigned> perm_rows, const
std::vector<unsigned> perm_cols) {
00761     if (perm_rows.empty()) throw std::runtime_error("Row permutation vector is empty");
00762     if (perm_cols.empty()) throw std::runtime_error("Column permutation vector is empty");
00763
00764     for (unsigned pc = 0; pc < perm_cols.size(); pc++)
00765         if (!perm_cols[pc] < A.cols()) throw std::out_of_range("Column index in permutation vector out
of range");
00766
00767     for (unsigned pr = 0; pr < perm_rows.size(); pr++)
00768         if (!perm_rows[pr] < A.rows()) throw std::out_of_range("Row index in permutation vector out of
range");
00769
00770     Matrix<T> B(perm_rows.size(), perm_cols.size());
00771
00772     for (unsigned pc = 0; pc < perm_cols.size(); pc++)
00773         for (unsigned pr = 0; pr < perm_rows.size(); pr++)
00774             B(pr,pc) = A(perm_rows[pr],perm_cols[pc]);
00775
00776     return B;
00777 }
00778
00794 template<typename T, bool transpose_first = false, bool transpose_second = false>
00795 Matrix<T> mult(const Matrix<T>& A, const Matrix<T>& B) {
00796     // Adjust dimensions based on transpositions
00797     unsigned rows_A = transpose_first ? A.cols() : A.rows();
00798     unsigned cols_A = transpose_first ? A.rows() : A.cols();
00799     unsigned rows_B = transpose_second ? B.cols() : B.rows();
00800     unsigned cols_B = transpose_second ? B.rows() : B.cols();
00801
00802     if (cols_A != rows_B) throw std::runtime_error("Unmatching matrix dimensions for mult");
00803
00804     Matrix<T> C(static_cast<T>(0), rows_A, cols_B);
00805
00806     for (unsigned i = 0; i < rows_A; i++)
00807         for (unsigned j = 0; j < cols_B; j++)
00808             for (unsigned k = 0; k < cols_A; k++)
00809                 C(i,j) += (transpose_first ? cconj(A(k,i)) : A(i,k)) *
00810                     (transpose_second ? cconj(B(j,k)) : B(k,j));
00811

```

```

00812     return C;
00813 }
00814
00830 template<typename T, bool transpose_first = false, bool transpose_second = false>
00831 Matrix<T> mult_hadamard(const Matrix<T>& A, const Matrix<T>& B) {
00832     // Adjust dimensions based on transpositions
00833     unsigned rows_A = transpose_first ? A.cols() : A.rows();
00834     unsigned cols_A = transpose_first ? A.rows() : A.cols();
00835     unsigned rows_B = transpose_second ? B.cols() : B.rows();
00836     unsigned cols_B = transpose_second ? B.rows() : B.cols();
00837
00838     if ((rows_A != rows_B) || (cols_A != cols_B)) throw std::runtime_error("Unmatching matrix dimensions
for mult_hadamard");
00839
00840     Matrix<T> C(static_cast<T>(0), rows_A, cols_A);
00841
00842     for (unsigned i = 0; i < rows_A; i++)
00843         for (unsigned j = 0; j < cols_A; j++)
00844             C(i,j) += (transpose_first ? cconj(A(j,i)) : A(i,j)) *
00845                       (transpose_second ? cconj(B(j,i)) : B(i,j));
00846
00847     return C;
00848 }
00849
00865 template<typename T, bool transpose_first = false, bool transpose_second = false>
00866 Matrix<T> add(const Matrix<T>& A, const Matrix<T>& B) {
00867     // Adjust dimensions based on transpositions
00868     unsigned rows_A = transpose_first ? A.cols() : A.rows();
00869     unsigned cols_A = transpose_first ? A.rows() : A.cols();
00870     unsigned rows_B = transpose_second ? B.cols() : B.rows();
00871     unsigned cols_B = transpose_second ? B.rows() : B.cols();
00872
00873     if ((rows_A != rows_B) || (cols_A != cols_B)) throw std::runtime_error("Unmatching matrix dimensions
for add");
00874
00875     Matrix<T> C(static_cast<T>(0), rows_A, cols_A);
00876
00877     for (unsigned i = 0; i < rows_A; i++)
00878         for (unsigned j = 0; j < cols_A; j++)
00879             C(i,j) += (transpose_first ? cconj(A(j,i)) : A(i,j)) +
00880                       (transpose_second ? cconj(B(j,i)) : B(i,j));
00881
00882     return C;
00883 }
00884
00900 template<typename T, bool transpose_first = false, bool transpose_second = false>
00901 Matrix<T> subtract(const Matrix<T>& A, const Matrix<T>& B) {
00902     // Adjust dimensions based on transpositions
00903     unsigned rows_A = transpose_first ? A.cols() : A.rows();
00904     unsigned cols_A = transpose_first ? A.rows() : A.cols();
00905     unsigned rows_B = transpose_second ? B.cols() : B.rows();
00906     unsigned cols_B = transpose_second ? B.rows() : B.cols();
00907
00908     if ((rows_A != rows_B) || (cols_A != cols_B)) throw std::runtime_error("Unmatching matrix dimensions
for subtract");
00909
00910     Matrix<T> C(static_cast<T>(0), rows_A, cols_A);
00911
00912     for (unsigned i = 0; i < rows_A; i++)
00913         for (unsigned j = 0; j < cols_A; j++)
00914             C(i,j) += (transpose_first ? cconj(A(j,i)) : A(i,j)) -
00915                       (transpose_second ? cconj(B(j,i)) : B(i,j));
00916
00917     return C;
00918 }
00919
00935 template<typename T, bool transpose_matrix = false>
00936 std::vector<T> mult(const Matrix<T>& A, const std::vector<T>& v) {
00937     // Adjust dimensions based on transpositions
00938     unsigned rows_A = transpose_matrix ? A.cols() : A.rows();
00939     unsigned cols_A = transpose_matrix ? A.rows() : A.cols();
00940
00941     if (cols_A != v.size()) throw std::runtime_error("Unmatching matrix dimensions for mult");
00942
00943     std::vector<T> u(rows_A, static_cast<T>(0));
00944     for (unsigned r = 0; r < rows_A; r++)
00945         for (unsigned c = 0; c < cols_A; c++)
00946             u[r] += v[c] * (transpose_matrix ? cconj(A(c,r)) : A(r,c));
00947
00948     return u;
00949 }
00950
00966 template<typename T, bool transpose_matrix = false>
00967 std::vector<T> mult(const std::vector<T>& v, const Matrix<T>& A) {
00968     // Adjust dimensions based on transpositions
00969     unsigned rows_A = transpose_matrix ? A.cols() : A.rows();
00970     unsigned cols_A = transpose_matrix ? A.rows() : A.cols();

```

```

00971
00972     if (rows_A != v.size()) throw std::runtime_error("Unmatching matrix dimensions for mult");
00973
00974     std::vector<T> u(cols_A, static_cast<T>(0));
00975     for (unsigned c = 0; c < cols_A; c++)
00976         for (unsigned r = 0; r < rows_A; r++)
00977             u[c] += v[r] * (transpose_matrix ? cconj(A(c,r)) : A(r,c));
00978
00979     return u;
00980 }
00981
00982 template<typename T>
00983 Matrix<T> add(const Matrix<T>& A, T s) {
00984     Matrix<T> B(A.rows(), A.cols());
00985     for (unsigned i = 0; i < A.numel(); i++)
00986         B(i) = A(i) + s;
00987     return B;
00988 }
00989
00990 template<typename T>
01001 Matrix<T> subtract(const Matrix<T>& A, T s) {
01002     Matrix<T> B(A.rows(), A.cols());
01003     for (unsigned i = 0; i < A.numel(); i++)
01004         B(i) = A(i) - s;
01005     return B;
01006 }
01007
01008 template<typename T>
01009 Matrix<T> mult(const Matrix<T>& A, T s) {
01010     Matrix<T> B(A.rows(), A.cols());
01011     for (unsigned i = 0; i < A.numel(); i++)
01012         B(i) = A(i) * s;
01013     return B;
01014 }
01015
01016 template<typename T>
01017 Matrix<T> div(const Matrix<T>& A, T s) {
01018     Matrix<T> B(A.rows(), A.cols());
01019     for (unsigned i = 0; i < A.numel(); i++)
01020         B(i) = A(i) / s;
01021     return B;
01022 }
01023
01024 template<typename T>
01025 std::ostream& operator<<(std::ostream& os, const Matrix<T>& A) {
01026     for (unsigned row = 0; row < A.rows(); row++) {
01027         for (unsigned col = 0; col < A.cols(); col++)
01028             os << A(row,col) << " ";
01029         if (row < static_cast<unsigned>(A.rows()-1)) os << std::endl;
01030     }
01031     return os;
01032 }
01033
01034 template<typename T>
01035 inline Matrix<T> operator+(const Matrix<T>& A, const Matrix<T>& B) {
01036     return add(A,B);
01037 }
01038
01039 template<typename T>
01040 inline Matrix<T> operator-(const Matrix<T>& A, const Matrix<T>& B) {
01041     return subtract(A,B);
01042 }
01043
01044 template<typename T>
01045 inline Matrix<T> operator^(const Matrix<T>& A, const Matrix<T>& B) {
01046     return mult_hadamard(A,B);
01047 }
01048
01049 template<typename T>
01050 inline Matrix<T> operator*(const Matrix<T>& A, const Matrix<T>& B) {
01051     return mult(A,B);
01052 }
01053
01054 template<typename T>
01055 inline std::vector<T> operator*(const Matrix<T>& A, const std::vector<T>& v) {
01056     return mult(A,v);
01057 }
01058
01059 template<typename T>
01060 inline std::vector<T> operator*(const std::vector<T>& v, const Matrix<T>& A) {
01061     return mult(v,A);
01062 }
01063
01064 template<typename T>
01065 inline Matrix<T> operator+(const Matrix<T>& A, T s) {
01066     return add(A,s);
01067 }
01068
01069
01070
01071
01072
01073
01074
01075
01076
01077
01078
01079
01080
01081
01082
01083
01084
01085
01086
01087
01088
01089
01090
01091
01092
01093
01094
01095
01096
01097
01098
01099
01100
01101
01102
01103
01104
01105
01106
01107
01108
01109
01110
01111

```

```

01112
01117 template<typename T>
01118 inline Matrix<T> operator-(const Matrix<T>& A, T s) {
01119     return subtract(A,s);
01120 }
01121
01126 template<typename T>
01127 inline Matrix<T> operator*(const Matrix<T>& A, T s) {
01128     return mult(A,s);
01129 }
01130
01135 template<typename T>
01136 inline Matrix<T> operator/(const Matrix<T>& A, T s) {
01137     return div(A,s);
01138 }
01139
01143 template<typename T>
01144 inline Matrix<T> operator+(T s, const Matrix<T>& A) {
01145     return add(A,s);
01146 }
01147
01152 template<typename T>
01153 inline Matrix<T> operator*(T s, const Matrix<T>& A) {
01154     return mult(A,s);
01155 }
01156
01161 template<typename T>
01162 inline Matrix<T>& operator+=(Matrix<T>& A, const Matrix<T>& B) {
01163     return A.add(B);
01164 }
01165
01170 template<typename T>
01171 inline Matrix<T>& operator-=(Matrix<T>& A, const Matrix<T>& B) {
01172     return A.subtract(B);
01173 }
01174
01179 template<typename T>
01180 inline Matrix<T>& operator*=(Matrix<T>& A, const Matrix<T>& B) {
01181     A = mult(A,B);
01182     return A;
01183 }
01184
01190 template<typename T>
01191 inline Matrix<T>& operator^=(Matrix<T>& A, const Matrix<T>& B) {
01192     return A.mult_hadamard(B);
01193 }
01194
01199 template<typename T>
01200 inline Matrix<T>& operator+=(Matrix<T>& A, T s) {
01201     return A.add(s);
01202 }
01203
01208 template<typename T>
01209 inline Matrix<T>& operator-=(Matrix<T>& A, T s) {
01210     return A.subtract(s);
01211 }
01212
01217 template<typename T>
01218 inline Matrix<T>& operator*=(Matrix<T>& A, T s) {
01219     return A.mult(s);
01220 }
01221
01226 template<typename T>
01227 inline Matrix<T>& operator/=(Matrix<T>& A, T s) {
01228     return A.div(s);
01229 }
01230
01235 template<typename T>
01236 inline bool operator==(const Matrix<T>& A, const Matrix<T>& b) {
01237     return A.isequal(b);
01238 }
01239
01244 template<typename T>
01245 inline bool operator!=(const Matrix<T>& A, const Matrix<T>& b) {
01246     return !(A.isequal(b));
01247 }
01248
01255 template<typename T>
01256 Matrix<T> kron(const Matrix<T>& A, const Matrix<T>& B) {
01257     const unsigned rows_A = A.rows();
01258     const unsigned cols_A = A.cols();
01259     const unsigned rows_B = B.rows();
01260     const unsigned cols_B = B.cols();
01261
01262     unsigned rows_C = rows_A * rows_B;
01263     unsigned cols_C = cols_A * cols_B;
01264

```



```

01265     Matrix<T> C(rows_C, cols_C);
01266
01267     for (unsigned i = 0; i < rows_A; i++)
01268         for (unsigned j = 0; j < cols_A; j++)
01269             for (unsigned k = 0; k < rows_B; k++)
01270                 for (unsigned l = 0; l < cols_B; l++)
01271                     C(i+rows_B + k, j*cols_B + l) = A(i, j) * B(k, l);
01272
01273     return C;
01274 }
01275
01283 template<typename T>
01284 Matrix<T> adj(const Matrix<T>& A) {
01285     if (! A.issquare()) throw std::runtime_error("Input matrix is not square");
01286
01287     Matrix<T> B(A.rows(), A.cols());
01288     if (A.rows() == 1) {
01289         B(0) = static_cast<T>(1.0);
01290     } else {
01291         for (unsigned i = 0; i < A.rows(); i++) {
01292             for (unsigned j = 0; j < A.cols(); j++) {
01293                 T sgn = static_cast<T>(1.0) (((i + j) % 2 == 0) ? (1.0) : (-1.0));
01294                 B(j, i) = sgn * det(cofactor(A, i, j));
01295             }
01296         }
01297     }
01298     return B;
01299 }
01300
01314 template<typename T>
01315 Matrix<T> cofactor(const Matrix<T>& A, unsigned p, unsigned q) {
01316     if (! A.issquare()) throw std::runtime_error("Input matrix is not square");
01317     if (! (p < A.rows())) throw std::out_of_range("Row index out of range");
01318     if (! (q < A.cols())) throw std::out_of_range("Column index out of range");
01319     if (A.cols() < 2) throw std::runtime_error("Cofactor calculation requested for matrix with less than
2 rows");
01320
01321     Matrix<T> c(A.rows()-1, A.cols()-1);
01322     unsigned i = 0;
01323     unsigned j = 0;
01324
01325     for (unsigned row = 0; row < A.rows(); row++) {
01326         if (row != p) {
01327             for (unsigned col = 0; col < A.cols(); col++)
01328                 if (col != q) c(i, j++) = A(row, col);
01329             j = 0;
01330             i++;
01331         }
01332     }
01333
01334     return c;
01335 }
01336
01348 template<typename T>
01349 T det_lu(const Matrix<T>& A) {
01350     if (! A.issquare()) throw std::runtime_error("Input matrix is not square");
01351
01352     // LU decomposition with pivoting
01353     auto res = lup(A);
01354
01355     // Determinants of LU
01356     T detLU = static_cast<T>(1);
01357
01358     for (unsigned i = 0; i < res.L.rows(); i++)
01359         detLU *= res.L(i, i) * res.U(i, i);
01360
01361     // Determinant of P
01362     unsigned len = res.P.size();
01363     T detP = static_cast<T>(1);
01364
01365     std::vector<unsigned> p(res.P);
01366     std::vector<unsigned> q;
01367     q.resize(len);
01368
01369     for (unsigned i = 0; i < len; i++)
01370         q[p[i]] = i;
01371
01372     for (unsigned i = 0; i < len; i++) {
01373         unsigned j = p[i];
01374         unsigned k = q[i];
01375         if (j != i) {
01376             p[k] = p[i];
01377             q[j] = q[i];
01378             detP = - detP;
01379         }
01380     }
01381

```

```

01382     return detLU * detP;
01383 }
01384
01394 template<typename T>
01395 T det(const Matrix<T>& A) {
01396     if (! A.issquare()) throw std::runtime_error("Input matrix is not square");
01397
01398     if (A.rows() == 1)
01399         return A(0,0);
01400     else if (A.rows() == 2)
01401         return A(0,0)*A(1,1) - A(0,1)*A(1,0);
01402     else if (A.rows() == 3)
01403         return A(0,0)*(A(1,1)*A(2,2) - A(1,2)*A(2,1)) -
01404             A(0,1)*(A(1,0)*A(2,2) - A(1,2)*A(2,0)) +
01405             A(0,2)*(A(1,0)*A(2,1) - A(1,1)*A(2,0));
01406     else
01407         return det_lu(A);
01408 }
01409
01419 template<typename T>
01420 LU_result<T> lu(const Matrix<T>& A) {
01421     const unsigned M = A.rows();
01422     const unsigned N = A.cols();
01423
01424     LU_result<T> res;
01425     res.L = eye<T>(M);
01426     res.U = Matrix<T>(A);
01427
01428     // aliases
01429     auto& L = res.L;
01430     auto& U = res.U;
01431
01432     if (A.numel() == 0)
01433         return res;
01434
01435     for (unsigned k = 0; k < M-1; k++) {
01436         for (unsigned i = k+1; i < M; i++) {
01437             L(i,k) = U(i,k) / U(k,k);
01438             for (unsigned l = k+1; l < N; l++) {
01439                 U(i,l) -= L(i,k) * U(k,l);
01440             }
01441         }
01442     }
01443
01444     for (unsigned col = 0; col < N; col++)
01445         for (unsigned row = col+1; row < M; row++)
01446             U(row,col) = 0;
01447
01448     return res;
01449 }
01450
01464 template<typename T>
01465 LUP_result<T> lup(const Matrix<T>& A) {
01466     const unsigned M = A.rows();
01467     const unsigned N = A.cols();
01468
01469     // Initialize L, U, and PP
01470     LUP_result<T> res;
01471
01472     if (A.numel() == 0)
01473         return res;
01474
01475     res.L = eye<T>(M);
01476     res.U = Matrix<T>(A);
01477     std::vector<unsigned> PP;
01478
01479     // aliases
01480     auto& L = res.L;
01481     auto& U = res.U;
01482
01483     PP.resize(N);
01484     for (unsigned i = 0; i < N; i++)
01485         PP[i] = i;
01486
01487     for (unsigned k = 0; k < M-1; k++) {
01488         // Find the column with the largest absolute value in the current row
01489         auto max_col_value = std::abs(U(k,k));
01490         unsigned max_col_index = k;
01491         for (unsigned l = k+1; l < N; l++) {
01492             auto val = std::abs(U(k,l));
01493             if (val > max_col_value) {
01494                 max_col_value = val;
01495                 max_col_index = l;
01496             }
01497         }
01498
01499         // Swap columns k and max_col_index in U and update P

```

```

01500     if (max_col_index != k) {
01501         U.swap_cols(k, max_col_index); // TODO: This could be reworked to avoid column swap in U during
every iteration by:
01502                                     // 1. using PP[k] for column indexing across iterations
01503                                     // 2. doing just one permutation of U at the end
01504         std::swap(PP[k], PP[max_col_index]);
01505     }
01506
01507     // Update L and U
01508     for (unsigned i = k+1; i < M; i++) {
01509         L(i,k) = U(i,k) / U(k,k);
01510         for (unsigned l = k+1; l < N; l++) {
01511             U(i,l) -= L(i,k) * U(k,l);
01512         }
01513     }
01514 }
01515
01516 // Set elements in lower triangular part of U to zero
01517 for (unsigned col = 0; col < N; col++)
01518     for (unsigned row = col+1; row < M; row++)
01519         U(row,col) = 0;
01520
01521 // Transpose indices in permutation vector
01522 res.P.resize(N);
01523 for (unsigned i = 0; i < N; i++)
01524     res.P[PP[i]] = i;
01525
01526 return res;
01527 }
01528
01529 template<typename T>
01530 Matrix<T> inv_gauss_jordan(const Matrix<T>& A) {
01541     if (! A.issquare()) throw std::runtime_error("Input matrix is not square");
01542
01543     const unsigned N = A.rows();
01544     Matrix<T> AA(A);
01545     auto IA = eye<T>(N);
01546
01547     bool found_nonzero;
01548     for (unsigned j = 0; j < N; j++) {
01549         found_nonzero = false;
01550         for (unsigned i = j; i < N; i++) {
01551             if (AA(i,j) != static_cast<T>(0)) {
01552                 found_nonzero = true;
01553                 for (unsigned k = 0; k < N; k++) {
01554                     std::swap(AA(j,k), AA(i,k));
01555                     std::swap(IA(j,k), IA(i,k));
01556                 }
01557                 if (AA(j,j) != static_cast<T>(1)) {
01558                     T s = static_cast<T>(1) / AA(j,j);
01559                     for (unsigned k = 0; k < N; k++) {
01560                         AA(j,k) *= s;
01561                         IA(j,k) *= s;
01562                     }
01563                 }
01564                 for (unsigned l = 0; l < N; l++) {
01565                     if (l != j) {
01566                         T s = AA(l,j);
01567                         for (unsigned k = 0; k < N; k++) {
01568                             AA(l,k) -= s * AA(j,k);
01569                             IA(l,k) -= s * IA(j,k);
01570                         }
01571                     }
01572                 }
01573             }
01574             break;
01575         }
01576         // if a row full of zeros is found, the input matrix was singular
01577         if (!found_nonzero) throw singular_matrix_exception("Singular matrix in inv_gauss_jordan");
01578     }
01579     return IA;
01580 }
01581
01592 template<typename T>
01593 Matrix<T> inv_tril(const Matrix<T>& A) {
01594     if (! A.issquare()) throw std::runtime_error("Input matrix is not square");
01595
01596     const unsigned N = A.rows();
01597     auto IA = zeros<T>(N);
01598
01599     for (unsigned i = 0; i < N; i++) {
01600         if (A(i,i) == static_cast<T>(0.0)) throw singular_matrix_exception("Division by zero in
inv_tril");
01601         IA(i,i) = static_cast<T>(1.0) / A(i,i);
01602         for (unsigned j = 0; j < i; j++) {

```

```

01605         T s = 0.0;
01606         for (unsigned k = j; k < i; k++)
01607             s += A(i,k) * IA(k,j);
01608         IA(i,j) = -s * IA(i,i);
01609     }
01610 }
01611
01612 return IA;
01613 }
01614
01615 template<typename T>
01616 Matrix<T> inv_triu(const Matrix<T>& A) {
01617     if (!A.issquare()) throw std::runtime_error("Input matrix is not square");
01618     const unsigned N = A.rows();
01619     auto IA = zeros<T>(N);
01620     for (int i = N - 1; i >= 0; i--) {
01621         if (A(i,i) == static_cast<T>(0.0)) throw singular_matrix_exception("Division by zero in
01622 inv_triu");
01623         IA(i, i) = static_cast<T>(1.0) / A(i,i);
01624         for (int j = N - 1; j > i; j--) {
01625             T s = static_cast<T>(0.0);
01626             for (int k = i + 1; k <= j; k++)
01627                 s += A(i,k) * IA(k,j);
01628             IA(i,j) = -s * IA(i,i);
01629         }
01630     }
01631     return IA;
01632 }
01633
01634 template<typename T>
01635 Matrix<T> inv_posdef(const Matrix<T>& A) {
01636     auto L = cholinv(A);
01637     return mult<T,true,false>(L,L);
01638 }
01639
01640 template<typename T>
01641 Matrix<T> inv_square(const Matrix<T>& A) {
01642     if (!A.issquare()) throw std::runtime_error("Input matrix is not square");
01643     // LU decomposition with pivoting
01644     auto LU = lup(A);
01645     auto IL = inv_tril(LU.L);
01646     auto IU = inv_triu(LU.U);
01647     return permute_rows(IU * IL, LU.P);
01648 }
01649
01650 template<typename T>
01651 Matrix<T> inv(const Matrix<T>& A) {
01652     if (!A.issquare()) throw std::runtime_error("Input matrix is not square");
01653     if (A.numel() == 0) {
01654         return Matrix<T>();
01655     } else if (A.rows() < 4) {
01656         T d = det(A);
01657         if (d == static_cast<T>(0.0)) throw singular_matrix_exception("Singular matrix in inv");
01658         Matrix<T> IA(A.rows(), A.rows());
01659         T invdet = static_cast<T>(1.0) / d;
01660         if (A.rows() == 1) {
01661             IA(0,0) = invdet;
01662         } else if (A.rows() == 2) {
01663             IA(0,0) = A(1,1) * invdet;
01664             IA(0,1) = -A(0,1) * invdet;
01665             IA(1,0) = -A(1,0) * invdet;
01666             IA(1,1) = A(0,0) * invdet;
01667         } else if (A.rows() == 3) {
01668             IA(0,0) = (A(1,1)*A(2,2) - A(2,1)*A(1,2)) * invdet;
01669             IA(0,1) = (A(0,2)*A(2,1) - A(0,1)*A(2,2)) * invdet;
01670             IA(0,2) = (A(0,1)*A(1,2) - A(0,2)*A(1,1)) * invdet;
01671             IA(1,0) = (A(1,2)*A(2,0) - A(1,0)*A(2,2)) * invdet;
01672             IA(1,1) = (A(0,0)*A(2,2) - A(0,2)*A(2,0)) * invdet;
01673             IA(1,2) = (A(1,0)*A(0,2) - A(0,0)*A(1,2)) * invdet;
01674             IA(2,0) = (A(1,0)*A(2,1) - A(2,0)*A(1,1)) * invdet;
01675             IA(2,1) = (A(2,0)*A(0,1) - A(0,0)*A(2,1)) * invdet;
01676             IA(2,2) = (A(0,0)*A(1,1) - A(1,0)*A(0,1)) * invdet;
01677         }
01678         return IA;
01679     } else {

```

```

01734     return inv_square(A);
01735 }
01736 }
01737
01747 template<typename T>
01748 Matrix<T> pinv(const Matrix<T>& A) {
01749     if (A.rows() > A.cols()) {
01750         auto AH_A = mult<T,true,false>(A, A);
01751         auto Linv = inv_posdef(AH_A);
01752         return mult<T,false,true>(Linv, A);
01753     } else {
01754         auto AA_H = mult<T,false,true>(A, A);
01755         auto Linv = inv_posdef(AA_H);
01756         return mult<T,true,false>(A, Linv);
01757     }
01758 }
01759
01765 template<typename T>
01766 T trace(const Matrix<T>& A) {
01767     T t = static_cast<T>(0);
01768     for (int i = 0; i < A.rows(); i++)
01769         t += A(i,i);
01770     return t;
01771 }
01772
01781 template<typename T>
01782 double cond(const Matrix<T>& A) {
01783     try {
01784         auto A_inv = inv(A);
01785         return norm_fro(A) * norm_fro(A_inv);
01786     } catch (singular_matrix_exception& e) {
01787         return std::numeric_limits<double>::max();
01788     }
01789 }
01790
01809 template<typename T, bool is_upper = false>
01810 Matrix<T> chol(const Matrix<T>& A) {
01811     if (!A.issquare()) throw std::runtime_error("Input matrix is not square");
01812
01813     const unsigned N = A.rows();
01814
01815     // Calculate lower or upper triangular, depending on template parameter.
01816     // Calculation is the same - the difference is in transposed row and column indexing.
01817     Matrix<T> C = is_upper ? triu(A) : tril(A);
01818
01819     for (unsigned j = 0; j < N; j++) {
01820         if (C(j,j) == static_cast<T>(0.0)) throw singular_matrix_exception("Singular matrix in chol");
01821
01822         C(j,j) = std::sqrt(C(j,j));
01823
01824         for (unsigned k = j+1; k < N; k++)
01825             if (is_upper)
01826                 C(j,k) /= C(j,j);
01827             else
01828                 C(k,j) /= C(j,j);
01829
01830         for (unsigned k = j+1; k < N; k++)
01831             for (unsigned i = k; i < N; i++)
01832                 if (is_upper)
01833                     C(k,i) -= C(j,i) * cconj(C(j,k));
01834                 else
01835                     C(i,k) -= C(i,j) * cconj(C(k,j));
01836     }
01837
01838     return C;
01839 }
01840
01852 template<typename T>
01853 Matrix<T> cholinv(const Matrix<T>& A) {
01854     if (!A.issquare()) throw std::runtime_error("Input matrix is not square");
01855
01856     const unsigned N = A.rows();
01857     Matrix<T> L(A);
01858     auto Linv = eye<T>(N);
01859
01860     for (unsigned j = 0; j < N; j++) {
01861         if (L(j,j) == static_cast<T>(0.0)) throw singular_matrix_exception("Singular matrix in cholinv");
01862
01863         L(j,j) = static_cast<T>(1.0) / std::sqrt(L(j,j));
01864
01865         for (unsigned k = j+1; k < N; k++)
01866             L(k,j) = L(k,j) * L(j,j);
01867
01868         for (unsigned k = j+1; k < N; k++)
01869             for (unsigned i = k; i < N; i++)
01870                 L(i,k) = L(i,k) - L(i,j) * cconj(L(k,j));
01871     }

```

```

01872
01873     for (unsigned k = 0; k < N; k++) {
01874         for (unsigned i = k; i < N; i++) {
01875             Linv(i,k) = Linv(i,k) * L(i,i);
01876             for (unsigned j = i+1; j < N; j++)
01877                 Linv(j,k) = Linv(j,k) - L(j,i) * Linv(i,k);
01878         }
01879     }
01880
01881     return Linv;
01882 }
01883
01889 template<typename T>
01900 LDL_result<T> ldl(const Matrix<T>& A) {
01901     if (! A.issquare()) throw std::runtime_error("Input matrix is not square");
01902
01903     const unsigned N = A.rows();
01904
01905     LDL_result<T> res;
01906
01907     // aliases
01908     auto& L = res.L;
01909     auto& d = res.d;
01910
01911     L = eye<T>(N);
01912     d.resize(N);
01913
01914     for (unsigned m = 0; m < N; m++) {
01915         d[m] = A(m,m);
01916
01917         for (unsigned k = 0; k < m; k++)
01918             d[m] -= L(m,k) * cconj(L(m,k)) * d[k];
01919
01920         if (d[m] == static_cast<T>(0.0)) throw singular_matrix_exception("Singular matrix in ldl");
01921
01922         for (unsigned n = m+1; n < N; n++) {
01923             L(n,m) = A(n,m);
01924             for (unsigned k = 0; k < m; k++)
01925                 L(n,m) -= L(n,k) * cconj(L(m,k)) * d[k];
01926             L(n,m) /= d[m];
01927         }
01928     }
01929
01930     return res;
01931 }
01932
01944 template<typename T>
01945 QR_result<T> qr_red_gs(const Matrix<T>& A) {
01946     const int rows = A.rows();
01947     const int cols = A.cols();
01948
01949     QR_result<T> res;
01950
01951     //aliases
01952     auto& Q = res.Q;
01953     auto& R = res.R;
01954
01955     Q = zeros<T>(rows, cols);
01956     R = zeros<T>(cols, cols);
01957
01958     for (int c = 0; c < cols; c++) {
01959         Matrix<T> v = A.get_submatrix(0, rows-1, c, c);
01960         for (int r = 0; r < c; r++) {
01961             for (int k = 0; k < rows; k++)
01962                 R(r,c) = R(r,c) + cconj(Q(k,r)) * A(k,c);
01963             for (int k = 0; k < rows; k++)
01964                 v(k) = v(k) - R(r,c) * Q(k,r);
01965         }
01966
01967         R(c,c) = static_cast<T>(norm_fro(v));
01968
01969         if (R(c,c) == static_cast<T>(0.0)) throw singular_matrix_exception("Division by 0 in QR GS");
01970
01971         for (int k = 0; k < rows; k++)
01972             Q(k,c) = v(k) / R(c,c);
01973     }
01974
01975     return res;
01976 }
01977
01985 template<typename T>
01986 Matrix<T> householder_reflection(const Matrix<T>& a) {
01987     if (a.cols() != 1) throw std::runtime_error("Input not a column vector");
01988
01989     static const T ISQRT2 = static_cast<T>(0.707106781186547);
01990
01991     Matrix<T> v(a);

```

```

01992     v(0) += csign(v(0)) * norm_fro(v);
01993     auto vn = norm_fro(v) * ISQRT2;
01994     for (unsigned i = 0; i < v.numel(); i++)
01995         v(i) /= vn;
01996     return v;
01997 }
01998
01999 template<typename T>
02000 QR_result<T> qr_householder(const Matrix<T>& A, bool calculate_Q = true) {
02001     const unsigned rows = A.rows();
02002     const unsigned cols = A.cols();
02003
02004     QR_result<T> res;
02005
02006     //aliases
02007     auto& Q = res.Q;
02008     auto& R = res.R;
02009
02010     R = Matrix<T>(A);
02011
02012     if (calculate_Q)
02013         Q = eye<T>(rows);
02014
02015     const unsigned N = (rows > cols) ? cols : rows;
02016
02017     for (unsigned j = 0; j < N; j++) {
02018         auto v = householder_reflection(R.get_submatrix(j, rows-1, j, j));
02019
02020         auto R1 = R.get_submatrix(j, rows-1, j, cols-1);
02021         auto WR = v * mult<T,true,false>(v, R1);
02022         for (unsigned c = j; c < cols; c++)
02023             for (unsigned r = j; r < rows; r++)
02024                 R(r,c) -= WR(r-j,c-j);
02025
02026         if (calculate_Q) {
02027             auto Q1 = Q.get_submatrix(0, rows-1, j, rows-1);
02028             auto WQ = mult<T,false,true>(Q1 * v, v);
02029             for (unsigned c = j; c < rows; c++)
02030                 for (unsigned r = 0; r < rows; r++)
02031                     Q(r,c) -= WQ(r,c-j);
02032         }
02033     }
02034
02035     for (unsigned col = 0; col < R.cols(); col++)
02036         for (unsigned row = col+1; row < R.rows(); row++)
02037             R(row,col) = 0;
02038
02039     return res;
02040 }
02041
02042 template<typename T>
02043 inline QR_result<T> qr(const Matrix<T>& A, bool calculate_Q = true) {
02044     return qr_householder(A, calculate_Q);
02045 }
02046
02047 template<typename T>
02048 Hessenberg_result<T> hessenberg(const Matrix<T>& A, bool calculate_Q = true) {
02049     if (! A.issquare()) throw std::runtime_error("Input matrix is not square");
02050
02051     Hessenberg_result<T> res;
02052
02053     // aliases
02054     auto& H = res.H;
02055     auto& Q = res.Q;
02056
02057     const unsigned N = A.rows();
02058     H = Matrix<T>(A);
02059
02060     if (calculate_Q)
02061         Q = eye<T>(N);
02062
02063     for (unsigned k = 1; k < N-1; k++) {
02064         auto v = householder_reflection(H.get_submatrix(k, N-1, k-1, k-1));
02065
02066         auto H1 = H.get_submatrix(k, N-1, 0, N-1);
02067         auto W1 = v * mult<T,true,false>(v, H1);
02068         for (unsigned c = 0; c < N; c++)
02069             for (unsigned r = k; r < N; r++)
02070                 H(r,c) -= W1(r-k,c);
02071
02072         auto H2 = H.get_submatrix(0, N-1, k, N-1);
02073         auto W2 = mult<T,false,true>(H2 * v, v);
02074         for (unsigned c = k; c < N; c++)
02075             for (unsigned r = 0; r < N; r++)
02076                 H(r,c) -= W2(r,c-k);
02077
02078         if (calculate_Q) {

```

```

02111     auto Q1 = Q.get_submatrix(0, N-1, k, N-1);
02112     auto W3 = mult<T,false,true>(Q1 * v, v);
02113     for (unsigned c = k; c < N; c++)
02114         for (unsigned r = 0; r < N; r++)
02115             Q(r,c) -= W3(r,c-k);
02116     }
02117 }
02118
02119 for (unsigned row = 2; row < N; row++)
02120     for (unsigned col = 0; col < row-2; col++)
02121         H(row,col) = static_cast<T>(0);
02122
02123 return res;
02124 }
02125
02126 template<typename T>
02127 std::complex<T> wilkinson_shift(const Matrix<std::complex<T>& H, T tol = 1e-10) {
02128     if (!H.issquare()) throw std::runtime_error("Input matrix is not square");
02129
02130     const unsigned n = H.rows();
02131     std::complex<T> mu;
02132
02133     if (std::abs(H(n-1,n-2)) < tol) {
02134         mu = H(n-2,n-2);
02135     } else {
02136         auto trA = H(n-2,n-2) + H(n-1,n-1);
02137         auto detA = H(n-2,n-2) * H(n-1,n-1) - H(n-2, n-1) * H(n-1, n-2);
02138         mu = (trA + std::sqrt(trA*trA - 4.0*detA)) / 2.0;
02139     }
02140
02141     return mu;
02142 }
02143
02144 template<typename T>
02145 Eigenvalues_result<T> eigenvalues(const Matrix<std::complex<T>& A, T tol = 1e-12, unsigned max_iter =
100) {
02146     if (!A.issquare()) throw std::runtime_error("Input matrix is not square");
02147
02148     const unsigned N = A.rows();
02149     Matrix<std::complex<T>> H;
02150     bool success = false;
02151
02152     QR_result<std::complex<T>> QR;
02153
02154     // aliases
02155     auto& Q = QR.Q;
02156     auto& R = QR.R;
02157
02158     // Transfer A to Hessenberg form to improve convergence (skip calculation of Q)
02159     H = hessenberg(A, false).H;
02160
02161     for (unsigned iter = 0; iter < max_iter; iter++) {
02162         auto mu = wilkinson_shift(H, tol);
02163
02164         // subtract mu from diagonal
02165         for (unsigned n = 0; n < N; n++)
02166             H(n,n) -= mu;
02167
02168         // QR factorization with shifted H
02169         QR = qr(H);
02170         H = R * Q;
02171
02172         // add back mu to diagonal
02173         for (unsigned n = 0; n < N; n++)
02174             H(n,n) += mu;
02175
02176         // Check for convergence
02177         if (std::abs(H(N-2,N-1)) <= tol) {
02178             success = true;
02179             break;
02180         }
02181     }
02182
02183     Eigenvalues_result<T> res;
02184     res.eig = diag(H);
02185     res.err = std::abs(H(N-2,N-1));
02186     res.converged = success;
02187
02188     return res;
02189 }
02190
02191 template<typename T>
02192 Eigenvalues_result<T> eigenvalues(const Matrix<T>& A, T tol = 1e-12, unsigned max_iter = 100) {
02193     auto A_cplx = make_complex(A);
02194     return eigenvalues(A_cplx, tol, max_iter);
02195 }
02196
02197
02198

```



```

02240 template<typename T>
02241 Matrix<T> solve_triu(const Matrix<T>& U, const Matrix<T>& B) {
02242     if (! U.issquare()) throw std::runtime_error("Input matrix is not square");
02243     if (U.rows() != B.rows()) throw std::runtime_error("Unmatching matrix dimensions for solve");
02244
02245     const unsigned N = U.rows();
02246     const unsigned M = B.cols();
02247
02248     if (U.numel() == 0)
02249         return Matrix<T>();
02250
02251     Matrix<T> X(B);
02252
02253     for (unsigned m = 0; m < M; m++) {
02254         // backwards substitution for each column of B
02255         for (int n = N-1; n >= 0; n--) {
02256             for (unsigned j = n + 1; j < N; j++)
02257                 X(n,m) -= U(n,j) * X(j,m);
02258
02259             if (U(n,n) == static_cast<T>(0.0)) throw singular_matrix_exception("Singular matrix in
solve_triu");
02260
02261             X(n,m) /= U(n,n);
02262         }
02263     }
02264
02265     return X;
02266 }
02267
02283 template<typename T>
02284 Matrix<T> solve_tril(const Matrix<T>& L, const Matrix<T>& B) {
02285     if (! L.issquare()) throw std::runtime_error("Input matrix is not square");
02286     if (L.rows() != B.rows()) throw std::runtime_error("Unmatching matrix dimensions for solve");
02287
02288     const unsigned N = L.rows();
02289     const unsigned M = B.cols();
02290
02291     if (L.numel() == 0)
02292         return Matrix<T>();
02293
02294     Matrix<T> X(B);
02295
02296     for (unsigned m = 0; m < M; m++) {
02297         // forwards substitution for each column of B
02298         for (unsigned n = 0; n < N; n++) {
02299             for (unsigned j = 0; j < n; j++)
02300                 X(n,m) -= L(n,j) * X(j,m);
02301
02302             if (L(n,n) == static_cast<T>(0.0)) throw singular_matrix_exception("Singular matrix in
solve_tril");
02303
02304             X(n,m) /= L(n,n);
02305         }
02306     }
02307
02308     return X;
02309 }
02310
02326 template<typename T>
02327 Matrix<T> solve_square(const Matrix<T>& A, const Matrix<T>& B) {
02328     if (! A.issquare()) throw std::runtime_error("Input matrix is not square");
02329     if (A.rows() != B.rows()) throw std::runtime_error("Unmatching matrix dimensions for solve");
02330
02331     if (A.numel() == 0)
02332         return Matrix<T>();
02333
02334     Matrix<T> L;
02335     Matrix<T> U;
02336     std::vector<unsigned> P;
02337
02338     // LU decomposition with pivoting
02339     auto lup_res = lup(A);
02340
02341     auto y = solve_tril(lup_res.L, B);
02342     auto x = solve_triu(lup_res.U, y);
02343
02344     return permute_rows(x, lup_res.P);
02345 }
02346
02362 template<typename T>
02363 Matrix<T> solve_posdef(const Matrix<T>& A, const Matrix<T>& B) {
02364     if (! A.issquare()) throw std::runtime_error("Input matrix is not square");
02365     if (A.rows() != B.rows()) throw std::runtime_error("Unmatching matrix dimensions for solve");
02366
02367     if (A.numel() == 0)
02368         return Matrix<T>();
02369

```

```

02370 // LU decomposition with pivoting
02371 auto L = chol(A);
02372
02373 auto Y = solve_tril(L, B);
02374 return solve_triu(L.ctranspose(), Y);
02375 }
02376
02381 template<typename T>
02382 class Matrix {
02383 public:
02388     Matrix();
02389
02394     Matrix(unsigned size);
02395
02400     Matrix(unsigned nrows, unsigned ncols);
02401
02406     Matrix(T x, unsigned nrows, unsigned ncols);
02407
02414     Matrix(const T* array, unsigned nrows, unsigned ncols);
02415
02425     Matrix(const std::vector<T>& vec, unsigned nrows, unsigned ncols);
02426
02436     Matrix(std::initializer_list<T> init_list, unsigned nrows, unsigned ncols);
02437
02440     Matrix(const Matrix &);
02441
02444     virtual ~Matrix();
02445
02454     Matrix<T> get_submatrix(unsigned row_first, unsigned row_last, unsigned col_first, unsigned
col_last) const;
02455
02464     void set_submatrix(const Matrix<T>& smtx, unsigned row_first, unsigned col_first);
02465
02470     void clear();
02471
02479     void reshape(unsigned rows, unsigned cols);
02480
02486     void resize(unsigned rows, unsigned cols);
02487
02493     bool exists(unsigned row, unsigned col) const;
02494
02500     T* ptr(unsigned row, unsigned col);
02501
02509     T* ptr();
02510
02514     void fill(T value);
02515
02522     void fill_col(T value, unsigned col);
02523
02530     void fill_row(T value, unsigned row);
02531
02536     bool isempty() const;
02537
02541     bool issquare() const;
02542
02547     bool isequal(const Matrix<T>&) const;
02548
02554     bool isequal(const Matrix<T>&, T) const;
02555
02560     unsigned numel() const;
02561
02566     unsigned rows() const;
02567
02572     unsigned cols() const;
02573
02579     std::pair<unsigned,unsigned> shape() const;
02580
02585     Matrix<T> transpose() const;
02586
02592     Matrix<T> ctranspose() const;
02593
02601     Matrix<T>& add(const Matrix<T>&);
02602
02610     Matrix<T>& subtract(const Matrix<T>&);
02611
02620     Matrix<T>& mult_hadamard(const Matrix<T>&);
02621
02627     Matrix<T>& add(T);
02628
02634     Matrix<T>& subtract(T);
02635
02641     Matrix<T>& mult(T);
02642
02648     Matrix<T>& div(T);
02649
02654     Matrix<T>& operator=(const Matrix<T>&);
02655

```

```

02661     Matrix<T>& operator=(T);
02662
02668     explicit operator std::vector<T>() const;
02669     std::vector<T> to_vector() const;
02670
02677     T& operator()(unsigned nel);
02678     T operator()(unsigned nel) const;
02679     T& at(unsigned nel);
02680     T at(unsigned nel) const;
02681
02688     T& operator()(unsigned row, unsigned col);
02689     T operator()(unsigned row, unsigned col) const;
02690     T& at(unsigned row, unsigned col);
02691     T at(unsigned row, unsigned col) const;
02692
02700     void add_row_to_another(unsigned to, unsigned from);
02701
02709     void add_col_to_another(unsigned to, unsigned from);
02710
02718     void mult_row_by_another(unsigned to, unsigned from);
02719
02727     void mult_col_by_another(unsigned to, unsigned from);
02728
02735     void swap_rows(unsigned i, unsigned j);
02736
02743     void swap_cols(unsigned i, unsigned j);
02744
02751     std::vector<T> col_to_vector(unsigned col) const;
02752
02759     std::vector<T> row_to_vector(unsigned row) const;
02760
02769     void col_from_vector(const std::vector<T>&, unsigned col);
02770
02779     void row_from_vector(const std::vector<T>&, unsigned row);
02780
02781 private:
02782     unsigned nrows;
02783     unsigned ncols;
02784     std::vector<T> data;
02785 };
02786
02787 /*
02788  * Implementation of Matrix class methods
02789  */
02790
02791 template<typename T>
02792 Matrix<T>::Matrix() : nrows(0), ncols(0), data() { }
02793
02794 template<typename T>
02795 Matrix<T>::Matrix(unsigned size) : Matrix(size, size) { }
02796
02797 template<typename T>
02798 Matrix<T>::Matrix(unsigned rows, unsigned cols) : nrows(rows), ncols(cols) {
02799     data.resize(numel());
02800 }
02801
02802 template<typename T>
02803 Matrix<T>::Matrix(T x, unsigned rows, unsigned cols) : Matrix(rows, cols) {
02804     fill(x);
02805 }
02806
02807 template<typename T>
02808 Matrix<T>::Matrix(const T* array, unsigned rows, unsigned cols) : Matrix(rows, cols) {
02809     data.assign(array, array + numel());
02810 }
02811
02812 template<typename T>
02813 Matrix<T>::Matrix(const std::vector<T>& vec, unsigned rows, unsigned cols) : Matrix(rows, cols) {
02814     if (vec.size() != numel()) throw std::runtime_error("Size of initialization vector not consistent
02815 with matrix dimensions");
02816     data.assign(vec.begin(), vec.end());
02817 }
02818
02819 template<typename T>
02820 Matrix<T>::Matrix(std::initializer_list<T> init_list, unsigned rows, unsigned cols) : Matrix(rows,
02821 cols) {
02822     if (init_list.size() != numel()) throw std::runtime_error("Size of initialization list not
02823 consistent with matrix dimensions");
02824
02825     auto it = init_list.begin();
02826     for (unsigned row = 0; row < this->nrows; row++)
02827         for (unsigned col = 0; col < this->ncols; col++)
02828             this->at(row, col) = *(it++);
02829 }

```

```

02830 template<typename T>
02831 Matrix<T>::Matrix(const Matrix & other) : Matrix(other.nrows, other.ncols) {
02832     this->data.assign(other.data.begin(), other.data.end());
02833 }
02834
02835 template<typename T>
02836 Matrix<T> & Matrix<T>::operator=(const Matrix<T> & other) {
02837     this->nrows = other.nrows;
02838     this->ncols = other.ncols;
02839     this->data.assign(other.data.begin(), other.data.end());
02840     return *this;
02841 }
02842
02843 template<typename T>
02844 Matrix<T> & Matrix<T>::operator=(T s) {
02845     fill(s);
02846     return *this;
02847 }
02848
02849 template<typename T>
02850 inline Matrix<T>::operator std::vector<T>() const {
02851     return data;
02852 }
02853
02854 template<typename T>
02855 inline void Matrix<T>::clear() {
02856     this->nrows = 0;
02857     this->ncols = 0;
02858     data.resize(0);
02859 }
02860
02861 template<typename T>
02862 void Matrix<T>::reshape(unsigned rows, unsigned cols) {
02863     if (this->numel() != rows * cols) throw std::runtime_error("Illegal attempt to change number of
elements via reshape");
02864
02865     this->nrows = rows;
02866     this->ncols = cols;
02867 }
02868
02869 template<typename T>
02870 void Matrix<T>::resize(unsigned rows, unsigned cols) {
02871     this->nrows = rows;
02872     this->ncols = cols;
02873     data.resize(nrows*ncols);
02874 }
02875
02876 template<typename T>
02877 Matrix<T> Matrix<T>::get_submatrix(unsigned row_base, unsigned row_lim, unsigned col_base, unsigned
col_lim) const {
02878     if (row_base > row_lim) throw std::out_of_range("Row index of submatrix out of range");
02879     if (col_base > col_lim) throw std::out_of_range("Column index of submatrix out of range");
02880     if (row_lim >= this->rows()) throw std::out_of_range("Row index of submatrix out of range");
02881     if (col_lim >= this->cols()) throw std::out_of_range("Column index of submatrix out of range");
02882
02883     unsigned num_rows = row_lim - row_base + 1;
02884     unsigned num_cols = col_lim - col_base + 1;
02885     Matrix<T> S(num_rows, num_cols);
02886     for (unsigned i = 0; i < num_rows; i++) {
02887         for (unsigned j = 0; j < num_cols; j++) {
02888             S(i,j) = at(row_base + i, col_base + j);
02889         }
02890     }
02891     return S;
02892 }
02893
02894 template<typename T>
02895 void Matrix<T>::set_submatrix(const Matrix<T> & S, unsigned row_base, unsigned col_base) {
02896     if (this->isempty()) throw std::runtime_error("Invalid attempt to set submatrix in empty matrix");
02897
02898     const unsigned row_lim = row_base + S.rows() - 1;
02899     const unsigned col_lim = col_base + S.cols() - 1;
02900
02901     if (row_base > row_lim) throw std::out_of_range("Row index of submatrix out of range");
02902     if (col_base > col_lim) throw std::out_of_range("Column index of submatrix out of range");
02903     if (row_lim >= this->rows()) throw std::out_of_range("Row index of submatrix out of range");
02904     if (col_lim >= this->cols()) throw std::out_of_range("Column index of submatrix out of range");
02905
02906     unsigned num_rows = row_lim - row_base + 1;
02907     unsigned num_cols = col_lim - col_base + 1;
02908     for (unsigned i = 0; i < num_rows; i++)
02909         for (unsigned j = 0; j < num_cols; j++)
02910             at(row_base + i, col_base + j) = S(i,j);
02911 }
02912
02913 template<typename T>
02914 inline T & Matrix<T>::operator()(unsigned nel) {

```

```

02915     return at(nel);
02916 }
02917
02918 template<typename T>
02919 inline T & Matrix<T>::operator()(unsigned row, unsigned col) {
02920     return at(row, col);
02921 }
02922
02923 template<typename T>
02924 inline T Matrix<T>::operator()(unsigned nel) const {
02925     return at(nel);
02926 }
02927
02928 template<typename T>
02929 inline T Matrix<T>::operator()(unsigned row, unsigned col) const {
02930     return at(row, col);
02931 }
02932
02933 template<typename T>
02934 inline T & Matrix<T>::at(unsigned nel) {
02935     if (!(nel < numel())) throw std::out_of_range("Element index out of range");
02936     return data[nel];
02937 }
02938
02939 template<typename T>
02940 inline T & Matrix<T>::at(unsigned row, unsigned col) {
02941     if (!(row < rows() && col < cols())) throw std::out_of_range("Element index out of range");
02942     return data[nrows * col + row];
02943 }
02944
02945 template<typename T>
02946 inline T Matrix<T>::at(unsigned nel) const {
02947     if (!(nel < numel())) throw std::out_of_range("Element index out of range");
02948     return data[nel];
02949 }
02950
02951 template<typename T>
02952 inline T Matrix<T>::at(unsigned row, unsigned col) const {
02953     if (!(row < rows() && col < cols())) throw std::out_of_range("Row index out of range");
02954     if (!(col < cols())) throw std::out_of_range("Column index out of range");
02955     return data[nrows * col + row];
02956 }
02957
02958 template<typename T>
02959 inline void Matrix<T>::fill(T value) {
02960     for (unsigned i = 0; i < numel(); i++)
02961         data[i] = value;
02962 }
02963
02964 template<typename T>
02965 inline void Matrix<T>::fill_col(T value, unsigned col) {
02966     if (!(col < cols())) throw std::out_of_range("Column index out of range");
02967     for (unsigned i = col * nrows; i < (col+1) * nrows; i++)
02968         data[i] = value;
02969 }
02970
02971 template<typename T>
02972 inline void Matrix<T>::fill_row(T value, unsigned row) {
02973     if (!(row < rows())) throw std::out_of_range("Row index out of range");
02974     for (unsigned i = 0; i < ncols; i++)
02975         data[row + i * nrows] = value;
02976 }
02977
02978 template<typename T>
02979 inline bool Matrix<T>::exists(unsigned row, unsigned col) const {
02980     return (row < nrows && col < ncols);
02981 }
02982
02983 template<typename T>
02984 inline T* Matrix<T>::ptr(unsigned row, unsigned col) {
02985     if (!(row < rows())) throw std::out_of_range("Row index out of range");
02986     if (!(col < cols())) throw std::out_of_range("Column index out of range");
02987     return data.data() + nrows * col + row;
02988 }
02989
02990 template<typename T>
02991 inline T* Matrix<T>::ptr() {
02992     return data.data();
02993 }
02994
02995
03000 }
03001

```

```

03002 template<typename T>
03003 inline bool Matrix<T>::isempty() const {
03004     return (nrows == 0) || (ncols == 0);
03005 }
03006
03007 template<typename T>
03008 inline bool Matrix<T>::issquare() const {
03009     return (nrows == ncols) && !isempty();
03010 }
03011
03012 template<typename T>
03013 bool Matrix<T>::isequal(const Matrix<T>& A) const {
03014     bool ret = true;
03015     if (nrows != A.rows() || ncols != A.cols()) {
03016         ret = false;
03017     } else {
03018         for (unsigned i = 0; i < numel(); i++) {
03019             if (at(i) != A(i)) {
03020                 ret = false;
03021                 break;
03022             }
03023         }
03024     }
03025     return ret;
03026 }
03027
03028 template<typename T>
03029 bool Matrix<T>::isequal(const Matrix<T>& A, T tol) const {
03030     bool ret = true;
03031     if (rows() != A.rows() || cols() != A.cols()) {
03032         ret = false;
03033     } else {
03034         auto abs_tol = std::abs(tol); // workaround for complex
03035         for (unsigned i = 0; i < A.numel(); i++) {
03036             if (abs_tol < std::abs(at(i) - A(i))) {
03037                 ret = false;
03038                 break;
03039             }
03040         }
03041     }
03042     return ret;
03043 }
03044
03045 template<typename T>
03046 inline unsigned Matrix<T>::numel() const {
03047     return nrows * ncols;
03048 }
03049
03050 template<typename T>
03051 inline unsigned Matrix<T>::rows() const {
03052     return nrows;
03053 }
03054
03055 template<typename T>
03056 inline unsigned Matrix<T>::cols() const {
03057     return ncols;
03058 }
03059
03060 template<typename T>
03061 inline std::pair<unsigned,unsigned> Matrix<T>::shape() const {
03062     return std::pair<unsigned,unsigned>(nrows,ncols);
03063 }
03064
03065 template<typename T>
03066 inline Matrix<T> Matrix<T>::transpose() const {
03067     Matrix<T> res(ncols, nrows);
03068     for (unsigned c = 0; c < ncols; c++)
03069         for (unsigned r = 0; r < nrows; r++)
03070             res(c,r) = at(r,c);
03071     return res;
03072 }
03073
03074 template<typename T>
03075 inline Matrix<T> Matrix<T>::ctranspose() const {
03076     Matrix<T> res(ncols, nrows);
03077     for (unsigned c = 0; c < ncols; c++)
03078         for (unsigned r = 0; r < nrows; r++)
03079             res(c,r) = cconj(at(r,c));
03080     return res;
03081 }
03082
03083 template<typename T>
03084 Matrix<T>& Matrix<T>::add(const Matrix<T>& m) {
03085     if (!(m.rows() == rows() && m.cols() == cols())) throw std::runtime_error("Unmatching matrix
dimensions for iadd");
03086
03087     for (unsigned i = 0; i < numel(); i++)

```

```

03088     data[i] += m(i);
03089     return *this;
03090 }
03091
03092 template<typename T>
03093 Matrix<T>& Matrix<T>::subtract(const Matrix<T>& m) {
03094     if (! (m.rows() == rows() && m.cols() == cols())) throw std::runtime_error("Unmatching matrix
dimensions for isubtract");
03095
03096     for (unsigned i = 0; i < numel(); i++)
03097         data[i] -= m(i);
03098     return *this;
03099 }
03100
03101 template<typename T>
03102 Matrix<T>& Matrix<T>::mult_hadamard(const Matrix<T>& m) {
03103     if (! (m.rows() == rows() && m.cols() == cols())) throw std::runtime_error("Unmatching matrix
dimensions for ihprod");
03104
03105     for (unsigned i = 0; i < numel(); i++)
03106         data[i] *= m(i);
03107     return *this;
03108 }
03109
03110 template<typename T>
03111 Matrix<T>& Matrix<T>::add(T s) {
03112     for (auto& x : data)
03113         x += s;
03114     return *this;
03115 }
03116
03117 template<typename T>
03118 Matrix<T>& Matrix<T>::subtract(T s) {
03119     for (auto& x : data)
03120         x -= s;
03121     return *this;
03122 }
03123
03124 template<typename T>
03125 Matrix<T>& Matrix<T>::mult(T s) {
03126     for (auto& x : data)
03127         x *= s;
03128     return *this;
03129 }
03130
03131 template<typename T>
03132 Matrix<T>& Matrix<T>::div(T s) {
03133     for (auto& x : data)
03134         x /= s;
03135     return *this;
03136 }
03137
03138 template<typename T>
03139 void Matrix<T>::add_row_to_another(unsigned to, unsigned from) {
03140     if (! (to < rows() && from < rows())) throw std::out_of_range("Row index out of range");
03141
03142     for (unsigned k = 0; k < cols(); k++)
03143         at(to, k) += at(from, k);
03144 }
03145
03146 template<typename T>
03147 void Matrix<T>::add_col_to_another(unsigned to, unsigned from) {
03148     if (! (to < cols() && from < cols())) throw std::out_of_range("Column index out of range");
03149
03150     for (unsigned k = 0; k < rows(); k++)
03151         at(k, to) += at(k, from);
03152 }
03153
03154 template<typename T>
03155 void Matrix<T>::mult_row_by_another(unsigned to, unsigned from) {
03156     if (! (to < rows() && from < rows())) throw std::out_of_range("Row index out of range");
03157
03158     for (unsigned k = 0; k < cols(); k++)
03159         at(to, k) *= at(from, k);
03160 }
03161
03162 template<typename T>
03163 void Matrix<T>::mult_col_by_another(unsigned to, unsigned from) {
03164     if (! (to < cols() && from < cols())) throw std::out_of_range("Column index out of range");
03165
03166     for (unsigned k = 0; k < rows(); k++)
03167         at(k, to) *= at(k, from);
03168 }
03169
03170 template<typename T>
03171 void Matrix<T>::swap_rows(unsigned i, unsigned j) {
03172     if (! (i < rows() && j < rows())) throw std::out_of_range("Row index out of range");

```

```

03173
03174     for (unsigned k = 0; k < cols(); k++) {
03175         T tmp = at(i,k);
03176         at(i,k) = at(j,k);
03177         at(j,k) = tmp;
03178     }
03179 }
03180
03181 template<typename T>
03182 void Matrix<T>::swap_cols(unsigned i, unsigned j) {
03183     if (!(i < cols() && j < cols())) throw std::out_of_range("Column index out of range");
03184
03185     for (unsigned k = 0; k < rows(); k++) {
03186         T tmp = at(k,i);
03187         at(k,i) = at(k,j);
03188         at(k,j) = tmp;
03189     }
03190 }
03191
03192 template<typename T>
03193 inline std::vector<T> Matrix<T>::to_vector() const {
03194     return data;
03195 }
03196
03197 template<typename T>
03198 inline std::vector<T> Matrix<T>::col_to_vector(unsigned col) const {
03199     std::vector<T> vec(rows());
03200     for (unsigned i = 0; i < rows(); i++)
03201         vec[i] = at(i,col);
03202     return vec;
03203 }
03204
03205 template<typename T>
03206 inline std::vector<T> Matrix<T>::row_to_vector(unsigned row) const {
03207     std::vector<T> vec(cols());
03208     for (unsigned i = 0; i < cols(); i++)
03209         vec[i] = at(row,i);
03210     return vec;
03211 }
03212
03213 template<typename T>
03214 inline void Matrix<T>::col_from_vector(const std::vector<T>& vec, unsigned col) {
03215     if (vec.size() != rows()) throw std::runtime_error("Vector size is not equal to number of rows");
03216     if (col >= cols()) throw std::out_of_range("Column index out of range");
03217
03218     for (unsigned i = 0; i < rows(); i++)
03219         data[col*rows() + i] = vec[i];
03220 }
03221
03222 template<typename T>
03223 inline void Matrix<T>::row_from_vector(const std::vector<T>& vec, unsigned row) {
03224     if (vec.size() != cols()) throw std::runtime_error("Vector size is not equal to number of columns");
03225     if (row >= rows()) throw std::out_of_range("Row index out of range");
03226
03227     for (unsigned i = 0; i < cols(); i++)
03228         data[row + i*rows()] = vec[i];
03229 }
03230
03231 template<typename T>
03232 Matrix<T>::~Matrix() { }
03233
03234 } // namespace Matrix_hpp
03235
03236 #endif // __MATRIX_HPP__

```