

Matrix HPP

Generated by Doxygen 1.9.8

| | |
|--|----------|
| 1 Matrix HPP - C++ library for matrix class container and linear algebra computations | 1 |
| 1.1 Installation | 1 |
| 1.2 Hello world example | 1 |
| 1.3 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] | 14 |
| 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] | 15 |
| 5.6.2.7 Matrix() [7/8] | 16 |
| 5.6.2.8 Matrix() [8/8] | 16 |
| 5.6.2.9 ~Matrix() | 16 |
| 5.6.3 Member Function Documentation | 16 |
| 5.6.3.1 add() [1/2] | 16 |
| 5.6.3.2 add() [2/2] | 17 |
| 5.6.3.3 add_col_to_another() | 17 |
| 5.6.3.4 add_row_to_another() | 17 |
| 5.6.3.5 clear() | 18 |
| 5.6.3.6 col_from_vector() | 18 |

| | |
|--|-----------|
| 5.6.3.7 col_to_vector() | 18 |
| 5.6.3.8 cols() | 18 |
| 5.6.3.9 ctranspose() | 19 |
| 5.6.3.10 div() | 19 |
| 5.6.3.11 exists() | 19 |
| 5.6.3.12 fill() | 20 |
| 5.6.3.13 fill_col() | 20 |
| 5.6.3.14 fill_row() | 20 |
| 5.6.3.15 get_submatrix() | 20 |
| 5.6.3.16 isempty() | 21 |
| 5.6.3.17 isequal() [1/2] | 21 |
| 5.6.3.18 isequal() [2/2] | 21 |
| 5.6.3.19 mult() | 22 |
| 5.6.3.20 mult_hadamard() | 22 |
| 5.6.3.21 numel() | 22 |
| 5.6.3.22 operator std::vector< T >() | 22 |
| 5.6.3.23 operator()() [1/2] | 23 |
| 5.6.3.24 operator()() [2/2] | 23 |
| 5.6.3.25 operator=() [1/2] | 23 |
| 5.6.3.26 operator=() [2/2] | 23 |
| 5.6.3.27 ptr() [1/2] | 24 |
| 5.6.3.28 ptr() [2/2] | 24 |
| 5.6.3.29 reshape() | 24 |
| 5.6.3.30 resize() | 24 |
| 5.6.3.31 row_from_vector() | 25 |
| 5.6.3.32 row_to_vector() | 25 |
| 5.6.3.33 rows() | 25 |
| 5.6.3.34 set_submatrix() | 26 |
| 5.6.3.35 subtract() [1/2] | 26 |
| 5.6.3.36 subtract() [2/2] | 26 |
| 5.6.3.37 swap_cols() | 27 |
| 5.6.3.38 swap_rows() | 27 |
| 5.6.3.39 transpose() | 27 |
| 5.7 Mtx::QR_result< T > Struct Template Reference | 28 |
| 5.7.1 Detailed Description | 28 |
| 5.8 Mtx::singular_matrix_exception Class Reference | 28 |
| 6 File Documentation | 29 |
| 6.1 examples.cpp File Reference | 29 |
| 6.1.1 Detailed Description | 29 |
| 6.2 matrix.hpp File Reference | 29 |
| 6.2.1 Function Documentation | 35 |

| | |
|--|----|
| 6.2.1.1 <code>add()</code> [1/2] | 35 |
| 6.2.1.2 <code>add()</code> [2/2] | 36 |
| 6.2.1.3 <code>adj()</code> | 36 |
| 6.2.1.4 <code>cconj()</code> | 36 |
| 6.2.1.5 <code>chol()</code> | 37 |
| 6.2.1.6 <code>cholinv()</code> | 37 |
| 6.2.1.7 <code>circshift()</code> | 38 |
| 6.2.1.8 <code>circulant()</code> [1/2] | 38 |
| 6.2.1.9 <code>circulant()</code> [2/2] | 39 |
| 6.2.1.10 <code>cofactor()</code> | 39 |
| 6.2.1.11 <code>cond()</code> | 40 |
| 6.2.1.12 <code>csign()</code> | 40 |
| 6.2.1.13 <code>ctranspose()</code> | 40 |
| 6.2.1.14 <code>det()</code> | 40 |
| 6.2.1.15 <code>det_lu()</code> | 41 |
| 6.2.1.16 <code>diag()</code> [1/3] | 41 |
| 6.2.1.17 <code>diag()</code> [2/3] | 42 |
| 6.2.1.18 <code>diag()</code> [3/3] | 42 |
| 6.2.1.19 <code>div()</code> | 43 |
| 6.2.1.20 <code>eigenvalues()</code> [1/2] | 43 |
| 6.2.1.21 <code>eigenvalues()</code> [2/2] | 43 |
| 6.2.1.22 <code>eye()</code> | 44 |
| 6.2.1.23 <code>foreach_elem()</code> | 44 |
| 6.2.1.24 <code>foreach_elem_copy()</code> | 45 |
| 6.2.1.25 <code>hessenberg()</code> | 45 |
| 6.2.1.26 <code>householder_reflection()</code> | 46 |
| 6.2.1.27 <code>imag()</code> | 46 |
| 6.2.1.28 <code>inv()</code> | 47 |
| 6.2.1.29 <code>inv_gauss_jordan()</code> | 47 |
| 6.2.1.30 <code>inv_posdef()</code> | 47 |
| 6.2.1.31 <code>inv_square()</code> | 48 |
| 6.2.1.32 <code>inv_tril()</code> | 48 |
| 6.2.1.33 <code>inv_triu()</code> | 49 |
| 6.2.1.34 <code>ishess()</code> | 49 |
| 6.2.1.35 <code>istril()</code> | 50 |
| 6.2.1.36 <code>istriu()</code> | 50 |
| 6.2.1.37 <code>kron()</code> | 50 |
| 6.2.1.38 <code>ldl()</code> | 50 |
| 6.2.1.39 <code>lu()</code> | 51 |
| 6.2.1.40 <code>lup()</code> | 51 |
| 6.2.1.41 <code>make_complex()</code> [1/2] | 52 |
| 6.2.1.42 <code>make_complex()</code> [2/2] | 52 |

| | |
|-------------------------------|----|
| 6.2.1.43 mult() [1/4] | 53 |
| 6.2.1.44 mult() [2/4] | 54 |
| 6.2.1.45 mult() [3/4] | 54 |
| 6.2.1.46 mult() [4/4] | 54 |
| 6.2.1.47 mult_hadamard() | 55 |
| 6.2.1.48 norm_fro() [1/2] | 55 |
| 6.2.1.49 norm_fro() [2/2] | 56 |
| 6.2.1.50 ones() [1/2] | 56 |
| 6.2.1.51 ones() [2/2] | 56 |
| 6.2.1.52 operator!==() | 57 |
| 6.2.1.53 operator*() [1/4] | 57 |
| 6.2.1.54 operator*() [2/4] | 57 |
| 6.2.1.55 operator*() [3/4] | 58 |
| 6.2.1.56 operator*() [4/4] | 58 |
| 6.2.1.57 operator*==() [1/2] | 58 |
| 6.2.1.58 operator*==() [2/2] | 58 |
| 6.2.1.59 operator+() [1/3] | 59 |
| 6.2.1.60 operator+() [2/3] | 59 |
| 6.2.1.61 operator+() [3/3] | 59 |
| 6.2.1.62 operator+==() [1/2] | 59 |
| 6.2.1.63 operator+==() [2/2] | 60 |
| 6.2.1.64 operator-() [1/2] | 60 |
| 6.2.1.65 operator-() [2/2] | 60 |
| 6.2.1.66 operator-==() [1/2] | 60 |
| 6.2.1.67 operator-==() [2/2] | 61 |
| 6.2.1.68 operator/() | 61 |
| 6.2.1.69 operator/==() | 61 |
| 6.2.1.70 operator<<() | 61 |
| 6.2.1.71 operator==() | 62 |
| 6.2.1.72 operator^() | 62 |
| 6.2.1.73 operator^==() | 62 |
| 6.2.1.74 permute_cols() | 62 |
| 6.2.1.75 permute_rows() | 63 |
| 6.2.1.76 pinv() | 64 |
| 6.2.1.77 qr() | 64 |
| 6.2.1.78 qr_householder() | 64 |
| 6.2.1.79 qr_red_gs() | 65 |
| 6.2.1.80 real() | 66 |
| 6.2.1.81 repmat() | 66 |
| 6.2.1.82 solve_posdef() | 66 |
| 6.2.1.83 solve_square() | 67 |
| 6.2.1.84 solve_tril() | 68 |

| | |
|----------------------------|----|
| 6.2.1.85 solve_triu() | 68 |
| 6.2.1.86 subtract() [1/2] | 69 |
| 6.2.1.87 subtract() [2/2] | 70 |
| 6.2.1.88 trace() | 70 |
| 6.2.1.89 transpose() | 70 |
| 6.2.1.90 tril() | 71 |
| 6.2.1.91 triu() | 71 |
| 6.2.1.92 wilkinson_shift() | 71 |
| 6.2.1.93 zeros() [1/2] | 72 |
| 6.2.1.94 zeros() [2/2] | 72 |
| 6.3 matrix.hpp | 72 |

Chapter 1

Matrix HPP - C++ 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 datatypes.
- 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 include directory of your project.

1.2 Hello world example

A simple hello world example is provided below. The program creates two matrices with two rows and three columns, and initializes their content with constants. Then, the matrices are added and the resulting matrix is printed to stdout.

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

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

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

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

    auto C = A + B;

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

For more examples, refer to `examples.cpp` file. Remark that not all features of the library are used in the provided examples.

1.3 License

MIT license was selected for this project.

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 | 28 |
| 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 > | 28 |

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 | 28 |
| Mtx::singular_matrix_exception | |
| Singular matrix exception | 28 |

Chapter 4

File Index

4.1 File List

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

| | |
|--|----|
| examples.cpp | 29 |
| matrix.hpp | 29 |

Chapter 5

Class Documentation

5.1 Mtx::Eigenvalues_result< T > Struct Template Reference

Result of eigenvalues.

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

Public Attributes

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

5.1.1 Detailed Description

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

Result of eigenvalues.

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

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

- [matrix.hpp](#)

5.2 Mtx::Hessenberg_result< T > Struct Template Reference

Result of Hessenberg decomposition.

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

Public Attributes

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

5.2.1 Detailed Description

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

Result of Hessenberg decomposition.

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

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

- [matrix.hpp](#)

5.3 Mtx::LDL_result< T > Struct Template Reference

Result of LDL decomposition.

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

Public Attributes

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

5.3.1 Detailed Description

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

Result of LDL decomposition.

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

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

- [matrix.hpp](#)

5.4 Mtx::LU_result< T > Struct Template Reference

Result of LU decomposition.

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

Public Attributes

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

5.4.1 Detailed Description

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

Result of LU decomposition.

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

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

- [matrix.hpp](#)

5.5 Mtx::LUP_result< T > Struct Template Reference

Result of LU decomposition with pivoting.

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

Public Attributes

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

5.5.1 Detailed Description

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

Result of LU decomposition with pivoting.

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

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

- [matrix.hpp](#)

5.6 Mtx::Matrix< T > Class Template Reference

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

Public Member Functions

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

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

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

5.6.1 Detailed Description

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

[Matrix](#) class definition.

5.6.2 Constructor & Destructor Documentation

5.6.2.1 Matrix() [1/8]

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

Default constructor.

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

5.6.2.2 Matrix() [2/8]

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

Square matrix constructor.

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

5.6.2.3 Matrix() [3/8]

```
template<typename T >
Mtx::Matrix< T >::Matrix (
    unsigned n_rows,
    unsigned n_cols )
```

Rectangular matrix constructor.

Constructs a matrix of size *n_rows* x *n_cols*. 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 n_rows,
    unsigned n_cols )
```

Rectangular matrix constructor with fill.

Constructs a matrix of size *n_rows* x *n_cols*. All of the matrix elements of are set to value *x*.

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

5.6.2.5 Matrix() [5/8]

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

Rectangular matrix constructor with initialization.

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

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

5.6.2.6 Matrix() [6/8]

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

Rectangular matrix constructor with initialization.

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

Exceptions

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

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

5.6.2.7 Matrix() [7/8]

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

Rectangular matrix constructor with initialization.

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

Exceptions

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

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

5.6.2.8 Matrix() [8/8]

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

Copy constructor.

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

5.6.2.9 ~Matrix()

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

Destructor.

5.6.3 Member Function Documentation

5.6.3.1 add() [1/2]

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

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

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

Exceptions

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

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

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

5.6.3.2 add() [2/2]

```
template<typename T >
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*.

Exceptions

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

5.6.3.4 add_row_to_another()

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

Row addition.

Adds values of elements in row *from* to the elements of row *to*.

Exceptions

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

5.6.3.5 clear()

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

Clears the matrix.

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

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

5.6.3.6 col_from_vector()

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

Column from vector.

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

Exceptions

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

5.6.3.7 col_to_vector()

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

Column to vector.

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

Exceptions

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

5.6.3.8 cols()

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

Number of columns.

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

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

5.6.3.9 ctranspose()

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

Transpose a complex matrix.

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

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

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

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

5.6.3.10 div()

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

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

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

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

5.6.3.11 exists()

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

Element exist check.

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

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

5.6.3.12 fill()

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

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

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

5.6.3.13 fill_col()

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

Fill column with a scalar.

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

Exceptions

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

5.6.3.14 fill_row()

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

Fill row with a scalar.

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

Exceptions

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

5.6.3.15 get_submatrix()

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

Extract a submatrix.

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

Exceptions

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

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

5.6.3.16 isempty()

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

Emptiness check.

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

5.6.3.17 isequal() [1/2]

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

[Matrix](#) equality check.

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

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

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

5.6.3.18 isequal() [2/2]

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

[Matrix](#) equality check with tolerance.

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

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

5.6.3.19 mult()

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

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

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

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

5.6.3.20 mult_hadamard()

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

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

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

Exceptions

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

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

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

5.6.3.21 numel()

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

[Matrix](#) capacity.

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

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

5.6.3.22 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 $nrows \times ncols$ elements. Element order in the vector follow column-major format.

5.6.3.23 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 |
|--------------------------------|------------------------------------|

5.6.3.24 operator()(row, col) [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 |
|--------------------------------|---|

5.6.3.25 operator=(other) [1/2]

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

Matrix assignment.

Performs deep-copy of another matrix.

5.6.3.26 operator=(s) [2/2]

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

Matrix fill operator.

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

5.6.3.27 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 |
|--------------------------------|--|

5.6.3.28 ptr() [2/2]

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

Memory pointer.

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

5.6.3.29 reshape()

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

Matrix dimension reshape.

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

Exceptions

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

5.6.3.30 resize()

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


Resize the matrix.

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

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

5.6.3.31 row_from_vector()

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

Row from vector.

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

Exceptions

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

5.6.3.32 row_to_vector()

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

Row to vector.

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

Exceptions

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

5.6.3.33 rows()

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

Number of rows.

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

Referenced by [Mtx::Matrix< T >::add\(\)](#), [Mtx::add\(\)](#), [Mtx::add\(\)](#), [Mtx::adj\(\)](#), [Mtx::chol\(\)](#), [Mtx::cholinv\(\)](#), [Mtx::circshift\(\)](#), [Mtx::cofactor\(\)](#), [Mtx::det\(\)](#), [Mtx::diag\(\)](#), [Mtx::div\(\)](#), [Mtx::eigenvalues\(\)](#), [Mtx::hessenberg\(\)](#), [Mtx::inv\(\)](#), [Mtx::inv_gauss_jordan\(\)](#),

[Mtx::inv_tril\(\)](#), [Mtx::inv_triu\(\)](#), [Mtx::Matrix< T >::isequal\(\)](#), [Mtx::Matrix< T >::isequal\(\)](#), [Mtx::ishess\(\)](#), [Mtx::istril\(\)](#), [Mtx::istriu\(\)](#), [Mtx::kron\(\)](#), [Mtx::ldl\(\)](#), [Mtx::lu\(\)](#), [Mtx::lup\(\)](#), [Mtx::make_complex\(\)](#), [Mtx::make_complex\(\)](#), [Mtx::mult\(\)](#), [Mtx::mult\(\)](#), [Mtx::mult\(\)](#), [Mtx::Matrix< T >::mult_hadamard\(\)](#), [Mtx::mult_hadamard\(\)](#), [Mtx::operator<<\(\)](#), [Mtx::permute_cols\(\)](#), [Mtx::permute_rows\(\)](#), [Mtx::qr_householder\(\)](#), [Mtx::qr_red_gs\(\)](#), [Mtx::repmat\(\)](#), [Mtx::Matrix< T >::set_submatrix\(\)](#), [Mtx::solve_posdef\(\)](#), [Mtx::solve_square\(\)](#), [Mtx::solve_tril\(\)](#), [Mtx::solve_triu\(\)](#), [Mtx::Matrix< T >::subtract\(\)](#), [Mtx::subtract\(\)](#), [Mtx::subtract\(\)](#), [Mtx::trace\(\)](#), [Mtx::tril\(\)](#), and [Mtx::triu\(\)](#).

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

5.6.3.35 subtract() [1/2]

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

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

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

Exceptions

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

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

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

5.6.3.36 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.

5.6.3.37 swap_cols()

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

Column swap.

Swaps element values between two columns.

Exceptions

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

5.6.3.38 swap_rows()

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

Row swap.

Swaps element values of two columns.

Exceptions

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

5.6.3.39 transpose()

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

Transpose a matrix.

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

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

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

- [matrix.hpp](#)

5.7 Mtx::QR_result< T > Struct Template Reference

Result of QR decomposition.

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

Public Attributes

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

5.7.1 Detailed Description

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

Result of QR decomposition.

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

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

- [matrix.hpp](#)

5.8 Mtx::singular_matrix_exception Class Reference

Singular matrix exception.

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

Inheritance diagram for Mtx::singular_matrix_exception:

Chapter 6

File Documentation

6.1 examples.cpp File Reference

6.1.1 Detailed Description

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

6.2 matrix.hpp File Reference

Classes

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

Functions

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

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

- Matrix subtraction.*

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

- Matrix product with scalar.*

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

Matrix sum.

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

Matrix subtraction.

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

Matrix product.

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

Matrix Hadamard product.

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

Matrix sum with scalar.

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

Matrix subtraction with scalar.

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

Matrix product with scalar.

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

Matrix division by scalar.

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

Matrix equality check operator.

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

Matrix non-equality check operator.

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

Kronecker product.

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

Adjugate matrix.

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

Cofactor matrix.

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

Matrix determinant from on LU decomposition.

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

Matrix determinant.

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

LU decomposition.

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

LU decomposition with pivoting.

- `template<typename T >`
`Matrix< T > Mtx::inv_gauss_jordan (const Matrix< T > &A)`
Matrix inverse using Gauss-Jordan elimination.
- `template<typename T >`
`Matrix< T > Mtx::inv_tril (const Matrix< T > &A)`
Matrix inverse for lower triangular matrix.
- `template<typename T >`
`Matrix< T > Mtx::inv_triu (const Matrix< T > &A)`
Matrix inverse for upper triangular matrix.
- `template<typename T >`
`Matrix< T > Mtx::inv_posdef (const Matrix< T > &A)`
Matrix inverse for Hermitian positive-definite matrix.
- `template<typename T >`
`Matrix< T > Mtx::inv_square (const Matrix< T > &A)`
Matrix inverse for general square matrix.
- `template<typename T >`
`Matrix< T > Mtx::inv (const Matrix< T > &A)`
Matrix inverse (universal).
- `template<typename T >`
`Matrix< T > Mtx::pinv (const Matrix< T > &A)`
Moore-Penrose 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 >`
`Matrix< T > Mtx::chol (const Matrix< T > &A)`
Cholesky decomposition.
- `template<typename T >`
`Matrix< T > Mtx::cholinv (const Matrix< T > &A)`
Inverse of Cholesky decomposition.
- `template<typename T >`
`LDL_result< T > Mtx::ldl (const Matrix< T > &A)`
LDL decomposition.
- `template<typename T >`
`QR_result< T > Mtx::qr_red_gs (const Matrix< T > &A)`
Reduced QR decomposition based on Gram-Schmidt method.
- `template<typename T >`
`Matrix< T > Mtx::householder_reflection (const Matrix< T > &a)`
Generate Householder reflection.
- `template<typename T >`
`QR_result< T > Mtx::qr_householder (const Matrix< T > &A, bool calculate_Q=true)`
QR decomposition based on Householder method.
- `template<typename T >`
`QR_result< T > Mtx::qr (const Matrix< T > &A, bool calculate_Q=true)`
QR decomposition.
- `template<typename T >`
`Hessenberg_result< T > Mtx::hessenberg (const Matrix< T > &A, bool calculate_Q=true)`
Hessenberg decomposition.
- `template<typename T >`
`std::complex< T > Mtx::wilkinson_shift (const Matrix< std::complex< T > > &H, T tol=1e-10)`

Wilkinson's shift for complex eigenvalues.

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

Matrix eigenvalues of complex matrix.

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

Matrix eigenvalues of real matrix.

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

Solves the upper triangular system.

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

Solves the lower triangular system.

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

Solves the square system.

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

Solves the positive definite (Hermitian) system.

6.2.1 Function Documentation

6.2.1.1 add() [1/2]

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

Matrix addition.

Performs addition of two matrices.

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

Template Parameters

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

Parameters

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

Returns

output matrix of size $N \times M$

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

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

6.2.1.2 add() [2/2]

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

Addition of scalar to matrix.

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

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

6.2.1.3 adj()

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

Adjugate matrix.

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

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

Exceptions

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

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

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

6.2.1.4 cconj()

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

Complex conjugate helper.

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

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

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

6.2.1.5 chol()

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

Cholesky decomposition.

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

$$A = LL^H$$

where L is a lower triangular matrix with real and positive diagonal entries, and L^H denotes the conjugate transpose of L .

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

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

Exceptions

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

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

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

6.2.1.6 cholinv()

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

Inverse of Cholesky decomposition.

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

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

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

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

Exceptions

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

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

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

6.2.1.7 circshift()

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

Circular shift.

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

Parameters

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

Returns

matrix inverse

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

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

6.2.1.8 circulant() [1/2]

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

Circulant matrix from std::vector.

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

Parameters

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

Returns

circulant matrix

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

6.2.1.9 `circulant()` [2/2]

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

Circulant matrix from array.

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

Parameters

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

Returns

circulant matrix

References `Mtx::circulant()`.

Referenced by `Mtx::circulant()`, and `Mtx::circulant()`.

6.2.1.10 `cofactor()`

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

Cofactor matrix.

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

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

Parameters

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

Exceptions

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

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

6.2.1.11 cond()

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

Condition number of a matrix.

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

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

Frobenius norm is used for the sake of calculations.

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

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

6.2.1.12 csign()

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

Complex sign helper.

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

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

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

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

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

6.2.1.13 ctranspose()

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

Transpose a complex matrix.

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

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

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

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

6.2.1.14 det()

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

Matrix determinant.

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

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

Exceptions

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

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

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

6.2.1.15 det_lu()

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

Matrix determinant from on LU decomposition.

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

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

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

Exceptions

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

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

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

6.2.1.16 diag() [1/3]

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

Diagonal extraction.

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

Parameters

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

Returns

vector of diagonal elements

Exceptions

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

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

6.2.1.17 diag() [2/3]

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

Diagonal matrix from `std::vector`.

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

Parameters

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

Returns

diagonal matrix

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

6.2.1.18 diag() [3/3]

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

Diagonal matrix from array.

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

Parameters

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

Returns

diagonal matrix

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

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

6.2.1.19 div()

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

Division of matrix by scalar.

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

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

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

6.2.1.20 eigenvalues() [1/2]

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

Matrix eigenvalues of complex matrix.

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

Parameters

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

Returns

structure containing the result and status of eigenvalue calculation

Exceptions

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

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

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

6.2.1.21 eigenvalues() [2/2]

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

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

Matrix eigenvalues of real matrix.

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

Parameters

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

Returns

structure containing the result and status of eigenvalue calculation

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

6.2.1.22 eye()

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

Identity matrix.

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

Parameters

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

Returns

zeros matrix

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

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

6.2.1.23 foreach_elem()

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

Applies custom function element-wise in-place.

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

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

Parameters

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

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

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

6.2.1.24 foreach_elem_copy()

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

Applies custom function element-wise with matrix copy.

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

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

Parameters

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

Returns

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

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

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

6.2.1.25 hessenberg()

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

Hessenberg decomposition.

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

Parameters

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

Returns

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

Exceptions

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

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

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

6.2.1.26 householder_reflection()

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

Generate Householder reflection.

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

Parameters

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

Returns

column vector with Householder reflection of *a*

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

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

6.2.1.27 imag()

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

Get imaginary part of complex matrix.

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

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

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

6.2.1.28 inv()

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

Matrix inverse (universal).

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

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

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

Exceptions

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

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

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

6.2.1.29 inv_gauss_jordan()

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

Matrix inverse using Gauss-Jordan elimination.

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

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

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

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

Exceptions

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

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

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

6.2.1.30 inv_posdef()

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

Matrix inverse for Hermitian positive-definite matrix.

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

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

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

Exceptions

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

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

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

6.2.1.31 inv_square()

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

Matrix inverse for general square matrix.

Calculates an inverse of square matrix using matrix.

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

Exceptions

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

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

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

6.2.1.32 inv_tril()

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

Matrix inverse for lower triangular matrix.

Calculates an inverse of lower triangular matrix.

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

Exceptions

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

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

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

6.2.1.33 inv_triu()

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

Matrix inverse for upper triangular matrix.

Calculates an inverse of upper triangular matrix.

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

Exceptions

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

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

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

6.2.1.34 ishess()

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

Hessenberg matrix check.

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

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

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

6.2.1.35 istril()

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

Lower triangular matrix check.

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

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

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

6.2.1.36 istriu()

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

Lower triangular matrix check.

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

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

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

6.2.1.37 kron()

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

Kronecker product.

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

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

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

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

6.2.1.38 ldl()

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

LDL decomposition.

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

$$A = LDL^H$$

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

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

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

Parameters

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

Returns

structure encapsulating calculated L and D

Exceptions

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

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

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

6.2.1.39 lu()

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

LU decomposition.

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

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

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

Parameters

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

Returns

structure containing calculated L and U matrices

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

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

6.2.1.40 lup()

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

LU decomposition with pivoting.

Performs LU factorization with partial pivoting, employing column permutations.

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

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

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

Parameters

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

Returns

structure containing L , U and P .

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

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

6.2.1.41 make_complex() [1/2]

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

Create complex matrix from real matrix.

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

Parameters

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

Returns

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

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

6.2.1.42 make_complex() [2/2]

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

Create complex matrix from real and imaginary matrices.

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

Parameters

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

Returns

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

Exceptions

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

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

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

6.2.1.43 mult() [1/4]

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

Matrix multiplication.

Performs multiplication of two matrices.

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

Template Parameters

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

Parameters

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

Returns

output matrix of size $N \times K$

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

6.2.1.44 mult() [2/4]

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

Multiplication of matrix by std::vector.

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

Parameters

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

Returns

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

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

6.2.1.45 mult() [3/4]

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

Multiplication of matrix by scalar.

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

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

6.2.1.46 mult() [4/4]

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

Multiplication of std::vector by matrix.

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

Parameters

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

Returns

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

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

6.2.1.47 mult_hadamard()

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

Matrix Hadamard (elementwise) multiplication.

Performs Hadamard (elementwise) multiplication of two matrices.

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

Template Parameters

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

Parameters

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

Returns

output matrix of size $N \times M$

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

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

6.2.1.48 norm_fro() [1/2]

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

Frobenius norm of complex matrix.

Calculates Frobenius norm of complex matrix.

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

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

6.2.1.49 norm_fro() [2/2]

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

Frobenius norm.

Calculates Frobenius norm of real matrix.

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

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

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

6.2.1.50 ones() [1/2]

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

Square matrix of ones.

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

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

Parameters

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

Returns

zeros matrix

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

6.2.1.51 ones() [2/2]

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

Matrix of ones.

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

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

Parameters

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

Returns

ones matrix

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

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

6.2.1.52 operator!=(())

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

Matrix non-equality check operator.

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

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

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

6.2.1.53 operator*() [1/4]

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

Matrix product.

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

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

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

6.2.1.54 operator*() [2/4]

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

Matrix and std::vector product.

Calculates product between a matrix and a std::vector $A \cdot v$.

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

6.2.1.55 operator*() [3/4]

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

Matrix product with scalar.

Multiplies each element of the matrix by a scalar s .

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

6.2.1.56 operator*() [4/4]

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

Matrix product with scalar.

Multiplies each element of the matrix by a scalar s .

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

6.2.1.57 operator*=() [1/2]

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

Matrix product.

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

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

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

6.2.1.58 operator*=() [2/2]

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

Matrix product with scalar.

Multiplies each element of the matrix by a scalar s .

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

6.2.1.59 operator+() [1/3]

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

Matrix sum.

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

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

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

6.2.1.60 operator+() [2/3]

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

Matrix sum with scalar.

Adds a scalar s to each element of the matrix.

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

6.2.1.61 operator+() [3/3]

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

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

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

6.2.1.62 operator+=() [1/2]

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

Matrix sum.

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

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

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

6.2.1.63 operator+=() [2/2]

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

Matrix sum with scalar.

Adds a scalar s to each element of the matrix.

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

6.2.1.64 operator-() [1/2]

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

Matrix subtraction.

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

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

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

6.2.1.65 operator-() [2/2]

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

Matrix subtraction with scalar.

Subtracts a scalar s from each element of the matrix.

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

6.2.1.66 operator-=() [1/2]

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

Matrix subtraction.

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

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

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

6.2.1.67 operator-=() [2/2]

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

Matrix subtraction with scalar.

Subtracts a scalar s from each element of the matrix.

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

6.2.1.68 operator/()

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

Matrix division by scalar.

Divides each element of the matrix by a scalar s .

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

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

6.2.1.69 operator/=()

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

Matrix division by scalar.

Divides each element of the matrix by a scalar s .

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

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

6.2.1.70 operator<<()

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

Matrix ostream operator.

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

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

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

6.2.1.71 operator==()

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

Matrix equality check operator.

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

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

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

6.2.1.72 operator^()

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

Matrix Hadamard product.

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

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

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

6.2.1.73 operator^=()

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

Matrix Hadamard product.

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

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

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

6.2.1.74 permute_cols()

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

Permute columns of the matrix.

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

Parameters

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

Returns

output matrix created by column permutation of *A*

Exceptions

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

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

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

6.2.1.75 permute_rows()

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

Permute rows of the matrix.

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

Parameters

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

Returns

output matrix created by row permutation of *A*

Exceptions

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

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

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

6.2.1.76 pinv()

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

Moore-Penrose pseudo inverse.

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

$A^+ = (A'A)^{-1}A'$

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

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

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

6.2.1.77 qr()

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

QR decomposition.

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

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

Parameters

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

Returns

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

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

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

6.2.1.78 qr_householder()

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


QR decomposition based on Householder method.

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

This function implements QR decomposition based on Householder reflections method.

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

Parameters

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

Returns

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

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

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

6.2.1.79 qr_red_gs()

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

Reduced QR decomposition based on Gram-Schmidt method.

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

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

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

Parameters

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

Returns

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

Exceptions

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

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

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

6.2.1.80 real()

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

Get real part of complex matrix.

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

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

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

6.2.1.81 repmat()

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

Repeat matrix.

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

Parameters

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

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

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

6.2.1.82 solve_posdef()

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

Solves the positive definite (Hermitian) system.

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

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

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

Parameters

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

Returns

solution matrix of size $N \times M$.

Exceptions

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

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

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

6.2.1.83 solve_square()

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

Solves the square system.

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

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

Parameters

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

Returns

solution matrix of size $N \times M$.

Exceptions

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

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

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

6.2.1.84 solve_tril()

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

Solves the lower triangular system.

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

Parameters

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

Returns

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

Exceptions

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

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

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

6.2.1.85 solve_triu()

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

Solves the upper triangular system.

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

Parameters

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

Returns

solution matrix of size $N \times M$.

Exceptions

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

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

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

6.2.1.86 subtract() [1/2]

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

Matrix subtraction.

Performs subtraction of two matrices.

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

Template Parameters

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

Parameters

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

Returns

output matrix of size $N \times M$

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

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

6.2.1.87 subtract() [2/2]

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

Subtraction of scalar from matrix.

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

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

6.2.1.88 trace()

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

Matrix trace.

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

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

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

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

6.2.1.89 transpose()

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

Transpose a matrix.

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

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

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

6.2.1.90 tril()

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

Extract triangular lower part.

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

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

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

6.2.1.91 triu()

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

Extract triangular upper part.

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

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

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

6.2.1.92 wilkinson_shift()

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

Wilkinson's shift for complex eigenvalues.

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

Exceptions

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

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

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

6.2.1.93 zeros() [1/2]

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

Square matrix of zeros.

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

Parameters

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

Returns

zeros matrix

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

6.2.1.94 zeros() [2/2]

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

Matrix of zeros.

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

Parameters

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

Returns

zeros matrix

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

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

6.3 matrix.hpp

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

00001
00002


```

00003 #ifndef __MATRIX_HPP__
00004 #define __MATRIX_HPP__
00005
00006 #include <ostream>
00007 #include <complex>
00008 #include <vector>
00009 #include <initializer_list>
00010 #include <limits>
00011 #include <functional>
00012 #include <algorithm>
00013
00014 namespace Mtx {
00015
00016 template<typename T> class Matrix;
00017
00018 template<class T> struct is_complex : std::false_type {};
00019 template<class T> struct is_complex<std::complex<T> > : std::true_type {};
00020
00027 template<typename T, typename std::enable_if<!is_complex<T>::value,int>::type = 0>
00028 inline T cconj(T x) {
00029     return x;
00030 }
00031
00032 template<typename T, typename std::enable_if<is_complex<T>::value,int>::type = 0>
00033 inline T cconj(T x) {
00034     return std::conj(x);
00035 }
00036
00043 template<typename T, typename std::enable_if<!is_complex<T>::value,int>::type = 0>
00044 inline T csign(T x) {
00045     return (x > static_cast<T>(0)) ? static_cast<T>(1) : static_cast<T>(-1);
00046 }
00047
00048 template<typename T, typename std::enable_if<is_complex<T>::value,int>::type = 0>
00049 inline T csign(T x) {
00050     auto x_arg = std::arg(x);
00051     T y(0, x_arg);
00052     return std::exp(y);
00053 }
00054
00062 class singular_matrix_exception : public std::domain_error {
00063 public:
00064     singular_matrix_exception(const std::string& message) : std::domain_error(message) {}
00065 };
00066
00071 template<typename T>
00072 struct LU_result {
00073     Matrix<T> L;
00074
00075     Matrix<T> U;
00076 };
00077
00086 template<typename T>
00087 struct LUP_result {
00088     Matrix<T> L;
00089
00090     Matrix<T> U;
00091
00092     std::vector<unsigned> P;
00093 };
00094
00106 template<typename T>
00107 struct QR_result {
00108     Matrix<T> Q;
00109
00110     Matrix<T> R;
00111 };
00112
00121 template<typename T>
00122 struct Hessenberg_result {
00123     Matrix<T> H;
00124
00125     Matrix<T> Q;
00126 };
00127
00136 template<typename T>
00137 struct LDL_result {
00138     Matrix<T> L;
00139
00140     std::vector<T> d;
00141 };
00142
00151 template<typename T>
00152 struct Eigenvalues_result {
00153     std::vector<std::complex<T> > eig;
00154
00155     bool converged;
00156
00157 };

```

```

00160
00163     T err;
00164 };
00165
00166
00174 template<typename T>
00175 inline Matrix<T> zeros(unsigned nrows, unsigned ncols) {
00176     return Matrix<T>(static_cast<T>(0), nrows, ncols);
00177 }
00178
00185 template<typename T>
00186 inline Matrix<T> zeros(unsigned n) {
00187     return zeros<T>(n,n);
00188 }
00189
00198 template<typename T>
00199 inline Matrix<T> ones(unsigned nrows, unsigned ncols) {
00200     return Matrix<T>(static_cast<T>(1), nrows, ncols);
00201 }
00202
00210 template<typename T>
00211 inline Matrix<T> ones(unsigned n) {
00212     return ones<T>(n,n);
00213 }
00214
00222 template<typename T>
00223 Matrix<T> eye(unsigned n) {
00224     Matrix<T> A(static_cast<T>(0), n, n);
00225     for (unsigned i = 0; i < n; i++)
00226         A(i,i) = static_cast<T>(1);
00227     return A;
00228 }
00229
00237 template<typename T>
00238 Matrix<T> diag(const T* array, size_t n) {
00239     Matrix<T> A(static_cast<T>(0), n, n);
00240     for (unsigned i = 0; i < n; i++) {
00241         A(i,i) = array[i];
00242     }
00243     return A;
00244 }
00245
00253 template<typename T>
00254 inline Matrix<T> diag(const std::vector<T>& v) {
00255     return diag(v.data(), v.size());
00256 }
00257
00266 template<typename T>
00267 std::vector<T> diag(const Matrix<T>& A) {
00268     if (! A.issquare()) throw std::runtime_error("Input matrix is not square");
00269
00270     std::vector<T> v;
00271     v.resize(A.rows());
00272
00273     for (unsigned i = 0; i < A.rows(); i++)
00274         v[i] = A(i,i);
00275     return v;
00276 }
00277
00285 template<typename T>
00286 Matrix<T> circulant(const T* array, unsigned n) {
00287     Matrix<T> A(n, n);
00288     for (unsigned j = 0; j < n; j++)
00289         for (unsigned i = 0; i < n; i++)
00290             A((i+j) % n, j) = array[i];
00291     return A;
00292 }
00293
00304 template<typename T>
00305 Matrix<std::complex<T>> make_complex(const Matrix<T>& Re, const Matrix<T>& Im) {
00306     if (Re.rows() != Im.rows() || Re.cols() != Im.cols()) throw std::runtime_error("Size of input
matrices does not match");
00307
00308     Matrix<std::complex<T>> C(Re.rows(), Re.cols());
00309     for (unsigned n = 0; n < Re.numel(); n++) {
00310         C(n).real(Re(n));
00311         C(n).imag(Im(n));
00312     }
00313
00314     return C;
00315 }
00316
00323 template<typename T>
00324 Matrix<std::complex<T>> make_complex(const Matrix<T>& Re) {
00325     Matrix<std::complex<T>> C(Re.rows(), Re.cols());
00326
00327     for (unsigned n = 0; n < Re.numel(); n++) {

```

```

00328     C(n).real(Re(n));
00329     C(n).imag(static_cast<T>(0));
00330 }
00331
00332     return C;
00333 }
00334
00335 template<typename T>
00336 Matrix<T> real(const Matrix<std::complex<T>& C) {
00337     Matrix<T> Re(C.rows(), C.cols());
00338
00339     for (unsigned n = 0; n < C.numel(); n++)
00340         Re(n) = C(n).real();
00341
00342     return Re;
00343 }
00344
00345 template<typename T>
00346 Matrix<T> imag(const Matrix<std::complex<T>& C) {
00347     Matrix<T> Re(C.rows(), C.cols());
00348
00349     for (unsigned n = 0; n < C.numel(); n++)
00350         Re(n) = C(n).imag();
00351
00352     return Re;
00353 }
00354
00355 template<typename T>
00356 inline Matrix<T> circulant(const std::vector<T>& v) {
00357     return circulant(v.data(), v.size());
00358 }
00359
00360 template<typename T>
00361 inline Matrix<T> transpose(const Matrix<T>& A) {
00362     return A.transpose();
00363 }
00364
00365 template<typename T>
00366 inline Matrix<T> ctranspose(const Matrix<T>& A) {
00367     return A.ctranspose();
00368 }
00369
00370 template<typename T>
00371 Matrix<T> circshift(const Matrix<T>& A, int row_shift, int col_shift) {
00372     Matrix<T> B(A.rows(), A.cols());
00373     for (int i = 0; i < A.rows(); i++) {
00374         int ii = (i + row_shift) % A.rows();
00375         for (int j = 0; j < A.cols(); j++) {
00376             int jj = (j + col_shift) % A.cols();
00377             B(ii, jj) = A(i, j);
00378         }
00379     }
00380     return B;
00381 }
00382
00383 template<typename T>
00384 Matrix<T> repmat(const Matrix<T>& A, unsigned m, unsigned n) {
00385     Matrix<T> B(m * A.rows(), n * A.cols());
00386
00387     for (unsigned cb = 0; cb < n; cb++)
00388         for (unsigned rb = 0; rb < m; rb++)
00389             for (unsigned c = 0; c < A.cols(); c++)
00390                 for (unsigned r = 0; r < A.rows(); r++)
00391                     B(rb*A.rows() + r, cb*A.cols() + c) = A(r, c);
00392
00393     return B;
00394 }
00395
00396 template<typename T>
00397 double norm_fro(const Matrix<T>& A) {
00398     double sum = 0;
00399
00400     for (unsigned i = 0; i < A.numel(); i++)
00401         sum += A(i) * A(i);
00402
00403     return std::sqrt(sum);
00404 }
00405
00406 template<typename T>
00407 double norm_fro(const Matrix<std::complex<T>>& A) {
00408     double sum = 0;
00409
00410     for (unsigned i = 0; i < A.numel(); i++) {
00411         T x = std::abs(A(i));
00412         sum += x * x;
00413     }
00414
00415     return sum;
00416 }

```

```

00466     return std::sqrt(sum);
00467 }
00468
00473 template<typename T>
00474 Matrix<T> tril(const Matrix<T>& A) {
00475     Matrix<T> B(A);
00476
00477     for (unsigned row = 0; row < B.rows(); row++)
00478         for (unsigned col = row+1; col < B.cols(); col++)
00479             B(row,col) = 0;
00480
00481     return B;
00482 }
00483
00488 template<typename T>
00489 Matrix<T> triu(const Matrix<T>& A) {
00490     Matrix<T> B(A);
00491
00492     for (unsigned col = 0; col < B.cols(); col++)
00493         for (unsigned row = col+1; row < B.rows(); row++)
00494             B(row,col) = 0;
00495
00496     return B;
00497 }
00498
00504 template<typename T>
00505 bool istril(const Matrix<T>& A) {
00506     for (unsigned row = 0; row < A.rows(); row++)
00507         for (unsigned col = row+1; col < A.cols(); col++)
00508             if (A(row,col) != static_cast<T>(0)) return false;
00509     return true;
00510 }
00511
00517 template<typename T>
00518 bool istriu(const Matrix<T>& A) {
00519     for (unsigned col = 0; col < A.cols(); col++)
00520         for (unsigned row = col+1; row < A.rows(); row++)
00521             if (A(row,col) != static_cast<T>(0)) return false;
00522     return true;
00523 }
00524
00530 template<typename T>
00531 bool ishess(const Matrix<T>& A) {
00532     if (!A.issquare())
00533         return false;
00534     for (unsigned row = 2; row < A.rows(); row++)
00535         for (unsigned col = 0; col < row-2; col++)
00536             if (A(row,col) != static_cast<T>(0)) return false;
00537     return true;
00538 }
00539
00548 template<typename T>
00549 inline void foreach_elem(Matrix<T>& A, std::function<T(T)> func) {
00550     for (unsigned i = 0; i < A.numel(); i++)
00551         A(i) = func(A(i));
00552 }
00553
00562 template<typename T>
00563 inline Matrix<T> foreach_elem_copy(const Matrix<T>& A, std::function<T(T)> func) {
00564     Matrix<T> B(A);
00565     foreach_elem(B, func);
00566     return B;
00567 }
00568
00581 template<typename T>
00582 Matrix<T> permute_rows(const Matrix<T>& A, const std::vector<unsigned> perm) {
00583     if (perm.empty()) throw std::runtime_error("Permutation vector is empty");
00584
00585     Matrix<T> B(perm.size(), A.cols());
00586
00587     for (unsigned p = 0; p < perm.size(); p++) {
00588         if (!{perm[p] < A.rows()}) throw std::out_of_range("Index in permutation vector out of range");
00589
00590         for (unsigned c = 0; c < A.cols(); c++)
00591             B(p,c) = A(perm[p],c);
00592     }
00593
00594     return B;
00595 }
00596
00609 template<typename T>
00610 Matrix<T> permute_cols(const Matrix<T>& A, const std::vector<unsigned> perm) {
00611     if (perm.empty()) throw std::runtime_error("Permutation vector is empty");
00612
00613     Matrix<T> B(A.rows(), perm.size());
00614
00615     for (unsigned p = 0; p < perm.size(); p++) {

```

```

00616         if (!(perm[p] < A.cols())) throw std::out_of_range("Index in permutation vector out of range");
00617
00618         for (unsigned r = 0; r < A.rows(); r++)
00619             B(r,p) = A(r,perm[p]);
00620     }
00621
00622     return B;
00623 }
00624
00639 template<typename T, bool transpose_first = false, bool transpose_second = false>
00640 Matrix<T> mult(const Matrix<T>& A, const Matrix<T>& B) {
00641     // Adjust dimensions based on transpositions
00642     unsigned rows_A = transpose_first ? A.cols() : A.rows();
00643     unsigned cols_A = transpose_first ? A.rows() : A.cols();
00644     unsigned rows_B = transpose_second ? B.cols() : B.rows();
00645     unsigned cols_B = transpose_second ? B.rows() : B.cols();
00646
00647     if (cols_A != rows_B) throw std::runtime_error("Unmatching matrix dimensions for mult");
00648
00649     Matrix<T> C(static_cast<T>(0), rows_A, cols_B);
00650
00651     for (unsigned i = 0; i < rows_A; i++)
00652         for (unsigned j = 0; j < cols_B; j++)
00653             for (unsigned k = 0; k < cols_A; k++)
00654                 C(i,j) += (transpose_first ? cconj(A(k,i)) : A(i,k)) *
00655                     (transpose_second ? cconj(B(j,k)) : B(k,j));
00656
00657     return C;
00658 }
00659
00674 template<typename T, bool transpose_first = false, bool transpose_second = false>
00675 Matrix<T> mult_hadamard(const Matrix<T>& A, const Matrix<T>& B) {
00676     // Adjust dimensions based on transpositions
00677     unsigned rows_A = transpose_first ? A.cols() : A.rows();
00678     unsigned cols_A = transpose_first ? A.rows() : A.cols();
00679     unsigned rows_B = transpose_second ? B.cols() : B.rows();
00680     unsigned cols_B = transpose_second ? B.rows() : B.cols();
00681
00682     if ((rows_A != rows_B) || (cols_A != cols_B)) throw std::runtime_error("Unmatching matrix dimensions
for mult_hadamard");
00683
00684     Matrix<T> C(static_cast<T>(0), rows_A, cols_A);
00685
00686     for (unsigned i = 0; i < rows_A; i++)
00687         for (unsigned j = 0; j < cols_A; j++)
00688             C(i,j) += (transpose_first ? cconj(A(j,i)) : A(i,j)) *
00689                 (transpose_second ? cconj(B(j,i)) : B(i,j));
00690
00691     return C;
00692 }
00693
00708 template<typename T, bool transpose_first = false, bool transpose_second = false>
00709 Matrix<T> add(const Matrix<T>& A, const Matrix<T>& B) {
00710     // Adjust dimensions based on transpositions
00711     unsigned rows_A = transpose_first ? A.cols() : A.rows();
00712     unsigned cols_A = transpose_first ? A.rows() : A.cols();
00713     unsigned rows_B = transpose_second ? B.cols() : B.rows();
00714     unsigned cols_B = transpose_second ? B.rows() : B.cols();
00715
00716     if ((rows_A != rows_B) || (cols_A != cols_B)) throw std::runtime_error("Unmatching matrix dimensions
for add");
00717
00718     Matrix<T> C(static_cast<T>(0), rows_A, cols_A);
00719
00720     for (unsigned i = 0; i < rows_A; i++)
00721         for (unsigned j = 0; j < cols_A; j++)
00722             C(i,j) += (transpose_first ? cconj(A(j,i)) : A(i,j)) +
00723                 (transpose_second ? cconj(B(j,i)) : B(i,j));
00724
00725     return C;
00726 }
00727
00742 template<typename T, bool transpose_first = false, bool transpose_second = false>
00743 Matrix<T> subtract(const Matrix<T>& A, const Matrix<T>& B) {
00744     // Adjust dimensions based on transpositions
00745     unsigned rows_A = transpose_first ? A.cols() : A.rows();
00746     unsigned cols_A = transpose_first ? A.rows() : A.cols();
00747     unsigned rows_B = transpose_second ? B.cols() : B.rows();
00748     unsigned cols_B = transpose_second ? B.rows() : B.cols();
00749
00750     if ((rows_A != rows_B) || (cols_A != cols_B)) throw std::runtime_error("Unmatching matrix dimensions
for subtract");
00751
00752     Matrix<T> C(static_cast<T>(0), rows_A, cols_A);
00753
00754     for (unsigned i = 0; i < rows_A; i++)
00755         for (unsigned j = 0; j < cols_A; j++)

```

```

00756         C(i,j) += (transpose_first ? cconj(A(j,i)) : A(i,j)) -
00757                     (transpose_second ? cconj(B(j,i)) : B(i,j));
00758
00759     return C;
00760 }
00761
00770 template<typename T>
00771 std::vector<T> mult(const Matrix<T>& A, const std::vector<T>& v) {
00772     if (A.cols() != v.size()) throw std::runtime_error("Unmatching matrix dimensions for mult");
00773
00774     std::vector<T> u(A.rows(), static_cast<T>(0));
00775     for (unsigned r = 0; r < A.rows(); r++)
00776         for (unsigned c = 0; c < A.cols(); c++)
00777             u[r] += v[c] * A(r,c);
00778     return u;
00779 }
00780
00789 template<typename T>
00790 std::vector<T> mult(const std::vector<T>& v, const Matrix<T>& A) {
00791     if (A.rows() != v.size()) throw std::runtime_error("Unmatching matrix dimensions for mult");
00792
00793     std::vector<T> u(A.rows(), static_cast<T>(0));
00794     for (unsigned c = 0; c < A.cols(); c++)
00795         for (unsigned r = 0; r < A.rows(); r++)
00796             u[c] += v[r] * A(r,c);
00797     return u;
00798 }
00799
00805 template<typename T>
00806 Matrix<T> add(const Matrix<T>& A, T s) {
00807     Matrix<T> B(A.rows(), A.cols());
00808     for (unsigned i = 0; i < A.numel(); i++)
00809         B(i) = A(i) + s;
00810     return B;
00811 }
00812
00818 template<typename T>
00819 Matrix<T> subtract(const Matrix<T>& A, T s) {
00820     Matrix<T> B(A.rows(), A.cols());
00821     for (unsigned i = 0; i < A.numel(); i++)
00822         B(i) = A(i) - s;
00823     return B;
00824 }
00825
00831 template<typename T>
00832 Matrix<T> mult(const Matrix<T>& A, T s) {
00833     Matrix<T> B(A.rows(), A.cols());
00834     for (unsigned i = 0; i < A.numel(); i++)
00835         B(i) = A(i) * s;
00836     return B;
00837 }
00838
00844 template<typename T>
00845 Matrix<T> div(const Matrix<T>& A, T s) {
00846     Matrix<T> B(A.rows(), A.cols());
00847     for (unsigned i = 0; i < A.numel(); i++)
00848         B(i) = A(i) / s;
00849     return B;
00850 }
00851
00857 template<typename T>
00858 std::ostream& operator<<(std::ostream& os, const Matrix<T>& A) {
00859     for (unsigned row = 0; row < A.rows(); row++) {
00860         for (unsigned col = 0; col < A.cols(); col++)
00861             os << A(row,col) << " ";
00862         if (row < static_cast<unsigned>(A.rows()-1)) os << std::endl;
00863     }
00864     return os;
00865 }
00866
00871 template<typename T>
00872 inline Matrix<T> operator+(const Matrix<T>& A, const Matrix<T>& B) {
00873     return add(A,B);
00874 }
00875
00880 template<typename T>
00881 inline Matrix<T> operator-(const Matrix<T>& A, const Matrix<T>& B) {
00882     return subtract(A,B);
00883 }
00884
00890 template<typename T>
00891 inline Matrix<T> operator^(const Matrix<T>& A, const Matrix<T>& B) {
00892     return mult_hadamard(A,B);
00893 }
00894
00899 template<typename T>
00900 inline Matrix<T> operator*(const Matrix<T>& A, const Matrix<T>& B) {

```

```

00901     return mult (A,B);
00902 }
00903
00904 template<typename T>
00905 inline std::vector<T> operator*(const Matrix<T>& A, const std::vector<T>& v) {
00906     return mult (A,v);
00907 }
00908
00909 template<typename T>
00910 inline Matrix<T> operator+(const Matrix<T>& A, T s) {
00911     return add(A,s);
00912 }
00913
00914 template<typename T>
00915 inline Matrix<T> operator-(const Matrix<T>& A, T s) {
00916     return subtract(A,s);
00917 }
00918
00919 template<typename T>
00920 inline Matrix<T> operator*(const Matrix<T>& A, T s) {
00921     return mult (A,s);
00922 }
00923
00924 template<typename T>
00925 inline Matrix<T> operator/(const Matrix<T>& A, T s) {
00926     return div(A,s);
00927 }
00928
00929 template<typename T>
00930 inline Matrix<T> operator+(T s, const Matrix<T>& A) {
00931     return add(A,s);
00932 }
00933
00934 template<typename T>
00935 inline Matrix<T> operator*(T s, const Matrix<T>& A) {
00936     return mult (A,s);
00937 }
00938
00939 template<typename T>
00940 inline Matrix<T>& operator+=(Matrix<T>& A, const Matrix<T>& B) {
00941     return A.add(B);
00942 }
00943
00944 template<typename T>
00945 inline Matrix<T>& operator-=(Matrix<T>& A, const Matrix<T>& B) {
00946     return A.subtract(B);
00947 }
00948
00949 template<typename T>
00950 inline Matrix<T>& operator*=(Matrix<T>& A, const Matrix<T>& B) {
00951     A = mult (A,B);
00952     return A;
00953 }
00954
00955 template<typename T>
00956 inline Matrix<T>& operator^=(Matrix<T>& A, const Matrix<T>& B) {
00957     return A.mult_hadamard(B);
00958 }
00959
00960 template<typename T>
00961 inline Matrix<T>& operator+=(Matrix<T>& A, T s) {
00962     return A.add(s);
00963 }
00964
00965 template<typename T>
00966 inline Matrix<T>& operator-=(Matrix<T>& A, T s) {
00967     return A.subtract(s);
00968 }
00969
00970 template<typename T>
00971 inline Matrix<T>& operator*=(Matrix<T>& A, T s) {
00972     return A.mult (s);
00973 }
00974
00975 template<typename T>
00976 inline Matrix<T>& operator/=(Matrix<T>& A, T s) {
00977     return A.div(s);
00978 }
00979
00980 template<typename T>
00981 inline bool operator==(const Matrix<T>& A, const Matrix<T>& b) {
00982     return A.isequal(b);
00983 }
00984
00985 template<typename T>
00986 inline bool operator!=(const Matrix<T>& A, const Matrix<T>& b) {
00987     return !(A.isequal(b));
00988 }

```

```

01056 }
01057
01063 template<typename T>
01064 Matrix<T> kron(const Matrix<T>& A, const Matrix<T>& B) {
01065     const unsigned rows_A = A.rows();
01066     const unsigned cols_A = A.cols();
01067     const unsigned rows_B = B.rows();
01068     const unsigned cols_B = B.cols();
01069
01070     unsigned rows_C = rows_A * rows_B;
01071     unsigned cols_C = cols_A * cols_B;
01072
01073     Matrix<T> C(rows_C, cols_C);
01074
01075     for (unsigned i = 0; i < rows_A; i++)
01076         for (unsigned j = 0; j < cols_A; j++)
01077             for (unsigned k = 0; k < rows_B; k++)
01078                 for (unsigned l = 0; l < cols_B; l++)
01079                     C(i*rows_B + k, j*cols_B + l) = A(i,j) * B(k,l);
01080
01081     return C;
01082 }
01083
01091 template<typename T>
01092 Matrix<T> adj(const Matrix<T>& A) {
01093     if (! A.issquare()) throw std::runtime_error("Input matrix is not square");
01094
01095     Matrix<T> B(A.rows(), A.cols());
01096     if (A.rows() == 1) {
01097         B(0) = 1.0;
01098     } else {
01099         for (unsigned i = 0; i < A.rows(); i++) {
01100             for (unsigned j = 0; j < A.cols(); j++) {
01101                 T sgn = ((i + j) % 2 == 0) ? 1.0 : -1.0;
01102                 B(j,i) = sgn * det(cofactor(A,i,j));
01103             }
01104         }
01105     }
01106     return B;
01107 }
01108
01121 template<typename T>
01122 Matrix<T> cofactor(const Matrix<T>& A, unsigned p, unsigned q) {
01123     if (! A.issquare()) throw std::runtime_error("Input matrix is not square");
01124     if (!(p < A.rows())) throw std::out_of_range("Row index out of range");
01125     if (!(q < A.cols())) throw std::out_of_range("Column index out of range");
01126     if (A.cols() < 2) throw std::runtime_error("Cofactor calculation requested for matrix with less than
2 rows");
01127
01128     Matrix<T> c(A.rows()-1,A.cols()-1);
01129     unsigned i = 0;
01130     unsigned j = 0;
01131
01132     for (unsigned row = 0; row < A.rows(); row++) {
01133         if (row != p) {
01134             for (unsigned col = 0; col < A.cols(); col++)
01135                 if (col != q) c(i,j++) = A(row,col);
01136             j = 0;
01137             i++;
01138         }
01139     }
01140
01141     return c;
01142 }
01143
01155 template<typename T>
01156 T det_lu(const Matrix<T>& A) {
01157     if (! A.issquare()) throw std::runtime_error("Input matrix is not square");
01158
01159     // LU decomposition with pivoting
01160     auto res = lup(A);
01161
01162     // Determinants of LU
01163     T detLU = static_cast<T>(1);
01164
01165     for (unsigned i = 0; i < res.L.rows(); i++)
01166         detLU *= res.L(i,i) * res.U(i,i);
01167
01168     // Determinant of P
01169     unsigned len = res.P.size();
01170     T detP = 1;
01171
01172     std::vector<unsigned> p(res.P);
01173     std::vector<unsigned> q;
01174     q.resize(len);
01175
01176     for (unsigned i = 0; i < len; i++)

```



```

01177     q[p[i]] = i;
01178
01179     for (unsigned i = 0; i < len; i++) {
01180         unsigned j = p[i];
01181         unsigned k = q[i];
01182         if (j != i) {
01183             p[k] = p[i];
01184             q[j] = q[i];
01185             detP = - detP;
01186         }
01187     }
01188
01189     return detLU * detP;
01190 }
01191
01200 template<typename T>
01201 T det(const Matrix<T>& A) {
01202     if (! A.issquare()) throw std::runtime_error("Input matrix is not square");
01203
01204     if (A.rows() == 1)
01205         return A(0,0);
01206     else if (A.rows() == 2)
01207         return A(0,0)*A(1,1) - A(0,1)*A(1,0);
01208     else if (A.rows() == 3)
01209         return A(0,0)*(A(1,1)*A(2,2) - A(1,2)*A(2,1)) -
01210             A(0,1)*(A(1,0)*A(2,2) - A(1,2)*A(2,0)) +
01211             A(0,2)*(A(1,0)*A(2,1) - A(1,1)*A(2,0));
01212     else
01213         return det_lu(A);
01214 }
01215
01224 template<typename T>
01225 LU_result<T> lu(const Matrix<T>& A) {
01226     const unsigned M = A.rows();
01227     const unsigned N = A.cols();
01228
01229     LU_result<T> res;
01230     res.L = eye<T>(M);
01231     res.U = Matrix<T>(A);
01232
01233     // aliases
01234     auto& L = res.L;
01235     auto& U = res.U;
01236
01237     if (A.numel() == 0)
01238         return res;
01239
01240     for (unsigned k = 0; k < M-1; k++) {
01241         for (unsigned i = k+1; i < M; i++) {
01242             L(i,k) = U(i,k) / U(k,k);
01243             for (unsigned l = k+1; l < N; l++) {
01244                 U(i,l) -= L(i,k) * U(k,l);
01245             }
01246         }
01247     }
01248
01249     for (unsigned col = 0; col < N; col++)
01250         for (unsigned row = col+1; row < M; row++)
01251             U(row,col) = 0;
01252
01253     return res;
01254 }
01255
01269 template<typename T>
01270 LUP_result<T> lup(const Matrix<T>& A) {
01271     const unsigned M = A.rows();
01272     const unsigned N = A.cols();
01273
01274     // Initialize L, U, and PP
01275     LUP_result<T> res;
01276
01277     if (A.numel() == 0)
01278         return res;
01279
01280     res.L = eye<T>(M);
01281     res.U = Matrix<T>(A);
01282     std::vector<unsigned> PP;
01283
01284     // aliases
01285     auto& L = res.L;
01286     auto& U = res.U;
01287
01288     PP.resize(N);
01289     for (unsigned i = 0; i < N; i++)
01290         PP[i] = i;
01291
01292     for (unsigned k = 0; k < M-1; k++) {

```

```

01293     // Find the column with the largest absolute value in the current row
01294     auto max_col_value = std::abs(U(k,k));
01295     unsigned max_col_index = k;
01296     for (unsigned l = k+1; l < N; l++) {
01297         auto val = std::abs(U(k,l));
01298         if (val > max_col_value) {
01299             max_col_value = val;
01300             max_col_index = l;
01301         }
01302     }
01303
01304     // Swap columns k and max_col_index in U and update P
01305     if (max_col_index != k) {
01306         U.swap_cols(k, max_col_index); // TODO: This could be reworked to avoid column swap in U during
every iteration by:
01307                                     // 1. using PP[k] for column indexing across iterations
01308                                     // 2. doing just one permutation of U at the end
01309         std::swap(PP[k], PP[max_col_index]);
01310     }
01311
01312     // Update L and U
01313     for (unsigned i = k+1; i < M; i++) {
01314         L(i,k) = U(i,k) / U(k,k);
01315         for (unsigned l = k+1; l < N; l++) {
01316             U(i,l) -= L(i,k) * U(k,l);
01317         }
01318     }
01319 }
01320
01321 // Set elements in lower triangular part of U to zero
01322 for (unsigned col = 0; col < N; col++)
01323     for (unsigned row = col+1; row < M; row++)
01324         U(row,col) = 0;
01325
01326 // Transpose indices in permutation vector
01327 res.P.resize(N);
01328 for (unsigned i = 0; i < N; i++)
01329     res.P[PP[i]] = i;
01330
01331 return res;
01332 }
01333
01334 template<typename T>
01335 Matrix<T> inv_gauss_jordan(const Matrix<T>& A) {
01336     if (! A.issquare()) throw std::runtime_error("Input matrix is not square");
01337
01338     const unsigned N = A.rows();
01339     Matrix<T> AA(A);
01340     auto IA = eye<T>(N);
01341
01342     bool found_nonzero;
01343     for (unsigned j = 0; j < N; j++) {
01344         found_nonzero = false;
01345         for (unsigned i = j; i < N; i++) {
01346             if (AA(i,j) != static_cast<T>(0)) {
01347                 found_nonzero = true;
01348                 for (unsigned k = 0; k < N; k++) {
01349                     std::swap(AA(j,k), AA(i,k));
01350                     std::swap(IA(j,k), IA(i,k));
01351                 }
01352                 if (AA(j,j) != static_cast<T>(1)) {
01353                     T s = static_cast<T>(1) / AA(j,j);
01354                     for (unsigned k = 0; k < N; k++) {
01355                         AA(j,k) *= s;
01356                         IA(j,k) *= s;
01357                     }
01358                 }
01359                 for (unsigned l = 0; l < N; l++) {
01360                     if (l != j) {
01361                         T s = AA(l,j);
01362                         for (unsigned k = 0; k < N; k++) {
01363                             AA(l,k) -= s * AA(j,k);
01364                             IA(l,k) -= s * IA(j,k);
01365                         }
01366                     }
01367                 }
01368             }
01369             break;
01370         }
01371         if (! found_nonzero)
01372             throw singular_matrix_exception("Singular matrix in inv_gauss_jordan");
01373     }
01374     return IA;
01375 }
01376
01377 template<typename T>
01378 Matrix<T> inv_tril(const Matrix<T>& A) {

```

```

01399     if (! A.issquare()) throw std::runtime_error("Input matrix is not square");
01400
01401     const unsigned N = A.rows();
01402
01403     auto IA = zeros<T>(N);
01404
01405     for (unsigned i = 0; i < N; i++) {
01406         if (A(i,i) == 0.0) throw singular_matrix_exception("Division by zero in inv_tril");
01407
01408         IA(i,i) = static_cast<T>(1.0) / A(i,i);
01409         for (unsigned j = 0; j < i; j++) {
01410             T s = 0.0;
01411             for (unsigned k = j; k < i; k++)
01412                 s += A(i,k) * IA(k,j);
01413             IA(i,j) = -s * IA(i,i) ;
01414         }
01415     }
01416
01417     return IA;
01418 }
01419
01430 template<typename T>
01431 Matrix<T> inv_triu(const Matrix<T>& A) {
01432     if (!A.issquare()) throw std::runtime_error("Input matrix is not square");
01433
01434     const unsigned N = A.rows();
01435
01436     auto IA = zeros<T>(N);
01437
01438     for (int i = N - 1; i >= 0; i--) {
01439         if (A(i,i) == 0.0) throw singular_matrix_exception("Division by zero in inv_triu");
01440
01441         IA(i, i) = static_cast<T>(1.0) / A(i,i);
01442         for (int j = N - 1; j > i; j--) {
01443             T s = 0.0;
01444             for (int k = i + 1; k <= j; k++)
01445                 s += A(i,k) * IA(k,j);
01446             IA(i,j) = -s * IA(i,i);
01447         }
01448     }
01449
01450     return IA;
01451 }
01452
01465 template<typename T>
01466 Matrix<T> inv_posdef(const Matrix<T>& A) {
01467     auto L = cholinv(A);
01468     return mult<T,true,false>(L,L);
01469 }
01470
01481 template<typename T>
01482 Matrix<T> inv_square(const Matrix<T>& A) {
01483     if (! A.issquare()) throw std::runtime_error("Input matrix is not square");
01484
01485     // LU decomposition with pivoting
01486     auto LU = lup(A);
01487     auto IL = inv_tril(LU.L);
01488     auto IU = inv_triu(LU.U);
01489
01490     return permute_rows(IU * IL, LU.P);
01491 }
01492
01503 template<typename T>
01504 Matrix<T> inv(const Matrix<T>& A) {
01505     if (! A.issquare()) throw std::runtime_error("Input matrix is not square");
01506
01507     if (A.numel() == 0) {
01508         return Matrix<T>();
01509     } else if (A.rows() < 4) {
01510         T d = det(A);
01511
01512         if (d == 0.0) throw singular_matrix_exception("Singular matrix in inv");
01513
01514         Matrix<T> IA(A.rows(), A.rows());
01515         T invdet = static_cast<T>(1.0) / d;
01516
01517         if (A.rows() == 1) {
01518             IA(0,0) = invdet;
01519         } else if (A.rows() == 2) {
01520             IA(0,0) = A(1,1) * invdet;
01521             IA(0,1) = - A(0,1) * invdet;
01522             IA(1,0) = - A(1,0) * invdet;
01523             IA(1,1) = A(0,0) * invdet;
01524         } else if (A.rows() == 3) {
01525             IA(0,0) = (A(1,1)*A(2,2) - A(2,1)*A(1,2)) * invdet;
01526             IA(0,1) = (A(0,2)*A(2,1) - A(0,1)*A(2,2)) * invdet;
01527             IA(0,2) = (A(0,1)*A(1,2) - A(0,2)*A(1,1)) * invdet;

```

```

01528     IA(1,0) = (A(1,2)*A(2,0) - A(1,0)*A(2,2)) * invdet;
01529     IA(1,1) = (A(0,0)*A(2,2) - A(0,2)*A(2,0)) * invdet;
01530     IA(1,2) = (A(1,0)*A(0,2) - A(0,0)*A(1,2)) * invdet;
01531     IA(2,0) = (A(1,0)*A(2,1) - A(2,0)*A(1,1)) * invdet;
01532     IA(2,1) = (A(2,0)*A(0,1) - A(0,0)*A(2,1)) * invdet;
01533     IA(2,2) = (A(0,0)*A(1,1) - A(1,0)*A(0,1)) * invdet;
01534 }
01535
01536     return IA;
01537 } else {
01538     return inv_square(A);
01539 }
01540 }
01541
01542 template<typename T>
01543 Matrix<T> pinv(const Matrix<T>& A) {
01544     auto AH_A = mult<T,true,false>(A, A);
01545     auto Linv = inv_posdef(AH_A);
01546     return mult<T,false,true>(Linv, A);
01547 }
01548
01549 template<typename T>
01550 T trace(const Matrix<T>& A) {
01551     T t = static_cast<T>(0);
01552     for (int i = 0; i < A.rows(); i++)
01553         t += A(i,i);
01554     return t;
01555 }
01556
01557 template<typename T>
01558 double cond(const Matrix<T>& A) {
01559     try {
01560         auto A_inv = inv(A);
01561         return norm_fro(A) * norm_fro(A_inv);
01562     } catch (singular_matrix_exception& e) {
01563         return std::numeric_limits<double>::max();
01564     }
01565 }
01566
01567 template<typename T>
01568 Matrix<T> chol(const Matrix<T>& A) {
01569     if (! A.issquare()) throw std::runtime_error("Input matrix is not square");
01570
01571     const unsigned N = A.rows();
01572     Matrix<T> L = tril(A);
01573
01574     for (unsigned j = 0; j < N; j++) {
01575         if (L(j,j) == 0.0) throw singular_matrix_exception("Singular matrix in chol");
01576
01577         L(j,j) = std::sqrt(L(j,j));
01578
01579         for (unsigned k = j+1; k < N; k++)
01580             L(k,j) = L(k,j) / L(j,j);
01581
01582         for (unsigned k = j+1; k < N; k++)
01583             for (unsigned i = k; i < N; i++)
01584                 L(i,k) = L(i,k) - L(i,j) * cconj(L(k,j));
01585     }
01586
01587     return L;
01588 }
01589
01590 template<typename T>
01591 Matrix<T> cholinv(const Matrix<T>& A) {
01592     if (! A.issquare()) throw std::runtime_error("Input matrix is not square");
01593
01594     const unsigned N = A.rows();
01595     Matrix<T> L(A);
01596     auto Linv = eye<T>(N);
01597
01598     for (unsigned j = 0; j < N; j++) {
01599         if (L(j,j) == 0.0) throw singular_matrix_exception("Singular matrix in cholinv");
01600
01601         L(j,j) = 1.0 / std::sqrt(L(j,j));
01602
01603         for (unsigned k = j+1; k < N; k++)
01604             L(k,j) = L(k,j) * L(j,j);
01605
01606         for (unsigned k = j+1; k < N; k++)
01607             for (unsigned i = k; i < N; i++)
01608                 L(i,k) = L(i,k) - L(i,j) * cconj(L(k,j));
01609     }
01610
01611     for (unsigned k = 0; k < N; k++) {
01612         for (unsigned i = k; i < N; i++) {
01613             Linv(i,k) = Linv(i,k) * L(i,i);
01614         }
01615         for (unsigned j = i+1; j < N; j++)

```

```

01654         Linv(j,k) = Linv(j,k) - L(j,i) * Linv(i,k);
01655     }
01656 }
01657
01658 return Linv;
01659 }
01660
01661 template<typename T>
01662 LDL_result<T> ldl(const Matrix<T>& A) {
01663     if (! A.issquare()) throw std::runtime_error("Input matrix is not square");
01664
01665     const unsigned N = A.rows();
01666     LDL_result<T> res;
01667
01668     // aliases
01669     auto& L = res.L;
01670     auto& d = res.d;
01671
01672     L = eye<T>(N);
01673     d.resize(N);
01674
01675     for (unsigned m = 0; m < N; m++) {
01676         d[m] = A(m,m);
01677
01678         for (unsigned k = 0; k < m; k++)
01679             d[m] -= L(m,k) * cconj(L(m,k)) * d[k];
01680
01681         if (d[m] == 0.0) throw singular_matrix_exception("Singular matrix in ldl");
01682
01683         for (unsigned n = m+1; n < N; n++) {
01684             L(n,m) = A(n,m);
01685             for (unsigned k = 0; k < m; k++)
01686                 L(n,m) -= L(n,k) * cconj(L(m,k)) * d[k];
01687             L(n,m) /= d[m];
01688         }
01689     }
01690
01691     return res;
01692 }
01693
01694 template<typename T>
01695 QR_result<T> qr_red_gs(const Matrix<T>& A) {
01696     const int rows = A.rows();
01697     const int cols = A.cols();
01698
01699     QR_result<T> res;
01700
01701     //aliases
01702     auto& Q = res.Q;
01703     auto& R = res.R;
01704
01705     Q = zeros<T>(rows, cols);
01706     R = zeros<T>(cols, cols);
01707
01708     for (int c = 0; c < cols; c++) {
01709         Matrix<T> v = A.get_submatrix(0, rows-1, c, c);
01710         for (int r = 0; r < c; r++) {
01711             for (int k = 0; k < rows; k++)
01712                 R(r,c) = R(r,c) + cconj(Q(k,r)) * A(k,c);
01713             for (int k = 0; k < rows; k++)
01714                 v(k) = v(k) - R(r,c) * Q(k,r);
01715         }
01716
01717         R(c,c) = static_cast<T>(norm_fro(v));
01718
01719         if (R(c,c) == 0.0) throw singular_matrix_exception("Division by 0 in QR GS");
01720
01721         for (int k = 0; k < rows; k++)
01722             Q(k,c) = v(k) / R(c,c);
01723     }
01724
01725     return res;
01726 }
01727
01728 template<typename T>
01729 Matrix<T> householder_reflection(const Matrix<T>& a) {
01730     if (a.cols() != 1) throw std::runtime_error("Input not a column vector");
01731
01732     static const T ISQRT2 = static_cast<T>(0.707106781186547);
01733
01734     Matrix<T> v(a);
01735     v(0) += csign(v(0)) * norm_fro(v);
01736     auto vn = norm_fro(v) * ISQRT2;
01737     for (unsigned i = 0; i < v.numel(); i++)
01738         v(i) /= vn;
01739     return v;
01740 }

```

```

01773 }
01774
01786 template<typename T>
01787 QR_result<T> qr_householder(const Matrix<T>& A, bool calculate_Q = true) {
01788     const unsigned rows = A.rows();
01789     const unsigned cols = A.cols();
01790
01791     QR_result<T> res;
01792
01793     //aliases
01794     auto& Q = res.Q;
01795     auto& R = res.R;
01796
01797     R = Matrix<T>(A);
01798
01799     if (calculate_Q)
01800         Q = eye<T>(rows);
01801
01802     const unsigned N = (rows > cols) ? cols : rows;
01803
01804     for (unsigned j = 0; j < N; j++) {
01805         auto v = householder_reflection(R.get_submatrix(j, rows-1, j, j));
01806
01807         auto R1 = R.get_submatrix(j, rows-1, j, cols-1);
01808         auto WR = v * mult<T,true,false>(v, R1);
01809         for (unsigned c = j; c < cols; c++)
01810             for (unsigned r = j; r < rows; r++)
01811                 R(r,c) -= WR(r-j,c-j);
01812
01813         if (calculate_Q) {
01814             auto Q1 = Q.get_submatrix(0, rows-1, j, rows-1);
01815             auto WQ = mult<T,false,true>(Q1 * v, v);
01816             for (unsigned c = j; c < rows; c++)
01817                 for (unsigned r = 0; r < rows; r++)
01818                     Q(r,c) -= WQ(r,c-j);
01819         }
01820     }
01821
01822     for (unsigned col = 0; col < R.cols(); col++)
01823         for (unsigned row = col+1; row < R.rows(); row++)
01824             R(row,col) = 0;
01825
01826     return res;
01827 }
01828
01839 template<typename T>
01840 inline QR_result<T> qr(const Matrix<T>& A, bool calculate_Q = true) {
01841     return qr_householder(A, calculate_Q);
01842 }
01843
01854 template<typename T>
01855 Hessenberg_result<T> hessenberg(const Matrix<T>& A, bool calculate_Q = true) {
01856     if (! A.issquare()) throw std::runtime_error("Input matrix is not square");
01857
01858     Hessenberg_result<T> res;
01859
01860     // aliases
01861     auto& H = res.H;
01862     auto& Q = res.Q;
01863
01864     const unsigned N = A.rows();
01865     H = Matrix<T>(A);
01866
01867     if (calculate_Q)
01868         Q = eye<T>(N);
01869
01870     for (unsigned k = 1; k < N-1; k++) {
01871         auto v = householder_reflection(H.get_submatrix(k, N-1, k-1, k-1));
01872
01873         auto H1 = H.get_submatrix(k, N-1, 0, N-1);
01874         auto W1 = v * mult<T,true,false>(v, H1);
01875         for (unsigned c = 0; c < N; c++)
01876             for (unsigned r = k; r < N; r++)
01877                 H(r,c) -= W1(r-k,c);
01878
01879         auto H2 = H.get_submatrix(0, N-1, k, N-1);
01880         auto W2 = mult<T,false,true>(H2 * v, v);
01881         for (unsigned c = k; c < N; c++)
01882             for (unsigned r = 0; r < N; r++)
01883                 H(r,c) -= W2(r,c-k);
01884
01885         if (calculate_Q) {
01886             auto Q1 = Q.get_submatrix(0, N-1, k, N-1);
01887             auto W3 = mult<T,false,true>(Q1 * v, v);
01888             for (unsigned c = k; c < N; c++)
01889                 for (unsigned r = 0; r < N; r++)
01890                     Q(r,c) -= W3(r,c-k);

```

```

01891     }
01892 }
01893
01894 for (unsigned row = 2; row < N; row++)
01895     for (unsigned col = 0; col < row-2; col++)
01896         H(row,col) = static_cast<T>(0);
01897
01898 return res;
01899 }
01900
01901 template<typename T>
01902 std::complex<T> wilkinson_shift(const Matrix<std::complex<T>& H, T tol = 1e-10) {
01903     if (! H.issquare()) throw std::runtime_error("Input matrix is not square");
01904
01905     const unsigned n = H.rows();
01906     std::complex<T> mu;
01907
01908     if (std::abs(H(n-1,n-2)) < tol) {
01909         mu = H(n-2,n-2);
01910     } else {
01911         auto trA = H(n-2,n-2) + H(n-1,n-1);
01912         auto detA = H(n-2,n-2) * H(n-1,n-1) - H(n-2, n-1) * H(n-1, n-2);
01913         mu = (trA + std::sqrt(trA*trA - 4.0*detA)) / 2.0;
01914     }
01915
01916 return mu;
01917 }
01918
01919 template<typename T>
01920 Eigenvalues_result<T> eigenvalues(const Matrix<std::complex<T>& A, T tol = 1e-12, unsigned max_iter =
100) {
01921     if (! A.issquare()) throw std::runtime_error("Input matrix is not square");
01922
01923     const unsigned N = A.rows();
01924     Matrix<std::complex<T>> H;
01925     bool success = false;
01926
01927     QR_result<std::complex<T>> QR;
01928
01929     // aliases
01930     auto& Q = QR.Q;
01931     auto& R = QR.R;
01932
01933     // Transfer A to Hessenberg form to improve convergence (skip calculation of Q)
01934     H = hessenberg(A, false).H;
01935
01936     for (unsigned iter = 0; iter < max_iter; iter++) {
01937         auto mu = wilkinson_shift(H, tol);
01938
01939         // subtract mu from diagonal
01940         for (unsigned n = 0; n < N; n++)
01941             H(n,n) -= mu;
01942
01943         // QR factorization with shifted H
01944         QR = qr(H);
01945         H = R * Q;
01946
01947         // add back mu to diagonal
01948         for (unsigned n = 0; n < N; n++)
01949             H(n,n) += mu;
01950
01951         // Check for convergence
01952         if (std::abs(H(N-2,N-1)) <= tol) {
01953             success = true;
01954             break;
01955         }
01956     }
01957
01958     Eigenvalues_result<T> res;
01959     res.eig = diag(H);
01960     res.err = std::abs(H(N-2,N-1));
01961     res.converged = success;
01962
01963     return res;
01964 }
01965
01966 template<typename T>
01967 Eigenvalues_result<T> eigenvalues(const Matrix<T>& A, T tol = 1e-12, unsigned max_iter = 100) {
01968     auto A_cplx = make_complex(A);
01969     return eigenvalues(A_cplx, tol, max_iter);
01970 }
01971
01972 template<typename T>
01973 Matrix<T> solve_triu(const Matrix<T>& U, const Matrix<T>& B) {
01974     if (! U.issquare()) throw std::runtime_error("Input matrix is not square");
01975     if (U.rows() != B.rows()) throw std::runtime_error("Unmatching matrix dimensions for solve");
01976

```

```

02019     const unsigned N = U.rows();
02020     const unsigned M = B.cols();
02021
02022     if (U.numel() == 0)
02023         return Matrix<T>();
02024
02025     Matrix<T> X(B);
02026
02027     for (unsigned m = 0; m < M; m++) {
02028         // backwards substitution for each column of B
02029         for (int n = N-1; n >= 0; n--) {
02030             for (unsigned j = n + 1; j < N; j++)
02031                 X(n,m) -= U(n,j) * X(j,m);
02032
02033             if (U(n,n) == 0.0) throw singular_matrix_exception("Singular matrix in solve_triu");
02034
02035             X(n,m) /= U(n,n);
02036         }
02037     }
02038
02039     return X;
02040 }
02041
02056 template<typename T>
02057 Matrix<T> solve_tril(const Matrix<T>& L, const Matrix<T>& B) {
02058     if (! L.issquare()) throw std::runtime_error("Input matrix is not square");
02059     if (L.rows() != B.rows()) throw std::runtime_error("Unmatching matrix dimensions for solve");
02060
02061     const unsigned N = L.rows();
02062     const unsigned M = B.cols();
02063
02064     if (L.numel() == 0)
02065         return Matrix<T>();
02066
02067     Matrix<T> X(B);
02068
02069     for (unsigned m = 0; m < M; m++) {
02070         // forwards substitution for each column of B
02071         for (unsigned n = 0; n < N; n++) {
02072             for (unsigned j = 0; j < n; j++)
02073                 X(n,m) -= L(n,j) * X(j,m);
02074
02075             if (L(n,n) == 0.0) throw singular_matrix_exception("Singular matrix in solve_tril");
02076
02077             X(n,m) /= L(n,n);
02078         }
02079     }
02080
02081     return X;
02082 }
02083
02098 template<typename T>
02099 Matrix<T> solve_square(const Matrix<T>& A, const Matrix<T>& B) {
02100     if (! A.issquare()) throw std::runtime_error("Input matrix is not square");
02101     if (A.rows() != B.rows()) throw std::runtime_error("Unmatching matrix dimensions for solve");
02102
02103     if (A.numel() == 0)
02104         return Matrix<T>();
02105
02106     Matrix<T> L;
02107     Matrix<T> U;
02108     std::vector<unsigned> P;
02109
02110     // LU decomposition with pivoting
02111     auto lup_res = lup(A);
02112
02113     auto y = solve_tril(lup_res.L, B);
02114     auto x = solve_triu(lup_res.U, y);
02115
02116     return permute_rows(x, lup_res.P);
02117 }
02118
02133 template<typename T>
02134 Matrix<T> solve_posdef(const Matrix<T>& A, const Matrix<T>& B) {
02135     if (! A.issquare()) throw std::runtime_error("Input matrix is not square");
02136     if (A.rows() != B.rows()) throw std::runtime_error("Unmatching matrix dimensions for solve");
02137
02138     if (A.numel() == 0)
02139         return Matrix<T>();
02140
02141     // LU decomposition with pivoting
02142     auto L = chol(A);
02143
02144     auto Y = solve_tril(L, B);
02145     return solve_triu(L.ctranspose(), Y);
02146 }
02147

```



```

02152 template<typename T>
02153 class Matrix {
02154 public:
02159     Matrix();
02160
02165     Matrix(unsigned size);
02166
02171     Matrix(unsigned nrows, unsigned ncols);
02172
02177     Matrix(T x, unsigned nrows, unsigned ncols);
02178
02184     Matrix(const T* array, unsigned nrows, unsigned ncols);
02185
02193     Matrix(const std::vector<T>& vec, unsigned nrows, unsigned ncols);
02194
02202     Matrix(std::initializer_list<T> init_list, unsigned nrows, unsigned ncols);
02203
02206     Matrix(const Matrix &);
02207
02210     virtual ~Matrix();
02211
02219     Matrix<T> get_submatrix(unsigned row_first, unsigned row_last, unsigned col_first, unsigned
col_last) const;
02220
02229     void set_submatrix(const Matrix<T>& smtx, unsigned row_first, unsigned col_first);
02230
02235     void clear();
02236
02244     void reshape(unsigned rows, unsigned cols);
02245
02251     void resize(unsigned rows, unsigned cols);
02252
02258     bool exists(unsigned row, unsigned col) const;
02259
02264     T* ptr(unsigned row, unsigned col);
02265
02272     T* ptr();
02273
02277     void fill(T value);
02278
02285     void fill_col(T value, unsigned col);
02286
02293     void fill_row(T value, unsigned row);
02294
02299     bool isempty() const;
02300
02304     bool issquare() const;
02305
02310     bool isequal(const Matrix<T>&) const;
02311
02317     bool isequal(const Matrix<T>&, T) const;
02318
02323     unsigned numel() const;
02324
02329     unsigned rows() const;
02330
02335     unsigned cols() const;
02336
02341     Matrix<T> transpose() const;
02342
02348     Matrix<T> ctranspose() const;
02349
02357     Matrix<T>& add(const Matrix<T>&);
02358
02366     Matrix<T>& subtract(const Matrix<T>&);
02367
02376     Matrix<T>& mult_hadamard(const Matrix<T>&);
02377
02383     Matrix<T>& add(T);
02384
02390     Matrix<T>& subtract(T);
02391
02397     Matrix<T>& mult(T);
02398
02404     Matrix<T>& div(T);
02405
02410     Matrix<T>& operator=(const Matrix<T>&);
02411
02416     Matrix<T>& operator=(T);
02417
02422     explicit operator std::vector<T>() const;
02423     std::vector<T> to_vector() const;
02424
02431     T& operator()(unsigned nel);
02432     T operator()(unsigned nel) const;
02433     T& at(unsigned nel);
02434     T at(unsigned nel) const;

```

```

02435
02442     T& operator()(unsigned row, unsigned col);
02443     T operator()(unsigned row, unsigned col) const;
02444     T& at(unsigned row, unsigned col);
02445     T at(unsigned row, unsigned col) const;
02446
02453     void add_row_to_another(unsigned to, unsigned from);
02454
02461     void add_col_to_another(unsigned to, unsigned from);
02462
02469     void swap_rows(unsigned i, unsigned j);
02470
02477     void swap_cols(unsigned i, unsigned j);
02478
02485     std::vector<T> col_to_vector(unsigned col) const;
02486
02493     std::vector<T> row_to_vector(unsigned row) const;
02494
02502     void col_from_vector(const std::vector<T>&, unsigned col);
02503
02511     void row_from_vector(const std::vector<T>&, unsigned row);
02512
02513 private:
02514     unsigned nrows;
02515     unsigned ncols;
02516     std::vector<T> data;
02517 };
02518
02519 /*
02520  * Implementation of Matrix class methods
02521  */
02522
02523 template<typename T>
02524 Matrix<T>::Matrix() : nrows(0), ncols(0), data() { }
02525
02526 template<typename T>
02527 Matrix<T>::Matrix(unsigned size) : Matrix(size, size) { }
02528
02529 template<typename T>
02530 Matrix<T>::Matrix(unsigned rows, unsigned cols) : nrows(rows), ncols(cols) {
02531     data.resize(numel());
02532 }
02533
02534 template<typename T>
02535 Matrix<T>::Matrix(T x, unsigned rows, unsigned cols) : Matrix(rows, cols) {
02536     fill(x);
02537 }
02538
02539 template<typename T>
02540 Matrix<T>::Matrix(const T* array, unsigned rows, unsigned cols) : Matrix(rows, cols) {
02541     data.assign(array, array + numel());
02542 }
02543
02544 template<typename T>
02545 Matrix<T>::Matrix(const std::vector<T>& vec, unsigned rows, unsigned cols) : Matrix(rows, cols) {
02546     if (vec.size() != numel()) throw std::runtime_error("Size of initialization vector not consistent
with matrix dimensions");
02547
02548     data.assign(vec.begin(), vec.end());
02549 }
02550
02551 template<typename T>
02552 Matrix<T>::Matrix(std::initializer_list<T> init_list, unsigned rows, unsigned cols) : Matrix(rows,
cols) {
02553     if (init_list.size() != numel()) throw std::runtime_error("Size of initialization list not
consistent with matrix dimensions");
02554
02555     auto it = init_list.begin();
02556
02557     for (unsigned row = 0; row < this->nrows; row++)
02558         for (unsigned col = 0; col < this->ncols; col++)
02559             this->at(row, col) = *(it++);
02560 }
02561
02562 template<typename T>
02563 Matrix<T>::Matrix(const Matrix & other) : Matrix(other.nrows, other.ncols) {
02564     this->data.assign(other.data.begin(), other.data.end());
02565 }
02566
02567 template<typename T>
02568 Matrix<T> & Matrix<T>::operator=(const Matrix<T> & other) {
02569     this->nrows = other.nrows;
02570     this->ncols = other.ncols;
02571     this->data.assign(other.data.begin(), other.data.end());
02572     return *this;
02573 }
02574

```

```

02575 template<typename T>
02576 Matrix<T>& Matrix<T>::operator=(T s) {
02577     fill(s);
02578     return *this;
02579 }
02580
02581 template<typename T>
02582 inline Matrix<T>::operator std::vector<T>() const {
02583     return data;
02584 }
02585
02586 template<typename T>
02587 inline void Matrix<T>::clear() {
02588     this->nrows = 0;
02589     this->ncols = 0;
02590     data.resize(0);
02591 }
02592
02593 template<typename T>
02594 void Matrix<T>::reshape(unsigned rows, unsigned cols) {
02595     if (this->numel() != rows * cols) throw std::runtime_error("Illegal attempt to change number of
elements via reshape");
02596     this->nrows = rows;
02597     this->ncols = cols;
02598 }
02599
02600
02601 template<typename T>
02602 void Matrix<T>::resize(unsigned rows, unsigned cols) {
02603     this->nrows = rows;
02604     this->ncols = cols;
02605     data.resize(nrows*ncols);
02606 }
02607
02608 template<typename T>
02609 Matrix<T> Matrix<T>::get_submatrix(unsigned row_base, unsigned row_lim, unsigned col_base, unsigned
col_lim) const {
02610     if (row_base > row_lim) throw std::out_of_range("Row index of submatrix out of range");
02611     if (col_base > col_lim) throw std::out_of_range("Column index of submatrix out of range");
02612     if (row_lim >= this->rows()) throw std::out_of_range("Row index of submatrix out of range");
02613     if (col_lim >= this->cols()) throw std::out_of_range("Column index of submatrix out of range");
02614
02615     unsigned num_rows = row_lim - row_base + 1;
02616     unsigned num_cols = col_lim - col_base + 1;
02617     Matrix<T> S(num_rows, num_cols);
02618     for (unsigned i = 0; i < num_rows; i++) {
02619         for (unsigned j = 0; j < num_cols; j++) {
02620             S(i,j) = at(row_base + i, col_base + j);
02621         }
02622     }
02623     return S;
02624 }
02625
02626 template<typename T>
02627 void Matrix<T>::set_submatrix(const Matrix<T>& S, unsigned row_base, unsigned col_base) {
02628     if (this->isempty()) throw std::runtime_error("Invalid attempt to set submatrix in empty matrix");
02629
02630     const unsigned row_lim = row_base + S.rows() - 1;
02631     const unsigned col_lim = col_base + S.cols() - 1;
02632
02633     if (row_base > row_lim) throw std::out_of_range("Row index of submatrix out of range");
02634     if (col_base > col_lim) throw std::out_of_range("Column index of submatrix out of range");
02635     if (row_lim >= this->rows()) throw std::out_of_range("Row index of submatrix out of range");
02636     if (col_lim >= this->cols()) throw std::out_of_range("Column index of submatrix out of range");
02637
02638     unsigned num_rows = row_lim - row_base + 1;
02639     unsigned num_cols = col_lim - col_base + 1;
02640     for (unsigned i = 0; i < num_rows; i++)
02641         for (unsigned j = 0; j < num_cols; j++)
02642             at(row_base + i, col_base + j) = S(i,j);
02643 }
02644
02645 template<typename T>
02646 inline T & Matrix<T>::operator()(unsigned nel) {
02647     return at(nel);
02648 }
02649
02650 template<typename T>
02651 inline T & Matrix<T>::operator()(unsigned row, unsigned col) {
02652     return at(row, col);
02653 }
02654
02655 template<typename T>
02656 inline T Matrix<T>::operator()(unsigned nel) const {
02657     return at(nel);
02658 }
02659

```

```

02660 template<typename T>
02661 inline T Matrix<T>::operator()(unsigned row, unsigned col) const {
02662     return at(row, col);
02663 }
02664
02665 template<typename T>
02666 inline T & Matrix<T>::at(unsigned nel) {
02667     if (!(nel < numel())) throw std::out_of_range("Element index out of range");
02668
02669     return data[nel];
02670 }
02671
02672 template<typename T>
02673 inline T & Matrix<T>::at(unsigned row, unsigned col) {
02674     if (!(row < rows() && col < cols())) std::cout << "at() failed at " << row << ", " << col << std::endl;
02675
02676     return data[nrows * col + row];
02677 }
02678
02679 template<typename T>
02680 inline T Matrix<T>::at(unsigned nel) const {
02681     if (!(nel < numel())) throw std::out_of_range("Element index out of range");
02682
02683     return data[nel];
02684 }
02685
02686 template<typename T>
02687 inline T Matrix<T>::at(unsigned row, unsigned col) const {
02688     if (!(row < rows())) throw std::out_of_range("Row index out of range");
02689     if (!(col < cols())) throw std::out_of_range("Column index out of range");
02690
02691     return data[nrows * col + row];
02692 }
02693
02694 template<typename T>
02695 inline void Matrix<T>::fill(T value) {
02696     for (unsigned i = 0; i < numel(); i++)
02697         data[i] = value;
02698 }
02699
02700 template<typename T>
02701 inline void Matrix<T>::fill_col(T value, unsigned col) {
02702     if (!(col < cols())) throw std::out_of_range("Column index out of range");
02703
02704     for (unsigned i = col * nrows; i < (col+1) * nrows; i++)
02705         data[i] = value;
02706 }
02707
02708 template<typename T>
02709 inline void Matrix<T>::fill_row(T value, unsigned row) {
02710     if (!(row < rows())) throw std::out_of_range("Row index out of range");
02711
02712     for (unsigned i = 0; i < ncols; i++)
02713         data[row + i * nrows] = value;
02714 }
02715
02716 template<typename T>
02717 inline bool Matrix<T>::exists(unsigned row, unsigned col) const {
02718     return (row < nrows && col < ncols);
02719 }
02720
02721 template<typename T>
02722 inline T* Matrix<T>::ptr(unsigned row, unsigned col) {
02723     if (!(row < rows())) throw std::out_of_range("Row index out of range");
02724     if (!(col < cols())) throw std::out_of_range("Column index out of range");
02725
02726     return data.data() + nrows * col + row;
02727 }
02728
02729 template<typename T>
02730 inline T* Matrix<T>::ptr() {
02731     return data.data();
02732 }
02733
02734 template<typename T>
02735 inline bool Matrix<T>::isempty() const {
02736     return (nrows == 0) || (ncols == 0);
02737 }
02738
02739 template<typename T>
02740 inline bool Matrix<T>::issquare() const {
02741     return (nrows == ncols) && !isempty();
02742 }
02743
02744 template<typename T>
02745 bool Matrix<T>::isequal(const Matrix<T>& A) const {
02746     bool ret = true;

```

```

02747     if (nrows != A.rows() || ncols != A.cols()) {
02748         ret = false;
02749     } else {
02750         for (unsigned i = 0; i < numel(); i++) {
02751             if (at(i) != A(i)) {
02752                 ret = false;
02753                 break;
02754             }
02755         }
02756     }
02757     return ret;
02758 }
02759
02760 template<typename T>
02761 bool Matrix<T>::isequal(const Matrix<T>& A, T tol) const {
02762     bool ret = true;
02763     if (rows() != A.rows() || cols() != A.cols()) {
02764         ret = false;
02765     } else {
02766         auto abs_tol = std::abs(tol); // workaround for complex
02767         for (unsigned i = 0; i < A.numel(); i++) {
02768             if (abs_tol < std::abs(at(i) - A(i))) {
02769                 ret = false;
02770                 break;
02771             }
02772         }
02773     }
02774     return ret;
02775 }
02776
02777 template<typename T>
02778 inline unsigned Matrix<T>::numel() const {
02779     return nrows * ncols;
02780 }
02781
02782 template<typename T>
02783 inline unsigned Matrix<T>::rows() const {
02784     return nrows;
02785 }
02786
02787 template<typename T>
02788 inline unsigned Matrix<T>::cols() const {
02789     return ncols;
02790 }
02791
02792 template<typename T>
02793 inline Matrix<T> Matrix<T>::transpose() const {
02794     Matrix<T> res(ncols, nrows);
02795     for (unsigned c = 0; c < ncols; c++)
02796         for (unsigned r = 0; r < nrows; r++)
02797             res(c,r) = at(r,c);
02798     return res;
02799 }
02800
02801 template<typename T>
02802 inline Matrix<T> Matrix<T>::ctranspose() const {
02803     Matrix<T> res(ncols, nrows);
02804     for (unsigned c = 0; c < ncols; c++)
02805         for (unsigned r = 0; r < nrows; r++)
02806             res(c,r) = cconj(at(r,c));
02807     return res;
02808 }
02809
02810 template<typename T>
02811 Matrix<T>& Matrix<T>::add(const Matrix<T>& m) {
02812     if (!(m.rows() == rows() && m.cols() == cols())) throw std::runtime_error("Unmatching matrix
dimensions for iadd");
02813     for (unsigned i = 0; i < numel(); i++)
02814         data[i] += m(i);
02815     return *this;
02816 }
02817
02818 template<typename T>
02819 Matrix<T>& Matrix<T>::subtract(const Matrix<T>& m) {
02820     if (!(m.rows() == rows() && m.cols() == cols())) throw std::runtime_error("Unmatching matrix
dimensions for isubtract");
02821     for (unsigned i = 0; i < numel(); i++)
02822         data[i] -= m(i);
02823     return *this;
02824 }
02825
02826 template<typename T>
02827 Matrix<T>& Matrix<T>::mult_hadamard(const Matrix<T>& m) {
02828     if (!(m.rows() == rows() && m.cols() == cols())) throw std::runtime_error("Unmatching matrix
dimensions for ihprod");

```

```

02831
02832     for (unsigned i = 0; i < numel(); i++)
02833         data[i] *= m(i);
02834     return *this;
02835 }
02836
02837 template<typename T>
02838 Matrix<T>& Matrix<T>::add(T s) {
02839     for (auto& x : data)
02840         x += s;
02841     return *this;
02842 }
02843
02844 template<typename T>
02845 Matrix<T>& Matrix<T>::subtract(T s) {
02846     for (auto& x : data)
02847         x -= s;
02848     return *this;
02849 }
02850
02851 template<typename T>
02852 Matrix<T>& Matrix<T>::mult(T s) {
02853     for (auto& x : data)
02854         x *= s;
02855     return *this;
02856 }
02857
02858 template<typename T>
02859 Matrix<T>& Matrix<T>::div(T s) {
02860     for (auto& x : data)
02861         x /= s;
02862     return *this;
02863 }
02864
02865 template<typename T>
02866 void Matrix<T>::add_row_to_another(unsigned to, unsigned from) {
02867     if (!(to < rows() && from < rows())) throw std::out_of_range("Row index out of range");
02868
02869     for (unsigned k = 0; k < cols(); k++)
02870         at(to, k) += at(from, k);
02871 }
02872
02873 template<typename T>
02874 void Matrix<T>::add_col_to_another(unsigned to, unsigned from) {
02875     if (!(to < cols() && from < cols())) throw std::out_of_range("Column index out of range");
02876
02877     for (unsigned k = 0; k < rows(); k++)
02878         at(k, to) += at(k, from);
02879 }
02880
02881 template<typename T>
02882 void Matrix<T>::swap_rows(unsigned i, unsigned j) {
02883     if (!(i < rows() && j < rows())) throw std::out_of_range("Row index out of range");
02884
02885     for (unsigned k = 0; k < cols(); k++) {
02886         T tmp = at(i, k);
02887         at(i, k) = at(j, k);
02888         at(j, k) = tmp;
02889     }
02890 }
02891
02892 template<typename T>
02893 void Matrix<T>::swap_cols(unsigned i, unsigned j) {
02894     if (!(i < cols() && j < cols())) throw std::out_of_range("Column index out of range");
02895
02896     for (unsigned k = 0; k < rows(); k++) {
02897         T tmp = at(k, i);
02898         at(k, i) = at(k, j);
02899         at(k, j) = tmp;
02900     }
02901 }
02902
02903 template<typename T>
02904 inline std::vector<T> Matrix<T>::to_vector() const {
02905     return data;
02906 }
02907
02908 template<typename T>
02909 inline std::vector<T> Matrix<T>::col_to_vector(unsigned col) const {
02910     std::vector<T> vec(rows());
02911     for (unsigned i = 0; i < rows(); i++)
02912         vec[i] = at(i, col);
02913     return vec;
02914 }
02915
02916 template<typename T>
02917 inline std::vector<T> Matrix<T>::row_to_vector(unsigned row) const {

```

```
02918     std::vector<T> vec(cols());
02919     for (unsigned i = 0; i < cols(); i++)
02920         vec[i] = at(row,i);
02921     return vec;
02922 }
02923
02924 template<typename T>
02925 inline void Matrix<T>::col_from_vector(const std::vector<T>& vec, unsigned col) {
02926     if (vec.size() != rows()) throw std::runtime_error("Vector size is not equal to number of rows");
02927     if (col >= cols()) throw std::out_of_range("Column index out of range");
02928
02929     for (unsigned i = 0; i < rows(); i++)
02930         data[col*rows() + i] = vec[i];
02931 }
02932
02933 template<typename T>
02934 inline void Matrix<T>::row_from_vector(const std::vector<T>& vec, unsigned row) {
02935     if (vec.size() != rows()) throw std::runtime_error("Vector size is not equal to number of columns");
02936     if (row >= rows()) throw std::out_of_range("Row index out of range");
02937
02938     for (unsigned i = 0; i < cols(); i++)
02939         data[row + i*rows()] = vec[i];
02940 }
02941
02942 template<typename T>
02943 Matrix<T>::~Matrix() { }
02944
02945 } // namespace Matrix_hpp
02946
02947 #endif // __MATRIX_HPP__
```

