

Matrix HPP

Generated by Doxygen 1.9.8

| | |
|--|-----------|
| 1 Matrix HPP - C++11 library for matrix class container and linear algebra computations | 1 |
| 1.1 Installation | 1 |
| 1.2 Functionality | 1 |
| 1.3 Hello world example | 2 |
| 1.4 Debugging | 2 |
| 1.5 Tests | 2 |
| 1.6 License | 2 |
| 2 Namespace Index | 3 |
| 2.1 Namespace List | 3 |
| 3 Hierarchical Index | 5 |
| 3.1 Class Hierarchy | 5 |
| 4 Class Index | 7 |
| 4.1 Class List | 7 |
| 5 File Index | 9 |
| 5.1 File List | 9 |
| 6 Namespace Documentation | 11 |
| 6.1 Mtx::Util Namespace Reference | 11 |
| 6.1.1 Detailed Description | 11 |
| 6.1.2 Function Documentation | 11 |
| 6.1.2.1 cconj() | 11 |
| 6.1.2.2 creal() | 12 |
| 6.1.2.3 csign() | 12 |
| 7 Class Documentation | 13 |
| 7.1 Mtx::Eigenvalues_result< T > Struct Template Reference | 13 |
| 7.1.1 Detailed Description | 13 |
| 7.2 Mtx::Hessenberg_result< T > Struct Template Reference | 13 |
| 7.2.1 Detailed Description | 14 |
| 7.3 Mtx::Util::is_complex< T > Struct Template Reference | 14 |
| 7.4 Mtx::Util::is_complex< std::complex< T > > Struct Template Reference | 14 |
| 7.5 Mtx::LDL_result< T > Struct Template Reference | 14 |
| 7.5.1 Detailed Description | 15 |
| 7.6 Mtx::LU_result< T > Struct Template Reference | 15 |
| 7.6.1 Detailed Description | 15 |
| 7.7 Mtx::LUP_result< T > Struct Template Reference | 16 |
| 7.7.1 Detailed Description | 16 |
| 7.8 Mtx::Matrix< T > Class Template Reference | 16 |
| 7.8.1 Detailed Description | 18 |
| 7.8.2 Constructor & Destructor Documentation | 19 |

| | |
|--------------------------------------|----|
| 7.8.2.1 Matrix() [1/8] | 19 |
| 7.8.2.2 Matrix() [2/8] | 19 |
| 7.8.2.3 Matrix() [3/8] | 19 |
| 7.8.2.4 Matrix() [4/8] | 19 |
| 7.8.2.5 Matrix() [5/8] | 20 |
| 7.8.2.6 Matrix() [6/8] | 20 |
| 7.8.2.7 Matrix() [7/8] | 20 |
| 7.8.2.8 Matrix() [8/8] | 21 |
| 7.8.2.9 ~Matrix() | 21 |
| 7.8.3 Member Function Documentation | 21 |
| 7.8.3.1 add() [1/2] | 21 |
| 7.8.3.2 add() [2/2] | 21 |
| 7.8.3.3 add_col_to_another() | 22 |
| 7.8.3.4 add_row_to_another() | 22 |
| 7.8.3.5 clear() | 22 |
| 7.8.3.6 col_from_vector() | 23 |
| 7.8.3.7 col_to_vector() | 23 |
| 7.8.3.8 cols() | 23 |
| 7.8.3.9 ctranspose() | 24 |
| 7.8.3.10 div() | 24 |
| 7.8.3.11 exists() | 24 |
| 7.8.3.12 fill() | 25 |
| 7.8.3.13 fill_col() | 25 |
| 7.8.3.14 fill_row() | 25 |
| 7.8.3.15 get_submatrix() | 25 |
| 7.8.3.16 isempty() | 26 |
| 7.8.3.17 isequal() [1/2] | 26 |
| 7.8.3.18 isequal() [2/2] | 26 |
| 7.8.3.19 mult() | 27 |
| 7.8.3.20 mult_col_by_another() | 27 |
| 7.8.3.21 mult_hadamard() | 27 |
| 7.8.3.22 mult_row_by_another() | 28 |
| 7.8.3.23 numel() | 28 |
| 7.8.3.24 operator std::vector< T >() | 28 |
| 7.8.3.25 operator()() [1/2] | 28 |
| 7.8.3.26 operator()() [2/2] | 29 |
| 7.8.3.27 operator=() [1/2] | 29 |
| 7.8.3.28 operator=() [2/2] | 29 |
| 7.8.3.29 ptr() [1/2] | 29 |
| 7.8.3.30 ptr() [2/2] | 30 |
| 7.8.3.31 reshape() | 30 |
| 7.8.3.32 resize() | 30 |

| | |
|---|-----------|
| 7.8.3.33 row_from_vector() | 31 |
| 7.8.3.34 row_to_vector() | 31 |
| 7.8.3.35 rows() | 31 |
| 7.8.3.36 set_submatrix() | 32 |
| 7.8.3.37 shape() | 32 |
| 7.8.3.38 subtract() [1/2] | 32 |
| 7.8.3.39 subtract() [2/2] | 33 |
| 7.8.3.40 swap_cols() | 33 |
| 7.8.3.41 swap_rows() | 33 |
| 7.8.3.42 transpose() | 34 |
| 7.9 Mtx::QR_result< T > Struct Template Reference | 34 |
| 7.9.1 Detailed Description | 34 |
| 7.10 Mtx::singular_matrix_exception Class Reference | 34 |
| 8 File Documentation | 35 |
| 8.1 matrix.hpp File Reference | 35 |
| 8.1.1 Function Documentation | 42 |
| 8.1.1.1 add() [1/2] | 42 |
| 8.1.1.2 add() [2/2] | 42 |
| 8.1.1.3 adj() | 43 |
| 8.1.1.4 chol() | 43 |
| 8.1.1.5 cholinv() | 44 |
| 8.1.1.6 circshift() | 44 |
| 8.1.1.7 circulant() [1/2] | 45 |
| 8.1.1.8 circulant() [2/2] | 45 |
| 8.1.1.9 cofactor() | 46 |
| 8.1.1.10 concatenate_horizontal() | 46 |
| 8.1.1.11 concatenate_vertical() | 47 |
| 8.1.1.12 cond() | 47 |
| 8.1.1.13 ctranspose() | 47 |
| 8.1.1.14 det() | 48 |
| 8.1.1.15 det_lu() | 48 |
| 8.1.1.16 diag() [1/3] | 48 |
| 8.1.1.17 diag() [2/3] | 49 |
| 8.1.1.18 diag() [3/3] | 49 |
| 8.1.1.19 div() | 50 |
| 8.1.1.20 eigenvalues() [1/2] | 50 |
| 8.1.1.21 eigenvalues() [2/2] | 51 |
| 8.1.1.22 eye() | 51 |
| 8.1.1.23 foreach_elem() | 52 |
| 8.1.1.24 foreach_elem_copy() | 52 |
| 8.1.1.25 hessenberg() | 53 |

| | |
|--|----|
| 8.1.1.26 <code>householder_reflection()</code> | 53 |
| 8.1.1.27 <code>imag()</code> | 54 |
| 8.1.1.28 <code>inv()</code> | 54 |
| 8.1.1.29 <code>inv_gauss_jordan()</code> | 55 |
| 8.1.1.30 <code>inv_posdef()</code> | 55 |
| 8.1.1.31 <code>inv_square()</code> | 56 |
| 8.1.1.32 <code>inv_tril()</code> | 56 |
| 8.1.1.33 <code>inv_triu()</code> | 56 |
| 8.1.1.34 <code>ishess()</code> | 57 |
| 8.1.1.35 <code>istril()</code> | 57 |
| 8.1.1.36 <code>istriu()</code> | 58 |
| 8.1.1.37 <code>kron()</code> | 58 |
| 8.1.1.38 <code>ldl()</code> | 58 |
| 8.1.1.39 <code>lu()</code> | 59 |
| 8.1.1.40 <code>lup()</code> | 59 |
| 8.1.1.41 <code>make_complex()</code> [1/2] | 60 |
| 8.1.1.42 <code>make_complex()</code> [2/2] | 60 |
| 8.1.1.43 <code>mult()</code> [1/4] | 61 |
| 8.1.1.44 <code>mult()</code> [2/4] | 62 |
| 8.1.1.45 <code>mult()</code> [3/4] | 62 |
| 8.1.1.46 <code>mult()</code> [4/4] | 63 |
| 8.1.1.47 <code>mult_and_add()</code> | 64 |
| 8.1.1.48 <code>mult_hadamard()</code> | 65 |
| 8.1.1.49 <code>norm_fro()</code> | 65 |
| 8.1.1.50 <code>norm_inf()</code> | 66 |
| 8.1.1.51 <code>norm_p1()</code> | 66 |
| 8.1.1.52 <code>ones()</code> [1/2] | 66 |
| 8.1.1.53 <code>ones()</code> [2/2] | 67 |
| 8.1.1.54 <code>operator"!=""()</code> | 67 |
| 8.1.1.55 <code>operator*()</code> [1/5] | 67 |
| 8.1.1.56 <code>operator*()</code> [2/5] | 68 |
| 8.1.1.57 <code>operator*()</code> [3/5] | 68 |
| 8.1.1.58 <code>operator*()</code> [4/5] | 68 |
| 8.1.1.59 <code>operator*()</code> [5/5] | 68 |
| 8.1.1.60 <code>operator*=""()</code> [1/2] | 69 |
| 8.1.1.61 <code>operator*=""()</code> [2/2] | 69 |
| 8.1.1.62 <code>operator+()</code> [1/3] | 69 |
| 8.1.1.63 <code>operator+()</code> [2/3] | 69 |
| 8.1.1.64 <code>operator+()</code> [3/3] | 70 |
| 8.1.1.65 <code>operator+=""()</code> [1/2] | 70 |
| 8.1.1.66 <code>operator+=""()</code> [2/2] | 70 |
| 8.1.1.67 <code>operator-()</code> [1/2] | 70 |

| | |
|-----------------------------------|----|
| 8.1.1.68 operator-() [2/2] | 71 |
| 8.1.1.69 operator-=() [1/2] | 71 |
| 8.1.1.70 operator-=() [2/2] | 71 |
| 8.1.1.71 operator/() | 71 |
| 8.1.1.72 operator/=() | 72 |
| 8.1.1.73 operator<<() | 72 |
| 8.1.1.74 operator==() | 72 |
| 8.1.1.75 operator^() | 73 |
| 8.1.1.76 operator^()=() | 73 |
| 8.1.1.77 permute_cols() | 73 |
| 8.1.1.78 permute_rows() | 74 |
| 8.1.1.79 permute_rows_and_cols() | 74 |
| 8.1.1.80 pinv() | 75 |
| 8.1.1.81 qr() | 76 |
| 8.1.1.82 qr_householder() | 76 |
| 8.1.1.83 qr_red_gs() | 77 |
| 8.1.1.84 real() | 77 |
| 8.1.1.85 repmat() | 78 |
| 8.1.1.86 rref() | 78 |
| 8.1.1.87 solve_posdef() | 79 |
| 8.1.1.88 solve_square() | 79 |
| 8.1.1.89 solve_tril() | 80 |
| 8.1.1.90 solve_triu() | 81 |
| 8.1.1.91 subtract() [1/2] | 81 |
| 8.1.1.92 subtract() [2/2] | 82 |
| 8.1.1.93 to_string() | 82 |
| 8.1.1.94 trace() | 83 |
| 8.1.1.95 transpose() | 83 |
| 8.1.1.96 tril() | 83 |
| 8.1.1.97 triu() | 84 |
| 8.1.1.98 wilkinson_shift() | 84 |
| 8.1.1.99 zeros() [1/2] | 84 |
| 8.1.1.100 zeros() [2/2] | 85 |
| 8.2 matrix.hpp | 85 |

Chapter 1

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

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

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

1.1 Installation

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

1.2 Functionality

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

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

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

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

1.3 Hello world example

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

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

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

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

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

    auto C = A + B;

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

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

1.4 Debugging

The `MATRIX_STRICT_BOUNDS_CHECK` preprocessor macro controls whether runtime bounds checking is performed for element access operations, e.g., using `operator()`, within the `Matrix` class. When enabled, out-of-bounds access attempts will throw a `std::out_of_range` exception. Please refer to documentation of a specific function for information if it is affected by the `MATRIX_STRICT_BOUNDS_CHECK` preprocessor macro or not.

Disabling bounds checking improves performance but removes protection against errors. It should be disabled only for optimized release builds where peak performance is required.

1.5 Tests

Unit tests are compiled with `make tests`.

1.6 License

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

Chapter 2

Namespace Index

2.1 Namespace List

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

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

Chapter 3

Hierarchical Index

3.1 Class Hierarchy

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

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

Chapter 4

Class Index

4.1 Class List

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

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

Chapter 5

File Index

5.1 File List

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

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

Chapter 6

Namespace Documentation

6.1 Mtx::Util Namespace Reference

Classes

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

Functions

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

6.1.1 Detailed Description

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

6.1.2 Function Documentation

6.1.2.1 cconj()

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

Complex conjugate helper.

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

For real numbers, this function returns the input argument unchanged. For complex numbers, this function calls `std::conj`.

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

6.1.2.2 creal()

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

Complex real part helper.

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

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

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

6.1.2.3 csign()

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

Complex sign helper.

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

For real numbers, this function returns sign bit, i.e., 1 when the value is non-negative and -1 otherwise. For complex numbers, this function calculates $e^{i \cdot \arg(x)}$.

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

Chapter 7

Class Documentation

7.1 Mtx::Eigenvalues_result< T > Struct Template Reference

Result of eigenvalues.

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

Public Attributes

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

7.1.1 Detailed Description

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

Result of eigenvalues.

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

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

- [matrix.hpp](#)

7.2 Mtx::Hessenberg_result< T > Struct Template Reference

Result of Hessenberg decomposition.

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

Public Attributes

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

7.2.1 Detailed Description

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

Result of Hessenberg decomposition.

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

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

- [matrix.hpp](#)

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

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

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

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

- [matrix.hpp](#)

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

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

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

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

- [matrix.hpp](#)

7.5 Mtx::LDL_result< T > Struct Template Reference

Result of LDL decomposition.

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

Public Attributes

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

7.5.1 Detailed Description

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

Result of LDL decomposition.

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

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

- [matrix.hpp](#)

7.6 Mtx::LU_result< T > Struct Template Reference

Result of LU decomposition.

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

Public Attributes

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

7.6.1 Detailed Description

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

Result of LU decomposition.

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

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

- [matrix.hpp](#)

7.7 Mtx::LUP_result< T > Struct Template Reference

Result of LU decomposition with pivoting.

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

Public Attributes

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

7.7.1 Detailed Description

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

Result of LU decomposition with pivoting.

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

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

- [matrix.hpp](#)

7.8 Mtx::Matrix< T > Class Template Reference

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

Public Member Functions

- [Matrix \(\)](#)
Default constructor.
- [Matrix \(unsigned size\)](#)
Square matrix constructor.
- [Matrix \(unsigned nrows, unsigned ncols\)](#)
Rectangular matrix constructor.
- [Matrix \(T x, unsigned nrows, unsigned ncols\)](#)
Rectangular matrix constructor with fill.
- [Matrix \(const T *array, unsigned nrows, unsigned ncols\)](#)
Rectangular matrix constructor with initialization.
- [Matrix \(const std::vector< T > &vec, unsigned nrows, unsigned ncols\)](#)
Rectangular matrix constructor with initialization.
- [Matrix \(std::initializer_list< T > init_list, unsigned nrows, unsigned ncols\)](#)

Rectangular matrix constructor with initialization.

- `Matrix (const Matrix &)`
- `virtual ~Matrix ()`
- `Matrix< T > get_submatrix (unsigned row_first, unsigned row_last, unsigned col_first, unsigned col_last) const`

Extract a submatrix.

- `void set_submatrix (const Matrix< T > &smtx, unsigned row_first, unsigned col_first)`

Embed a submatrix.

- `void clear ()`

Clears the matrix.

- `void reshape (unsigned rows, unsigned cols)`

Matrix dimension reshape.

- `void resize (unsigned rows, unsigned cols)`

Resize the matrix.

- `bool exists (unsigned row, unsigned col) const`

Element exist check.

- `T * ptr (unsigned row, unsigned col)`

Memory pointer.

- `T * ptr ()`

Memory pointer.

- `void fill (T value)`
- `void fill_col (T value, unsigned col)`

Fill column with a scalar.

- `void fill_row (T value, unsigned row)`

Fill row with a scalar.

- `bool isempty () const`

Emptiness check.

- `bool issquare () const`

Squareness check. Check if the matrix is square, i.e., the width of the first and the second dimensions are equal.

- `bool isequal (const Matrix< T > &) const`

Matrix equality check.

- `bool isequal (const Matrix< T > &, T) const`

Matrix equality check with tolerance.

- `unsigned numel () const`

Matrix capacity.

- `unsigned rows () const`

Number of rows.

- `unsigned cols () const`

Number of columns.

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

Matrix shape.

- `Matrix< T > transpose () const`

Transpose a matrix.

- `Matrix< T > ctranspose () const`

Transpose a complex matrix.

- `Matrix< T > & add (const Matrix< T > &)`

Matrix sum (in-place).

- `Matrix< T > & subtract (const Matrix< T > &)`

Matrix subtraction (in-place).

- `Matrix< T > & mult_hadamard (const Matrix< T > &)`

Matrix Hadamard product (in-place).

- [Matrix< T > & add \(T\)](#)
Matrix sum with scalar (in-place).
- [Matrix< T > & subtract \(T\)](#)
Matrix subtraction with scalar (in-place).
- [Matrix< T > & mult \(T\)](#)
Matrix product with scalar (in-place).
- [Matrix< T > & div \(T\)](#)
Matrix division by scalar (in-place).
- [Matrix< T > & operator= \(const Matrix< T > &\)](#)
Matrix assignment.
- [Matrix< T > & operator= \(T\)](#)
Matrix fill operator.
- [operator std::vector< T > \(\) const](#)
Vector cast operator.
- [std::vector< T > to_vector \(\) const](#)
- [T & operator\(\) \(unsigned nel\)](#)
Element access operator (1D)
- [T operator\(\) \(unsigned nel\) const](#)
- [T & at \(unsigned nel\)](#)
- [T at \(unsigned nel\) const](#)
- [T & operator\(\) \(unsigned row, unsigned col\)](#)
Element access operator (2D)
- [T operator\(\) \(unsigned row, unsigned col\) const](#)
- [T & at \(unsigned row, unsigned col\)](#)
- [T at \(unsigned row, unsigned col\) const](#)
- [void add_row_to_another \(unsigned to, unsigned from\)](#)
Row addition.
- [void add_col_to_another \(unsigned to, unsigned from\)](#)
Column addition.
- [void mult_row_by_another \(unsigned to, unsigned from\)](#)
Row multiplication.
- [void mult_col_by_another \(unsigned to, unsigned from\)](#)
Column multiplication.
- [void swap_rows \(unsigned i, unsigned j\)](#)
Row swap.
- [void swap_cols \(unsigned i, unsigned j\)](#)
Column swap.
- [std::vector< T > col_to_vector \(unsigned col\) const](#)
Column to vector.
- [std::vector< T > row_to_vector \(unsigned row\) const](#)
Row to vector.
- [void col_from_vector \(const std::vector< T > &, unsigned col\)](#)
Column from vector.
- [void row_from_vector \(const std::vector< T > &, unsigned row\)](#)
Row from vector.

7.8.1 Detailed Description

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

[Matrix](#) class definition.

7.8.2 Constructor & Destructor Documentation

7.8.2.1 Matrix() [1/8]

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

Default constructor.

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

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

7.8.2.2 Matrix() [2/8]

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

Square matrix constructor.

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

7.8.2.3 Matrix() [3/8]

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

Rectangular matrix constructor.

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

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

7.8.2.4 Matrix() [4/8]

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

Rectangular matrix constructor with fill.

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

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

7.8.2.5 `Matrix()` [5/8]

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

Rectangular matrix constructor with initialization.

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

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

7.8.2.6 `Matrix()` [6/8]

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

Rectangular matrix constructor with initialization.

Constructs a matrix of size *nrows* x *ncols*. The elements of the matrix are initialized using the elements stored in the input `std::vector`. Size of the vector must be equal to the number of matrix elements given by *nrows* and *ncols*.

The elements of the matrix are filled in a column-major order.

Exceptions

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

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

7.8.2.7 `Matrix()` [7/8]

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

Rectangular matrix constructor with initialization.

Constructs a matrix of size *nrows* x *ncols*. The elements of the matrix are initialized using the elements stored in the input `std::initializer_list`. Number of elements in the list must be equal to the number of matrix elements given by *nrows* and *ncols*.

The elements of the matrix are filled in a column-major order.

Exceptions

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

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

7.8.2.8 Matrix() [8/8]

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

Copy constructor.

7.8.2.9 ~Matrix()

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

Destructor.

7.8.3 Member Function Documentation

7.8.3.1 add() [1/2]

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

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

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

Exceptions

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

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

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

7.8.3.2 add() [2/2]

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

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

Adds a scalar *s* to each element of the matrix.

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

7.8.3.3 add_col_to_another()

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

Column addition.

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

Exceptions

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

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

7.8.3.4 add_row_to_another()

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

Row addition.

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

Exceptions

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

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

7.8.3.5 clear()

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

Clears the matrix.

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

7.8.3.6 col_from_vector()

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

Column from vector.

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

Exceptions

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

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

7.8.3.7 col_to_vector()

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

Column to vector.

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

Exceptions

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

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

7.8.3.8 cols()

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

Number of columns.

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

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

[Mtx::permute_cols\(\)](#), [Mtx::permute_rows\(\)](#), [Mtx::permute_rows_and_cols\(\)](#), [Mtx::pinv\(\)](#), [Mtx::Matrix< T >::ptr\(\)](#), [Mtx::qr_householder\(\)](#), [Mtx::qr_red_gs\(\)](#), [Mtx::real\(\)](#), [Mtx::repmat\(\)](#), [Mtx::Matrix< T >::reshape\(\)](#), [Mtx::Matrix< T >::resize\(\)](#), [Mtx::Matrix< T >::row_from_vector\(\)](#), [Mtx::Matrix< T >::row_to_vector\(\)](#), [Mtx::rref\(\)](#), [Mtx::Matrix< T >::set_submatrix\(\)](#), [Mtx::solve_tril\(\)](#), [Mtx::solve_triu\(\)](#), [Mtx::Matrix< T >::subtract\(\)](#), [Mtx::subtract\(\)](#), [Mtx::subtract\(\)](#), [Mtx::Matrix< T >::swap_cols\(\)](#), [Mtx::Matrix< T >::swap_rows\(\)](#), [Mtx::to_string\(\)](#), [Mtx::tril\(\)](#), and [Mtx::triu\(\)](#).

7.8.3.9 ctranspose()

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

Transpose a complex matrix.

Returns a matrix that is a conjugate (Hermitian) transposition of an input matrix. Conjugate transpose applies a conjugate operation to all elements in addition to matrix transposition.

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

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

7.8.3.10 div()

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

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

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

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

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

7.8.3.11 exists()

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

Element exist check.

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

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

7.8.3.12 fill()

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

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

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

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

7.8.3.13 fill_col()

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

Fill column with a scalar.

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

Exceptions

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

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

7.8.3.14 fill_row()

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

Fill row with a scalar.

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

Exceptions

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

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

7.8.3.15 get_submatrix()

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

```

    unsigned row_first,
    unsigned row_last,
    unsigned col_first,
    unsigned col_last ) const

```

Extract a submatrix.

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

Exceptions

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

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

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

7.8.3.16 isempty()

```

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

```

Emptiness check.

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

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

7.8.3.17 isequal() [1/2]

```

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

```

[Matrix](#) equality check.

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

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

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

7.8.3.18 isequal() [2/2]

```

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

```

[Matrix](#) equality check with tolerance.

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

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

7.8.3.19 mult()

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

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

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

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

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

7.8.3.20 mult_col_by_another()

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

Column multiplication.

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

Exceptions

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

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

7.8.3.21 mult_hadamard()

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

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

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

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

Exceptions

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

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

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

7.8.3.22 mult_row_by_another()

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

Row multiplication.

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

Exceptions

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

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

7.8.3.23 numel()

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

[Matrix](#) capacity.

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

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

7.8.3.24 operator std::vector< T >()

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

Vector cast operator.

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

7.8.3.25 operator()() [1/2]

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

Element access operator (1D)

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

Exceptions

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

7.8.3.26 operator() [2/2]

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

Element access operator (2D)

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

Exceptions

| | |
|--------------------------------|---|
| <code>std::out_of_range</code> | when row or column index is out of range of matrix dimensions. Thrown only when MATRIX_STRICT_BOUNDS_CHECK is enabled during compilation. |
|--------------------------------|---|

7.8.3.27 operator=() [1/2]

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

[Matrix](#) assignment.

Performs deep-copy of the matrix.

7.8.3.28 operator=() [2/2]

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

[Matrix](#) fill operator.

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

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

7.8.3.29 ptr() [1/2]

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

Memory pointer.

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

Exceptions

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

7.8.3.30 ptr() [2/2]

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

Memory pointer.

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

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

7.8.3.31 reshape()

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

[Matrix](#) dimension reshape.

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

Exceptions

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

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

7.8.3.32 resize()

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

Resize the matrix.

Clears the content of the matrix and changes it dimensions to be equal to the specified number of rows and columns.

Remark that the content of the matrix is lost after calling the reshape method.

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

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

7.8.3.33 row_from_vector()

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

Row from vector.

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

Exceptions

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

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

7.8.3.34 row_to_vector()

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

Row to vector.

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

Exceptions

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

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

7.8.3.35 rows()

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

Number of rows.

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

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

Mtx::Matrix< T >::mult_col_by_another(), Mtx::Matrix< T >::mult_hadamard(), Mtx::mult_hadamard(), Mtx::Matrix< T >::mult_row_ Mtx::norm_inf(), Mtx::norm_p1(), Mtx::operator<<(), Mtx::permute_cols(), Mtx::permute_rows(), Mtx::permute_rows_and_cols(), Mtx::pinv(), Mtx::Matrix< T >::ptr(), Mtx::qr_householder(), Mtx::qr_red_gs(), Mtx::real(), Mtx::repmat(), Mtx::Matrix< T >::reshape(), Mtx::Matrix< T >::resize(), Mtx::Matrix< T >::row_from_vector(), Mtx::rref(), Mtx::Matrix< T >::set_submatrix(), Mtx::solve_posdef(), Mtx::solve_square(), Mtx::solve_tril(), Mtx::solve_triu(), Mtx::Matrix< T >::subtract(), Mtx::subtract(), Mtx::subtract(), Mtx::Matrix< T >::swap_cols(), Mtx::Matrix< T >::swap_rows(), Mtx::to_string(), Mtx::trace(), Mtx::tril(), Mtx::triu(), and Mtx::wilkinson_shift().

7.8.3.36 set_submatrix()

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

Embed a submatrix.

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

Exceptions

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

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

7.8.3.37 shape()

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

[Matrix](#) shape.

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

7.8.3.38 subtract() [1/2]

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

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

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

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

Exceptions

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

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

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

7.8.3.39 subtract() [2/2]

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

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

Subtracts a scalar *s* from each element of the matrix.

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

7.8.3.40 swap_cols()

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

Column swap.

Swaps element values between two columns.

Exceptions

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

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

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

7.8.3.41 swap_rows()

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

Row swap.

Swaps element values of two columns.

Exceptions

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

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

7.8.3.42 transpose()

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

Transpose a matrix.

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

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

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

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

- [matrix.hpp](#)

7.9 Mtx::QR_result< T > Struct Template Reference

Result of QR decomposition.

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

Public Attributes

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

7.9.1 Detailed Description

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

Result of QR decomposition.

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

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

- [matrix.hpp](#)

7.10 Mtx::singular_matrix_exception Class Reference

Singular matrix exception.

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

Inheritance diagram for [Mtx::singular_matrix_exception](#):

Chapter 8

File Documentation

8.1 matrix.hpp File Reference

Classes

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

Namespaces

- namespace [Mtx::Util](#)

Functions

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

- Circulant matrix from std::vector.*

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

Transpose a matrix.

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

Transpose a complex matrix.

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

Circular shift.

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

Repeat matrix.

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

Horizontal matrix concatenation.

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

Vertical matrix concatenation.

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

Frobenius norm.

 - `template<typename T >`
`double Mtx::norm_p1 (const Matrix< T > &A)`
Matrix $p = 1$ norm (column norm).
 - `template<typename T >`
`double Mtx::norm_inf (const Matrix< T > &A)`
Matrix $p = \infty$ norm (row norm).
 - `template<typename T >`
`Matrix< T > Mtx::tril (const Matrix< T > &A)`
Extract triangular lower part.
 - `template<typename T >`
`Matrix< T > Mtx::triu (const Matrix< T > &A)`
Extract triangular upper part.
 - `template<typename T >`
`bool Mtx::istril (const Matrix< T > &A)`
Lower triangular matrix check.
 - `template<typename T >`
`bool Mtx::istriu (const Matrix< T > &A)`
Lower triangular matrix check.
 - `template<typename T >`
`bool Mtx::ishess (const Matrix< T > &A)`
Hessenberg matrix check.
 - `template<typename T >`
`void Mtx::foreach_elem (Matrix< T > &A, std::function< T(T)> func)`
Applies custom function element-wise in-place.
 - `template<typename T >`
`Matrix< T > Mtx::foreach_elem_copy (const Matrix< T > &A, std::function< T(T)> func)`
Applies custom function element-wise with matrix copy.
 - `template<typename T >`
`Matrix< T > Mtx::permute_rows (const Matrix< T > &A, const std::vector< unsigned > perm)`
Permute rows of the matrix.

- `template<typename T >`
`Matrix< T > Mtx::permute_cols (const Matrix< T > &A, const std::vector< unsigned > perm)`
Permute columns of the matrix.
- `template<typename T >`
`Matrix< T > Mtx::permute_rows_and_cols (const Matrix< T > &A, const std::vector< unsigned > perm_rows, const std::vector< unsigned > perm_cols)`
Permute both rows and columns of the matrix.
- `template<typename T , bool transpose_first = false, bool transpose_second = false>`
`Matrix< T > Mtx::mult (const Matrix< T > &A, const Matrix< T > &B)`
Matrix multiplication.
- `template<typename T , bool transpose_A = false, bool transpose_B = false, bool transpose_C = false>`
`Matrix< T > Mtx::mult_and_add (const Matrix< T > &A, const Matrix< T > &B, const Matrix< T > &C)`
Matrix multiplication with addition.
- `template<typename T , bool transpose_first = false, bool transpose_second = false>`
`Matrix< T > Mtx::mult_hadamard (const Matrix< T > &A, const Matrix< T > &B)`
Matrix Hadamard (element-wise) multiplication.
- `template<typename T , bool transpose_first = false, bool transpose_second = false>`
`Matrix< T > Mtx::add (const Matrix< T > &A, const Matrix< T > &B)`
Matrix addition.
- `template<typename T , bool transpose_first = false, bool transpose_second = false>`
`Matrix< T > Mtx::subtract (const Matrix< T > &A, const Matrix< T > &B)`
Matrix subtraction.
- `template<typename T , bool transpose_matrix = false>`
`std::vector< T > Mtx::mult (const Matrix< T > &A, const std::vector< T > &v)`
Multiplication of matrix by std::vector.
- `template<typename T , bool transpose_matrix = false>`
`std::vector< T > Mtx::mult (const std::vector< T > &v, const Matrix< T > &A)`
Multiplication of std::vector by matrix.
- `template<typename T >`
`Matrix< T > Mtx::add (const Matrix< T > &A, T s)`
Addition of scalar to matrix.
- `template<typename T >`
`Matrix< T > Mtx::subtract (const Matrix< T > &A, T s)`
Subtraction of scalar from matrix.
- `template<typename T >`
`Matrix< T > Mtx::mult (const Matrix< T > &A, T s)`
Multiplication of matrix by scalar.
- `template<typename T >`
`Matrix< T > Mtx::div (const Matrix< T > &A, T s)`
Division of matrix by scalar.
- `template<typename T >`
`std::string Mtx::to_string (const Matrix< T > &A, char col_separator=' ', char row_separator='\n')`
Converts matrix to std::string.
- `template<typename T >`
`std::ostream & Mtx::operator<< (std::ostream &os, const Matrix< T > &A)`
Matrix ostream operator.
- `template<typename T >`
`Matrix< T > Mtx::operator+ (const Matrix< T > &A, const Matrix< T > &B)`
Matrix sum.
- `template<typename T >`
`Matrix< T > Mtx::operator- (const Matrix< T > &A, const Matrix< T > &B)`
Matrix subtraction.

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

- `template<typename T >`
`bool Mtx::operator== (const Matrix< T > &A, const Matrix< T > &b)`
Matrix equality check operator.
- `template<typename T >`
`bool Mtx::operator!= (const Matrix< T > &A, const Matrix< T > &b)`
Matrix non-equality check operator.
- `template<typename T >`
`Matrix< T > Mtx::kron (const Matrix< T > &A, const Matrix< T > &B)`
Kronecker product.
- `template<typename T >`
`Matrix< T > Mtx::adj (const Matrix< T > &A)`
Adjugate matrix.
- `template<typename T >`
`Matrix< T > Mtx::cofactor (const Matrix< T > &A, unsigned p, unsigned q)`
Cofactor matrix.
- `template<typename T >`
`T Mtx::det_lu (const Matrix< T > &A)`
Matrix determinant from on LU decomposition.
- `template<typename T >`
`T Mtx::det (const Matrix< T > &A)`
Matrix determinant.
- `template<typename T >`
`LU_result< T > Mtx::lu (const Matrix< T > &A)`
LU decomposition.
- `template<typename T >`
`LUP_result< T > Mtx::lup (const Matrix< T > &A)`
LU decomposition with pivoting.
- `template<typename T >`
`Matrix< T > Mtx::rref (const Matrix< T > &A, T tol=0)`
Reduced row echelon form.
- `template<typename T >`
`Matrix< T > Mtx::inv_gauss_jordan (const Matrix< T > &A)`
Matrix inverse using Gauss-Jordan elimination.
- `template<typename T >`
`Matrix< T > Mtx::inv_tril (const Matrix< T > &A)`
Matrix inverse for lower triangular matrix.
- `template<typename T >`
`Matrix< T > Mtx::inv_triu (const Matrix< T > &A)`
Matrix inverse for upper triangular matrix.
- `template<typename T >`
`Matrix< T > Mtx::inv_posdef (const Matrix< T > &A)`
Matrix inverse for Hermitian positive-definite matrix.
- `template<typename T >`
`Matrix< T > Mtx::inv_square (const Matrix< T > &A)`
Matrix inverse for general square matrix.
- `template<typename T >`
`Matrix< T > Mtx::inv (const Matrix< T > &A)`
Matrix inverse (universal).
- `template<typename T >`
`Matrix< T > Mtx::pinv (const Matrix< T > &A)`
Moore-Penrose pseudo-inverse.
- `template<typename T >`
`T Mtx::trace (const Matrix< T > &A)`

- Matrix trace.*

 - `template<typename T >`
`double Mtx::cond (const Matrix< T > &A)`
Condition number of a matrix.
- `template<typename T , bool is_upper = false>`
`Matrix< T > Mtx::chol (const Matrix< T > &A)`
Cholesky decomposition.
- `template<typename T >`
`Matrix< T > Mtx::cholinv (const Matrix< T > &A)`
Inverse of Cholesky decomposition.
- `template<typename T >`
`LDL_result< T > Mtx::ldl (const Matrix< T > &A)`
LDL decomposition.
- `template<typename T >`
`QR_result< T > Mtx::qr_red_gs (const Matrix< T > &A)`
Reduced QR decomposition based on Gram-Schmidt method.
- `template<typename T >`
`Matrix< T > Mtx::householder_reflection (const Matrix< T > &a)`
Generate Householder reflection.
- `template<typename T >`
`QR_result< T > Mtx::qr_householder (const Matrix< T > &A, bool calculate_Q=true)`
QR decomposition based on Householder method.
- `template<typename T >`
`QR_result< T > Mtx::qr (const Matrix< T > &A, bool calculate_Q=true)`
QR decomposition.
- `template<typename T >`
`Hessenberg_result< T > Mtx::hessenberg (const Matrix< T > &A, bool calculate_Q=true)`
Hessenberg decomposition.
- `template<typename T >`
`std::complex< T > Mtx::wilkinson_shift (const Matrix< std::complex< T > > &H, T tol=1e-10)`
Wilkinson's shift for complex eigenvalues.
- `template<typename T >`
`Eigenvalues_result< T > Mtx::eigenvalues (const Matrix< std::complex< T > > &A, T tol=1e-12, unsigned max_iter=100)`
Matrix eigenvalues of complex matrix.
- `template<typename T >`
`Eigenvalues_result< T > Mtx::eigenvalues (const Matrix< T > &A, T tol=1e-12, unsigned max_iter=100)`
Matrix eigenvalues of real matrix.
- `template<typename T >`
`Matrix< T > Mtx::solve_triu (const Matrix< T > &U, const Matrix< T > &B)`
Solves the upper triangular system.
- `template<typename T >`
`Matrix< T > Mtx::solve_tril (const Matrix< T > &L, const Matrix< T > &B)`
Solves the lower triangular system.
- `template<typename T >`
`Matrix< T > Mtx::solve_square (const Matrix< T > &A, const Matrix< T > &B)`
Solves the square system.
- `template<typename T >`
`Matrix< T > Mtx::solve_posdef (const Matrix< T > &A, const Matrix< T > &B)`
Solves the positive definite (Hermitian) system.

8.1.1 Function Documentation

8.1.1.1 `add()` [1/2]

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

Matrix addition.

Performs addition of two matrices.

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

Template Parameters

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

Parameters

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

Returns

output matrix of size $N \times M$

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

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

8.1.1.2 `add()` [2/2]

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

Addition of scalar to matrix.

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

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

8.1.1.3 adj()

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

Adjugate matrix.

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

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

Exceptions

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

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

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

8.1.1.4 chol()

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

Cholesky decomposition.

The Cholesky decomposition of a Hermitian positive-definite matrix A is a decomposition of the form $A = LL^H$, where L is a lower triangular matrix with real and positive diagonal entries, and H denotes the conjugate transpose.

Alternatively, the decomposition can be computed as $A = U^H U$ with U being upper-triangular matrix. Selection between lower and upper triangular factor can be done via template parameter.

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

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

Template Parameters

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

Exceptions

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

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

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

8.1.1.5 cholinv()

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

Inverse of Cholesky decomposition.

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

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

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

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

Exceptions

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

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

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

8.1.1.6 circshift()

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

Circular shift.

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

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

Parameters

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

Returns

matrix inverse

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

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

8.1.1.7 circulant() [1/2]

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

Circulant matrix from std::vector.

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

Parameters

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

Returns

circulant matrix

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

8.1.1.8 circulant() [2/2]

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

Circulant matrix from array.

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

Parameters

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

Returns

circulant matrix

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

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

8.1.1.9 cofactor()

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

Cofactor matrix.

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

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

Parameters

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

Exceptions

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

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

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

8.1.1.10 concatenate_horizontal()

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

Horizontal matrix concatenation.

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

Exceptions

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

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

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

8.1.1.11 concatenate_vertical()

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

Vertical matrix concatenation.

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

Exceptions

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

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

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

8.1.1.12 cond()

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

Condition number of a matrix.

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

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

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

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

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

8.1.1.13 ctranspose()

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

Transpose a complex matrix.

Returns a matrix that is a conjugate (Hermitian) transposition of an input matrix. Conjugate transpose applies a conjugate operation to all elements in addition to element transposition.

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

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

8.1.1.14 det()

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

Matrix determinant.

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

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

Exceptions

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

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

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

8.1.1.15 det_lu()

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

Matrix determinant from on LU decomposition.

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

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

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

Exceptions

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

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

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

8.1.1.16 diag() [1/3]

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


Diagonal extraction.

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

Parameters

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

Returns

vector of diagonal elements

Exceptions

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

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

8.1.1.17 diag() [2/3]

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

Diagonal matrix from `std::vector`.

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

Parameters

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

Returns

diagonal matrix

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

8.1.1.18 diag() [3/3]

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

Diagonal matrix from array.

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

Parameters

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

Returns

diagonal matrix

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

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

8.1.1.19 div()

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

Division of matrix by scalar.

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

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

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

8.1.1.20 eigenvalues() [1/2]

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

Matrix eigenvalues of complex matrix.

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

Parameters

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

Returns

structure containing the result and status of eigenvalue calculation

Exceptions

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

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

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

8.1.1.21 `eigenvalues()` [2/2]

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

Matrix eigenvalues of real matrix.

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

Parameters

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

Returns

structure containing the result and status of eigenvalue calculation

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

8.1.1.22 `eye()`

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

Identity matrix.

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

Parameters

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

Returns

zeros matrix

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

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

8.1.1.23 foreach_elem()

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

Applies custom function element-wise in-place.

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

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

Parameters

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

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

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

8.1.1.24 foreach_elem_copy()

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

Applies custom function element-wise with matrix copy.

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

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

Parameters

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

Returns

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

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

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

8.1.1.25 hessenberg()

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

Hessenberg decomposition.

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

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

Parameters

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

Returns

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

Exceptions

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

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

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

8.1.1.26 householder_reflection()

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

Generate Householder reflection.

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

Parameters

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

Returns

column vector with Householder reflection of *a*

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

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

8.1.1.27 imag()

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

Get imaginary part of complex matrix.

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

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

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

8.1.1.28 inv()

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

Matrix inverse (universal).

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

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

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

Exceptions

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

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

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

8.1.1.29 inv_gauss_jordan()

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

Matrix inverse using Gauss-Jordan elimination.

Calculates an inverse of a square matrix recursively using Gauss-Jordan elimination. If the inverse doesn't exist, e.g., because the input matrix was singular, an empty matrix is returned.

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

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

Exceptions

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

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

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

8.1.1.30 inv_posdef()

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

Matrix inverse for Hermitian positive-definite matrix.

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

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

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

Exceptions

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

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

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

8.1.1.31 inv_square()

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

Matrix inverse for general square matrix.

Calculates an inverse of square matrix using matrix.

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

Exceptions

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

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

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

8.1.1.32 inv_tril()

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

Matrix inverse for lower triangular matrix.

Calculates an inverse of lower triangular matrix.

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

Exceptions

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

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

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

8.1.1.33 inv_triu()

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


Matrix inverse for upper triangular matrix.

Calculates an inverse of upper triangular matrix.

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

Exceptions

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

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

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

8.1.1.34 ishess()

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

Hessenberg matrix check.

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

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

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

8.1.1.35 istril()

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

Lower triangular matrix check.

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

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

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

8.1.1.36 istriu()

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

Lower triangular matrix check.

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

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

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

8.1.1.37 kron()

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

Kronecker product.

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

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

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

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

8.1.1.38 ldl()

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

LDL decomposition.

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

$$A = LDL^H$$

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

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

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

Parameters

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

Returns

structure encapsulating calculated L and D

Exceptions

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

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

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

8.1.1.39 lu()

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

LU decomposition.

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

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

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

Parameters

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

Returns

structure containing calculated L and U matrices

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

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

8.1.1.40 lup()

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

LU decomposition with pivoting.

Performs LU factorization with partial pivoting, employing column permutations.

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

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

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

Parameters

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

Returns

structure containing L , U and P .

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

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

8.1.1.41 make_complex() [1/2]

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

Create complex matrix from real matrix.

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

Parameters

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

Returns

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

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

8.1.1.42 make_complex() [2/2]

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

Create complex matrix from real and imaginary matrices.

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

Parameters

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

Returns

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

Exceptions

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

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

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

8.1.1.43 mult() [1/4]

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

Matrix multiplication.

Performs multiplication of two matrices.

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

Template Parameters

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

Parameters

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

Returns

output matrix of size $N \times K$

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

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

8.1.1.44 `mult()` [2/4]

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

Multiplication of matrix by `std::vector`.

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

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

Template Parameters

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

Parameters

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

Returns

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

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

8.1.1.45 `mult()` [3/4]

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

Multiplication of matrix by scalar.

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

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

8.1.1.46 mult() [4/4]

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

Multiplication of std::vector by matrix.

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

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

Template Parameters

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

Parameters

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

Returns

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

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

8.1.1.47 mult_and_add()

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

Matrix multiplication with addition.

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

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

Template Parameters

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

Parameters

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

Returns

output matrix of size $N \times K$

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

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

8.1.1.48 mult_hadamard()

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

Matrix Hadamard (element-wise) multiplication.

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

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

Template Parameters

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

Parameters

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

Returns

output matrix of size $N \times M$

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

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

8.1.1.49 norm_fro()

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

Frobenius norm.

Calculates Frobenius norm of a matrix.

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

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

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

8.1.1.50 norm_inf()

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

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

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

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

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

8.1.1.51 norm_p1()

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

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

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

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

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

8.1.1.52 ones() [1/2]

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

Square matrix of ones.

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

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

Parameters

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

Returns

zeros matrix

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

8.1.1.53 ones() [2/2]

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

Matrix of ones.

Construct a matrix of size *nrows* x *ncols* and fill it with all elements set to 1.

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

Parameters

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

Returns

ones matrix

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

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

8.1.1.54 operator!=(())

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

Matrix non-equality check operator.

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

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

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

8.1.1.55 operator*() [1/5]

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

Matrix product.

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

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

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

8.1.1.56 operator*() [2/5]

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

Matrix and std::vector product.

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

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

8.1.1.57 operator*() [3/5]

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

Matrix product with scalar.

Multiplies each element of the matrix by a scalar s .

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

8.1.1.58 operator*() [4/5]

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

std::vector and matrix product.

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

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

8.1.1.59 operator*() [5/5]

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

Matrix product with scalar.

Multiplies each element of the matrix by a scalar s .

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

8.1.1.60 operator*=() [1/2]

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

Matrix product.

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

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

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

8.1.1.61 operator*=() [2/2]

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

Matrix product with scalar.

Multiplies each element of the matrix by a scalar s .

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

8.1.1.62 operator+() [1/3]

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

Matrix sum.

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

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

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

8.1.1.63 operator+() [2/3]

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

Matrix sum with scalar.

Adds a scalar s to each element of the matrix.

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

8.1.1.64 operator+() [3/3]

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

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

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

8.1.1.65 operator+=() [1/2]

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

Matrix sum.

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

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

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

8.1.1.66 operator+=() [2/2]

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

Matrix sum with scalar.

Adds a scalar s to each element of the matrix.

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

8.1.1.67 operator-() [1/2]

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

Matrix subtraction.

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

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

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

8.1.1.68 operator-() [2/2]

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

Matrix subtraction with scalar.

Subtracts a scalar s from each element of the matrix.

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

8.1.1.69 operator-=() [1/2]

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

Matrix subtraction.

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

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

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

8.1.1.70 operator-=() [2/2]

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

Matrix subtraction with scalar.

Subtracts a scalar s from each element of the matrix.

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

8.1.1.71 operator/()

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

Matrix division by scalar.

Divides each element of the matrix by a scalar s .

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

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

8.1.1.72 operator/=()

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

Matrix division by scalar.

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

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

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

8.1.1.73 operator<<()

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

Matrix ostream operator.

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

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

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

8.1.1.74 operator==()

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

Matrix equality check operator.

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

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

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

8.1.1.75 operator^()

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

Matrix Hadamard product.

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

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

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

8.1.1.76 operator^=()

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

Matrix Hadamard product.

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

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

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

8.1.1.77 permute_cols()

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

Permute columns of the matrix.

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

Parameters

| | |
|--------|--|
| A | input matrix |
| $perm$ | permutation vector with column indices |

Returns

output matrix created by column permutation of *A*

Exceptions

| | |
|---------------------------------|--|
| <code>std::runtime_error</code> | when permutation vector is empty |
| <code>std::out_of_range</code> | when any index in permutation vector is out of range Thrown only when <code>MATRIX_STRICT_BOUNDS_CHECK</code> is enabled during compilation. |

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

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

8.1.1.78 permute_rows()

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

Permute rows of the matrix.

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

Parameters

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

Returns

output matrix created by row permutation of *A*

Exceptions

| | |
|---------------------------------|--|
| <code>std::runtime_error</code> | when permutation vector is empty |
| <code>std::out_of_range</code> | when any index in permutation vector is out of range Thrown only when <code>MATRIX_STRICT_BOUNDS_CHECK</code> is enabled during compilation. |

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

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

8.1.1.79 permute_rows_and_cols()

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

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

Permute both rows and columns of the matrix.

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

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

Parameters

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

Returns

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

Exceptions

| | |
|---------------------------|--|
| <i>std::runtime_error</i> | when any of permutation vectors is empty |
| <i>std::out_of_range</i> | when any index in permutation vector is out of range. Thrown only when MATRIX_STRICT_BOUNDS_CHECK is enabled during compilation. |

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

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

8.1.1.80 pinv()

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

Moore-Penrose pseudo-inverse.

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

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

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

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

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

8.1.1.81 qr()

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

QR decomposition.

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

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

Parameters

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

Returns

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

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

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

8.1.1.82 qr_householder()

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

QR decomposition based on Householder method.

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

This function implements QR decomposition based on Householder reflections method.

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

Parameters

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

Returns

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

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

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

8.1.1.83 qr_red_gs()

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

Reduced QR decomposition based on Gram-Schmidt method.

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

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

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

Parameters

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

Returns

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

Exceptions

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

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

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

8.1.1.84 real()

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

Get real part of complex matrix.

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

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

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

8.1.1.85 repmat()

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

Repeat matrix.

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

Parameters

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

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

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

8.1.1.86 rref()

```
template<typename T >
Matrix< T > Mtx::rref (
    const Matrix< T > & A,
    T tol = 0 )
```

Reduced row echelon form.

Computes the reduced row echelon form of a matrix using the Gauss-Jordan elimination method by applying a sequence of elementary row operations.

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

Parameters

| | |
|-------|--|
| A | input matrix to be reduced |
| tol | numerical precision tolerance to determine zero element, defaults to 0 |

Returns

reduced row echelon form of matrix A

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

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

8.1.1.87 solve_posdef()

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

Solves the positive definite (Hermitian) system.

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

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

Parameters

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

Returns

solution matrix of size $N \times M$.

Exceptions

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

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

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

8.1.1.88 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\(\)](#).

8.1.1.89 solve_tril()

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

Solves the lower triangular system.

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

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

Parameters

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

Returns

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

Exceptions

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

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

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

8.1.1.90 solve_triu()

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

Solves the upper triangular system.

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

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

Parameters

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

Returns

solution matrix of size $N \times M$.

Exceptions

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

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

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

8.1.1.91 subtract() [1/2]

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

Matrix subtraction.

Performs subtraction of two matrices.

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

Template Parameters

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

Parameters

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

Returns

output matrix of size $N \times M$

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

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

8.1.1.92 subtract() [2/2]

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

Subtraction of scalar from matrix.

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

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

8.1.1.93 to_string()

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

Converts matrix to std::string.

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

Parameters

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

Returns

std::string representation of the input matrix

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

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

8.1.1.94 trace()

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

Matrix trace.

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

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

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

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

8.1.1.95 transpose()

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

Transpose a matrix.

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

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

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

8.1.1.96 tril()

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

Extract triangular lower part.

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

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

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

8.1.1.97 triu()

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

Extract triangular upper part.

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

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

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

8.1.1.98 wilkinson_shift()

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

Wilkinson's shift for complex eigenvalues.

Computes Wilkinson's shift value μ for complex eigenvalues of input matrix. Wilkinson's shift is calculated as eigenvalue of the bottom 2 x 2 principal minor closest to the corner entry of the matrix.

Input must be a square matrix in Hessenberg form.

Exceptions

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

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

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

8.1.1.99 zeros() [1/2]

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

Square matrix of zeros.

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

Parameters

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

Returns

zeros matrix

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

8.1.1.100 zeros() [2/2]

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

Matrix of zeros.

Create a matrix of size *nrows* x *ncols* and fill it with all elements set to 0.

Parameters

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

Returns

zeros matrix

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

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

8.2 matrix.hpp

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

```
00001
00002
00003 /* MIT License
00004 *
00005 * Copyright (c) 2024 gc1905
00006 *
00007 * Permission is hereby granted, free of charge, to any person obtaining a copy
00008 * of this software and associated documentation files (the "Software"), to deal
00009 * in the Software without restriction, including without limitation the rights
00010 * to use, copy, modify, merge, publish, distribute, sublicense, and/or sell
00011 * copies of the Software, and to permit persons to whom the Software is
00012 * furnished to do so, subject to the following conditions:
00013 *
00014 * The above copyright notice and this permission notice shall be included in all
00015 * copies or substantial portions of the Software.
00016 *
00017 * THE SOFTWARE IS PROVIDED "AS IS", WITHOUT WARRANTY OF ANY KIND, EXPRESS OR
00018 * IMPLIED, INCLUDING BUT NOT LIMITED TO THE WARRANTIES OF MERCHANTABILITY,
00019 * FITNESS FOR A PARTICULAR PURPOSE AND NONINFRINGEMENT. IN NO EVENT SHALL THE
00020 * AUTHORS OR COPYRIGHT HOLDERS BE LIABLE FOR ANY CLAIM, DAMAGES OR OTHER
00021 * LIABILITY, WHETHER IN AN ACTION OF CONTRACT, TORT OR OTHERWISE, ARISING FROM,
00022 * OUT OF OR IN CONNECTION WITH THE SOFTWARE OR THE USE OR OTHER DEALINGS IN THE
00023 * SOFTWARE.
00024 */
00025
00026 #ifndef __MATRIX_HPP__
00027 #define __MATRIX_HPP__
00028
```

```

00029 #include <ostream>
00030 #include <complex>
00031 #include <vector>
00032 #include <initializer_list>
00033 #include <limits>
00034 #include <functional>
00035 #include <algorithm>
00036 #include <utility>
00037
00038 namespace Mtx {
00039
00040 template<typename T> class Matrix;
00041
00042 namespace Util {
00043     template<class T> struct is_complex : std::false_type {};
00044     template<class T> struct is_complex<std::complex<T> > : std::true_type {};
00045
00046     template<typename T, typename std::enable_if<!is_complex<T>::value,int>::type = 0>
00047     inline T cconj(T x) {
00048         return x;
00049     }
00050
00051     template<typename T, typename std::enable_if<is_complex<T>::value,int>::type = 0>
00052     inline T cconj(T x) {
00053         return std::conj(x);
00054     }
00055
00056     template<typename T, typename std::enable_if<!is_complex<T>::value,int>::type = 0>
00057     inline T csign(T x) {
00058         return (x > static_cast<T>(0)) ? static_cast<T>(1) : static_cast<T>(-1);
00059     }
00060
00061     template<typename T, typename std::enable_if<is_complex<T>::value,int>::type = 0>
00062     inline T csign(T x) {
00063         auto x_arg = std::arg(x);
00064         T y(0, x_arg);
00065         return std::exp(y);
00066     }
00067
00068     template<typename T, typename std::enable_if<!is_complex<T>::value,int>::type = 0>
00069     inline T creal(std::complex<T> x) {
00070         return std::real(x);
00071     }
00072
00073     template<typename T, typename std::enable_if<!is_complex<T>::value,int>::type = 0>
00074     inline T creal(T x) {
00075         return x;
00076     }
00077 } // namespace Util
00078
00079 class singular_matrix_exception : public std::domain_error {
00080 public:
00081     singular_matrix_exception(const std::string& message) : std::domain_error(message) {}
00082 };
00083
00084 template<typename T>
00085 struct LU_result {
00086     Matrix<T> L;
00087     Matrix<T> U;
00088 };
00089
00090 template<typename T>
00091 struct LUP_result {
00092     Matrix<T> L;
00093     Matrix<T> U;
00094     std::vector<unsigned> P;
00095 };
00096
00097 template<typename T>
00098 struct QR_result {
00099     Matrix<T> Q;
00100     Matrix<T> R;
00101 };
00102
00103 template<typename T>
00104 struct Hessenberg_result {
00105     Matrix<T> H;
00106     Matrix<T> Q;
00107 };
00108
00109 template<typename T>
00110 struct LDL_result {

```

```

00190     Matrix<T> L;
00191
00194     std::vector<T> d;
00195 };
00196
00201 template<typename T>
00202 struct Eigenvalues_result {
00205     std::vector<std::complex<T>> eig;
00206
00209     bool converged;
00210
00213     T err;
00214 };
00215
00216
00224 template<typename T>
00225 inline Matrix<T> zeros(unsigned nrows, unsigned ncols) {
00226     return Matrix<T>(static_cast<T>(0), nrows, ncols);
00227 }
00228
00235 template<typename T>
00236 inline Matrix<T> zeros(unsigned n) {
00237     return zeros<T>(n,n);
00238 }
00239
00250 template<typename T>
00251 inline Matrix<T> ones(unsigned nrows, unsigned ncols) {
00252     return Matrix<T>(static_cast<T>(1), nrows, ncols);
00253 }
00254
00264 template<typename T>
00265 inline Matrix<T> ones(unsigned n) {
00266     return ones<T>(n,n);
00267 }
00268
00276 template<typename T>
00277 Matrix<T> eye(unsigned n) {
00278     Matrix<T> A(static_cast<T>(0), n, n);
00279     for (unsigned i = 0; i < n; i++)
00280         A(i,i) = static_cast<T>(1);
00281     return A;
00282 }
00283
00292 template<typename T>
00293 Matrix<T> diag(const T* array, size_t n) {
00294     Matrix<T> A(static_cast<T>(0), n, n);
00295     for (unsigned i = 0; i < n; i++) {
00296         A(i,i) = array[i];
00297     }
00298     return A;
00299 }
00300
00309 template<typename T>
00310 inline Matrix<T> diag(const std::vector<T>& v) {
00311     return diag(v.data(), v.size());
00312 }
00313
00323 template<typename T>
00324 std::vector<T> diag(const Matrix<T>& A) {
00325     if (! A.issquare()) throw std::runtime_error("Input matrix is not square");
00326
00327     std::vector<T> v;
00328     v.resize(A.rows());
00329
00330     for (unsigned i = 0; i < A.rows(); i++)
00331         v[i] = A(i,i);
00332     return v;
00333 }
00334
00343 template<typename T>
00344 Matrix<T> circulant(const T* array, unsigned n) {
00345     Matrix<T> A(n, n);
00346     for (unsigned j = 0; j < n; j++)
00347         for (unsigned i = 0; i < n; i++)
00348             A((i+j) % n, j) = array[i];
00349     return A;
00350 }
00351
00363 template<typename T>
00364 Matrix<std::complex<T>> make_complex(const Matrix<T>& Re, const Matrix<T>& Im) {
00365     if (Re.rows() != Im.rows() || Re.cols() != Im.cols()) throw std::runtime_error("Size of input
matrices does not match");
00366
00367     Matrix<std::complex<T>> C(Re.rows(), Re.cols());
00368     for (unsigned n = 0; n < Re.numel(); n++) {
00369         C(n).real(Re(n));
00370         C(n).imag(Im(n));

```

```

00371     }
00372
00373     return C;
00374 }
00375
00383 template<typename T>
00384 Matrix<std::complex<T>> make_complex(const Matrix<T>& Re) {
00385     Matrix<std::complex<T>> C(Re.rows(), Re.cols());
00386
00387     for (unsigned n = 0; n < Re.numel(); n++) {
00388         C(n).real(Re(n));
00389         C(n).imag(static_cast<T>(0));
00390     }
00391
00392     return C;
00393 }
00394
00399 template<typename T>
00400 Matrix<T> real(const Matrix<std::complex<T>& C) {
00401     Matrix<T> Re(C.rows(), C.cols());
00402
00403     for (unsigned n = 0; n < C.numel(); n++)
00404         Re(n) = C(n).real();
00405
00406     return Re;
00407 }
00408
00413 template<typename T>
00414 Matrix<T> imag(const Matrix<std::complex<T>& C) {
00415     Matrix<T> Re(C.rows(), C.cols());
00416
00417     for (unsigned n = 0; n < C.numel(); n++)
00418         Re(n) = C(n).imag();
00419
00420     return Re;
00421 }
00422
00431 template<typename T>
00432 inline Matrix<T> circulant(const std::vector<T>& v) {
00433     return circulant(v.data(), v.size());
00434 }
00435
00440 template<typename T>
00441 inline Matrix<T> transpose(const Matrix<T>& A) {
00442     return A.transpose();
00443 }
00444
00450 template<typename T>
00451 inline Matrix<T> ctranspose(const Matrix<T>& A) {
00452     return A.ctranspose();
00453 }
00454
00467 template<typename T>
00468 Matrix<T> circshift(const Matrix<T>& A, int row_shift, int col_shift) {
00469     Matrix<T> B(A.rows(), A.cols());
00470     for (int i = 0; i < A.rows(); i++) {
00471         int ii = (i + row_shift) % A.rows();
00472         for (int j = 0; j < A.cols(); j++) {
00473             int jj = (j + col_shift) % A.cols();
00474             B(ii, jj) = A(i, j);
00475         }
00476     }
00477     return B;
00478 }
00479
00488 template<typename T>
00489 Matrix<T> repmat(const Matrix<T>& A, unsigned m, unsigned n) {
00490     Matrix<T> B(m * A.rows(), n * A.cols());
00491
00492     for (unsigned cb = 0; cb < n; cb++)
00493         for (unsigned rb = 0; rb < m; rb++)
00494             for (unsigned c = 0; c < A.cols(); c++)
00495                 for (unsigned r = 0; r < A.rows(); r++)
00496                     B(rb*A.rows() + r, cb*A.cols() + c) = A(r, c);
00497
00498     return B;
00499 }
00500
00507 template<typename T>
00508 Matrix<T> concatenate_horizontal(const Matrix<T>& A, const Matrix<T>& B) {
00509     if (A.rows() != B.rows()) throw std::runtime_error("Unmatching number of rows for horizontal
concatenation");
00510
00511     Matrix<T> C(A.rows(), A.cols() + B.cols());
00512
00513     for (unsigned c = 0; c < A.cols(); c++)
00514         for (unsigned r = 0; r < A.rows(); r++)

```



```

00515         C(r,c) = A(r,c);
00516
00517     for (unsigned c = 0; c < B.cols(); c++)
00518     for (unsigned r = 0; r < B.rows(); r++)
00519         C(r,c+A.cols()) = B(r,c);
00520
00521     return C;
00522 }
00523
00530 template<typename T>
00531 Matrix<T> concatenate_vertical(const Matrix<T>& A, const Matrix<T>& B) {
00532     if (A.cols() != B.cols()) throw std::runtime_error("Unmatching number of columns for vertical
concatenation");
00533
00534     Matrix<T> C(A.rows() + B.rows(), A.cols());
00535
00536     for (unsigned c = 0; c < A.cols(); c++)
00537     for (unsigned r = 0; r < A.rows(); r++)
00538         C(r,c) = A(r,c);
00539
00540     for (unsigned c = 0; c < B.cols(); c++)
00541     for (unsigned r = 0; r < B.rows(); r++)
00542         C(r+A.rows(),c) = B(r,c);
00543
00544     return C;
00545 }
00546
00553 template<typename T>
00554 double norm_fro(const Matrix<T>& A) {
00555     double sum = 0;
00556
00557     for (unsigned i = 0; i < A.numel(); i++)
00558         sum += Util::creal(A(i) * Util::conj(A(i)));
00559
00560     return std::sqrt(sum);
00561 }
00562
00568 template<typename T>
00569 double norm_p1(const Matrix<T>& A) {
00570     double max_sum = 0.0;
00571
00572     for (unsigned c = 0; c < A.cols(); c++) {
00573         double sum = 0.0;
00574
00575         for (unsigned r = 0; r < A.rows(); r++)
00576             sum += std::abs(A(r,c));
00577
00578         if (sum > max_sum)
00579             max_sum = sum;
00580     }
00581
00582     return max_sum;
00583 }
00584
00590 template<typename T>
00591 double norm_inf(const Matrix<T>& A) {
00592     double max_sum = 0.0;
00593
00594     for (unsigned r = 0; r < A.rows(); r++) {
00595         double sum = 0.0;
00596
00597         for (unsigned c = 0; c < A.cols(); c++)
00598             sum += std::abs(A(r,c));
00599
00600         if (sum > max_sum)
00601             max_sum = sum;
00602     }
00603
00604     return max_sum;
00605 }
00606
00612 template<typename T>
00613 Matrix<T> tril(const Matrix<T>& A) {
00614     Matrix<T> B(A);
00615
00616     for (unsigned row = 0; row < B.rows(); row++)
00617     for (unsigned col = row+1; col < B.cols(); col++)
00618         B(row,col) = static_cast<T>(0);
00619
00620     return B;
00621 }
00622
00628 template<typename T>
00629 Matrix<T> triu(const Matrix<T>& A) {
00630     Matrix<T> B(A);
00631
00632     for (unsigned col = 0; col < B.cols(); col++)

```

```

00633     for (unsigned row = col+1; row < B.rows(); row++)
00634         B(row,col) = static_cast<T>(0);
00635
00636     return B;
00637 }
00638
00644 template<typename T>
00645 bool istril(const Matrix<T>& A) {
00646     for (unsigned row = 0; row < A.rows(); row++)
00647         for (unsigned col = row+1; col < A.cols(); col++)
00648             if (A(row,col) != static_cast<T>(0)) return false;
00649     return true;
00650 }
00651
00657 template<typename T>
00658 bool istriu(const Matrix<T>& A) {
00659     for (unsigned col = 0; col < A.cols(); col++)
00660         for (unsigned row = col+1; row < A.rows(); row++)
00661             if (A(row,col) != static_cast<T>(0)) return false;
00662     return true;
00663 }
00664
00670 template<typename T>
00671 bool ishess(const Matrix<T>& A) {
00672     if (!A.issquare())
00673         return false;
00674     for (unsigned row = 2; row < A.rows(); row++)
00675         for (unsigned col = 0; col < row-2; col++)
00676             if (A(row,col) != static_cast<T>(0)) return false;
00677     return true;
00678 }
00679
00691 template<typename T>
00692 inline void foreach_elem(Matrix<T>& A, std::function<T(T)> func) {
00693     for (unsigned i = 0; i < A.numel(); i++)
00694         A(i) = func(A(i));
00695 }
00696
00709 template<typename T>
00710 inline Matrix<T> foreach_elem_copy(const Matrix<T>& A, std::function<T(T)> func) {
00711     Matrix<T> B(A);
00712     foreach_elem(B, func);
00713     return B;
00714 }
00715
00731 template<typename T>
00732 Matrix<T> permute_rows(const Matrix<T>& A, const std::vector<unsigned> perm) {
00733     if (perm.empty()) throw std::runtime_error("Permutation vector is empty");
00734
00735     #ifdef MATRIX_STRICT_BOUNDS_CHECK
00736         for (unsigned p = 0; p < perm.size(); p++)
00737             if (!{perm[p] < A.rows()}) throw std::out_of_range("Index in permutation vector out of range");
00738     #endif
00739
00740     Matrix<T> B(perm.size(), A.cols());
00741
00742     for (unsigned p = 0; p < perm.size(); p++)
00743         for (unsigned c = 0; c < A.cols(); c++)
00744             B(p,c) = A(perm[p],c);
00745
00746     return B;
00747 }
00748
00763 template<typename T>
00764 Matrix<T> permute_cols(const Matrix<T>& A, const std::vector<unsigned> perm) {
00765     if (perm.empty()) throw std::runtime_error("Permutation vector is empty");
00766
00767     #ifdef MATRIX_STRICT_BOUNDS_CHECK
00768         for (unsigned p = 0; p < perm.size(); p++)
00769             if (!{perm[p] < A.cols()}) throw std::out_of_range("Index in permutation vector out of range");
00770     #endif
00771
00772     Matrix<T> B(A.rows(), perm.size());
00773
00774     for (unsigned p = 0; p < perm.size(); p++)
00775         for (unsigned r = 0; r < A.rows(); r++)
00776             B(r,p) = A(r,perm[p]);
00777
00778     return B;
00779 }
00780
00798 template<typename T>
00799 Matrix<T> permute_rows_and_cols(const Matrix<T>& A, const std::vector<unsigned> perm_rows, const
std::vector<unsigned> perm_cols) {
00800     if (perm_rows.empty()) throw std::runtime_error("Row permutation vector is empty");
00801     if (perm_cols.empty()) throw std::runtime_error("Column permutation vector is empty");
00802

```

```

00803 #ifndef MATRIX_STRICT_BOUNDS_CHECK
00804     for (unsigned pc = 0; pc < perm_cols.size(); pc++)
00805         if (!perm_cols[pc] < A.cols()) throw std::out_of_range("Column index in permutation vector out
of range");
00806
00807     for (unsigned pr = 0; pr < perm_rows.size(); pr++)
00808         if (!perm_rows[pr] < A.rows()) throw std::out_of_range("Row index in permutation vector out of
range");
00809 #endif
00810
00811     Matrix<T> B(perm_rows.size(), perm_cols.size());
00812
00813     for (unsigned pc = 0; pc < perm_cols.size(); pc++)
00814         for (unsigned pr = 0; pr < perm_rows.size(); pr++)
00815             B(pr,pc) = A(perm_rows[pr],perm_cols[pc]);
00816
00817     return B;
00818 }
00819
00835 template<typename T, bool transpose_first = false, bool transpose_second = false>
00836 Matrix<T> mult(const Matrix<T>& A, const Matrix<T>& B) {
00837     // Adjust dimensions based on transpositions
00838     unsigned rows_A = transpose_first ? A.cols() : A.rows();
00839     unsigned cols_A = transpose_first ? A.rows() : A.cols();
00840     unsigned rows_B = transpose_second ? B.cols() : B.rows();
00841     unsigned cols_B = transpose_second ? B.rows() : B.cols();
00842
00843     if (cols_A != rows_B) throw std::runtime_error("Unmatching matrix dimensions for mult");
00844
00845     Matrix<T> C(static_cast<T>(0), rows_A, cols_B);
00846
00847     for (unsigned i = 0; i < rows_A; i++)
00848         for (unsigned j = 0; j < cols_B; j++)
00849             for (unsigned k = 0; k < cols_A; k++)
00850                 C(i,j) += (transpose_first ? Util::cconj(A(k,i)) : A(i,k)) *
00851                     (transpose_second ? Util::cconj(B(j,k)) : B(k,j));
00852
00853     return C;
00854 }
00855
00873 template<typename T, bool transpose_A = false, bool transpose_B = false, bool transpose_C = false>
00874 Matrix<T> mult_and_add(const Matrix<T>& A, const Matrix<T>& B, const Matrix<T>& C) {
00875     // Adjust dimensions based on transpositions
00876     unsigned rows_A = transpose_A ? A.cols() : A.rows();
00877     unsigned cols_A = transpose_A ? A.rows() : A.cols();
00878     unsigned rows_B = transpose_B ? B.cols() : B.rows();
00879     unsigned cols_B = transpose_B ? B.rows() : B.cols();
00880     unsigned rows_C = transpose_C ? C.cols() : C.rows();
00881     unsigned cols_C = transpose_C ? C.rows() : C.cols();
00882
00883     if ((cols_A != rows_B) || (rows_A != rows_C) || (cols_B != cols_C))
00884         throw std::runtime_error("Unmatching matrix dimensions for mult_and_add");
00885
00886     Matrix<T> D(rows_C, cols_C);
00887
00888     for (unsigned i = 0; i < rows_A; i++) {
00889         for (unsigned j = 0; j < cols_B; j++) {
00890             D(i,j) = transpose_C ? Util::cconj(C(j,i)) : C(i,j);
00891             for (unsigned k = 0; k < cols_A; k++) {
00892                 D(i,j) += (transpose_A ? Util::cconj(A(k,i)) : A(i,k)) *
00893                     (transpose_B ? Util::cconj(B(j,k)) : B(k,j));
00894             }
00895         }
00896     }
00897
00898     return D;
00899 }
00900
00916 template<typename T, bool transpose_first = false, bool transpose_second = false>
00917 Matrix<T> mult_hadamard(const Matrix<T>& A, const Matrix<T>& B) {
00918     // Adjust dimensions based on transpositions
00919     unsigned rows_A = transpose_first ? A.cols() : A.rows();
00920     unsigned cols_A = transpose_first ? A.rows() : A.cols();
00921     unsigned rows_B = transpose_second ? B.cols() : B.rows();
00922     unsigned cols_B = transpose_second ? B.rows() : B.cols();
00923
00924     if ((rows_A != rows_B) || (cols_A != cols_B)) throw std::runtime_error("Unmatching matrix dimensions
for mult_hadamard");
00925
00926     Matrix<T> C(static_cast<T>(0), rows_A, cols_A);
00927
00928     for (unsigned i = 0; i < rows_A; i++)
00929         for (unsigned j = 0; j < cols_A; j++)
00930             C(i,j) += (transpose_first ? Util::cconj(A(j,i)) : A(i,j)) *
00931                 (transpose_second ? Util::cconj(B(j,i)) : B(i,j));
00932
00933     return C;

```

```

00934 }
00935
00951 template<typename T, bool transpose_first = false, bool transpose_second = false>
00952 Matrix<T> add(const Matrix<T>& A, const Matrix<T>& B) {
00953     // Adjust dimensions based on transpositions
00954     unsigned rows_A = transpose_first ? A.cols() : A.rows();
00955     unsigned cols_A = transpose_first ? A.rows() : A.cols();
00956     unsigned rows_B = transpose_second ? B.cols() : B.rows();
00957     unsigned cols_B = transpose_second ? B.rows() : B.cols();
00958
00959     if ((rows_A != rows_B) || (cols_A != cols_B)) throw std::runtime_error("Unmatching matrix dimensions
for add");
00960
00961     Matrix<T> C(static_cast<T>(0), rows_A, cols_A);
00962
00963     for (unsigned i = 0; i < rows_A; i++)
00964         for (unsigned j = 0; j < cols_A; j++)
00965             C(i,j) += (transpose_first ? Util::cconj(A(j,i)) : A(i,j)) +
00966                       (transpose_second ? Util::cconj(B(j,i)) : B(i,j));
00967
00968     return C;
00969 }
00970
00986 template<typename T, bool transpose_first = false, bool transpose_second = false>
00987 Matrix<T> subtract(const Matrix<T>& A, const Matrix<T>& B) {
00988     // Adjust dimensions based on transpositions
00989     unsigned rows_A = transpose_first ? A.cols() : A.rows();
00990     unsigned cols_A = transpose_first ? A.rows() : A.cols();
00991     unsigned rows_B = transpose_second ? B.cols() : B.rows();
00992     unsigned cols_B = transpose_second ? B.rows() : B.cols();
00993
00994     if ((rows_A != rows_B) || (cols_A != cols_B)) throw std::runtime_error("Unmatching matrix dimensions
for subtract");
00995
00996     Matrix<T> C(static_cast<T>(0), rows_A, cols_A);
00997
00998     for (unsigned i = 0; i < rows_A; i++)
00999         for (unsigned j = 0; j < cols_A; j++)
01000             C(i,j) += (transpose_first ? Util::cconj(A(j,i)) : A(i,j)) -
01001                       (transpose_second ? Util::cconj(B(j,i)) : B(i,j));
01002
01003     return C;
01004 }
01005
01021 template<typename T, bool transpose_matrix = false>
01022 std::vector<T> mult(const Matrix<T>& A, const std::vector<T>& v) {
01023     // Adjust dimensions based on transpositions
01024     unsigned rows_A = transpose_matrix ? A.cols() : A.rows();
01025     unsigned cols_A = transpose_matrix ? A.rows() : A.cols();
01026
01027     if (cols_A != v.size()) throw std::runtime_error("Unmatching matrix dimensions for mult");
01028
01029     std::vector<T> u(rows_A, static_cast<T>(0));
01030     for (unsigned r = 0; r < rows_A; r++)
01031         for (unsigned c = 0; c < cols_A; c++)
01032             u[r] += v[c] * (transpose_matrix ? Util::cconj(A(c,r)) : A(r,c));
01033
01034     return u;
01035 }
01036
01052 template<typename T, bool transpose_matrix = false>
01053 std::vector<T> mult(const std::vector<T>& v, const Matrix<T>& A) {
01054     // Adjust dimensions based on transpositions
01055     unsigned rows_A = transpose_matrix ? A.cols() : A.rows();
01056     unsigned cols_A = transpose_matrix ? A.rows() : A.cols();
01057
01058     if (rows_A != v.size()) throw std::runtime_error("Unmatching matrix dimensions for mult");
01059
01060     std::vector<T> u(cols_A, static_cast<T>(0));
01061     for (unsigned c = 0; c < cols_A; c++)
01062         for (unsigned r = 0; r < rows_A; r++)
01063             u[c] += v[r] * (transpose_matrix ? Util::cconj(A(c,r)) : A(r,c));
01064
01065     return u;
01066 }
01067
01073 template<typename T>
01074 Matrix<T> add(const Matrix<T>& A, T s) {
01075     Matrix<T> B(A.rows(), A.cols());
01076     for (unsigned i = 0; i < A.numel(); i++)
01077         B(i) = A(i) + s;
01078     return B;
01079 }
01080
01086 template<typename T>
01087 Matrix<T> subtract(const Matrix<T>& A, T s) {
01088     Matrix<T> B(A.rows(), A.cols());

```

```

01089     for (unsigned i = 0; i < A.numel(); i++)
01090         B(i) = A(i) - s;
01091     return B;
01092 }
01093
01094 template<typename T>
01100 Matrix<T> mult(const Matrix<T>& A, T s) {
01101     Matrix<T> B(A.rows(), A.cols());
01102     for (unsigned i = 0; i < A.numel(); i++)
01103         B(i) = A(i) * s;
01104     return B;
01105 }
01106
01112 template<typename T>
01113 Matrix<T> div(const Matrix<T>& A, T s) {
01114     Matrix<T> B(A.rows(), A.cols());
01115     for (unsigned i = 0; i < A.numel(); i++)
01116         B(i) = A(i) / s;
01117     return B;
01118 }
01119
01131 template<typename T>
01132 std::string to_string(const Matrix<T>& A, char col_separator = ' ', char row_separator = '\n') {
01133     std::stringstream ss;
01134     for (unsigned row = 0; row < A.rows(); row++) {
01135         for (unsigned col = 0; col < A.cols(); col++)
01136             ss << A(row,col) << col_separator;
01137         if (row < static_cast<unsigned>(A.rows()-1)) ss << row_separator;
01138     }
01139     return ss.str();
01140 }
01141
01150 template<typename T>
01151 std::ostream& operator<<(std::ostream& os, const Matrix<T>& A) {
01152     for (unsigned row = 0; row < A.rows(); row++) {
01153         for (unsigned col = 0; col < A.cols(); col++)
01154             os << A(row,col) << " ";
01155         if (row < static_cast<unsigned>(A.rows()-1)) os << std::endl;
01156     }
01157     return os;
01158 }
01159
01164 template<typename T>
01165 inline Matrix<T> operator+(const Matrix<T>& A, const Matrix<T>& B) {
01166     return add(A,B);
01167 }
01168
01173 template<typename T>
01174 inline Matrix<T> operator-(const Matrix<T>& A, const Matrix<T>& B) {
01175     return subtract(A,B);
01176 }
01177
01183 template<typename T>
01184 inline Matrix<T> operator^(const Matrix<T>& A, const Matrix<T>& B) {
01185     return mult_hadamard(A,B);
01186 }
01187
01192 template<typename T>
01193 inline Matrix<T> operator*(const Matrix<T>& A, const Matrix<T>& B) {
01194     return mult(A,B);
01195 }
01196
01202 template<typename T>
01203 inline std::vector<T> operator*(const Matrix<T>& A, const std::vector<T>& v) {
01204     return mult(A,v);
01205 }
01206
01212 template<typename T>
01213 inline std::vector<T> operator*(const std::vector<T>& v, const Matrix<T>& A) {
01214     return mult(v,A);
01215 }
01216
01221 template<typename T>
01222 inline Matrix<T> operator+(const Matrix<T>& A, T s) {
01223     return add(A,s);
01224 }
01225
01230 template<typename T>
01231 inline Matrix<T> operator-(const Matrix<T>& A, T s) {
01232     return subtract(A,s);
01233 }
01234
01239 template<typename T>
01240 inline Matrix<T> operator*(const Matrix<T>& A, T s) {
01241     return mult(A,s);
01242 }
01243

```

```

01248 template<typename T>
01249 inline Matrix<T> operator/(const Matrix<T>& A, T s) {
01250     return div(A,s);
01251 }
01252
01256 template<typename T>
01257 inline Matrix<T> operator+(T s, const Matrix<T>& A) {
01258     return add(A,s);
01259 }
01260
01265 template<typename T>
01266 inline Matrix<T> operator*(T s, const Matrix<T>& A) {
01267     return mult(A,s);
01268 }
01269
01274 template<typename T>
01275 inline Matrix<T>& operator+=(Matrix<T>& A, const Matrix<T>& B) {
01276     return A.add(B);
01277 }
01278
01283 template<typename T>
01284 inline Matrix<T>& operator-=(Matrix<T>& A, const Matrix<T>& B) {
01285     return A.subtract(B);
01286 }
01287
01292 template<typename T>
01293 inline Matrix<T>& operator*=(Matrix<T>& A, const Matrix<T>& B) {
01294     A = mult(A,B);
01295     return A;
01296 }
01297
01303 template<typename T>
01304 inline Matrix<T>& operator^=(Matrix<T>& A, const Matrix<T>& B) {
01305     return A.mult_hadamard(B);
01306 }
01307
01312 template<typename T>
01313 inline Matrix<T>& operator+=(Matrix<T>& A, T s) {
01314     return A.add(s);
01315 }
01316
01321 template<typename T>
01322 inline Matrix<T>& operator-=(Matrix<T>& A, T s) {
01323     return A.subtract(s);
01324 }
01325
01330 template<typename T>
01331 inline Matrix<T>& operator*=(Matrix<T>& A, T s) {
01332     return A.mult(s);
01333 }
01334
01339 template<typename T>
01340 inline Matrix<T>& operator/=(Matrix<T>& A, T s) {
01341     return A.div(s);
01342 }
01343
01348 template<typename T>
01349 inline bool operator==(const Matrix<T>& A, const Matrix<T>& b) {
01350     return A.isequal(b);
01351 }
01352
01357 template<typename T>
01358 inline bool operator!=(const Matrix<T>& A, const Matrix<T>& b) {
01359     return !(A.isequal(b));
01360 }
01361
01369 template<typename T>
01370 Matrix<T> kron(const Matrix<T>& A, const Matrix<T>& B) {
01371     const unsigned rows_A = A.rows();
01372     const unsigned cols_A = A.cols();
01373     const unsigned rows_B = B.rows();
01374     const unsigned cols_B = B.cols();
01375
01376     unsigned rows_C = rows_A * rows_B;
01377     unsigned cols_C = cols_A * cols_B;
01378
01379     Matrix<T> C(rows_C, cols_C);
01380
01381     for (unsigned i = 0; i < rows_A; i++)
01382         for (unsigned j = 0; j < cols_A; j++)
01383             for (unsigned k = 0; k < rows_B; k++)
01384                 for (unsigned l = 0; l < cols_B; l++)
01385                     C(i*rows_B + k, j*cols_B + l) = A(i,j) * B(k,l);
01386
01387     return C;
01388 }
01389

```

```

01398 template<typename T>
01399 Matrix<T> adj(const Matrix<T>& A) {
01400     if (! A.issquare()) throw std::runtime_error("Input matrix is not square");
01401
01402     Matrix<T> B(A.rows(), A.cols());
01403     if (A.rows() == 1) {
01404         B(0) = static_cast<T>(1.0);
01405     } else {
01406         for (unsigned i = 0; i < A.rows(); i++) {
01407             for (unsigned j = 0; j < A.cols(); j++) {
01408                 T sgn = static_cast<T>(1.0) (((i + j) % 2 == 0) ? (1.0) : (-1.0));
01409                 B(j,i) = sgn * det(cofactor(A,i,j));
01410             }
01411         }
01412     }
01413     return B;
01414 }
01415
01430 template<typename T>
01431 Matrix<T> cofactor(const Matrix<T>& A, unsigned p, unsigned q) {
01432     if (! A.issquare()) throw std::runtime_error("Input matrix is not square");
01433     if (! (p < A.rows())) throw std::out_of_range("Row index out of range");
01434     if (! (q < A.cols())) throw std::out_of_range("Column index out of range");
01435     if (A.cols() < 2) throw std::runtime_error("Cofactor calculation requested for matrix with less than
2 rows");
01436
01437     Matrix<T> c(A.rows()-1,A.cols()-1);
01438     unsigned i = 0;
01439     unsigned j = 0;
01440
01441     for (unsigned row = 0; row < A.rows(); row++) {
01442         if (row != p) {
01443             for (unsigned col = 0; col < A.cols(); col++)
01444                 if (col != q) c(i,j++) = A(row,col);
01445             j = 0;
01446             i++;
01447         }
01448     }
01449
01450     return c;
01451 }
01452
01466 template<typename T>
01467 T det_lu(const Matrix<T>& A) {
01468     if (! A.issquare()) throw std::runtime_error("Input matrix is not square");
01469
01470     // LU decomposition with pivoting
01471     auto res = lup(A);
01472
01473     // Determinants of LU
01474     T detLU = static_cast<T>(1);
01475
01476     for (unsigned i = 0; i < res.L.rows(); i++)
01477         detLU *= res.L(i,i) * res.U(i,i);
01478
01479     // Determinant of P
01480     unsigned len = res.P.size();
01481     T detP = static_cast<T>(1);
01482
01483     std::vector<unsigned> p(res.P);
01484     std::vector<unsigned> q;
01485     q.resize(len);
01486
01487     for (unsigned i = 0; i < len; i++)
01488         q[p[i]] = i;
01489
01490     for (unsigned i = 0; i < len; i++) {
01491         unsigned j = p[i];
01492         unsigned k = q[i];
01493         if (j != i) {
01494             p[k] = p[i];
01495             q[j] = q[i];
01496             detP = - detP;
01497         }
01498     }
01499
01500     return detLU * detP;
01501 }
01502
01513 template<typename T>
01514 T det(const Matrix<T>& A) {
01515     if (! A.issquare()) throw std::runtime_error("Input matrix is not square");
01516
01517     if (A.rows() == 1)
01518         return A(0,0);
01519     else if (A.rows() == 2)
01520         return A(0,0)*A(1,1) - A(0,1)*A(1,0);

```

```

01521     else if (A.rows() == 3)
01522         return A(0,0)*(A(1,1)*A(2,2) - A(1,2)*A(2,1)) -
01523             A(0,1)*(A(1,0)*A(2,2) - A(1,2)*A(2,0)) +
01524             A(0,2)*(A(1,0)*A(2,1) - A(1,1)*A(2,0));
01525     else
01526         return det_lu(A);
01527 }
01528
01540 template<typename T>
01541 LU_result<T> lu(const Matrix<T>& A) {
01542     const unsigned M = A.rows();
01543     const unsigned N = A.cols();
01544
01545     LU_result<T> res;
01546     res.L = eye<T>(M);
01547     res.U = Matrix<T>(A);
01548
01549     // aliases
01550     auto& L = res.L;
01551     auto& U = res.U;
01552
01553     if (A.numel() == 0)
01554         return res;
01555
01556     for (unsigned k = 0; k < M-1; k++) {
01557         for (unsigned i = k+1; i < M; i++) {
01558             L(i,k) = U(i,k) / U(k,k);
01559             for (unsigned l = k+1; l < N; l++) {
01560                 U(i,l) -= L(i,k) * U(k,l);
01561             }
01562         }
01563     }
01564
01565     for (unsigned col = 0; col < N; col++)
01566         for (unsigned row = col+1; row < M; row++)
01567             U(row,col) = 0;
01568
01569     return res;
01570 }
01571
01586 template<typename T>
01587 LUP_result<T> lup(const Matrix<T>& A) {
01588     const unsigned M = A.rows();
01589     const unsigned N = A.cols();
01590
01591     // Initialize L, U, and PP
01592     LUP_result<T> res;
01593
01594     if (A.numel() == 0)
01595         return res;
01596
01597     res.L = eye<T>(M);
01598     res.U = Matrix<T>(A);
01599     std::vector<unsigned> PP;
01600
01601     // aliases
01602     auto& L = res.L;
01603     auto& U = res.U;
01604
01605     PP.resize(N);
01606     for (unsigned i = 0; i < N; i++)
01607         PP[i] = i;
01608
01609     for (unsigned k = 0; k < M-1; k++) {
01610         // Find the column with the largest absolute value in the current row
01611         auto max_col_value = std::abs(U(k,k));
01612         unsigned max_col_index = k;
01613         for (unsigned l = k+1; l < N; l++) {
01614             auto val = std::abs(U(k,l));
01615             if (val > max_col_value) {
01616                 max_col_value = val;
01617                 max_col_index = l;
01618             }
01619         }
01620
01621         // Swap columns k and max_col_index in U and update P
01622         if (max_col_index != k) {
01623             U.swap_cols(k, max_col_index); // TODO: This could be reworked to avoid column swap in U during
every iteration by:
01624                                     // 1. using PP[k] for column indexing across iterations
01625                                     // 2. doing just one permutation of U at the end
01626             std::swap(PP[k], PP[max_col_index]);
01627         }
01628
01629         // Update L and U
01630         for (unsigned i = k+1; i < M; i++) {
01631             L(i,k) = U(i,k) / U(k,k);

```



```

01632         for (unsigned l = k+1; l < N; l++) {
01633             U(i,l) -= L(i,k) * U(k,l);
01634         }
01635     }
01636 }
01637
01638 // Set elements in lower triangular part of U to zero
01639 for (unsigned col = 0; col < N; col++)
01640     for (unsigned row = col+1; row < M; row++)
01641         U(row,col) = 0;
01642
01643 // Transpose indices in permutation vector
01644 res.P.resize(N);
01645 for (unsigned i = 0; i < N; i++)
01646     res.P[PP[i]] = i;
01647
01648 return res;
01649 }
01650
01662 template<typename T>
01663 Matrix<T> rref(const Matrix<T>& A, T tol = 0) {
01664     unsigned row = 0;
01665
01666     Matrix<T> B(A);
01667
01668     for (unsigned c = 0; c < B.cols(); c++) {
01669         // stop if already found pivots for all rows
01670         if (row >= B.rows())
01671             break;
01672
01673         // find the pivot row
01674         T max_val = static_cast<T>(0);
01675         unsigned pivot_row = row;
01676
01677         for (unsigned i = row; i < B.rows(); i++) {
01678             T x = static_cast<T>(std::abs(B(i,c)));
01679             if (Util::creal(x) > Util::creal(max_val)) {
01680                 max_val = x;
01681                 pivot_row = i;
01682             }
01683         }
01684
01685         if (Util::creal(max_val) <= Util::creal(tol)) {
01686             // skip column c
01687             for (unsigned i = row; i < B.rows(); i++)
01688                 B(i,c) = static_cast<T>(0);
01689         } else {
01690             // swap current row with the pivot row
01691             if (pivot_row != row)
01692                 for (unsigned j = c; j < B.cols(); j++)
01693                     std::swap(B(row,j), B(pivot_row,j));
01694
01695             // normalize pivot row if not normalized
01696             if (B(row,c) != static_cast<T>(1)) {
01697                 auto factor = static_cast<T>(1) / B(row,c);
01698                 for (unsigned j = c; j < B.cols(); j++)
01699                     B(row,j) *= factor;
01700             }
01701
01702             // eliminate current column
01703             for (unsigned i = 0; i < B.rows(); i++) {
01704                 if (i != row) {
01705                     auto factor = B(i,c);
01706                     for (unsigned j = c; j < B.cols(); j++)
01707                         B(i,j) -= factor * B(row,j);
01708                 }
01709             }
01710
01711             row++;
01712         }
01713     }
01714
01715     return B;
01716 }
01717
01730 template<typename T>
01731 Matrix<T> inv_gauss_jordan(const Matrix<T>& A) {
01732     if (! A.issquare()) throw std::runtime_error("Input matrix is not square");
01733
01734     const unsigned N = A.rows();
01735     Matrix<T> AA(A);
01736     auto IA = eye<T>(N);
01737
01738     bool found_nonzero;
01739     for (unsigned j = 0; j < N; j++) {
01740         found_nonzero = false;
01741         for (unsigned i = j; i < N; i++) {

```

```

01742     if (AA(i,j) != static_cast<T>(0)) {
01743         found_nonzero = true;
01744         for (unsigned k = 0; k < N; k++) {
01745             std::swap(AA(j,k), AA(i,k));
01746             std::swap(IA(j,k), IA(i,k));
01747         }
01748         if (AA(j,j) != static_cast<T>(1)) {
01749             T s = static_cast<T>(1) / AA(j,j);
01750             for (unsigned k = 0; k < N; k++) {
01751                 AA(j,k) *= s;
01752                 IA(j,k) *= s;
01753             }
01754         }
01755         for (unsigned l = 0; l < N; l++) {
01756             if (l != j) {
01757                 T s = AA(l,j);
01758                 for (unsigned k = 0; k < N; k++) {
01759                     AA(l,k) -= s * AA(j,k);
01760                     IA(l,k) -= s * IA(j,k);
01761                 }
01762             }
01763         }
01764     }
01765     break;
01766 }
01767 // if a row full of zeros is found, the input matrix was singular
01768 if (!found_nonzero) throw singular_matrix_exception("Singular matrix in inv_gauss_jordan");
01769 }
01770 return IA;
01771 }
01772
01784 template<typename T>
01785 Matrix<T> inv_tril(const Matrix<T>& A) {
01786     if (!A.issquare()) throw std::runtime_error("Input matrix is not square");
01787
01788     const unsigned N = A.rows();
01789
01790     auto IA = zeros<T>(N);
01791
01792     for (unsigned i = 0; i < N; i++) {
01793         if (A(i,i) == static_cast<T>(0.0)) throw singular_matrix_exception("Division by zero in
01794 inv_tril");
01795         IA(i,i) = static_cast<T>(1.0) / A(i,i);
01796         for (unsigned j = 0; j < i; j++) {
01797             T s = 0.0;
01798             for (unsigned k = j; k < i; k++)
01799                 s += A(i,k) * IA(k,j);
01800             IA(i,j) = -s * IA(i,i);
01801         }
01802     }
01803
01804     return IA;
01805 }
01806
01818 template<typename T>
01819 Matrix<T> inv_triu(const Matrix<T>& A) {
01820     if (!A.issquare()) throw std::runtime_error("Input matrix is not square");
01821
01822     const unsigned N = A.rows();
01823
01824     auto IA = zeros<T>(N);
01825
01826     for (int i = N - 1; i >= 0; i--) {
01827         if (A(i,i) == static_cast<T>(0.0)) throw singular_matrix_exception("Division by zero in
01828 inv_triu");
01829         IA(i,i) = static_cast<T>(1.0) / A(i,i);
01830         for (int j = N - 1; j > i; j--) {
01831             T s = static_cast<T>(0.0);
01832             for (int k = i + 1; k <= j; k++)
01833                 s += A(i,k) * IA(k,j);
01834             IA(i,j) = -s * IA(i,i);
01835         }
01836     }
01837
01838     return IA;
01839 }
01840
01855 template<typename T>
01856 Matrix<T> inv_posdef(const Matrix<T>& A) {
01857     auto L = cholinv(A);
01858     return mult<T,true,false>(L,L);
01859 }
01860
01872 template<typename T>
01873 Matrix<T> inv_square(const Matrix<T>& A) {

```

```

01874     if (! A.issquare()) throw std::runtime_error("Input matrix is not square");
01875
01876     // LU decomposition with pivoting
01877     auto LU = lup(A);
01878     auto IL = inv_tril(LU.L);
01879     auto IU = inv_triu(LU.U);
01880
01881     return permute_rows(IU * IL, LU.P);
01882 }
01883
01897 template<typename T>
01898 Matrix<T> inv(const Matrix<T>& A) {
01899     if (! A.issquare()) throw std::runtime_error("Input matrix is not square");
01900
01901     if (A.numel() == 0) {
01902         return Matrix<T>();
01903     } else if (A.rows() < 4) {
01904         T d = det(A);
01905
01906         if (d == static_cast<T>(0.0)) throw singular_matrix_exception("Singular matrix in inv");
01907
01908         Matrix<T> IA(A.rows(), A.rows());
01909         T invdet = static_cast<T>(1.0) / d;
01910
01911         if (A.rows() == 1) {
01912             IA(0,0) = invdet;
01913         } else if (A.rows() == 2) {
01914             IA(0,0) = A(1,1) * invdet;
01915             IA(0,1) = - A(0,1) * invdet;
01916             IA(1,0) = - A(1,0) * invdet;
01917             IA(1,1) = A(0,0) * invdet;
01918         } else if (A.rows() == 3) {
01919             IA(0,0) = (A(1,1)*A(2,2) - A(2,1)*A(1,2)) * invdet;
01920             IA(0,1) = (A(0,2)*A(2,1) - A(0,1)*A(2,2)) * invdet;
01921             IA(0,2) = (A(0,1)*A(1,2) - A(0,2)*A(1,1)) * invdet;
01922             IA(1,0) = (A(1,2)*A(2,0) - A(1,0)*A(2,2)) * invdet;
01923             IA(1,1) = (A(0,0)*A(2,2) - A(0,2)*A(2,0)) * invdet;
01924             IA(1,2) = (A(1,0)*A(0,2) - A(0,0)*A(1,2)) * invdet;
01925             IA(2,0) = (A(1,0)*A(2,1) - A(2,0)*A(1,1)) * invdet;
01926             IA(2,1) = (A(2,0)*A(0,1) - A(0,0)*A(2,1)) * invdet;
01927             IA(2,2) = (A(0,0)*A(1,1) - A(1,0)*A(0,1)) * invdet;
01928         }
01929
01930         return IA;
01931     } else {
01932         return inv_square(A);
01933     }
01934 }
01935
01947 template<typename T>
01948 Matrix<T> pinv(const Matrix<T>& A) {
01949     if (A.rows() > A.cols()) {
01950         auto AH_A = mult<T,true,false>(A, A);
01951         auto Linv = inv_posdef(AH_A);
01952         return mult<T,false,true>(Linv, A);
01953     } else {
01954         auto AA_H = mult<T,false,true>(A, A);
01955         auto Linv = inv_posdef(AA_H);
01956         return mult<T,true,false>(A, Linv);
01957     }
01958 }
01959
01966 template<typename T>
01967 T trace(const Matrix<T>& A) {
01968     T t = static_cast<T>(0);
01969     for (int i = 0; i < A.rows(); i++)
01970         t += A(i,i);
01971     return t;
01972 }
01973
01984 template<typename T>
01985 double cond(const Matrix<T>& A) {
01986     try {
01987         auto A_inv = inv(A);
01988         return norm_fro(A) * norm_fro(A_inv);
01989     } catch (singular_matrix_exception& e) {
01990         return std::numeric_limits<double>::max();
01991     }
01992 }
01993
02015 template<typename T, bool is_upper = false>
02016 Matrix<T> chol(const Matrix<T>& A) {
02017     if (! A.issquare()) throw std::runtime_error("Input matrix is not square");
02018
02019     const unsigned N = A.rows();
02020
02021     // Calculate lower or upper triangular, depending on template parameter.

```

```

02022 // Calculation is the same - the difference is in transposed row and column indexing.
02023 Matrix<T> C = is_upper ? triu(A) : tril(A);
02024
02025 for (unsigned j = 0; j < N; j++) {
02026     if (C(j,j) == static_cast<T>(0.0)) throw singular_matrix_exception("Singular matrix in chol");
02027
02028     C(j,j) = std::sqrt(C(j,j));
02029
02030     for (unsigned k = j+1; k < N; k++)
02031         if (is_upper)
02032             C(j,k) /= C(j,j);
02033         else
02034             C(k,j) /= C(j,j);
02035
02036     for (unsigned k = j+1; k < N; k++)
02037         for (unsigned i = k; i < N; i++)
02038             if (is_upper)
02039                 C(k,i) -= C(j,i) * Util::cconj(C(j,k));
02040             else
02041                 C(i,k) -= C(i,j) * Util::cconj(C(k,j));
02042 }
02043
02044 return C;
02045 }
02046
02061 template<typename T>
02062 Matrix<T> cholinv(const Matrix<T>& A) {
02063     if (! A.issquare()) throw std::runtime_error("Input matrix is not square");
02064
02065     const unsigned N = A.rows();
02066     Matrix<T> L(A);
02067     auto Linv = eye<T>(N);
02068
02069     for (unsigned j = 0; j < N; j++) {
02070         if (L(j,j) == static_cast<T>(0.0)) throw singular_matrix_exception("Singular matrix in cholinv");
02071
02072         L(j,j) = static_cast<T>(1.0) / std::sqrt(L(j,j));
02073
02074         for (unsigned k = j+1; k < N; k++)
02075             L(k,j) = L(k,j) * L(j,j);
02076
02077         for (unsigned k = j+1; k < N; k++)
02078             for (unsigned i = k; i < N; i++)
02079                 L(i,k) = L(i,k) - L(i,j) * Util::cconj(L(k,j));
02080     }
02081
02082     for (unsigned k = 0; k < N; k++) {
02083         for (unsigned i = k; i < N; i++) {
02084             Linv(i,k) = Linv(i,k) * L(i,i);
02085             for (unsigned j = i+1; j < N; j++)
02086                 Linv(j,k) = Linv(j,k) - L(j,i) * Linv(i,k);
02087         }
02088     }
02089
02090     return Linv;
02091 }
02092
02113 template<typename T>
02114 LDL_result<T> ldl(const Matrix<T>& A) {
02115     if (! A.issquare()) throw std::runtime_error("Input matrix is not square");
02116
02117     const unsigned N = A.rows();
02118
02119     LDL_result<T> res;
02120
02121     // aliases
02122     auto& L = res.L;
02123     auto& d = res.d;
02124
02125     L = eye<T>(N);
02126     d.resize(N);
02127
02128     for (unsigned m = 0; m < N; m++) {
02129         d[m] = A(m,m);
02130
02131         for (unsigned k = 0; k < m; k++)
02132             d[m] -= L(m,k) * Util::cconj(L(m,k)) * d[k];
02133
02134         if (d[m] == static_cast<T>(0.0)) throw singular_matrix_exception("Singular matrix in ldl");
02135
02136         for (unsigned n = m+1; n < N; n++) {
02137             L(n,m) = A(n,m);
02138             for (unsigned k = 0; k < m; k++)
02139                 L(n,m) -= L(n,k) * Util::cconj(L(m,k)) * d[k];
02140             L(n,m) /= d[m];
02141         }
02142     }

```

```

02143
02144     return res;
02145 }
02146
02160 template<typename T>
02161 QR_result<T> qr_red_gs(const Matrix<T>& A) {
02162     const int rows = A.rows();
02163     const int cols = A.cols();
02164
02165     QR_result<T> res;
02166
02167     //aliases
02168     auto& Q = res.Q;
02169     auto& R = res.R;
02170
02171     Q = zeros<T>(rows, cols);
02172     R = zeros<T>(cols, cols);
02173
02174     for (int c = 0; c < cols; c++) {
02175         Matrix<T> v = A.get_submatrix(0, rows-1, c, c);
02176         for (int r = 0; r < c; r++) {
02177             for (int k = 0; k < rows; k++)
02178                 R(r,c) = R(r,c) + Util::cconj(Q(k,r)) * A(k,c);
02179             for (int k = 0; k < rows; k++)
02180                 v(k) = v(k) - R(r,c) * Q(k,r);
02181         }
02182
02183         R(c,c) = static_cast<T>(norm_fro(v));
02184
02185         if (R(c,c) == static_cast<T>(0.0)) throw singular_matrix_exception("Division by 0 in QR GS");
02186
02187         for (int k = 0; k < rows; k++)
02188             Q(k,c) = v(k) / R(c,c);
02189     }
02190
02191     return res;
02192 }
02193
02201 template<typename T>
02202 Matrix<T> householder_reflection(const Matrix<T>& a) {
02203     if (a.cols() != 1) throw std::runtime_error("Input not a column vector");
02204
02205     static const T ISQRT2 = static_cast<T>(0.707106781186547);
02206
02207     Matrix<T> v(a);
02208     v(0) += Util::csign(v(0)) * norm_fro(v);
02209     auto vn = norm_fro(v) * ISQRT2;
02210     for (unsigned i = 0; i < v.numel(); i++)
02211         v(i) /= vn;
02212     return v;
02213 }
02214
02229 template<typename T>
02230 QR_result<T> qr_householder(const Matrix<T>& A, bool calculate_Q = true) {
02231     const unsigned rows = A.rows();
02232     const unsigned cols = A.cols();
02233
02234     QR_result<T> res;
02235
02236     //aliases
02237     auto& Q = res.Q;
02238     auto& R = res.R;
02239
02240     R = Matrix<T>(A);
02241
02242     if (calculate_Q)
02243         Q = eye<T>(rows);
02244
02245     const unsigned N = (rows > cols) ? cols : rows;
02246
02247     for (unsigned j = 0; j < N; j++) {
02248         auto v = householder_reflection(R.get_submatrix(j, rows-1, j, j));
02249
02250         auto R1 = R.get_submatrix(j, rows-1, j, cols-1);
02251         auto WR = v * mult<T,true,false>(v, R1);
02252         for (unsigned c = j; c < cols; c++)
02253             for (unsigned r = j; r < rows; r++)
02254                 R(r,c) -= WR(r-j,c-j);
02255
02256         if (calculate_Q) {
02257             auto Q1 = Q.get_submatrix(0, rows-1, j, rows-1);
02258             auto WQ = mult<T,false,true>(Q1 * v, v);
02259             for (unsigned c = j; c < rows; c++)
02260                 for (unsigned r = 0; r < rows; r++)
02261                     Q(r,c) -= WQ(r,c-j);
02262         }
02263     }

```

```

02264
02265     for (unsigned col = 0; col < R.cols(); col++)
02266         for (unsigned row = col+1; row < R.rows(); row++)
02267             R(row,col) = 0;
02268
02269     return res;
02270 }
02271
02285 template<typename T>
02286 inline QR_result<T> qr(const Matrix<T>& A, bool calculate_Q = true) {
02287     return qr_householder(A, calculate_Q);
02288 }
02289
02302 template<typename T>
02303 Hessenberg_result<T> hessenberg(const Matrix<T>& A, bool calculate_Q = true) {
02304     if (! A.issquare()) throw std::runtime_error("Input matrix is not square");
02305
02306     Hessenberg_result<T> res;
02307
02308     // aliases
02309     auto& H = res.H;
02310     auto& Q = res.Q;
02311
02312     const unsigned N = A.rows();
02313     H = Matrix<T>(A);
02314
02315     if (calculate_Q)
02316         Q = eye<T>(N);
02317
02318     for (unsigned k = 1; k < N-1; k++) {
02319         auto v = householder_reflection(H.get_submatrix(k, N-1, k-1, k-1));
02320
02321         auto H1 = H.get_submatrix(k, N-1, 0, N-1);
02322         auto W1 = v * mult<T,true,false>(v, H1);
02323         for (unsigned c = 0; c < N; c++)
02324             for (unsigned r = k; r < N; r++)
02325                 H(r,c) -= W1(r-k,c);
02326
02327         auto H2 = H.get_submatrix(0, N-1, k, N-1);
02328         auto W2 = mult<T,false,true>(H2 * v, v);
02329         for (unsigned c = k; c < N; c++)
02330             for (unsigned r = 0; r < N; r++)
02331                 H(r,c) -= W2(r,c-k);
02332
02333         if (calculate_Q) {
02334             auto Q1 = Q.get_submatrix(0, N-1, k, N-1);
02335             auto W3 = mult<T,false,true>(Q1 * v, v);
02336             for (unsigned c = k; c < N; c++)
02337                 for (unsigned r = 0; r < N; r++)
02338                     Q(r,c) -= W3(r,c-k);
02339         }
02340     }
02341
02342     for (unsigned row = 2; row < N; row++)
02343         for (unsigned col = 0; col < row-2; col++)
02344             H(row,col) = static_cast<T>(0);
02345
02346     return res;
02347 }
02348
02358 template<typename T>
02359 std::complex<T> wilkinson_shift(const Matrix<std::complex<T>& H, T tol = 1e-10) {
02360     if (! H.issquare()) throw std::runtime_error("Input matrix is not square");
02361
02362     const unsigned n = H.rows();
02363     std::complex<T> mu;
02364
02365     if (std::abs(H(n-1,n-2)) < tol) {
02366         mu = H(n-2,n-2);
02367     } else {
02368         auto trA = H(n-2,n-2) + H(n-1,n-1);
02369         auto detA = H(n-2,n-2) * H(n-1,n-1) - H(n-2, n-1) * H(n-1, n-2);
02370         mu = (trA + std::sqrt(trA*trA - 4.0*detA)) / 2.0;
02371     }
02372
02373     return mu;
02374 }
02375
02387 template<typename T>
02388 Eigenvalues_result<T> eigenvalues(const Matrix<std::complex<T>& A, T tol = 1e-12, unsigned max_iter =
02389     100) {
02390     if (! A.issquare()) throw std::runtime_error("Input matrix is not square");
02391
02392     const unsigned N = A.rows();
02393     Matrix<std::complex<T>> H;
02394     bool success = false;

```

```

02395     QR_result<std::complex<T>> QR;
02396
02397     // aliases
02398     auto& Q = QR.Q;
02399     auto& R = QR.R;
02400
02401     // Transfer A to Hessenberg form to improve convergence (skip calculation of Q)
02402     H = hessenberg(A, false).H;
02403
02404     for (unsigned iter = 0; iter < max_iter; iter++) {
02405         auto mu = wilkinson_shift(H, tol);
02406
02407         // subtract mu from diagonal
02408         for (unsigned n = 0; n < N; n++)
02409             H(n,n) -= mu;
02410
02411         // QR factorization with shifted H
02412         QR = qr(H);
02413         H = R * Q;
02414
02415         // add back mu to diagonal
02416         for (unsigned n = 0; n < N; n++)
02417             H(n,n) += mu;
02418
02419         // Check for convergence
02420         if (std::abs(H(N-2,N-1)) <= tol) {
02421             success = true;
02422             break;
02423         }
02424     }
02425
02426     Eigenvalues_result<T> res;
02427     res.eig = diag(H);
02428     res.err = std::abs(H(N-2,N-1));
02429     res.converged = success;
02430
02431     return res;
02432 }
02433
02443 template<typename T>
02444 Eigenvalues_result<T> eigenvalues(const Matrix<T>& A, T tol = 1e-12, unsigned max_iter = 100) {
02445     auto A_cplx = make_complex(A);
02446     return eigenvalues(A_cplx, tol, max_iter);
02447 }
02448
02465 template<typename T>
02466 Matrix<T> solve_triu(const Matrix<T>& U, const Matrix<T>& B) {
02467     if (! U.issquare()) throw std::runtime_error("Input matrix is not square");
02468     if (U.rows() != B.rows()) throw std::runtime_error("Unmatching matrix dimensions for solve");
02469
02470     const unsigned N = U.rows();
02471     const unsigned M = B.cols();
02472
02473     if (U.numel() == 0)
02474         return Matrix<T>();
02475
02476     Matrix<T> X(B);
02477
02478     for (unsigned m = 0; m < M; m++) {
02479         // backwards substitution for each column of B
02480         for (int n = N-1; n >= 0; n--) {
02481             for (unsigned j = n + 1; j < N; j++)
02482                 X(n,m) -= U(n,j) * X(j,m);
02483
02484             if (U(n,n) == static_cast<T>(0.0)) throw singular_matrix_exception("Singular matrix in
solve_triu");
02485
02486             X(n,m) /= U(n,n);
02487         }
02488     }
02489
02490     return X;
02491 }
02492
02509 template<typename T>
02510 Matrix<T> solve_tril(const Matrix<T>& L, const Matrix<T>& B) {
02511     if (! L.issquare()) throw std::runtime_error("Input matrix is not square");
02512     if (L.rows() != B.rows()) throw std::runtime_error("Unmatching matrix dimensions for solve");
02513
02514     const unsigned N = L.rows();
02515     const unsigned M = B.cols();
02516
02517     if (L.numel() == 0)
02518         return Matrix<T>();
02519
02520     Matrix<T> X(B);
02521

```

```

02522     for (unsigned m = 0; m < M; m++) {
02523         // forwards substitution for each column of B
02524         for (unsigned n = 0; n < N; n++) {
02525             for (unsigned j = 0; j < n; j++)
02526                 X(n,m) -= L(n,j) * X(j,m);
02527
02528             if (L(n,n) == static_cast<T>(0.0)) throw singular_matrix_exception("Singular matrix in
solve_tril");
02529
02530             X(n,m) /= L(n,n);
02531         }
02532     }
02533     return X;
02534 }
02535
02536 template<typename T>
02537 Matrix<T> solve_square(const Matrix<T>& A, const Matrix<T>& B) {
02538     if (! A.issquare()) throw std::runtime_error("Input matrix is not square");
02539     if (A.rows() != B.rows()) throw std::runtime_error("Unmatching matrix dimensions for solve");
02540
02541     if (A.numel() == 0)
02542         return Matrix<T>();
02543
02544     Matrix<T> L;
02545     Matrix<T> U;
02546     std::vector<unsigned> P;
02547
02548     // LU decomposition with pivoting
02549     auto lup_res = lup(A);
02550
02551     auto y = solve_tril(lup_res.L, B);
02552     auto x = solve_triu(lup_res.U, y);
02553
02554     return permute_rows(x, lup_res.P);
02555 }
02556
02557 template<typename T>
02558 Matrix<T> solve_posdef(const Matrix<T>& A, const Matrix<T>& B) {
02559     if (! A.issquare()) throw std::runtime_error("Input matrix is not square");
02560     if (A.rows() != B.rows()) throw std::runtime_error("Unmatching matrix dimensions for solve");
02561
02562     if (A.numel() == 0)
02563         return Matrix<T>();
02564
02565     // LU decomposition with pivoting
02566     auto L = chol(A);
02567
02568     auto Y = solve_tril(L, B);
02569     return solve_triu(L.transpose(), Y);
02570 }
02571
02572 template<typename T>
02573 class Matrix {
02574 public:
02575     Matrix();
02576
02577     Matrix(unsigned size);
02578
02579     Matrix(unsigned nrow, unsigned ncol);
02580
02581     Matrix(T x, unsigned nrow, unsigned ncol);
02582
02583     Matrix(const T* array, unsigned nrow, unsigned ncol);
02584
02585     Matrix(const std::vector<T>& vec, unsigned nrow, unsigned ncol);
02586
02587     Matrix(std::initializer_list<T> init_list, unsigned nrow, unsigned ncol);
02588
02589     Matrix(const Matrix &);
02590
02591     virtual ~Matrix();
02592
02593     Matrix<T> get_submatrix(unsigned row_first, unsigned row_last, unsigned col_first, unsigned
col_last) const;
02594
02595     void set_submatrix(const Matrix<T>& smtx, unsigned row_first, unsigned col_first);
02596
02597     void clear();
02598
02599     void reshape(unsigned rows, unsigned cols);
02600
02601     void resize(unsigned rows, unsigned cols);
02602
02603     bool exists(unsigned row, unsigned col) const;
02604
02605     T* ptr(unsigned row, unsigned col);

```



```

02733
02741     T* ptr();
02742
02746     void fill(T value);
02747
02754     void fill_col(T value, unsigned col);
02755
02762     void fill_row(T value, unsigned row);
02763
02768     bool isempty() const;
02769
02773     bool issquare() const;
02774
02779     bool isequal(const Matrix<T>&) const;
02780
02786     bool isequal(const Matrix<T>&, T) const;
02787
02792     unsigned numel() const;
02793
02798     unsigned rows() const;
02799
02804     unsigned cols() const;
02805
02811     std::pair<unsigned,unsigned> shape() const;
02812
02817     Matrix<T> transpose() const;
02818
02824     Matrix<T> ctranspose() const;
02825
02833     Matrix<T>& add(const Matrix<T>&);
02834
02843     Matrix<T>& subtract(const Matrix<T>&);
02844
02854     Matrix<T>& mult_hadamard(const Matrix<T>&);
02855
02862     Matrix<T>& add(T);
02863
02870     Matrix<T>& subtract(T);
02871
02878     Matrix<T>& mult(T);
02879
02886     Matrix<T>& div(T);
02887
02892     Matrix<T>& operator=(const Matrix<T>&);
02893
02899     Matrix<T>& operator=(T);
02900
02906     explicit operator std::vector<T>() const;
02907     std::vector<T> to_vector() const;
02908
02915     T& operator()(unsigned nel);
02916     T operator()(unsigned nel) const;
02917     T& at(unsigned nel);
02918     T at(unsigned nel) const;
02919
02927     T& operator()(unsigned row, unsigned col);
02928     T operator()(unsigned row, unsigned col) const;
02929     T& at(unsigned row, unsigned col);
02930     T at(unsigned row, unsigned col) const;
02931
02939     void add_row_to_another(unsigned to, unsigned from);
02940
02948     void add_col_to_another(unsigned to, unsigned from);
02949
02957     void mult_row_by_another(unsigned to, unsigned from);
02958
02966     void mult_col_by_another(unsigned to, unsigned from);
02967
02974     void swap_rows(unsigned i, unsigned j);
02975
02982     void swap_cols(unsigned i, unsigned j);
02983
02990     std::vector<T> col_to_vector(unsigned col) const;
02991
02998     std::vector<T> row_to_vector(unsigned row) const;
02999
03008     void col_from_vector(const std::vector<T>&, unsigned col);
03009
03018     void row_from_vector(const std::vector<T>&, unsigned row);
03019
03020 private:
03021     unsigned nrows;
03022     unsigned ncols;
03023     std::vector<T> data;
03024 };
03025
03026 /*

```

```

03027  * Implementation of Matrix class methods
03028  */
03029
03030 template<typename T>
03031 Matrix<T>::Matrix() : nrows(0), ncols(0), data() { }
03032
03033 template<typename T>
03034 Matrix<T>::Matrix(unsigned size) : Matrix(size, size) { }
03035
03036 template<typename T>
03037 Matrix<T>::Matrix(unsigned rows, unsigned cols) : nrows(rows), ncols(cols) {
03038     data.resize(numel());
03039 }
03040
03041 template<typename T>
03042 Matrix<T>::Matrix(T x, unsigned rows, unsigned cols) : Matrix(rows, cols) {
03043     fill(x);
03044 }
03045
03046 template<typename T>
03047 Matrix<T>::Matrix(const T* array, unsigned rows, unsigned cols) : Matrix(rows, cols) {
03048     data.assign(array, array + numel());
03049 }
03050
03051 template<typename T>
03052 Matrix<T>::Matrix(const std::vector<T>& vec, unsigned rows, unsigned cols) : Matrix(rows, cols) {
03053     if (vec.size() != numel()) throw std::runtime_error("Size of initialization vector not consistent
    with matrix dimensions");
03054     data.assign(vec.begin(), vec.end());
03055 }
03056
03057 template<typename T>
03058 Matrix<T>::Matrix(std::initializer_list<T> init_list, unsigned rows, unsigned cols) : Matrix(rows,
    cols) {
03059     if (init_list.size() != numel()) throw std::runtime_error("Size of initialization list not
    consistent with matrix dimensions");
03060     auto it = init_list.begin();
03061     for (unsigned row = 0; row < this->nrows; row++)
03062         for (unsigned col = 0; col < this->ncols; col++)
03063             this->at(row, col) = *(it++);
03064 }
03065
03066 template<typename T>
03067 Matrix<T>::Matrix(const Matrix & other) : Matrix(other.nrows, other.ncols) {
03068     this->data.assign(other.data.begin(), other.data.end());
03069 }
03070
03071 template<typename T>
03072 Matrix<T>& Matrix<T>::operator=(const Matrix<T>& other) {
03073     this->nrows = other.nrows;
03074     this->ncols = other.ncols;
03075     this->data.assign(other.data.begin(), other.data.end());
03076     return *this;
03077 }
03078
03079 template<typename T>
03080 Matrix<T>& Matrix<T>::operator=(T s) {
03081     fill(s);
03082     return *this;
03083 }
03084
03085 template<typename T>
03086 inline Matrix<T>::operator std::vector<T>() const {
03087     return data;
03088 }
03089
03090 template<typename T>
03091 inline void Matrix<T>::clear() {
03092     this->nrows = 0;
03093     this->ncols = 0;
03094     data.resize(0);
03095 }
03096
03097 template<typename T>
03098 void Matrix<T>::reshape(unsigned rows, unsigned cols) {
03099     if (this->numel() != rows * cols) throw std::runtime_error("Illegal attempt to change number of
    elements via reshape");
03100     this->nrows = rows;
03101     this->ncols = cols;
03102 }
03103
03104 template<typename T>
03105 void Matrix<T>::resize(unsigned rows, unsigned cols) {

```

```

03110     this->nrows = rows;
03111     this->ncols = cols;
03112     data.resize(nrows*ncols);
03113 }
03114
03115 template<typename T>
03116 Matrix<T> Matrix<T>::get_submatrix(unsigned row_base, unsigned row_lim, unsigned col_base, unsigned
col_lim) const {
03117     if (row_base > row_lim) throw std::out_of_range("Row index of submatrix out of range");
03118     if (col_base > col_lim) throw std::out_of_range("Column index of submatrix out of range");
03119     if (row_lim >= this->nrows()) throw std::out_of_range("Row index of submatrix out of range");
03120     if (col_lim >= this->ncols()) throw std::out_of_range("Column index of submatrix out of range");
03121
03122     unsigned num_rows = row_lim - row_base + 1;
03123     unsigned num_cols = col_lim - col_base + 1;
03124     Matrix<T> S(num_rows, num_cols);
03125     for (unsigned i = 0; i < num_rows; i++) {
03126         for (unsigned j = 0; j < num_cols; j++) {
03127             S(i,j) = at(row_base + i, col_base + j);
03128         }
03129     }
03130     return S;
03131 }
03132
03133 template<typename T>
03134 void Matrix<T>::set_submatrix(const Matrix<T>& S, unsigned row_base, unsigned col_base) {
03135     if (this->isempty()) throw std::runtime_error("Invalid attempt to set submatrix in empty matrix");
03136
03137     const unsigned row_lim = row_base + S.rows() - 1;
03138     const unsigned col_lim = col_base + S.cols() - 1;
03139
03140     if (row_base > row_lim) throw std::out_of_range("Row index of submatrix out of range");
03141     if (col_base > col_lim) throw std::out_of_range("Column index of submatrix out of range");
03142     if (row_lim >= this->nrows()) throw std::out_of_range("Row index of submatrix out of range");
03143     if (col_lim >= this->ncols()) throw std::out_of_range("Column index of submatrix out of range");
03144
03145     unsigned num_rows = row_lim - row_base + 1;
03146     unsigned num_cols = col_lim - col_base + 1;
03147     for (unsigned i = 0; i < num_rows; i++)
03148         for (unsigned j = 0; j < num_cols; j++)
03149             at(row_base + i, col_base + j) = S(i,j);
03150 }
03151
03152 template<typename T>
03153 inline T & Matrix<T>::operator()(unsigned nel) {
03154     return at(nel);
03155 }
03156
03157 template<typename T>
03158 inline T & Matrix<T>::operator()(unsigned row, unsigned col) {
03159     return at(row, col);
03160 }
03161
03162 template<typename T>
03163 inline T Matrix<T>::operator()(unsigned nel) const {
03164     return at(nel);
03165 }
03166
03167 template<typename T>
03168 inline T Matrix<T>::operator()(unsigned row, unsigned col) const {
03169     return at(row, col);
03170 }
03171
03172 template<typename T>
03173 inline T & Matrix<T>::at(unsigned nel) {
03174     #ifdef MATRIX_STRICT_BOUNDS_CHECK
03175         if (!(nel < numel())) throw std::out_of_range("Element index out of range");
03176     #endif
03177     return data[nel];
03178 }
03179
03180 template<typename T>
03181 inline T & Matrix<T>::at(unsigned row, unsigned col) {
03182     #ifdef MATRIX_STRICT_BOUNDS_CHECK
03183         if (!(row < rows() && col < cols())) throw std::out_of_range("Element index out of range");
03184     #endif
03185     return data[nrows * col + row];
03186 }
03187
03188 template<typename T>
03189 inline T Matrix<T>::at(unsigned nel) const {
03190     #ifdef MATRIX_STRICT_BOUNDS_CHECK
03191         if (!(nel < numel())) throw std::out_of_range("Element index out of range");
03192     #endif
03193 }
03194
03195

```

```

03196     return data[nel];
03197 }
03198
03199 template<typename T>
03200 inline T Matrix<T>::at(unsigned row, unsigned col) const {
03201     #ifndef MATRIX_STRICT_BOUNDS_CHECK
03202         if (!(row < rows())) throw std::out_of_range("Row index out of range");
03203         if (!(col < cols())) throw std::out_of_range("Column index out of range");
03204     #endif
03205
03206     return data[nrows * col + row];
03207 }
03208
03209 template<typename T>
03210 inline void Matrix<T>::fill(T value) {
03211     for (unsigned i = 0; i < numel(); i++)
03212         data[i] = value;
03213 }
03214
03215 template<typename T>
03216 inline void Matrix<T>::fill_col(T value, unsigned col) {
03217     if (!(col < cols())) throw std::out_of_range("Column index out of range");
03218
03219     for (unsigned i = col * nrows; i < (col+1) * nrows; i++)
03220         data[i] = value;
03221 }
03222
03223 template<typename T>
03224 inline void Matrix<T>::fill_row(T value, unsigned row) {
03225     if (!(row < rows())) throw std::out_of_range("Row index out of range");
03226
03227     for (unsigned i = 0; i < ncols; i++)
03228         data[row + i * nrows] = value;
03229 }
03230
03231 template<typename T>
03232 inline bool Matrix<T>::exists(unsigned row, unsigned col) const {
03233     return (row < nrows && col < ncols);
03234 }
03235
03236 template<typename T>
03237 inline T* Matrix<T>::ptr(unsigned row, unsigned col) {
03238     if (!(row < rows())) throw std::out_of_range("Row index out of range");
03239     if (!(col < cols())) throw std::out_of_range("Column index out of range");
03240
03241     return data.data() + nrows * col + row;
03242 }
03243
03244 template<typename T>
03245 inline T* Matrix<T>::ptr() {
03246     return data.data();
03247 }
03248
03249 template<typename T>
03250 inline bool Matrix<T>::isempty() const {
03251     return (nrows == 0) || (ncols == 0);
03252 }
03253
03254 template<typename T>
03255 inline bool Matrix<T>::issquare() const {
03256     return (nrows == ncols) && !isempty();
03257 }
03258
03259 template<typename T>
03260 bool Matrix<T>::isequal(const Matrix<T>& A) const {
03261     bool ret = true;
03262     if (nrows != A.rows() || ncols != A.cols()) {
03263         ret = false;
03264     } else {
03265         for (unsigned i = 0; i < numel(); i++) {
03266             if (at(i) != A(i)) {
03267                 ret = false;
03268                 break;
03269             }
03270         }
03271     }
03272     return ret;
03273 }
03274
03275 template<typename T>
03276 bool Matrix<T>::isequal(const Matrix<T>& A, T tol) const {
03277     bool ret = true;
03278     if (rows() != A.rows() || cols() != A.cols()) {
03279         ret = false;
03280     } else {
03281         auto abs_tol = std::abs(tol); // workaround for complex
03282         for (unsigned i = 0; i < A.numel(); i++) {

```

```

03283         if (abs_tol < std::abs(at(i) - A(i))) {
03284             ret = false;
03285             break;
03286         }
03287     }
03288 }
03289 return ret;
03290 }
03291
03292 template<typename T>
03293 inline unsigned Matrix<T>::numel() const {
03294     return nrows * ncols;
03295 }
03296
03297 template<typename T>
03298 inline unsigned Matrix<T>::rows() const {
03299     return nrows;
03300 }
03301
03302 template<typename T>
03303 inline unsigned Matrix<T>::cols() const {
03304     return ncols;
03305 }
03306
03307 template<typename T>
03308 inline std::pair<unsigned,unsigned> Matrix<T>::shape() const {
03309     return std::pair<unsigned,unsigned>(nrows,ncols);
03310 }
03311
03312 template<typename T>
03313 inline Matrix<T> Matrix<T>::transpose() const {
03314     Matrix<T> res(ncols, nrows);
03315     for (unsigned c = 0; c < ncols; c++)
03316         for (unsigned r = 0; r < nrows; r++)
03317             res(c,r) = at(r,c);
03318     return res;
03319 }
03320
03321 template<typename T>
03322 inline Matrix<T> Matrix<T>::ctranspose() const {
03323     Matrix<T> res(ncols, nrows);
03324     for (unsigned c = 0; c < ncols; c++)
03325         for (unsigned r = 0; r < nrows; r++)
03326             res(c,r) = Util::cconj(at(r,c));
03327     return res;
03328 }
03329
03330 template<typename T>
03331 Matrix<T>& Matrix<T>::add(const Matrix<T>& m) {
03332     if (! (m.rows() == rows() && m.cols() == cols())) throw std::runtime_error("Unmatching matrix
dimensions for iadd");
03333     for (unsigned i = 0; i < numel(); i++)
03334         data[i] += m(i);
03335     return *this;
03336 }
03337
03338 template<typename T>
03339 Matrix<T>& Matrix<T>::subtract(const Matrix<T>& m) {
03340     if (! (m.rows() == rows() && m.cols() == cols())) throw std::runtime_error("Unmatching matrix
dimensions for isubtract");
03341     for (unsigned i = 0; i < numel(); i++)
03342         data[i] -= m(i);
03343     return *this;
03344 }
03345
03346 template<typename T>
03347 Matrix<T>& Matrix<T>::mult_hadamard(const Matrix<T>& m) {
03348     if (! (m.rows() == rows() && m.cols() == cols())) throw std::runtime_error("Unmatching matrix
dimensions for ihprod");
03349     for (unsigned i = 0; i < numel(); i++)
03350         data[i] *= m(i);
03351     return *this;
03352 }
03353
03354 template<typename T>
03355 Matrix<T>& Matrix<T>::add(T s) {
03356     for (auto& x : data)
03357         x += s;
03358     return *this;
03359 }
03360
03361 template<typename T>
03362 Matrix<T>& Matrix<T>::subtract(T s) {
03363     for (auto& x : data)

```

```

03367     x -= s;
03368     return *this;
03369 }
03370
03371 template<typename T>
03372 Matrix<T>& Matrix<T>::mult(T s) {
03373     for (auto& x : data)
03374         x *= s;
03375     return *this;
03376 }
03377
03378 template<typename T>
03379 Matrix<T>& Matrix<T>::div(T s) {
03380     for (auto& x : data)
03381         x /= s;
03382     return *this;
03383 }
03384
03385 template<typename T>
03386 void Matrix<T>::add_row_to_another(unsigned to, unsigned from) {
03387     if (!(to < rows() && from < rows())) throw std::out_of_range("Row index out of range");
03388     for (unsigned k = 0; k < cols(); k++)
03389         at(to, k) += at(from, k);
03390 }
03391
03392 template<typename T>
03393 void Matrix<T>::add_col_to_another(unsigned to, unsigned from) {
03394     if (!(to < cols() && from < cols())) throw std::out_of_range("Column index out of range");
03395     for (unsigned k = 0; k < rows(); k++)
03396         at(k, to) += at(k, from);
03397 }
03398
03399 template<typename T>
03400 void Matrix<T>::mult_row_by_another(unsigned to, unsigned from) {
03401     if (!(to < rows() && from < rows())) throw std::out_of_range("Row index out of range");
03402     for (unsigned k = 0; k < cols(); k++)
03403         at(to, k) *= at(from, k);
03404 }
03405
03406 template<typename T>
03407 void Matrix<T>::mult_col_by_another(unsigned to, unsigned from) {
03408     if (!(to < cols() && from < cols())) throw std::out_of_range("Column index out of range");
03409     for (unsigned k = 0; k < rows(); k++)
03410         at(k, to) *= at(k, from);
03411 }
03412
03413 template<typename T>
03414 void Matrix<T>::swap_rows(unsigned i, unsigned j) {
03415     if (!(i < rows() && j < rows())) throw std::out_of_range("Row index out of range");
03416     for (unsigned k = 0; k < cols(); k++)
03417         std::swap(at(i, k), at(j, k));
03418 }
03419
03420 template<typename T>
03421 void Matrix<T>::swap_cols(unsigned i, unsigned j) {
03422     if (!(i < cols() && j < cols())) throw std::out_of_range("Column index out of range");
03423     for (unsigned k = 0; k < rows(); k++)
03424         std::swap(at(k, i), at(k, j));
03425 }
03426
03427 template<typename T>
03428 inline std::vector<T> Matrix<T>::to_vector() const {
03429     return data;
03430 }
03431
03432 template<typename T>
03433 inline std::vector<T> Matrix<T>::col_to_vector(unsigned col) const {
03434     std::vector<T> vec(rows());
03435     for (unsigned i = 0; i < rows(); i++)
03436         vec[i] = at(i, col);
03437     return vec;
03438 }
03439
03440 template<typename T>
03441 inline std::vector<T> Matrix<T>::row_to_vector(unsigned row) const {
03442     std::vector<T> vec(cols());
03443     for (unsigned i = 0; i < cols(); i++)
03444         vec[i] = at(row, i);
03445     return vec;
03446 }
03447
03448
03449
03450
03451
03452
03453

```

```
03454 template<typename T>
03455 inline void Matrix<T>::col_from_vector(const std::vector<T>& vec, unsigned col) {
03456     if (vec.size() != rows()) throw std::runtime_error("Vector size is not equal to number of rows");
03457     if (col >= cols()) throw std::out_of_range("Column index out of range");
03458
03459     for (unsigned i = 0; i < rows(); i++)
03460         data[col*rows() + i] = vec[i];
03461 }
03462
03463 template<typename T>
03464 inline void Matrix<T>::row_from_vector(const std::vector<T>& vec, unsigned row) {
03465     if (vec.size() != rows()) throw std::runtime_error("Vector size is not equal to number of columns");
03466     if (row >= rows()) throw std::out_of_range("Row index out of range");
03467
03468     for (unsigned i = 0; i < cols(); i++)
03469         data[row + i*rows()] = vec[i];
03470 }
03471
03472 template<typename T>
03473 Matrix<T>::~Matrix() { }
03474
03475 } // namespace Matrix_hpp
03476
03477 #endif // __MATRIX_HPP__
```

