# Matrix HPP

Generated by Doxygen 1.9.8

# 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](LICENSE) for details.

# Chapter 2

# Namespace Index

## 2.1 Namespace List

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

# Chapter 3

# Hierarchical Index

## 3.1 Class Hierarchy

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

# Chapter 4

# Class Index

## 4.1 Class List

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

# Chapter 5

# File Index

## 5.1 File List

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

# 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

Colelction 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::ldl(), 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_vecto 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**

| | |
|---|---|
| *std::runtime_error* | 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**

| | |
|---|---|
| *std::runtime_error* | 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**

| | |
|---|---|
| *std::runtime_error* | 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**

| | |
|---|---|
| *std::out_of_range* | 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**

| | |
|---|---|
| *std::out_of_range* | 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**

| | |
|---|---|
| *std::runtime_error* | when std::vector size is not equal to number of rows |
| *std::out_of_range* | 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**

| | |
|---|---|
| *std::out_of_range* | 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 >::isequal(), Mtx::Matrix< T >::isequal(), Mtx::istril(), Mtx::istriu(), Mtx::kron(), Mtx::lu(), Mtx::lup(), Mtx::make_complex(), Mtx::make_complex(), Mtx::mult(), Mtx::mult(), Mtx::mult(), Mtx::mult(), Mtx::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::cconj(), 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**

| | |
|---|---|
| *std::out_of_range* | 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**

| | |
|---|---|
| *std::out_of_range* | 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**

| | |
|---|---|
| *std::out_of_range* | 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*=().

#### 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**

| | |
|---|---|
| *std::out_of_range* | 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**

| | |
|---|---|
| *std::runtime_error* | 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**

| | |
|---|---|
| *std::out_of_range* | 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 >::isequal(), Mtx::Matrix< T >::isequal(), Mtx::lu(), Mtx::lup(), Mtx::make_complex(), Mtx::make_complex(), Mtx::Matrix< T >::Matrix(), Mtx::Matrix< T >::Matrix(), Mtx::Matrix< T >::Matrix(), Mtx::Matrix< T >::Matrix(), Mtx::mult(), Mtx::Matrix< T >::mult_hadamard(), Mtx::norm_fro(), Mtx::real(), Mtx::Matrix< T >::reshape(), Mtx::solve_posdef(), Mtx::solve_square(), Mtx::solve_tril(), Mtx::solve_triu(), Mtx::Matrix< T >::subtract(), and Mtx::subtract().

#### 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**

| | |
|---|---|
| *std::out_of_range* | 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**

| | |
|---|---|
| *std::out_of_range* | 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**

| | |
|---|---|
| *std::out_of_range* | 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**

| | |
|---|---|
| *std::runtime_error* | 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**

| | |
|---|---|
| *std::runtime_error* | when std::vector size is not equal to number of columns |
| *std::out_of_range* | 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**

| | |
|---|---|
| *std::out_of_range* | 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**

| | |
|---|---|
| *std::out_of_range* | when row or column index is out of range of matrix dimensions |
| *std::runtime_error* | 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**

| | |
|---|---|
| *std::runtime_error* | 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 >
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**

| | |
|---|---|
| *std::out_of_range* | 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**

| | |
|---|---|
| *std::out_of_range* | 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**

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

**Parameters**

| | |
|---|---|
| *A* | left-side matrix of size *N* x *M* (after transposition) |
| *B* | right-side matrix of size *N* x *M* (after transposition) |

**Returns**

output matrix of size *N* x *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**

| | |
|---|---|
| *std::runtime_error* | 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**

| | |
|---|---|
| *is_upper* | 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**

| | |
|---|---|
| *std::runtime_error* | when the input matrix is not square |
| *singular_matrix_exception* | 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**

| | |
|---|---|
| *std::runtime_error* | when the input matrix is not square |
| *singular_matrix_exception* | 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**

| | |
|---|---|
| *A* | matrix |
| *row_shift* | row shift factor |
| *col_shift* | 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**

| | |
|---|---|
| *v* | 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* x *n* by taking the elements from *array* as the first column.

**Parameters**

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

**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)

**Parameters**

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

**Exceptions**

| | |
|---|---|
| *std::runtime_error* | when the input matrix is not square |
| *std::out_of_range* | when row index *p* or column index *q* are out of range |
| *std::runtime_error* | 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**

| | |
|---|---|
| *std::runtime_error* | 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**

| | |
|---|---|
| *std::runtime_error* | 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:

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

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**

| | |
|---|---|
| *std::runtime_error* | 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**

| | |
|---|---|
| *std::runtime_error* | when the input matrix is not square |

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

Referenced by Mtx::det(), and Mtx::det_lu().

#### 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**

| | |
|---|---|
| *A* | square matrix |

**Returns**

vector of diagonal elements

**Exceptions**

| | |
|---|---|
| *std::runtime_error* | 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**

| | |
|---|---|
| *v* | 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* x *n*, whose diagonal elements are set to the elements stored in the *array*.

**Parameters**

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

**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**

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

**Returns**

structure containing the result and status of eigenvalue calculation

**Exceptions**

| | |
|---|---|
| *std::runtime_error* | 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**

| | |
|---|---|
| *A* | input real matrix to be decomposed |
| *tol* | numerical precision tolerance for stop condition |
| *max_iter* | 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**

| | |
|---|---|
| *n* | 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**

| | |
|---|---|
| *A* | input matrix to be modified |
| *func* | function to be applied element-wise to *A*. It inputs one variable of template type T 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**

| | |
|---|---|
| *A* | input matrix |
| *func* | function to be applied element-wise to A. It inputs one variable of template type T 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* = True.

**Exceptions**

| | |
|---|---|
| *std::runtime_error* | 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**

| | |
|---|---|
| *a* | column vector of size *N* x *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**

| | |
|---|---|
| *std::runtime_error* | when the input matrix is not square |
| *singular_matrix_exception* | 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 exists, e.g., because the input matrix was singular, an empty matrix is returned.

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

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

**Exceptions**

| | |
|---|---|
| *std::runtime_error* | when the input matrix is not square |
| *singular_matrix_exception* | 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**

| | |
|---|---|
| *std::runtime_error* | when the input matrix is not square |
| *singular_matrix_exception* | 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**

| | |
|---|---|
| *std::runtime_error* | when the input matrix is not square |
| *singular_matrix_exception* | 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**

| | |
|---|---|
| *std::runtime_error* | when the input matrix is not square |
| *singular_matrix_exception* | 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**

| | |
|---|---|
| *std::runtime_error* | when the input matrix is not square |
| *singular_matrix_exception* | 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**

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

**Returns**

structure encapsulating calculated *L* and *D*

**Exceptions**

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

References Mtx::Util::cconj(), Mtx::Matrix< T >::issquare(), Mtx::ldl(), and Mtx::Matrix< T >::rows().

Referenced by Mtx::ldl().

#### 8.1.1.39 lu()

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

LU decomposition.

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

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

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

**Parameters**

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

**Returns**

structure containing calculated *L* and *U* matrices

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

Referenced by Mtx::lu().

#### 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**

| | |
|---|---|
| *Re* | real part matrix |
| *Im* | imaginary part matrix |

**Returns**

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

**Exceptions**

| | |
|---|---|
| *std::runtime_error* | when *Re* and *Im* 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**

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

**Parameters**

| | |
|---|---|
| *A* | left-side matrix of size *N* x *M* (after transposition) |
| *B* | right-side matrix of size *M* x *K* (after transposition) |

**Returns**

output matrix of size *N* x *K*

References Mtx::Util::cconj(), Mtx::Matrix< T >::cols(), Mtx::mult(), and Mtx::Matrix< T >::rows().

Referenced by Mtx::mult(), Mtx::mult(), Mtx::mult(), Mtx::mult(), Mtx::operator*(), Mtx::operator*(), Mtx::operator*(), Mtx::operator*(), Mtx::operator*(), and Mtx::operator*=().

#### 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**

| | |
|---|---|
| *transpose_matrix* | if set to true, the matrix will be transposed during operation |

**Parameters**

| | |
|---|---|
| *A* | input matrix of size $N$ x $M$ |
| *v* | std::vector of size $M$ |

**Returns**

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

References Mtx::Util::cconj(), 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**

| | |
|---|---|
| *transpose_matrix* | if set to true, the matrix will be transposed during operation |

**Parameters**

| | |
|---|---|
| *v* | std::vector of size *N* |
| *A* | input matrix of size *N* x *M* |

**Returns**

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

References Mtx::Util::cconj(), 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**

| | |
|---|---|
| *transpose←_A* | if set to true, matrix $A$ shall be transposed during operation |
| *transpose←_B* | if set to true, matrix $B$ shall be transposed during operation |
| *transpose←_C* | if set to true, matrix $C$ shall be transposed during operation |

**Parameters**

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

**Returns**

output matrix of size *N* x *K*

References Mtx::Util::cconj(), 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**

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

**Parameters**

| | |
|---|---|
| *A* | left-side matrix of size *N* x *M* (after transposition) |
| *B* | right-side matrix of size *N* x *M* (after transposition) |

**Returns**

output matrix of size *N* x *M*

References Mtx::Util::cconj(), 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::cconj(), 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* x *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**

| | |
|---|---|
| *nrows* | number of rows (the first dimension) |
| *ncols* | 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 endline 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()* x *perm.size()*.

**Parameters**

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

**Returns**

output matrix created by column permutation of *A*

**Exceptions**

| | |
|---|---|
| *std::runtime_error* | when permutation vector is empty |
| *std::out_of_range* | 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_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**

| | |
|---|---|
| *A* | input matrix |
| *perm* | permutation vector with row indices |

**Returns**

output matrix created by row permutation of *A*

**Exceptions**

| | |
|---|---|
| *std::runtime_error* | when permutation vector is empty |
| *std::out_of_range* | 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 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**

| | |
|---|---|
| *A* | input matrix |
| *perm_rows* | permutation vector with row indices |
| *perm_cols* | permutation vector with column indices |

**Returns**

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

**Exceptions**

| | |
|---|---|
| *std::runtime_error* | when any of permutation vectors is empty |
| *std::out_of_range* | 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* x *n* and *R* of size *n* x *m*. *Q* is calculated only when *calculate_Q* = 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* x *m* |
| *calculate_Q* | indicates if *Q* to be calculated |

**Returns**

structure encapsulating calculated *Q* of size *n* x *n* and *R* of size *n* x *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* x *m* |

**Returns**

structure encapsulating calculated *Q* of size *n* x *m*, and *R* of size *m* x *m*.

**Exceptions**

| | |
|---|---|
| *singular_matrix_exception* | when division by 0 is encountered during computation |

References Mtx::Util::cconj(), Mtx::Matrix< T >::cols(), Mtx::Matrix< T >::get_submatrix(), Mtx::norm_fro(), Mtx::qr_red_gs(), and Mtx::Matrix< T >::rows().

Referenced by Mtx::qr_red_gs().

#### 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* x *N*. Must be square and positive definite. |
| *B* | right hand side matrix of size *N* x *M*. |

**Returns**

solution matrix of size *N* x *M*.

**Exceptions**

| | |
|---|---|
| *std::runtime_error* | when the input matrix is not square |
| *std::runtime_error* | when number of rows is not equal between input matrices |
| *singular_matrix_exception* | 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**

| | |
|---|---|
| *A* | left side matrix of size *N* x *N*. Must be square. |
| *B* | right hand side matrix of size *N* x *M*. |

**Returns**

solution matrix of size *N* x *M*.

**Exceptions**

| | |
|---|---|
| *std::runtime_error* | when the input matrix is not square |
| *std::runtime_error* | when number of rows is not equal between input matrices |
| *singular_matrix_exception* | 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**

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

**Returns**

X solution matrix of size *N* x *M*.

**Exceptions**

| | |
|---|---|
| *std::runtime_error* | when the input matrix is not square |
| *std::runtime_error* | when number of rows is not equal between input matrices |
| *singular_matrix_exception* | 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* x *N*. Must be square and upper triangular |
| *B* | right hand side matrix of size *N* x *M*. |

**Returns**

solution matrix of size *N* x *M*.

**Exceptions**

| | |
|---|---|
| *std::runtime_error* | when the input matrix is not square |
| *std::runtime_error* | when number of rows is not equal between input matrices |
| *singular_matrix_exception* | 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**

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

**Parameters**

| | |
|---|---|
| *A* | left-side matrix of size *N* x *M* (after transposition) |
| *B* | right-side matrix of size *N* x *M* (after transposition) |

**Returns**

output matrix of size *N* x *M*

References Mtx::Util::cconj(), 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**

| | |
|---|---|
| *A* | inpur matrix to be converted |
| *col_separator* | character used to separate elements within the same row. The default character is the space |
| *row_separator* | 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:

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

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 *mu* for complex eigenvalues of input matrix. Wilkinson's shift is calculated as eigenvalue of the bottom 2 x 2 principal minor closest to the corner entry of the matrix.

Input must be a square matrix in Hessenberg form.

**Exceptions**

| | |
|---|---|
| *std::runtime_error* | 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* x *n* and fill it with all elements set to 0.

**Parameters**

| | |
|---|---|
| *n* | 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**

| | |
|---|---|
| *nrows* | number of rows (the first dimension) |
| *ncols* | 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.

```


/*  MIT License
 *
 *  Copyright (c) 2024 gc1905
 *
 *  Permission is hereby granted, free of charge, to any person obtaining a copy
 *  of this software and associated documentation files (the "Software"), to deal
 *  in the Software without restriction, including without limitation the rights
 *  to use, copy, modify, merge, publish, distribute, sublicense, and/or sell
 *  copies of the Software, and to permit persons to whom the Software is
 *  furnished to do so, subject to the following conditions:
 *
 *  The above copyright notice and this permission notice shall be included in all
 *  copies or substantial portions of the Software.
 *
 *  THE SOFTWARE IS PROVIDED "AS IS", WITHOUT WARRANTY OF ANY KIND, EXPRESS OR
 *  IMPLIED, INCLUDING BUT NOT LIMITED TO THE WARRANTIES OF MERCHANTABILITY,
 *  FITNESS FOR A PARTICULAR PURPOSE AND NONINFRINGEMENT. IN NO EVENT SHALL THE
 *  AUTHORS OR COPYRIGHT HOLDERS BE LIABLE FOR ANY CLAIM, DAMAGES OR OTHER
 *  LIABILITY, WHETHER IN AN ACTION OF CONTRACT, TORT OR OTHERWISE, ARISING FROM,
 *  OUT OF OR IN CONNECTION WITH THE SOFTWARE OR THE USE OR OTHER DEALINGS IN THE
 *  SOFTWARE.
 */

#ifndef __MATRIX_HPP__
#define __MATRIX_HPP__

```

```
#include <ostream>
#include <complex>
#include <vector>
#include <initializer_list>
#include <limits>
#include <functional>
#include <algorithm>
#include <utility>

namespace Mtx {

template<typename T> class Matrix;

namespace Util {
  template<class T> struct is_complex : std::false_type {};
  template<class T> struct is_complex<std::complex<T» : std::true_type {};

  template<typename T, typename std::enable_if<!is_complex<T>::value,int>::type = 0>
  inline T cconj(T x) {
    return x;
  }

  template<typename T, typename std::enable_if<is_complex<T>::value,int>::type = 0>
  inline T cconj(T x) {
    return std::conj(x);
  }

  template<typename T, typename std::enable_if<!is_complex<T>::value,int>::type = 0>
  inline T csign(T x) {
    return (x > static_cast<T>(0)) ? static_cast<T>(1) : static_cast<T>(-1);
  }

  template<typename T, typename std::enable_if<is_complex<T>::value,int>::type = 0>
  inline T csign(T x) {
    auto x_arg = std::arg(x);
    T y(0, x_arg);
    return std::exp(y);
  }

  template<typename T, typename std::enable_if<!is_complex<T>::value,int>::type = 0>
  inline T creal(std::complex<T> x) {
    return std::real(x);
  }

  template<typename T, typename std::enable_if<!is_complex<T>::value,int>::type = 0>
  inline T creal(T x) {
    return x;
  }
} // namespace Util

class singular_matrix_exception : public std::domain_error {
  public:
    singular_matrix_exception(const std::string& message) : std::domain_error(message) {}
};

template<typename T>
struct LU_result {
  Matrix<T> L;

  Matrix<T> U;
};

template<typename T>
struct LUP_result {
  Matrix<T> L;

  Matrix<T> U;

  std::vector<unsigned> P;
};

template<typename T>
struct QR_result {
  Matrix<T> Q;

  Matrix<T> R;
};

template<typename T>
struct Hessenberg_result {
  Matrix<T> H;

  Matrix<T> Q;
};

template<typename T>
struct LDL_result {
```

```
  Matrix<T> L;

  std::vector<T> d;
};

template<typename T>
struct Eigenvalues_result {
  std::vector<std::complex<T» eig;

  bool converged;

  T err;
};


template<typename T>
inline Matrix<T> zeros(unsigned nrows, unsigned ncols) {
  return Matrix<T>(static_cast<T>(0), nrows, ncols);
}

template<typename T>
inline Matrix<T> zeros(unsigned n) {
  return zeros<T>(n,n);
}

template<typename T>
inline Matrix<T> ones(unsigned nrows, unsigned ncols) {
  return Matrix<T>(static_cast<T>(1), nrows, ncols);
}

template<typename T>
inline Matrix<T> ones(unsigned n) {
  return ones<T>(n,n);
}

template<typename T>
Matrix<T> eye(unsigned n) {
  Matrix<T> A(static_cast<T>(0), n, n);
  for (unsigned i = 0; i < n; i++)
    A(i,i) = static_cast<T>(1);
  return A;
}

template<typename T>
Matrix<T> diag(const T* array, size_t n) {
  Matrix<T> A(static_cast<T>(0), n, n);
  for (unsigned i = 0; i < n; i++) {
    A(i,i) = array[i];
  }
  return A;
}

template<typename T>
inline Matrix<T> diag(const std::vector<T>& v) {
  return diag(v.data(), v.size());
}

template<typename T>
std::vector<T> diag(const Matrix<T>& A) {
  if (! A.issquare()) throw std::runtime_error("Input matrix is not square");

  std::vector<T> v;
  v.resize(A.rows());

  for (unsigned i = 0; i < A.rows(); i++)
    v[i] = A(i,i);
  return v;
}

template<typename T>
Matrix<T> circulant(const T* array, unsigned n) {
  Matrix<T> A(n, n);
  for (unsigned j = 0; j < n; j++)
    for (unsigned i = 0; i < n; i++)
      A((i+j) % n,j) = array[i];
  return A;
}

template<typename T>
Matrix<std::complex<T>> make_complex(const Matrix<T>& Re, const Matrix<T>& Im) {
  if (Re.rows() != Im.rows() || Re.cols() != Im.cols()) throw std::runtime_error("Size of input
matrices does not match");

  Matrix<std::complex<T> > C(Re.rows(),Re.cols());
  for (unsigned n = 0; n < Re.numel(); n++) {
    C(n).real(Re(n));
    C(n).imag(Im(n));
```

```
  }

  return C;
}

template<typename T>
Matrix<std::complex<T>> make_complex(const Matrix<T>& Re) {
  Matrix<std::complex<T>> C(Re.rows(),Re.cols());

  for (unsigned n = 0; n < Re.numel(); n++) {
    C(n).real(Re(n));
    C(n).imag(static_cast<T>(0));
  }

  return C;
}

template<typename T>
Matrix<T> real(const Matrix<std::complex<T>»& C) {
  Matrix<T> Re(C.rows(),C.cols());

  for (unsigned n = 0; n < C.numel(); n++)
    Re(n) = C(n).real();

  return Re;
}

template<typename T>
Matrix<T> imag(const Matrix<std::complex<T>»& C) {
  Matrix<T> Re(C.rows(),C.cols());

  for (unsigned n = 0; n < C.numel(); n++)
    Re(n) = C(n).imag();

  return Re;
}

template<typename T>
inline Matrix<T> circulant(const std::vector<T>& v) {
  return circulant(v.data(), v.size());
}

template<typename T>
inline Matrix<T> transpose(const Matrix<T>& A) {
  return A.transpose();
}

template<typename T>
inline Matrix<T> ctranspose(const Matrix<T>& A) {
  return A.ctranspose();
}

template<typename T>
Matrix<T> circshift(const Matrix<T>& A, int row_shift, int col_shift) {
  Matrix<T> B(A.rows(), A.cols());
  for (int i = 0; i < A.rows(); i++) {
    int ii = (i + row_shift) % A.rows();
    for (int j = 0; j < A.cols(); j++) {
      int jj = (j + col_shift) % A.cols();
      B(ii,jj) = A(i,j);
    }
  }
  return B;
}

template<typename T>
Matrix<T> repmat(const Matrix<T>& A, unsigned m, unsigned n) {
  Matrix<T> B(m * A.rows(), n * A.cols());

  for (unsigned cb = 0; cb < n; cb++)
    for (unsigned rb = 0; rb < m; rb++)
      for (unsigned c = 0; c < A.cols(); c++)
        for (unsigned r = 0; r < A.rows(); r++)
          B(rb*A.rows() + r, cb*A.cols() + c) = A(r, c);

  return B;
}

template<typename T>
Matrix<T> concatenate_horizontal(const Matrix<T>& A, const Matrix<T>& B) {
  if (A.rows() != B.rows()) throw std::runtime_error("Unmatching number of rows for horizontal
concatenation");

  Matrix<T> C(A.rows(), A.cols() + B.cols());

  for (unsigned c = 0; c < A.cols(); c++)
    for (unsigned r = 0; r < A.rows(); r++)
```

```
      C(r,c) = A(r,c);

  for (unsigned c = 0; c < B.cols(); c++)
    for (unsigned r = 0; r < B.rows(); r++)
      C(r,c+A.cols()) = B(r,c);

  return C;
}

template<typename T>
Matrix<T> concatenate_vertical(const Matrix<T>& A, const Matrix<T>& B) {
  if (A.cols() != B.cols()) throw std::runtime_error("Unmatching number of columns for vertical concatenation");

  Matrix<T> C(A.rows() + B.rows(), A.cols());

  for (unsigned c = 0; c < A.cols(); c++)
    for (unsigned r = 0; r < A.rows(); r++)
      C(r,c) = A(r,c);

  for (unsigned c = 0; c < B.cols(); c++)
    for (unsigned r = 0; r < B.rows(); r++)
      C(r+A.rows(),c) = B(r,c);

  return C;
}

template<typename T>
double norm_fro(const Matrix<T>& A) {
  double sum = 0;

  for (unsigned i = 0; i < A.numel(); i++)
    sum += Util::creal(A(i) * Util::cconj(A(i)));

  return std::sqrt(sum);
}

template<typename T>
double norm_p1(const Matrix<T>& A) {
  double max_sum = 0.0;

  for (unsigned c = 0; c < A.cols(); c++) {
    double sum = 0.0;

    for (unsigned r = 0; r < A.rows(); r++)
      sum += std::abs(A(r,c));

    if (sum > max_sum)
      max_sum = sum;
  }

  return max_sum;
}

template<typename T>
double norm_inf(const Matrix<T>& A) {
  double max_sum = 0.0;

  for (unsigned r = 0; r < A.rows(); r++) {
    double sum = 0.0;

    for (unsigned c = 0; c < A.cols(); c++)
      sum += std::abs(A(r,c));

    if (sum > max_sum)
      max_sum = sum;
  }

  return max_sum;
}

template<typename T>
Matrix<T> tril(const Matrix<T>& A) {
  Matrix<T> B(A);

  for (unsigned row = 0; row < B.rows(); row++)
    for (unsigned col = row+1; col < B.cols(); col++)
      B(row,col) = static_cast<T>(0);

  return B;
}

template<typename T>
Matrix<T> triu(const Matrix<T>& A) {
  Matrix<T> B(A);

  for (unsigned col = 0; col < B.cols(); col++)
```

```
    for (unsigned row = col+1; row < B.rows(); row++)
      B(row,col) = static_cast<T>(0);

  return B;
}

template<typename T>
bool istril(const Matrix<T>& A) {
  for (unsigned row = 0; row < A.rows(); row++)
    for (unsigned col = row+1; col < A.cols(); col++)
      if (A(row,col) != static_cast<T>(0)) return false;
  return true;
}

template<typename T>
bool istriu(const Matrix<T>& A) {
  for (unsigned col = 0; col < A.cols(); col++)
    for (unsigned row = col+1; row < A.rows(); row++)
      if (A(row,col) != static_cast<T>(0)) return false;
  return true;
}

template<typename T>
bool ishess(const Matrix<T>& A) {
  if (!A.issquare())
    return false;
  for (unsigned row = 2; row < A.rows(); row++)
    for (unsigned col = 0; col < row-2; col++)
      if (A(row,col) != static_cast<T>(0)) return false;
  return true;
}

template<typename T>
inline void foreach_elem(Matrix<T>& A, std::function<T(T)> func) {
  for (unsigned i = 0; i < A.numel(); i++)
    A(i) = func(A(i));
}

template<typename T>
inline Matrix<T> foreach_elem_copy(const Matrix<T>& A, std::function<T(T)> func) {
  Matrix<T> B(A);
  foreach_elem(B, func);
  return B;
}

template<typename T>
Matrix<T> permute_rows(const Matrix<T>& A, const std::vector<unsigned> perm) {
  if (perm.empty()) throw std::runtime_error("Permutation vector is empty");

#ifdef MATRIX_STRICT_BOUNDS_CHECK
  for (unsigned p = 0; p < perm.size(); p++)
    if (!(perm[p] < A.rows())) throw std::out_of_range("Index in permutation vector out of range");
#endif

  Matrix<T> B(perm.size(), A.cols());

  for (unsigned p = 0; p < perm.size(); p++)
    for (unsigned c = 0; c < A.cols(); c++)
      B(p,c) = A(perm[p],c);

  return B;
}

template<typename T>
Matrix<T> permute_cols(const Matrix<T>& A, const std::vector<unsigned> perm) {
  if (perm.empty()) throw std::runtime_error("Permutation vector is empty");

#ifdef MATRIX_STRICT_BOUNDS_CHECK
  for (unsigned p = 0; p < perm.size(); p++)
    if (!(perm[p] < A.cols())) throw std::out_of_range("Index in permutation vector out of range");
#endif

  Matrix<T> B(A.rows(), perm.size());

  for (unsigned p = 0; p < perm.size(); p++)
    for (unsigned r = 0; r < A.rows(); r++)
      B(r,p) = A(r,perm[p]);

  return B;
}

template<typename T>
Matrix<T> permute_rows_and_cols(const Matrix<T>& A, const std::vector<unsigned> perm_rows, const
std::vector<unsigned> perm_cols) {
  if (perm_rows.empty()) throw std::runtime_error("Row permutation vector is empty");
  if (perm_cols.empty()) throw std::runtime_error("Column permutation vector is empty");

```

```
#ifdef MATRIX_STRICT_BOUNDS_CHECK
  for (unsigned pc = 0; pc < perm_cols.size(); pc++)
    if (!(perm_cols[pc] < A.cols())) throw std::out_of_range("Column index in permutation vector out of range");

  for (unsigned pr = 0; pr < perm_rows.size(); pr++)
    if (!(perm_rows[pr] < A.rows())) throw std::out_of_range("Row index in permutation vector out of range");
#endif

  Matrix<T> B(perm_rows.size(), perm_cols.size());

  for (unsigned pc = 0; pc < perm_cols.size(); pc++)
    for (unsigned pr = 0; pr < perm_rows.size(); pr++)
      B(pr,pc) = A(perm_rows[pr],perm_cols[pc]);

  return B;
}

template<typename T, bool transpose_first = false, bool transpose_second = false>
Matrix<T> mult(const Matrix<T>& A, const Matrix<T>& B) {
  // Adjust dimensions based on transpositions
  unsigned rows_A = transpose_first ? A.cols() : A.rows();
  unsigned cols_A = transpose_first ? A.rows() : A.cols();
  unsigned rows_B = transpose_second ? B.cols() : B.rows();
  unsigned cols_B = transpose_second ? B.rows() : B.cols();

  if (cols_A != rows_B) throw std::runtime_error("Unmatching matrix dimensions for mult");

  Matrix<T> C(static_cast<T>(0), rows_A, cols_B);

  for (unsigned i = 0; i < rows_A; i++)
    for (unsigned j = 0; j < cols_B; j++)
      for (unsigned k = 0; k < cols_A; k++)
      C(i,j) += (transpose_first  ? Util::cconj(A(k,i)) : A(i,k)) *
                (transpose_second ? Util::cconj(B(j,k)) : B(k,j));

  return C;
}

template<typename T, bool transpose_A = false, bool transpose_B = false, bool transpose_C = false>
Matrix<T> mult_and_add(const Matrix<T>& A, const Matrix<T>& B, const Matrix<T>& C) {
  // Adjust dimensions based on transpositions
  unsigned rows_A = transpose_A ? A.cols() : A.rows();
  unsigned cols_A = transpose_A ? A.rows() : A.cols();
  unsigned rows_B = transpose_B ? B.cols() : B.rows();
  unsigned cols_B = transpose_B ? B.rows() : B.cols();
  unsigned rows_C = transpose_C ? C.cols() : C.rows();
  unsigned cols_C = transpose_C ? C.rows() : C.cols();

  if ((cols_A != rows_B) || (rows_A != rows_C) || (cols_B != cols_C))
    throw std::runtime_error("Unmatching matrix dimensions for mult_and_add");

  Matrix<T> D(rows_C, cols_C);

  for (unsigned i = 0; i < rows_A; i++) {
    for (unsigned j = 0; j < cols_B; j++) {
      D(i,j) = transpose_C ? Util::cconj(C(j,i)) : C(i,j);
      for (unsigned k = 0; k < cols_A; k++) {
        D(i,j) += (transpose_A ? Util::cconj(A(k,i)) : A(i,k)) *
                  (transpose_B ? Util::cconj(B(j,k)) : B(k,j));
      }
    }
  }

  return D;
}

template<typename T, bool transpose_first = false, bool transpose_second = false>
Matrix<T> mult_hadamard(const Matrix<T>& A, const Matrix<T>& B) {
  // Adjust dimensions based on transpositions
  unsigned rows_A = transpose_first ? A.cols() : A.rows();
  unsigned cols_A = transpose_first ? A.rows() : A.cols();
  unsigned rows_B = transpose_second ? B.cols() : B.rows();
  unsigned cols_B = transpose_second ? B.rows() : B.cols();

  if ((rows_A != rows_B) || (cols_A != cols_B)) throw std::runtime_error("Unmatching matrix dimensions for mult_hadamard");

  Matrix<T> C(static_cast<T>(0), rows_A, cols_A);

  for (unsigned i = 0; i < rows_A; i++)
    for (unsigned j = 0; j < cols_A; j++)
      C(i,j) += (transpose_first  ? Util::cconj(A(j,i)) : A(i,j)) *
                (transpose_second ? Util::cconj(B(j,i)) : B(i,j));

  return C;
```

```
}

template<typename T, bool transpose_first = false, bool transpose_second = false>
Matrix<T> add(const Matrix<T>& A, const Matrix<T>& B) {
  // Adjust dimensions based on transpositions
  unsigned rows_A = transpose_first ? A.cols() : A.rows();
  unsigned cols_A = transpose_first ? A.rows() : A.cols();
  unsigned rows_B = transpose_second ? B.cols() : B.rows();
  unsigned cols_B = transpose_second ? B.rows() : B.cols();

  if ((rows_A != rows_B) || (cols_A != cols_B)) throw std::runtime_error("Unmatching matrix dimensions for add");

  Matrix<T> C(static_cast<T>(0), rows_A, cols_A);

  for (unsigned i = 0; i < rows_A; i++)
    for (unsigned j = 0; j < cols_A; j++)
      C(i,j) += (transpose_first  ? Util::cconj(A(j,i)) : A(i,j)) +
                (transpose_second ? Util::cconj(B(j,i)) : B(i,j));

  return C;
}

template<typename T, bool transpose_first = false, bool transpose_second = false>
Matrix<T> subtract(const Matrix<T>& A, const Matrix<T>& B) {
  // Adjust dimensions based on transpositions
  unsigned rows_A = transpose_first ? A.cols() : A.rows();
  unsigned cols_A = transpose_first ? A.rows() : A.cols();
  unsigned rows_B = transpose_second ? B.cols() : B.rows();
  unsigned cols_B = transpose_second ? B.rows() : B.cols();

  if ((rows_A != rows_B) || (cols_A != cols_B)) throw std::runtime_error("Unmatching matrix dimensions for subtract");

  Matrix<T> C(static_cast<T>(0), rows_A, cols_A);

  for (unsigned i = 0; i < rows_A; i++)
    for (unsigned j = 0; j < cols_A; j++)
      C(i,j) += (transpose_first  ? Util::cconj(A(j,i)) : A(i,j)) -
                (transpose_second ? Util::cconj(B(j,i)) : B(i,j));

  return C;
}

template<typename T, bool transpose_matrix = false>
std::vector<T> mult(const Matrix<T>& A, const std::vector<T>& v) {
  // Adjust dimensions based on transpositions
  unsigned rows_A = transpose_matrix ? A.cols() : A.rows();
  unsigned cols_A = transpose_matrix ? A.rows() : A.cols();

  if (cols_A != v.size()) throw std::runtime_error("Unmatching matrix dimensions for mult");

  std::vector<T> u(rows_A, static_cast<T>(0));
  for (unsigned r = 0; r < rows_A; r++)
    for (unsigned c = 0; c < cols_A; c++)
      u[r] += v[c] * (transpose_matrix ? Util::cconj(A(c,r)) : A(r,c));

  return u;
}

template<typename T, bool transpose_matrix = false>
std::vector<T> mult(const std::vector<T>& v, const Matrix<T>& A) {
  // Adjust dimensions based on transpositions
  unsigned rows_A = transpose_matrix ? A.cols() : A.rows();
  unsigned cols_A = transpose_matrix ? A.rows() : A.cols();

  if (rows_A != v.size()) throw std::runtime_error("Unmatching matrix dimensions for mult");

  std::vector<T> u(cols_A, static_cast<T>(0));
  for (unsigned c = 0; c < cols_A; c++)
    for (unsigned r = 0; r < rows_A; r++)
      u[c] += v[r] * (transpose_matrix ? Util::cconj(A(c,r)) : A(r,c));

  return u;
}

template<typename T>
Matrix<T> add(const Matrix<T>& A, T s) {
  Matrix<T> B(A.rows(), A.cols());
  for (unsigned i = 0; i < A.numel(); i++)
    B(i) = A(i) + s;
  return B;
}

template<typename T>
Matrix<T> subtract(const Matrix<T>& A, T s) {
  Matrix<T> B(A.rows(), A.cols());
```

```
  for (unsigned i = 0; i < A.numel(); i++)
    B(i) = A(i) - s;
  return B;
}

template<typename T>
Matrix<T> mult(const Matrix<T>& A, T s) {
  Matrix<T> B(A.rows(), A.cols());
  for (unsigned i = 0; i < A.numel(); i++)
    B(i) = A(i) * s;
  return B;
}

template<typename T>
Matrix<T> div(const Matrix<T>& A, T s) {
  Matrix<T> B(A.rows(), A.cols());
  for (unsigned i = 0; i < A.numel(); i++)
    B(i) = A(i) / s;
  return B;
}

template<typename T>
std::string to_string(const Matrix<T>& A, char col_separator = ' ', char row_separator = '\n') {
  std::stringstream ss;
  for (unsigned row = 0; row < A.rows(); row ++) {
    for (unsigned col = 0; col < A.cols(); col ++)
      ss « A(row,col) « col_separator;
    if (row < static_cast<unsigned>(A.rows()-1)) ss « row_separator;
  }
  return ss.str();
}

template<typename T>
std::ostream& operator«(std::ostream& os, const Matrix<T>& A) {
  for (unsigned row = 0; row < A.rows(); row ++) {
    for (unsigned col = 0; col < A.cols(); col ++)
      os « A(row,col) « " ";
    if (row < static_cast<unsigned>(A.rows()-1)) os « std::endl;
  }
  return os;
}

template<typename T>
inline Matrix<T> operator+(const Matrix<T>& A, const Matrix<T>& B) {
  return add(A,B);
}

template<typename T>
inline Matrix<T> operator-(const Matrix<T>& A, const Matrix<T>& B) {
  return subtract(A,B);
}

template<typename T>
inline Matrix<T> operator^(const Matrix<T>& A, const Matrix<T>& B) {
  return mult_hadamard(A,B);
}

template<typename T>
inline Matrix<T> operator*(const Matrix<T>& A, const Matrix<T>& B) {
  return mult(A,B);
}

template<typename T>
inline std::vector<T> operator*(const Matrix<T>& A, const std::vector<T>& v) {
  return mult(A,v);
}

template<typename T>
inline std::vector<T> operator*(const std::vector<T>& v, const Matrix<T>& A) {
  return mult(v,A);
}

template<typename T>
inline Matrix<T> operator+(const Matrix<T>& A, T s) {
  return add(A,s);
}

template<typename T>
inline Matrix<T> operator-(const Matrix<T>& A, T s) {
  return subtract(A,s);
}

template<typename T>
inline Matrix<T> operator*(const Matrix<T>& A, T s) {
  return mult(A,s);
}

```

```
template<typename T>
inline Matrix<T> operator/(const Matrix<T>& A, T s) {
  return div(A,s);
}

template<typename T>
inline Matrix<T> operator+(T s, const Matrix<T>& A) {
  return add(A,s);
}

template<typename T>
inline Matrix<T> operator*(T s, const Matrix<T>& A) {
  return mult(A,s);
}

template<typename T>
inline Matrix<T>& operator+=(Matrix<T>& A, const Matrix<T>& B) {
  return A.add(B);
}

template<typename T>
inline Matrix<T>& operator-=(Matrix<T>& A, const Matrix<T>& B) {
  return A.subtract(B);
}

template<typename T>
inline Matrix<T>& operator*=(Matrix<T>& A, const Matrix<T>& B) {
  A = mult(A,B);
  return A;
}

template<typename T>
inline Matrix<T>& operator^=(Matrix<T>& A, const Matrix<T>& B) {
  return A.mult_hadamard(B);
}

template<typename T>
inline Matrix<T>& operator+=(Matrix<T>& A, T s) {
  return A.add(s);
}

template<typename T>
inline Matrix<T>& operator-=(Matrix<T>& A, T s) {
  return A.subtract(s);
}

template<typename T>
inline Matrix<T>& operator*=(Matrix<T>& A, T s) {
  return A.mult(s);
}

template<typename T>
inline Matrix<T>& operator/=(Matrix<T>& A, T s) {
  return A.div(s);
}

template<typename T>
inline bool operator==(const Matrix<T>& A, const Matrix<T>& b) {
  return A.isequal(b);
}

template<typename T>
inline bool operator!=(const Matrix<T>& A, const Matrix<T>& b) {
  return !(A.isequal(b));
}

template<typename T>
Matrix<T> kron(const Matrix<T>& A, const Matrix<T>& B) {
    const unsigned rows_A = A.rows();
    const unsigned cols_A = A.cols();
    const unsigned rows_B = B.rows();
    const unsigned cols_B = B.cols();

    unsigned rows_C = rows_A * rows_B;
    unsigned cols_C = cols_A * cols_B;

    Matrix<T> C(rows_C, cols_C);

    for (unsigned i = 0; i < rows_A; i++)
      for (unsigned j = 0; j < cols_A; j++)
        for (unsigned k = 0; k < rows_B; k++)
          for (unsigned l = 0; l < cols_B; l++)
            C(i*rows_B + k, j*cols_B + l) = A(i,j) * B(k,l);

    return C;
}

```

```
template<typename T>
Matrix<T> adj(const Matrix<T>& A) {
  if (! A.issquare()) throw std::runtime_error("Input matrix is not square");

  Matrix<T> B(A.rows(), A.cols());
  if (A.rows() == 1) {
    B(0) = static_cast<T>(1.0);
  } else {
    for (unsigned i = 0; i < A.rows(); i++) {
      for (unsigned j = 0; j < A.cols(); j++) {
        T sgn = static_cast<T>(1.0)(((i + j) % 2 == 0) ? (1.0) : (-1.0));
        B(j,i) = sgn * det(cofactor(A,i,j));
      }
    }
  }
  return B;
}

template<typename T>
Matrix<T> cofactor(const Matrix<T>& A, unsigned p, unsigned q) {
  if (! A.issquare()) throw std::runtime_error("Input matrix is not square");
  if (!(p < A.rows())) throw std::out_of_range("Row index out of range");
  if (!(q < A.cols())) throw std::out_of_range("Column index out of range");
  if (A.cols() < 2) throw std::runtime_error("Cofactor calculation requested for matrix with less than
 2 rows");

  Matrix<T> c(A.rows()-1,A.cols()-1);
  unsigned i = 0;
  unsigned j = 0;

  for (unsigned row = 0; row < A.rows(); row++) {
    if (row != p) {
      for (unsigned col = 0; col < A.cols(); col++)
        if (col != q) c(i,j++) = A(row,col);
      j = 0;
      i++;
    }
  }

  return c;
}

template<typename T>
T det_lu(const Matrix<T>& A) {
  if (! A.issquare()) throw std::runtime_error("Input matrix is not square");

  // LU decomposition with pivoting
  auto res = lup(A);

  // Determinants of LU
  T detLU = static_cast<T>(1);

  for (unsigned i = 0; i < res.L.rows(); i++)
    detLU *= res.L(i,i) * res.U(i,i);

  // Determinant of P
  unsigned len = res.P.size();
  T detP = static_cast<T>(1);

  std::vector<unsigned> p(res.P);
  std::vector<unsigned> q;
  q.resize(len);

  for (unsigned i = 0; i < len; i++)
    q[p[i]] = i;

  for (unsigned i = 0; i < len; i++) {
    unsigned j = p[i];
    unsigned k = q[i];
    if (j != i) {
      p[k] = p[i];
      q[j] = q[i];
      detP = - detP;
    }
  }

  return detLU * detP;
}

template<typename T>
T det(const Matrix<T>& A) {
  if (! A.issquare()) throw std::runtime_error("Input matrix is not square");

  if (A.rows() == 1)
    return A(0,0);
  else if (A.rows() == 2)
    return A(0,0)*A(1,1) - A(0,1)*A(1,0);
```

```
  else if (A.rows() == 3)
    return A(0,0)*(A(1,1)*A(2,2) - A(1,2)*A(2,1)) -
           A(0,1)*(A(1,0)*A(2,2) - A(1,2)*A(2,0)) +
           A(0,2)*(A(1,0)*A(2,1) - A(1,1)*A(2,0));
  else
    return det_lu(A);
}

template<typename T>
LU_result<T> lu(const Matrix<T>& A) {
  const unsigned M = A.rows();
  const unsigned N = A.cols();

  LU_result<T> res;
  res.L = eye<T>(M);
  res.U = Matrix<T>(A);

  // aliases
  auto& L = res.L;
  auto& U = res.U;

  if (A.numel() == 0)
    return res;

  for (unsigned k = 0; k < M-1; k++) {
    for (unsigned i = k+1; i < M; i++) {
      L(i,k) = U(i,k) / U(k,k);
      for (unsigned l = k+1; l < N; l++) {
        U(i,l) -= L(i,k) * U(k,l);
      }
    }
  }

  for (unsigned col = 0; col < N; col++)
    for (unsigned row = col+1; row < M; row++)
      U(row,col) = 0;

  return res;
}

template<typename T>
LUP_result<T> lup(const Matrix<T>& A) {
  const unsigned M = A.rows();
  const unsigned N = A.cols();

  // Initialize L, U, and PP
  LUP_result<T> res;

  if (A.numel() == 0)
    return res;

  res.L = eye<T>(M);
  res.U = Matrix<T>(A);
  std::vector<unsigned> PP;

  // aliases
  auto& L = res.L;
  auto& U = res.U;

  PP.resize(N);
  for (unsigned i = 0; i < N; i++)
    PP[i] = i;

  for (unsigned k = 0; k < M-1; k++) {
    // Find the column with the largest absolute value in the current row
    auto max_col_value = std::abs(U(k,k));
    unsigned max_col_index = k;
    for (unsigned l = k+1; l < N; l++) {
      auto val = std::abs(U(k,l));
      if (val > max_col_value) {
        max_col_value = val;
        max_col_index = l;
      }
    }

    // Swap columns k and max_col_index in U and update P
    if (max_col_index != k) {
      U.swap_cols(k, max_col_index); // TODO: This could be reworked to avoid column swap in U during every iteration by:
                                     //       1. using PP[k] for column indexing across iterations
                                     //       2. doing just one permutation of U at the end
      std::swap(PP[k], PP[max_col_index]);
    }

    // Update L and U
    for (unsigned i = k+1; i < M; i++) {
      L(i,k) = U(i,k) / U(k,k);
```

```
      for (unsigned l = k+1; l < N; l++) {
        U(i,l) -= L(i,k) * U(k,l);
      }
    }
  }

  // Set elements in lower triangular part of U to zero
  for (unsigned col = 0; col < N; col++)
    for (unsigned row = col+1; row < M; row++)
      U(row,col) = 0;

  // Transpose indices in permutation vector
  res.P.resize(N);
  for (unsigned i = 0; i < N; i++)
    res.P[PP[i]] = i;

  return res;
}

template<typename T>
Matrix<T> rref(const Matrix<T>& A, T tol = 0) {
  unsigned row = 0;

  Matrix<T> B(A);

  for (unsigned c = 0; c < B.cols(); c++) {
    // stop if already found pivots for all rows
    if (row >= B.rows())
      break;

    // find the pivot row
    T max_val = static_cast<T>(0);
    unsigned pivot_row = row;

    for (unsigned i = row; i < B.rows(); i++) {
      T x = static_cast<T>(std::abs(B(i,c)));
      if (Util::creal(x) > Util::creal(max_val)) {
        max_val = x;
        pivot_row = i;
      }
    }

    if (Util::creal(max_val) <= Util::creal(tol)) {
      // skip column c
      for (unsigned i = row; i < B.rows(); i++)
        B(i,c) = static_cast<T>(0);
    } else {
      // swap current row with the pivot row
      if (pivot_row != row)
        for (unsigned j = c; j < B.cols(); j++)
          std::swap(B(row,j), B(pivot_row,j));

      // normalize pivot row if not normalized
      if (B(row,c) != static_cast<T>(1)) {
        auto factor = static_cast<T>(1) / B(row,c);
        for (unsigned j = c; j < B.cols(); j++)
          B(row,j) *= factor;
      }

      // eliminate current column
      for (unsigned i = 0; i < B.rows(); i++) {
        if (i != row) {
          auto factor = B(i,c);
          for (unsigned j = c; j < B.cols(); j++)
            B(i,j) -= factor * B(row,j);
        }
      }

      row++;
    }
  }

  return B;
}

template<typename T>
Matrix<T> inv_gauss_jordan(const Matrix<T>& A) {
  if (! A.issquare()) throw std::runtime_error("Input matrix is not square");

  const unsigned N = A.rows();
  Matrix<T> AA(A);
  auto IA = eye<T>(N);

  bool found_nonzero;
  for (unsigned j = 0; j < N; j++) {
    found_nonzero = false;
    for (unsigned i = j; i < N; i++) {
```

```
      if (AA(i,j) != static_cast<T>(0)) {
        found_nonzero = true;
        for (unsigned k = 0; k < N; k++) {
          std::swap(AA(j,k), AA(i,k));
          std::swap(IA(j,k), IA(i,k));
        }
        if (AA(j,j) != static_cast<T>(1)) {
          T s = static_cast<T>(1) / AA(j,j);
          for (unsigned k = 0; k < N; k++) {
            AA(j,k) *= s;
            IA(j,k) *= s;
          }
        }
        for (unsigned l = 0; l < N; l++) {
          if (l != j) {
            T s = AA(l,j);
            for (unsigned k = 0; k < N; k++) {
              AA(l,k) -= s * AA(j,k);
              IA(l,k) -= s * IA(j,k);
            }
          }
        }
      }
      break;
    }
    // if a row full of zeros is found, the input matrix was singular
    if (!found_nonzero) throw singular_matrix_exception("Singular matrix in inv_gauss_jordan");
  }
  return IA;
}

template<typename T>
Matrix<T> inv_tril(const Matrix<T>& A) {
  if (! A.issquare()) throw std::runtime_error("Input matrix is not square");

  const unsigned N = A.rows();

  auto IA = zeros<T>(N);

  for (unsigned i = 0; i < N; i++) {
    if (A(i,i) == static_cast<T>(0.0)) throw singular_matrix_exception("Division by zero in inv_tril");

    IA(i,i) = static_cast<T>(1.0) / A(i,i);
    for (unsigned j = 0; j < i; j++) {
      T s = 0.0;
      for (unsigned k = j; k < i; k++)
        s += A(i,k) * IA(k,j);
      IA(i,j) = -s * IA(i,i) ;
    }
  }

  return IA;
}

template<typename T>
Matrix<T> inv_triu(const Matrix<T>& A) {
  if (!A.issquare()) throw std::runtime_error("Input matrix is not square");

  const unsigned N = A.rows();

  auto IA = zeros<T>(N);

  for (int i = N - 1; i >= 0; i--) {
    if (A(i,i) == static_cast<T>(0.0)) throw singular_matrix_exception("Division by zero in inv_triu");

    IA(i, i) = static_cast<T>(1.0) / A(i,i);
    for (int j = N - 1; j > i; j--) {
      T s = static_cast<T>(0.0);
      for (int k = i + 1; k <= j; k++)
        s += A(i,k) * IA(k,j);
      IA(i,j) = -s * IA(i,i);
    }
  }

  return IA;
}

template<typename T>
Matrix<T> inv_posdef(const Matrix<T>& A) {
  auto L = cholinv(A);
  return mult<T,true,false>(L,L);
}

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

```
  if (! A.issquare()) throw std::runtime_error("Input matrix is not square");

  // LU decomposition with pivoting
  auto LU = lup(A);
  auto IL = inv_tril(LU.L);
  auto IU = inv_triu(LU.U);

  return permute_rows(IU * IL, LU.P);
}

template<typename T>
Matrix<T> inv(const Matrix<T>& A) {
  if (! A.issquare()) throw std::runtime_error("Input matrix is not square");

  if (A.numel() == 0) {
    return Matrix<T>();
  } else if (A.rows() < 4) {
    T d = det(A);

    if (d == static_cast<T>(0.0)) throw singular_matrix_exception("Singular matrix in inv");

    Matrix<T> IA(A.rows(), A.rows());
    T invdet = static_cast<T>(1.0) / d;

    if (A.rows() == 1) {
      IA(0,0) = invdet;
    } else if (A.rows() == 2) {
      IA(0,0) =   A(1,1) * invdet;
      IA(0,1) = - A(0,1) * invdet;
      IA(1,0) = - A(1,0) * invdet;
      IA(1,1) =   A(0,0) * invdet;
    } else if (A.rows() == 3) {
      IA(0,0) = (A(1,1)*A(2,2) - A(2,1)*A(1,2)) * invdet;
      IA(0,1) = (A(0,2)*A(2,1) - A(0,1)*A(2,2)) * invdet;
      IA(0,2) = (A(0,1)*A(1,2) - A(0,2)*A(1,1)) * invdet;
      IA(1,0) = (A(1,2)*A(2,0) - A(1,0)*A(2,2)) * invdet;
      IA(1,1) = (A(0,0)*A(2,2) - A(0,2)*A(2,0)) * invdet;
      IA(1,2) = (A(1,0)*A(0,2) - A(0,0)*A(1,2)) * invdet;
      IA(2,0) = (A(1,0)*A(2,1) - A(2,0)*A(1,1)) * invdet;
      IA(2,1) = (A(2,0)*A(0,1) - A(0,0)*A(2,1)) * invdet;
      IA(2,2) = (A(0,0)*A(1,1) - A(1,0)*A(0,1)) * invdet;
    }

    return IA;
  } else {
    return inv_square(A);
  }
}

template<typename T>
Matrix<T> pinv(const Matrix<T>& A) {
  if (A.rows() > A.cols()) {
    auto AH_A = mult<T,true,false>(A, A);
    auto Linv = inv_posdef(AH_A);
    return mult<T,false,true>(Linv, A);
  } else {
    auto AA_H = mult<T,false,true>(A, A);
    auto Linv = inv_posdef(AA_H);
    return mult<T,true,false>(A, Linv);
  }
}

template<typename T>
T trace(const Matrix<T>& A) {
  T t = static_cast<T>(0);
  for (int i = 0; i < A.rows(); i++)
    t += A(i,i);
  return t;
}

template<typename T>
double cond(const Matrix<T>& A) {
  try {
    auto A_inv = inv(A);
    return norm_fro(A) * norm_fro(A_inv);
  } catch (singular_matrix_exception& e) {
    return std::numeric_limits<double>::max();
  }
}

template<typename T, bool is_upper = false>
Matrix<T> chol(const Matrix<T>& A) {
  if (! A.issquare()) throw std::runtime_error("Input matrix is not square");

  const unsigned N = A.rows();

  // Calculate lower or upper triangular, depending on template parameter.
```

```
  // Calculation is the same - the difference is in transposed row and column indexing.
  Matrix<T> C = is_upper ? triu(A) : tril(A);

  for (unsigned j = 0; j < N; j++) {
    if (C(j,j) == static_cast<T>(0.0)) throw singular_matrix_exception("Singular matrix in chol");

    C(j,j) = std::sqrt(C(j,j));

    for (unsigned k = j+1; k < N; k++)
      if (is_upper)
        C(j,k) /= C(j,j);
      else
        C(k,j) /= C(j,j);

    for (unsigned k = j+1; k < N; k++)
      for (unsigned i = k; i < N; i++)
        if (is_upper)
          C(k,i) -= C(j,i) * Util::cconj(C(j,k));
        else
          C(i,k) -= C(i,j) * Util::cconj(C(k,j));
  }

  return C;
}

template<typename T>
Matrix<T> cholinv(const Matrix<T>& A) {
  if (! A.issquare()) throw std::runtime_error("Input matrix is not square");

  const unsigned N = A.rows();
  Matrix<T> L(A);
  auto Linv = eye<T>(N);

  for (unsigned j = 0; j < N; j++) {
    if (L(j,j) == static_cast<T>(0.0)) throw singular_matrix_exception("Singular matrix in cholinv");

    L(j,j) = static_cast<T>(1.0) / std::sqrt(L(j,j));

    for (unsigned k = j+1; k < N; k++)
      L(k,j) = L(k,j) * L(j,j);

    for (unsigned k = j+1; k < N; k++)
      for (unsigned i = k; i < N; i++)
        L(i,k) = L(i,k) - L(i,j) * Util::cconj(L(k,j));
  }

  for (unsigned k = 0; k < N; k++) {
    for (unsigned i = k; i < N; i++) {
      Linv(i,k) = Linv(i,k) * L(i,i);
      for (unsigned j = i+1; j < N; j++)
        Linv(j,k) = Linv(j,k) - L(j,i) * Linv(i,k);
    }
  }

  return Linv;
}

template<typename T>
LDL_result<T> ldl(const Matrix<T>& A) {
  if (! A.issquare()) throw std::runtime_error("Input matrix is not square");

  const unsigned N = A.rows();

  LDL_result<T> res;

  // aliases
  auto& L = res.L;
  auto& d = res.d;

  L = eye<T>(N);
  d.resize(N);

  for (unsigned m = 0; m < N; m++) {
    d[m] = A(m,m);

    for (unsigned k = 0; k < m; k++)
      d[m] -= L(m,k) * Util::cconj(L(m,k)) * d[k];

    if (d[m] == static_cast<T>(0.0)) throw singular_matrix_exception("Singular matrix in ldl");

    for (unsigned n = m+1; n < N; n++) {
      L(n,m) = A(n,m);
      for (unsigned k = 0; k < m; k++)
        L(n,m) -= L(n,k) * Util::cconj(L(m,k)) * d[k];
      L(n,m) /= d[m];
    }
  }
```

```

  return res;
}

template<typename T>
QR_result<T> qr_red_gs(const Matrix<T>& A) {
  const int rows = A.rows();
  const int cols = A.cols();

  QR_result<T> res;

  //aliases
  auto& Q = res.Q;
  auto& R = res.R;

  Q = zeros<T>(rows, cols);
  R = zeros<T>(cols, cols);

  for (int c = 0; c < cols; c++) {
    Matrix<T> v = A.get_submatrix(0, rows-1, c, c);
    for (int r = 0; r < c; r++) {
      for (int k = 0; k < rows; k++)
        R(r,c) = R(r,c) + Util::cconj(Q(k,r)) * A(k,c);
      for (int k = 0; k < rows; k++)
        v(k) = v(k) - R(r,c) * Q(k,r);
    }

    R(c,c) = static_cast<T>(norm_fro(v));

    if (R(c,c) == static_cast<T>(0.0)) throw singular_matrix_exception("Division by 0 in QR GS");

    for (int k = 0; k < rows; k++)
      Q(k,c) = v(k) / R(c,c);
  }

  return res;
}

template<typename T>
Matrix<T> householder_reflection(const Matrix<T>& a) {
  if (a.cols() != 1) throw std::runtime_error("Input not a column vector");

  static const T ISQRT2 = static_cast<T>(0.707106781186547);

  Matrix<T> v(a);
  v(0) += Util::csign(v(0)) * norm_fro(v);
  auto vn = norm_fro(v) * ISQRT2;
  for (unsigned i = 0; i < v.numel(); i++)
    v(i) /= vn;
  return v;
}

template<typename T>
QR_result<T> qr_householder(const Matrix<T>& A, bool calculate_Q = true) {
  const unsigned rows = A.rows();
  const unsigned cols = A.cols();

  QR_result<T> res;

  //aliases
  auto& Q = res.Q;
  auto& R = res.R;

  R = Matrix<T>(A);

  if (calculate_Q)
    Q = eye<T>(rows);

  const unsigned N = (rows > cols) ? cols : rows;

  for (unsigned j = 0; j < N; j++) {
    auto v = householder_reflection(R.get_submatrix(j, rows-1, j, j));

    auto R1 = R.get_submatrix(j, rows-1, j, cols-1);
    auto WR = v * mult<T,true,false>(v, R1);
    for (unsigned c = j; c < cols; c++)
      for (unsigned r = j; r < rows; r++)
        R(r,c) -= WR(r-j,c-j);

    if (calculate_Q) {
      auto Q1 = Q.get_submatrix(0, rows-1, j, rows-1);
      auto WQ = mult<T,false,true>(Q1 * v, v);
      for (unsigned c = j; c < rows; c++)
        for (unsigned r = 0; r < rows; r++)
          Q(r,c) -= WQ(r,c-j);
    }
  }
```

```

  for (unsigned col = 0; col < R.cols(); col++)
    for (unsigned row = col+1; row < R.rows(); row++)
      R(row,col) = 0;

  return res;
}

template<typename T>
inline QR_result<T> qr(const Matrix<T>& A, bool calculate_Q = true) {
  return qr_householder(A, calculate_Q);
}

template<typename T>
Hessenberg_result<T> hessenberg(const Matrix<T>& A, bool calculate_Q = true) {
  if (! A.issquare()) throw std::runtime_error("Input matrix is not square");

  Hessenberg_result<T> res;

  // aliases
  auto& H = res.H;
  auto& Q = res.Q;

  const unsigned N = A.rows();
  H = Matrix<T>(A);

  if (calculate_Q)
    Q = eye<T>(N);

  for (unsigned k = 1; k < N-1; k++) {
    auto v = householder_reflection(H.get_submatrix(k, N-1, k-1, k-1));

    auto H1 = H.get_submatrix(k, N-1, 0, N-1);
    auto W1 = v * mult<T,true,false>(v, H1);
    for (unsigned c = 0; c < N; c++)
      for (unsigned r = k; r < N; r++)
        H(r,c) -= W1(r-k,c);

    auto H2 = H.get_submatrix(0, N-1, k, N-1);
    auto W2 = mult<T,false,true>(H2 * v, v);
    for (unsigned c = k; c < N; c++)
      for (unsigned r = 0; r < N; r++)
        H(r,c) -= W2(r,c-k);

    if (calculate_Q) {
      auto Q1 = Q.get_submatrix(0, N-1, k, N-1);
      auto W3 = mult<T,false,true>(Q1 * v, v);
      for (unsigned c = k; c < N; c++)
        for (unsigned r = 0; r < N; r++)
          Q(r,c) -= W3(r,c-k);
    }
  }

  for (unsigned row = 2; row < N; row++)
    for (unsigned col = 0; col < row-2; col++)
      H(row,col) = static_cast<T>(0);

  return res;
}

template<typename T>
std::complex<T> wilkinson_shift(const Matrix<std::complex<T»& H, T tol = 1e-10) {
  if (! H.issquare()) throw std::runtime_error("Input matrix is not square");

  const unsigned n = H.rows();
  std::complex<T> mu;

  if (std::abs(H(n-1,n-2)) < tol) {
    mu = H(n-2,n-2);
  } else {
    auto trA = H(n-2,n-2) + H(n-1,n-1);
    auto detA = H(n-2,n-2) * H(n-1,n-1) - H(n-2, n-1) * H(n-1, n-2);
    mu = (trA + std::sqrt(trA*trA - 4.0*detA)) / 2.0;
  }

  return mu;
}

template<typename T>
Eigenvalues_result<T> eigenvalues(const Matrix<std::complex<T»& A, T tol = 1e-12, unsigned max_iter = 100) {
  if (! A.issquare()) throw std::runtime_error("Input matrix is not square");

  const unsigned N = A.rows();
  Matrix<std::complex<T>> H;
  bool success = false;

```

```
  QR_result<std::complex<T>> QR;

  // aliases
  auto& Q = QR.Q;
  auto& R = QR.R;

  // Transfer A to Hessenberg form to improve convergence (skip calculation of Q)
  H = hessenberg(A, false).H;

  for (unsigned iter = 0; iter < max_iter; iter++) {
    auto mu = wilkinson_shift(H, tol);

    // subtract mu from diagonal
    for (unsigned n = 0; n < N; n++)
      H(n,n) -= mu;

    // QR factorization with shifted H
    QR = qr(H);
    H = R * Q;

    // add back mu to diagonal
    for (unsigned n = 0; n < N; n++)
      H(n,n) += mu;

    // Check for convergence
    if (std::abs(H(N-2,N-1)) <= tol) {
      success = true;
      break;
    }
  }

  Eigenvalues_result<T> res;
  res.eig = diag(H);
  res.err = std::abs(H(N-2,N-1));
  res.converged = success;

  return res;
}

template<typename T>
Eigenvalues_result<T> eigenvalues(const Matrix<T>& A, T tol = 1e-12, unsigned max_iter = 100) {
  auto A_cplx = make_complex(A);
  return eigenvalues(A_cplx, tol, max_iter);
}

template<typename T>
Matrix<T> solve_triu(const Matrix<T>& U, const Matrix<T>& B) {
  if (! U.issquare()) throw std::runtime_error("Input matrix is not square");
  if (U.rows() != B.rows()) throw std::runtime_error("Unmatching matrix dimensions for solve");

  const unsigned N = U.rows();
  const unsigned M = B.cols();

  if (U.numel() == 0)
    return Matrix<T>();

  Matrix<T> X(B);

  for (unsigned m = 0; m < M; m++) {
    // backwards substitution for each column of B
    for (int n = N-1; n >= 0; n--) {
      for (unsigned j = n + 1; j < N; j++)
        X(n,m) -= U(n,j) * X(j,m);

      if (U(n,n) == static_cast<T>(0.0)) throw singular_matrix_exception("Singular matrix in
solve_triu");

      X(n,m) /= U(n,n);
    }
  }

  return X;
}

template<typename T>
Matrix<T> solve_tril(const Matrix<T>& L, const Matrix<T>& B) {
  if (! L.issquare()) throw std::runtime_error("Input matrix is not square");
  if (L.rows() != B.rows()) throw std::runtime_error("Unmatching matrix dimensions for solve");

  const unsigned N = L.rows();
  const unsigned M = B.cols();

  if (L.numel() == 0)
    return Matrix<T>();

  Matrix<T> X(B);

```

```
  for (unsigned m = 0; m < M; m++) {
    // forwards substitution for each column of B
    for (unsigned n = 0; n < N; n++) {
      for (unsigned j = 0; j < n; j++)
        X(n,m) -= L(n,j) * X(j,m);

      if (L(n,n) == static_cast<T>(0.0)) throw singular_matrix_exception("Singular matrix in solve_tril");

      X(n,m) /= L(n,n);
    }
  }

  return X;
}

template<typename T>
Matrix<T> solve_square(const Matrix<T>& A, const Matrix<T>& B) {
  if (! A.issquare()) throw std::runtime_error("Input matrix is not square");
  if (A.rows() != B.rows()) throw std::runtime_error("Unmatching matrix dimensions for solve");

  if (A.numel() == 0)
    return Matrix<T>();

  Matrix<T> L;
  Matrix<T> U;
  std::vector<unsigned> P;

  // LU decomposition with pivoting
  auto lup_res = lup(A);

  auto y = solve_tril(lup_res.L, B);
  auto x = solve_triu(lup_res.U, y);

  return permute_rows(x, lup_res.P);
}

template<typename T>
Matrix<T> solve_posdef(const Matrix<T>& A, const Matrix<T>& B) {
  if (! A.issquare()) throw std::runtime_error("Input matrix is not square");
  if (A.rows() != B.rows()) throw std::runtime_error("Unmatching matrix dimensions for solve");

  if (A.numel() == 0)
    return Matrix<T>();

  // LU decomposition with pivoting
  auto L = chol(A);

  auto Y = solve_tril(L, B);
  return solve_triu(L.ctranspose(), Y);
}

template<typename T>
class Matrix {
  public:
    Matrix();

    Matrix(unsigned size);

    Matrix(unsigned nrows, unsigned ncols);

    Matrix(T x, unsigned nrows, unsigned ncols);

    Matrix(const T* array, unsigned nrows, unsigned ncols);

    Matrix(const std::vector<T>& vec, unsigned nrows, unsigned ncols);

    Matrix(std::initializer_list<T> init_list, unsigned nrows, unsigned ncols);

    Matrix(const Matrix &);

    virtual ~Matrix();

    Matrix<T> get_submatrix(unsigned row_first, unsigned row_last, unsigned col_first, unsigned col_last) const;

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

    void clear();

    void reshape(unsigned rows, unsigned cols);

    void resize(unsigned rows, unsigned cols);

    bool exists(unsigned row, unsigned col) const;

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

```
    T* ptr();

    void fill(T value);

    void fill_col(T value, unsigned col);

    void fill_row(T value, unsigned row);

    bool isempty() const;

    bool issquare() const;

    bool isequal(const Matrix<T>&) const;

    bool isequal(const Matrix<T>&, T) const;

    unsigned numel() const;

    unsigned rows() const;

    unsigned cols() const;

    std::pair<unsigned,unsigned> shape() const;

    Matrix<T> transpose() const;

    Matrix<T> ctranspose() const;

    Matrix<T>& add(const Matrix<T>&);

    Matrix<T>& subtract(const Matrix<T>&);

    Matrix<T>& mult_hadamard(const Matrix<T>&);

    Matrix<T>& add(T);

    Matrix<T>& subtract(T);

    Matrix<T>& mult(T);

    Matrix<T>& div(T);

    Matrix<T>& operator=(const Matrix<T>&);

    Matrix<T>& operator=(T);

    explicit operator std::vector<T>() const;
    std::vector<T> to_vector() const;

    T& operator()(unsigned nel);
    T  operator()(unsigned nel) const;
    T& at(unsigned nel);
    T  at(unsigned nel) const;

    T& operator()(unsigned row, unsigned col);
    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);

    void add_col_to_another(unsigned to, unsigned from);

    void mult_row_by_another(unsigned to, unsigned from);

    void mult_col_by_another(unsigned to, unsigned from);

    void swap_rows(unsigned i, unsigned j);

    void swap_cols(unsigned i, unsigned j);

    std::vector<T> col_to_vector(unsigned col) const;

    std::vector<T> row_to_vector(unsigned row) const;

    void col_from_vector(const std::vector<T>&, unsigned col);

    void row_from_vector(const std::vector<T>&, unsigned row);

  private:
    unsigned nrows;
    unsigned ncols;
    std::vector<T> data;
};

/*
```

```
 * Implementation of Matrix class methods
 */

template<typename T>
Matrix<T>::Matrix() : nrows(0), ncols(0), data() { }

template<typename T>
Matrix<T>::Matrix(unsigned size) : Matrix(size, size) { }

template<typename T>
Matrix<T>::Matrix(unsigned rows, unsigned cols) : nrows(rows), ncols(cols) {
  data.resize(numel());
}

template<typename T>
Matrix<T>::Matrix(T x, unsigned rows, unsigned cols) : Matrix(rows, cols)  {
  fill(x);
}

template<typename T>
Matrix<T>::Matrix(const T* array, unsigned rows, unsigned cols) : Matrix(rows, cols)  {
  data.assign(array, array + numel());
}

template<typename T>
Matrix<T>::Matrix(const std::vector<T>& vec, unsigned rows, unsigned cols) : Matrix(rows, cols) {
  if (vec.size() != numel()) throw std::runtime_error("Size of initialization vector not consistent with matrix dimensions");

  data.assign(vec.begin(), vec.end());
}

template<typename T>
Matrix<T>::Matrix(std::initializer_list<T> init_list, unsigned rows, unsigned cols) : Matrix(rows, cols) {
  if (init_list.size() != numel()) throw std::runtime_error("Size of initialization list not consistent with matrix dimensions");

  auto it = init_list.begin();

  for (unsigned row = 0; row < this->nrows; row++)
    for (unsigned col = 0; col < this->ncols; col++)
      this->at(row,col) = *(it++);
}

template<typename T>
Matrix<T>::Matrix(const Matrix & other) : Matrix(other.nrows, other.ncols) {
  this->data.assign(other.data.begin(), other.data.end());
}

template<typename T>
Matrix<T>& Matrix<T>::operator=(const Matrix<T>& other) {
  this->nrows = other.nrows;
  this->ncols = other.ncols;
  this->data.assign(other.data.begin(), other.data.end());
  return *this;
}

template<typename T>
Matrix<T>& Matrix<T>::operator=(T s) {
  fill(s);
  return *this;
}

template<typename T>
inline Matrix<T>::operator std::vector<T>() const {
  return data;
}

template<typename T>
inline void Matrix<T>::clear() {
  this->nrows = 0;
  this->ncols = 0;
  data.resize(0);
}

template<typename T>
void Matrix<T>::reshape(unsigned rows, unsigned cols) {
  if (this->numel() != rows * cols) throw std::runtime_error("Illegal attempt to change number of elements via reshape");

  this->nrows = rows;
  this->ncols = cols;
}

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

```
  this->nrows = rows;
  this->ncols = cols;
  data.resize(nrows*ncols);
}

template<typename T>
Matrix<T> Matrix<T>::get_submatrix(unsigned row_base, unsigned row_lim, unsigned col_base, unsigned col_lim) const {
  if (row_base > row_lim) throw std::out_of_range("Row index of submatrix out of range");
  if (col_base > col_lim) throw std::out_of_range("Column index of submatrix out of range");
  if (row_lim >= this->rows()) throw std::out_of_range("Row index of submatrix out of range");
  if (col_lim >= this->cols()) throw std::out_of_range("Column index of submatrix out of range");

  unsigned num_rows = row_lim - row_base + 1;
  unsigned num_cols = col_lim - col_base + 1;
  Matrix<T> S(num_rows, num_cols);
  for (unsigned i = 0; i < num_rows; i++) {
    for (unsigned j = 0; j < num_cols; j++) {
      S(i,j) = at(row_base + i, col_base + j);
    }
  }
  return S;
}

template<typename T>
void Matrix<T>::set_submatrix(const Matrix<T>& S, unsigned row_base, unsigned col_base) {
  if (this->isempty()) throw std::runtime_error("Invalid attempt to set submatrix in empty matrix");

  const unsigned row_lim = row_base + S.rows() - 1;
  const unsigned col_lim = col_base + S.cols() - 1;

  if (row_base > row_lim) throw std::out_of_range("Row index of submatrix out of range");
  if (col_base > col_lim) throw std::out_of_range("Column index of submatrix out of range");
  if (row_lim >= this->rows()) throw std::out_of_range("Row index of submatrix out of range");
  if (col_lim >= this->cols()) throw std::out_of_range("Column index of submatrix out of range");

  unsigned num_rows = row_lim - row_base + 1;
  unsigned num_cols = col_lim - col_base + 1;
  for (unsigned i = 0; i < num_rows; i++)
    for (unsigned j = 0; j < num_cols; j++)
      at(row_base + i, col_base + j) = S(i,j);
}

template<typename T>
inline T & Matrix<T>::operator()(unsigned nel) {
  return at(nel);
}

template<typename T>
inline T & Matrix<T>::operator()(unsigned row, unsigned col) {
  return at(row, col);
}

template<typename T>
inline T Matrix<T>::operator()(unsigned nel) const {
  return at(nel);
}

template<typename T>
inline T Matrix<T>::operator()(unsigned row, unsigned col) const {
  return at(row, col);
}

template<typename T>
inline T & Matrix<T>::at(unsigned nel) {
#ifdef MATRIX_STRICT_BOUNDS_CHECK
  if (!(nel < numel())) throw std::out_of_range("Element index out of range");
#endif

  return data[nel];
}

template<typename T>
inline T & Matrix<T>::at(unsigned row, unsigned col) {
#ifdef MATRIX_STRICT_BOUNDS_CHECK
  if (!(row < rows() && col < cols())) throw std::out_of_range("Element index out of range");
#endif

  return data[nrows * col + row];
}

template<typename T>
inline T Matrix<T>::at(unsigned nel) const {
#ifdef MATRIX_STRICT_BOUNDS_CHECK
  if (!(nel < numel())) throw std::out_of_range("Element index out of range");
#endif

```

```
  return data[nel];
}

template<typename T>
inline T Matrix<T>::at(unsigned row, unsigned col) const {
#ifdef MATRIX_STRICT_BOUNDS_CHECK
  if (!(row < rows())) throw std::out_of_range("Row index out of range");
  if (!(col < cols())) throw std::out_of_range("Column index out of range");
#endif

  return data[nrows * col + row];
}

template<typename T>
inline void Matrix<T>::fill(T value) {
  for (unsigned i = 0; i < numel(); i++)
    data[i] = value;
}

template<typename T>
inline void Matrix<T>::fill_col(T value, unsigned col) {
  if (!(col < cols())) throw std::out_of_range("Column index out of range");

  for (unsigned i = col * nrows; i < (col+1) * nrows; i++)
    data[i] = value;
}

template<typename T>
inline void Matrix<T>::fill_row(T value, unsigned row) {
  if (!(row < rows())) throw std::out_of_range("Row index out of range");

  for (unsigned i = 0; i < ncols; i++)
    data[row + i * nrows] = value;
}

template<typename T>
inline bool Matrix<T>::exists(unsigned row, unsigned col) const {
  return (row < nrows && col < ncols);
}

template<typename T>
inline T* Matrix<T>::ptr(unsigned row, unsigned col) {
  if (!(row < rows())) throw std::out_of_range("Row index out of range");
  if (!(col < cols())) throw std::out_of_range("Column index out of range");

  return data.data() + nrows * col + row;
}

template<typename T>
inline T* Matrix<T>::ptr() {
  return data.data();
}

template<typename T>
inline bool Matrix<T>::isempty() const {
  return (nrows == 0) || (ncols == 0);
}

template<typename T>
inline bool Matrix<T>::issquare() const {
  return (nrows == ncols) && !isempty();
}

template<typename T>
bool Matrix<T>::isequal(const Matrix<T>& A) const {
  bool ret = true;
  if (nrows != A.rows() || ncols != A.cols()) {
    ret = false;
  } else {
    for (unsigned i = 0; i < numel(); i++) {
      if (at(i) != A(i)) {
        ret = false;
        break;
      }
    }
  }
  return ret;
}

template<typename T>
bool Matrix<T>::isequal(const Matrix<T>& A, T tol) const {
  bool ret = true;
  if (rows() != A.rows() || cols() != A.cols()) {
    ret = false;
  } else {
    auto abs_tol = std::abs(tol); // workaround for complex
    for (unsigned i = 0; i < A.numel(); i++) {
```

```
      if (abs_tol < std::abs(at(i) - A(i))) {
        ret = false;
        break;
      }
    }
  }
  return ret;
}

template<typename T>
inline unsigned Matrix<T>::numel() const {
  return nrows * ncols;
}

template<typename T>
inline unsigned Matrix<T>::rows() const {
  return nrows;
}

template<typename T>
inline unsigned Matrix<T>::cols() const {
  return ncols;
}

template<typename T>
inline std::pair<unsigned,unsigned> Matrix<T>::shape() const {
  return std::pair<unsigned,unsigned>(nrows,ncols);
}

template<typename T>
inline Matrix<T> Matrix<T>::transpose() const {
  Matrix<T> res(ncols, nrows);
  for (unsigned c = 0; c < ncols; c++)
    for (unsigned r = 0; r < nrows; r++)
      res(c,r) = at(r,c);
  return res;
}

template<typename T>
inline Matrix<T> Matrix<T>::ctranspose() const {
  Matrix<T> res(ncols, nrows);
  for (unsigned c = 0; c < ncols; c++)
    for (unsigned r = 0; r < nrows; r++)
      res(c,r) = Util::cconj(at(r,c));
  return res;
}

template<typename T>
Matrix<T>& Matrix<T>::add(const Matrix<T>& m) {
  if (!(m.rows() == rows() && m.cols() == cols())) throw std::runtime_error("Unmatching matrix
dimensions for iadd");

  for (unsigned i = 0; i < numel(); i++)
    data[i] += m(i);
  return *this;
}

template<typename T>
Matrix<T>& Matrix<T>::subtract(const Matrix<T>& m) {
  if (!(m.rows() == rows() && m.cols() == cols())) throw std::runtime_error("Unmatching matrix
dimensions for isubtract");

  for (unsigned i = 0; i < numel(); i++)
    data[i] -= m(i);
  return *this;
}

template<typename T>
Matrix<T>& Matrix<T>::mult_hadamard(const Matrix<T>& m) {
  if (!(m.rows() == rows() && m.cols() == cols())) throw std::runtime_error("Unmatching matrix
dimensions for ihprod");

  for (unsigned i = 0; i < numel(); i++)
    data[i] *= m(i);
  return *this;
}

template<typename T>
Matrix<T>& Matrix<T>::add(T s) {
  for (auto& x : data)
    x += s;
  return *this;
}

template<typename T>
Matrix<T>& Matrix<T>::subtract(T s) {
  for (auto& x : data)
```

```
    x -= s;
  return *this;
}

template<typename T>
Matrix<T>& Matrix<T>::mult(T s) {
  for (auto& x : data)
    x *= s;
  return *this;
}

template<typename T>
Matrix<T>& Matrix<T>::div(T s) {
  for (auto& x : data)
    x /= s;
  return *this;
}

template<typename T>
void Matrix<T>::add_row_to_another(unsigned to, unsigned from) {
  if (!(to < rows() && from < rows())) throw std::out_of_range("Row index out of range");

  for (unsigned k = 0; k < cols(); k++)
    at(to, k) += at(from, k);
}

template<typename T>
void Matrix<T>::add_col_to_another(unsigned to, unsigned from) {
  if (!(to < cols() && from < cols())) throw std::out_of_range("Column index out of range");

  for (unsigned k = 0; k < rows(); k++)
    at(k, to) += at(k, from);
}

template<typename T>
void Matrix<T>::mult_row_by_another(unsigned to, unsigned from) {
  if (!(to < rows() && from < rows())) throw std::out_of_range("Row index out of range");

  for (unsigned k = 0; k < cols(); k++)
    at(to, k) *= at(from, k);
}

template<typename T>
void Matrix<T>::mult_col_by_another(unsigned to, unsigned from) {
  if (!(to < cols() && from < cols())) throw std::out_of_range("Column index out of range");

  for (unsigned k = 0; k < rows(); k++)
    at(k, to) *= at(k, from);
}

template<typename T>
void Matrix<T>::swap_rows(unsigned i, unsigned j) {
  if (!(i < rows() && j < rows())) throw std::out_of_range("Row index out of range");

  for (unsigned k = 0; k < cols(); k++)
    std::swap(at(i,k), at(j,k));
}

template<typename T>
void Matrix<T>::swap_cols(unsigned i, unsigned j) {
  if (!(i < cols() && j < cols())) throw std::out_of_range("Column index out of range");

  for (unsigned k = 0; k < rows(); k++)
    std::swap(at(k,i), at(k,j));
}

template<typename T>
inline std::vector<T> Matrix<T>::to_vector() const {
  return data;
}

template<typename T>
inline std::vector<T> Matrix<T>::col_to_vector(unsigned col) const {
  std::vector<T> vec(rows());
  for (unsigned i = 0; i < rows(); i++)
    vec[i] = at(i,col);
  return vec;
}

template<typename T>
inline std::vector<T> Matrix<T>::row_to_vector(unsigned row) const {
  std::vector<T> vec(cols());
  for (unsigned i = 0; i < cols(); i++)
    vec[i] = at(row,i);
  return vec;
}

```

```
template<typename T>
inline void Matrix<T>::col_from_vector(const std::vector<T>& vec, unsigned col) {
  if (vec.size() != rows()) throw std::runtime_error("Vector size is not equal to number of rows");
  if (col >= cols()) throw std::out_of_range("Column index out of range");

  for (unsigned i = 0; i < rows(); i++)
    data[col*rows() + i] = vec[i];
}

template<typename T>
inline void Matrix<T>::row_from_vector(const std::vector<T>& vec, unsigned row) {
  if (vec.size() != rows()) throw std::runtime_error("Vector size is not equal to number of columns");
  if (row >= rows()) throw std::out_of_range("Row index out of range");

  for (unsigned i = 0; i < cols(); i++)
    data[row + i*rows()] = vec[i];
}

template<typename T>
Matrix<T>::~Matrix() { }

} // namespace Matrix_hpp

#endif // __MATRIX_HPP__
```