

Matrix HPP

Generated by Doxygen 1.9.8

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

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

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

6.2.1.39 lu()	53
6.2.1.40 lup()	53
6.2.1.41 make_complex() [1/2]	54
6.2.1.42 make_complex() [2/2]	54
6.2.1.43 mult() [1/4]	55
6.2.1.44 mult() [2/4]	56
6.2.1.45 mult() [3/4]	56
6.2.1.46 mult() [4/4]	57
6.2.1.47 mult_hadamard()	58
6.2.1.48 norm_fro() [1/2]	59
6.2.1.49 norm_fro() [2/2]	59
6.2.1.50 ones() [1/2]	59
6.2.1.51 ones() [2/2]	60
6.2.1.52 operator!==()	60
6.2.1.53 operator*() [1/5]	60
6.2.1.54 operator*() [2/5]	61
6.2.1.55 operator*() [3/5]	61
6.2.1.56 operator*() [4/5]	61
6.2.1.57 operator*() [5/5]	61
6.2.1.58 operator*==() [1/2]	62
6.2.1.59 operator*==() [2/2]	62
6.2.1.60 operator+() [1/3]	62
6.2.1.61 operator+() [2/3]	62
6.2.1.62 operator+() [3/3]	63
6.2.1.63 operator+==() [1/2]	63
6.2.1.64 operator+==() [2/2]	63
6.2.1.65 operator-() [1/2]	63
6.2.1.66 operator-() [2/2]	64
6.2.1.67 operator-==() [1/2]	64
6.2.1.68 operator-==() [2/2]	64
6.2.1.69 operator/()	64
6.2.1.70 operator/==()	65
6.2.1.71 operator<<()	65
6.2.1.72 operator==()	65
6.2.1.73 operator^()	65
6.2.1.74 operator^==()	66
6.2.1.75 permute_cols()	66
6.2.1.76 permute_rows()	66
6.2.1.77 pinv()	67
6.2.1.78 qr()	67
6.2.1.79 qr_householder()	68
6.2.1.80 qr_red_gs()	68

6.2.1.81 <code>real()</code>	69
6.2.1.82 <code>repmat()</code>	69
6.2.1.83 <code>solve_posdef()</code>	70
6.2.1.84 <code>solve_square()</code>	70
6.2.1.85 <code>solve_tril()</code>	71
6.2.1.86 <code>solve_triu()</code>	72
6.2.1.87 <code>subtract()</code> [1/2]	72
6.2.1.88 <code>subtract()</code> [2/2]	73
6.2.1.89 <code>trace()</code>	73
6.2.1.90 <code>transpose()</code>	74
6.2.1.91 <code>tril()</code>	74
6.2.1.92 <code>triu()</code>	74
6.2.1.93 <code>wilkinson_shift()</code>	74
6.2.1.94 <code>zeros()</code> [1/2]	75
6.2.1.95 <code>zeros()</code> [2/2]	75
6.3 <code>matrix.hpp</code>	76

Chapter 1

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

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

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

1.1 Installation

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

1.2 Functionality

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

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

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

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

1.3 Hello world example

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

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

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

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

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

    auto C = A + B;

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

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

1.4 Tests

Unit tests are compiled with `make tests`.

1.5 License

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

Chapter 2

Hierarchical Index

2.1 Class Hierarchy

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

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

Chapter 3

Class Index

3.1 Class List

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

Mtx::Eigenvalues_result< T >	
Result of eigenvalues	9
Mtx::Hessenberg_result< T >	
Result of Hessenberg decomposition	9
Mtx::LDL_result< T >	
Result of LDL decomposition	10
Mtx::LU_result< T >	
Result of LU decomposition	11
Mtx::LUP_result< T >	
Result of LU decomposition with pivoting	11
Mtx::Matrix< T >	12
Mtx::QR_result< T >	
Result of QR decomposition	29
Mtx::singular_matrix_exception	
Singular matrix exception	30

Chapter 4

File Index

4.1 File List

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

examples.cpp	31
matrix.hpp	31

Chapter 5

Class Documentation

5.1 Mtx::Eigenvalues_result< T > Struct Template Reference

Result of eigenvalues.

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

Public Attributes

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

5.1.1 Detailed Description

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

Result of eigenvalues.

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

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

- [matrix.hpp](#)

5.2 Mtx::Hessenberg_result< T > Struct Template Reference

Result of Hessenberg decomposition.

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

Public Attributes

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

5.2.1 Detailed Description

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

Result of Hessenberg decomposition.

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

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

- [matrix.hpp](#)

5.3 Mtx::LDL_result< T > Struct Template Reference

Result of LDL decomposition.

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

Public Attributes

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

5.3.1 Detailed Description

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

Result of LDL decomposition.

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

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

- [matrix.hpp](#)

5.4 Mtx::LU_result< T > Struct Template Reference

Result of LU decomposition.

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

Public Attributes

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

5.4.1 Detailed Description

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

Result of LU decomposition.

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

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

- [matrix.hpp](#)

5.5 Mtx::LUP_result< T > Struct Template Reference

Result of LU decomposition with pivoting.

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

Public Attributes

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

5.5.1 Detailed Description

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

Result of LU decomposition with pivoting.

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

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

- [matrix.hpp](#)

5.6 Mtx::Matrix< T > Class Template Reference

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

Public Member Functions

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

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

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

5.6.1 Detailed Description

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

[Matrix](#) class definition.

5.6.2 Constructor & Destructor Documentation

5.6.2.1 Matrix() [1/8]

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

Default constructor.

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

5.6.2.2 Matrix() [2/8]

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

Square matrix constructor.

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

5.6.2.3 Matrix() [3/8]

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

Rectangular matrix constructor.

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

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

5.6.2.4 Matrix() [4/8]

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

Rectangular matrix constructor with fill.

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

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

5.6.2.5 Matrix() [5/8]

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

Rectangular matrix constructor with initialization.

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

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

5.6.2.6 Matrix() [6/8]

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

Rectangular matrix constructor with initialization.

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

Exceptions

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

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

5.6.2.7 Matrix() [7/8]

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

Rectangular matrix constructor with initialization.

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

Exceptions

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

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

5.6.2.8 Matrix() [8/8]

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

Copy constructor.

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

5.6.2.9 ~Matrix()

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

Destructor.

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

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

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

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

Exceptions

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

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

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

5.6.3.2 add() [2/2]

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

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

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

5.6.3.3 add_col_to_another()

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

Column addition.

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

Exceptions

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

5.6.3.4 add_row_to_another()

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

Row addition.

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

Exceptions

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

5.6.3.5 clear()

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

Clears the matrix.

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

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

5.6.3.6 col_from_vector()

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

Column from vector.

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

Exceptions

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

5.6.3.7 col_to_vector()

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

Column to vector.

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

Exceptions

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

5.6.3.8 cols()

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

Number of columns.

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

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

5.6.3.9 ctranspose()

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

Transpose a complex matrix.

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

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

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

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

5.6.3.10 div()

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

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

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

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

5.6.3.11 exists()

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

Element exist check.

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

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

5.6.3.12 fill()

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

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

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

5.6.3.13 fill_col()

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

Fill column with a scalar.

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

Exceptions

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

5.6.3.14 fill_row()

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

Fill row with a scalar.

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

Exceptions

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

5.6.3.15 get_submatrix()

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

Extract a submatrix.

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

Exceptions

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

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

5.6.3.16 isempty()

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

Emptiness check.

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

5.6.3.17 isequal() [1/2]

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

[Matrix](#) equality check.

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

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

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

5.6.3.18 isequal() [2/2]

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

[Matrix](#) equality check with tolerance.

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

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

5.6.3.19 mult()

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

Matrix product with scalar (in-place).

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

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

5.6.3.20 mult_col_by_another()

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

Column multiplication.

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

Exceptions

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

5.6.3.21 mult_hadamard()

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

Matrix Hadamard product (in-place).

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

Exceptions

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

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

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

5.6.3.22 mult_row_by_another()

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

Row multiplication.

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

Exceptions

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

5.6.3.23 numel()

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

Matrix capacity.

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

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

5.6.3.24 operator std::vector< T >()

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

Vector cast operator.

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

5.6.3.25 operator()() [1/2]

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

Element access operator (1D)

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

Exceptions

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

5.6.3.26 operator() [2/2]

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

Element access operator (2D)

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

Exceptions

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

5.6.3.27 operator=() [1/2]

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

Matrix assignment.

Performs deep-copy of another matrix.

5.6.3.28 operator=() [2/2]

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

Matrix fill operator.

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

5.6.3.29 ptr() [1/2]

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

Memory pointer.

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

Exceptions

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

5.6.3.30 ptr() [2/2]

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

Memory pointer.

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

5.6.3.31 reshape()

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

[Matrix](#) dimension reshape.

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

Exceptions

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

5.6.3.32 resize()

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

Resize the matrix.

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

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

5.6.3.33 row_from_vector()

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

Row from vector.

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

Exceptions

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

5.6.3.34 row_to_vector()

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

Row to vector.

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

Exceptions

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

5.6.3.35 rows()

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

Number of rows.

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

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

5.6.3.36 set_submatrix()

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

Embed a submatrix.

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

Exceptions

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

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

5.6.3.37 subtract() [1/2]

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

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

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

Exceptions

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

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

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

5.6.3.38 subtract() [2/2]

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

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

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

5.6.3.39 swap_cols()

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

Column swap.

Swaps element values between two columns.

Exceptions

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

5.6.3.40 swap_rows()

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

Row swap.

Swaps element values of two columns.

Exceptions

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

5.6.3.41 transpose()

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

Transpose a matrix.

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

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

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

- [matrix.hpp](#)

5.7 Mtx::QR_result< T > Struct Template Reference

Result of QR decomposition.

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

Public Attributes

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

5.7.1 Detailed Description

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

Result of QR decomposition.

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

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

- [matrix.hpp](#)

5.8 Mtx::singular_matrix_exception Class Reference

Singular matrix exception.

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

Inheritance diagram for Mtx::singular_matrix_exception:

Chapter 6

File Documentation

6.1 examples.cpp File Reference

6.1.1 Detailed Description

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

6.2 matrix.hpp File Reference

Classes

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

Functions

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

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

Matrix subtraction.

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

Multiplication of matrix by std::vector.

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

Multiplication of std::vector by matrix.

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

Addition of scalar to matrix.

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

Subtraction of scalar from matrix.

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

Multiplication of matrix by scalar.

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

Division of matrix by scalar.

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

Matrix ostream operator.

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

Matrix sum.

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

Matrix subtraction.

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

Matrix Hadamard product.

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

Matrix product.

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

Matrix and std::vector product.

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

std::vector and matrix product.

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

Matrix sum with scalar.

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

Matrix subtraction with scalar.

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

Matrix product with scalar.

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

Matrix division by scalar.

- `template<typename T>`
`Matrix< T> Mtx::operator+ (T s, const Matrix< T> &A)`
- `template<typename T>`
`Matrix< T> Mtx::operator* (T s, const Matrix< T> &A)`
Matrix product with scalar.
- `template<typename T>`
`Matrix< T> & Mtx::operator+= (Matrix< T> &A, const Matrix< T> &B)`
Matrix sum.
- `template<typename T>`
`Matrix< T> & Mtx::operator-= (Matrix< T> &A, const Matrix< T> &B)`
Matrix subtraction.
- `template<typename T>`
`Matrix< T> & Mtx::operator*= (Matrix< T> &A, const Matrix< T> &B)`
Matrix product.
- `template<typename T>`
`Matrix< T> & Mtx::operator^= (Matrix< T> &A, const Matrix< T> &B)`
Matrix Hadamard product.
- `template<typename T>`
`Matrix< T> & Mtx::operator+= (Matrix< T> &A, T s)`
Matrix sum with scalar.
- `template<typename T>`
`Matrix< T> & Mtx::operator-= (Matrix< T> &A, T s)`
Matrix subtraction with scalar.
- `template<typename T>`
`Matrix< T> & Mtx::operator*= (Matrix< T> &A, T s)`
Matrix product with scalar.
- `template<typename T>`
`Matrix< T> & Mtx::operator/= (Matrix< T> &A, T s)`
Matrix division by scalar.
- `template<typename T>`
`bool Mtx::operator== (const Matrix< T> &A, const Matrix< T> &b)`
Matrix equality check operator.
- `template<typename T>`
`bool Mtx::operator!= (const Matrix< T> &A, const Matrix< T> &b)`
Matrix non-equality check operator.
- `template<typename T>`
`Matrix< T> Mtx::kron (const Matrix< T> &A, const Matrix< T> &B)`
Kronecker product.
- `template<typename T>`
`Matrix< T> Mtx::adj (const Matrix< T> &A)`
Adjugate matrix.
- `template<typename T>`
`Matrix< T> Mtx::cofactor (const Matrix< T> &A, unsigned p, unsigned q)`
Cofactor matrix.
- `template<typename T>`
`T Mtx::det_lu (const Matrix< T> &A)`
Matrix determinant from on LU decomposition.
- `template<typename T>`
`T Mtx::det (const Matrix< T> &A)`
Matrix determinant.
- `template<typename T>`
`LU_result< T> Mtx::lu (const Matrix< T> &A)`
LU decomposition.

- `template<typename T >`
`LUP_result< T > Mtx::lup (const Matrix< T > &A)`
LU decomposition with pivoting.
- `template<typename T >`
`Matrix< T > Mtx::inv_gauss_jordan (const Matrix< T > &A)`
Matrix inverse using Gauss-Jordan elimination.
- `template<typename T >`
`Matrix< T > Mtx::inv_tril (const Matrix< T > &A)`
Matrix inverse for lower triangular matrix.
- `template<typename T >`
`Matrix< T > Mtx::inv_triu (const Matrix< T > &A)`
Matrix inverse for upper triangular matrix.
- `template<typename T >`
`Matrix< T > Mtx::inv_posdef (const Matrix< T > &A)`
Matrix inverse for Hermitian positive-definite matrix.
- `template<typename T >`
`Matrix< T > Mtx::inv_square (const Matrix< T > &A)`
Matrix inverse for general square matrix.
- `template<typename T >`
`Matrix< T > Mtx::inv (const Matrix< T > &A)`
Matrix inverse (universal).
- `template<typename T >`
`Matrix< T > Mtx::pinv (const Matrix< T > &A)`
Moore-Penrose pseudoinverse.
- `template<typename T >`
`T Mtx::trace (const Matrix< T > &A)`
Matrix trace.
- `template<typename T >`
`double Mtx::cond (const Matrix< T > &A)`
Condition number of a matrix.
- `template<typename T >`
`Matrix< T > Mtx::chol (const Matrix< T > &A)`
Cholesky decomposition.
- `template<typename T >`
`Matrix< T > Mtx::cholin (const Matrix< T > &A)`
Inverse of Cholesky decomposition.
- `template<typename T >`
`LDL_result< T > Mtx::ldl (const Matrix< T > &A)`
LDL decomposition.
- `template<typename T >`
`QR_result< T > Mtx::qr_red_gs (const Matrix< T > &A)`
Reduced QR decomposition based on Gram-Schmidt method.
- `template<typename T >`
`Matrix< T > Mtx::householder_reflection (const Matrix< T > &a)`
Generate Householder reflection.
- `template<typename T >`
`QR_result< T > Mtx::qr_householder (const Matrix< T > &A, bool calculate_Q=true)`
QR decomposition based on Householder method.
- `template<typename T >`
`QR_result< T > Mtx::qr (const Matrix< T > &A, bool calculate_Q=true)`
QR decomposition.
- `template<typename T >`
`Hessenberg_result< T > Mtx::hessenberg (const Matrix< T > &A, bool calculate_Q=true)`

- Hessenberg decomposition.*

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

Wilkinson's shift for complex eigenvalues.
- ```
template<typename T>
Eigenvalues_result< T> Mtx::eigenvalues (const Matrix< std::complex< T> > &A, T tol=1e-12, unsigned
max_iter=100)
```

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

Matrix eigenvalues of real matrix.
- ```
template<typename T>
Matrix< T> Mtx::solve_triu (const Matrix< T> &U, const Matrix< T> &B)
```

*Solves the upper triangular system.*
- ```
template<typename T>
Matrix< T> Mtx::solve_tril (const Matrix< T> &L, const Matrix< T> &B)
```

Solves the lower triangular system.
- ```
template<typename T>
Matrix< T> Mtx::solve_square (const Matrix< T> &A, const Matrix< T> &B)
```

*Solves the square system.*
- ```
template<typename T>
Matrix< T> Mtx::solve_posdef (const Matrix< T> &A, const Matrix< T> &B)
```

Solves the positive definite (Hermitian) system.

6.2.1 Function Documentation

6.2.1.1 add() [1/2]

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

Matrix addition.

Performs addition of two matrices.

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

Template Parameters

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

Parameters

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

Returns

output matrix of size $N \times M$

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

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

6.2.1.2 add() [2/2]

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

Addition of scalar to matrix.

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

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

6.2.1.3 adj()

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

Adjugate matrix.

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

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

Exceptions

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

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

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

6.2.1.4 cconj()

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

Complex conjugate helper.

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

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

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

6.2.1.5 chol()

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

Cholesky decomposition.

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

$$A = LL^H$$

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

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

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

Exceptions

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

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

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

6.2.1.6 cholinv()

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

Inverse of Cholesky decomposition.

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

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

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

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

Exceptions

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

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

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

6.2.1.7 circshift()

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

Circular shift.

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

Parameters

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

Returns

matrix inverse

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

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

6.2.1.8 circulant() [1/2]

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

Circulant matrix from std::vector.

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

Parameters

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

Returns

circulant matrix

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

6.2.1.9 `circulant()` [2/2]

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

Circulant matrix from array.

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

Parameters

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

Returns

circulant matrix

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

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

6.2.1.10 `cofactor()`

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

Cofactor matrix.

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

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

Parameters

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

Exceptions

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

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

6.2.1.11 cond()

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

Condition number of a matrix.

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

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

Frobenius norm is used for the sake of calculations.

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

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

6.2.1.12 csign()

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

Complex sign helper.

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

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

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

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

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

6.2.1.13 ctranspose()

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

Transpose a complex matrix.

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

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

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

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

6.2.1.14 det()

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

Matrix determinant.

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

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

Exceptions

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

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

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

6.2.1.15 det_lu()

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

Matrix determinant from on LU decomposition.

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

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

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

Exceptions

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

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

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

6.2.1.16 diag() [1/3]

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

Diagonal extraction.

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

Parameters

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

Returns

vector of diagonal elements

Exceptions

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

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

6.2.1.17 diag() [2/3]

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

Diagonal matrix from `std::vector`.

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

Parameters

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

Returns

diagonal matrix

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

6.2.1.18 diag() [3/3]

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

Diagonal matrix from array.

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

Parameters

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

Returns

diagonal matrix

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

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

6.2.1.19 div()

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

Division of matrix by scalar.

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

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

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

6.2.1.20 eigenvalues() [1/2]

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

Matrix eigenvalues of complex matrix.

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

Parameters

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

Returns

structure containing the result and status of eigenvalue calculation

Exceptions

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

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

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

6.2.1.21 eigenvalues() [2/2]

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

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

Matrix eigenvalues of real matrix.

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

Parameters

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

Returns

structure containing the result and status of eigenvalue calculation

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

6.2.1.22 eye()

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

Identity matrix.

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

Parameters

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

Returns

zeros matrix

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

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

6.2.1.23 foreach_elem()

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

Applies custom function element-wise in-place.

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

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

Parameters

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

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

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

6.2.1.24 foreach_elem_copy()

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

Applies custom function element-wise with matrix copy.

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

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

Parameters

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

Returns

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

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

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

6.2.1.25 hessenberg()

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

Hessenberg decomposition.

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

Parameters

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

Returns

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

Exceptions

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

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

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

6.2.1.26 householder_reflection()

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

Generate Householder reflection.

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

Parameters

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

Returns

column vector with Householder reflection of *a*

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

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

6.2.1.27 imag()

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

Get imaginary part of complex matrix.

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

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

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

6.2.1.28 inv()

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

Matrix inverse (universal).

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

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

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

Exceptions

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

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

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

6.2.1.29 inv_gauss_jordan()

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

Matrix inverse using Gauss-Jordan elimination.

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

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

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

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

Exceptions

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

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

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

6.2.1.30 inv_posdef()

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

Matrix inverse for Hermitian positive-definite matrix.

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

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

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

Exceptions

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

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

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

6.2.1.31 inv_square()

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

Matrix inverse for general square matrix.

Calculates an inverse of square matrix using matrix.

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

Exceptions

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

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

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

6.2.1.32 inv_tril()

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

Matrix inverse for lower triangular matrix.

Calculates an inverse of lower triangular matrix.

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

Exceptions

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

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

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

6.2.1.33 inv_triu()

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

Matrix inverse for upper triangular matrix.

Calculates an inverse of upper triangular matrix.

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

Exceptions

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

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

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

6.2.1.34 ishess()

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

Hessenberg matrix check.

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

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

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

6.2.1.35 istril()

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

Lower triangular matrix check.

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

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

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

6.2.1.36 istriu()

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

Lower triangular matrix check.

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

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

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

6.2.1.37 kron()

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

Kronecker product.

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

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

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

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

6.2.1.38 ldl()

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

LDL decomposition.

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

$$A = LDL^H$$

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

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

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

Parameters

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

Returns

structure encapsulating calculated L and D

Exceptions

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

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

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

6.2.1.39 lu()

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

LU decomposition.

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

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

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

Parameters

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

Returns

structure containing calculated L and U matrices

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

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

6.2.1.40 lup()

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

LU decomposition with pivoting.

Performs LU factorization with partial pivoting, employing column permutations.

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

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

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

Parameters

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

Returns

structure containing L , U and P .

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

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

6.2.1.41 make_complex() [1/2]

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

Create complex matrix from real matrix.

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

Parameters

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

Returns

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

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

6.2.1.42 make_complex() [2/2]

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

Create complex matrix from real and imaginary matrices.

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

Parameters

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

Returns

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

Exceptions

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

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

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

6.2.1.43 mult() [1/4]

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

Matrix multiplication.

Performs multiplication of two matrices.

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

Template Parameters

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

Parameters

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

Returns

output matrix of size $N \times K$

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

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

6.2.1.44 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 the operation is also a std::vector.

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

Template Parameters

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

Parameters

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

Returns

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

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

6.2.1.45 mult() [3/4]

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

Multiplication of matrix by scalar.

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

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

6.2.1.46 mult() [4/4]

```
template<typename T , 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 the operation is also a std::vector.

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

Template Parameters

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

Parameters

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

Returns

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

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

6.2.1.47 mult_hadamard()

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

Matrix Hadamard (elementwise) multiplication.

Performs Hadamard (elementwise) multiplication of two matrices.

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

Template Parameters

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

Parameters

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

Returns

output matrix of size *N* x *M*

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

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

6.2.1.48 norm_fro() [1/2]

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

Frobenius norm of complex matrix.

Calculates Frobenius norm of complex matrix.

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

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

6.2.1.49 norm_fro() [2/2]

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

Frobenius norm.

Calculates Frobenius norm of real matrix.

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

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

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

6.2.1.50 ones() [1/2]

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

Square matrix of ones.

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

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

Parameters

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

Returns

zeros matrix

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

6.2.1.51 ones() [2/2]

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

Matrix of ones.

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

Parameters

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

Returns

ones matrix

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

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

6.2.1.52 operator!=(=)

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

Matrix non-equality check operator.

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

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

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

6.2.1.53 operator*() [1/5]

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

Matrix product.

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

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

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

6.2.1.54 operator*() [2/5]

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

Matrix and std::vector product.

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

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

6.2.1.55 operator*() [3/5]

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

Matrix product with scalar.

Multiplies each element of the matrix by a scalar s .

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

6.2.1.56 operator*() [4/5]

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

std::vector and matrix product.

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

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

6.2.1.57 operator*() [5/5]

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

Matrix product with scalar.

Multiplies each element of the matrix by a scalar s .

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

6.2.1.58 operator*=() [1/2]

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

Matrix product.

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

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

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

6.2.1.59 operator*=() [2/2]

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

Matrix product with scalar.

Multiplies each element of the matrix by a scalar s .

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

6.2.1.60 operator+() [1/3]

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

Matrix sum.

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

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

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

6.2.1.61 operator+() [2/3]

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

Matrix sum with scalar.

Adds a scalar s to each element of the matrix.

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

6.2.1.62 operator+() [3/3]

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

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

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

6.2.1.63 operator+=() [1/2]

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

Matrix sum.

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

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

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

6.2.1.64 operator+=() [2/2]

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

Matrix sum with scalar.

Adds a scalar s to each element of the matrix.

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

6.2.1.65 operator-() [1/2]

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

Matrix subtraction.

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

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

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

6.2.1.66 operator-() [2/2]

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

Matrix subtraction with scalar.

Subtracts a scalar s from each element of the matrix.

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

6.2.1.67 operator-=() [1/2]

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

Matrix subtraction.

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

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

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

6.2.1.68 operator-=() [2/2]

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

Matrix subtraction with scalar.

Subtracts a scalar s from each element of the matrix.

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

6.2.1.69 operator/()

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

Matrix division by scalar.

Divides each element of the matrix by a scalar s .

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

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

6.2.1.70 operator/=()

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

Matrix division by scalar.

Divides each element of the matrix by a scalar s .

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

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

6.2.1.71 operator<<()

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

Matrix ostream operator.

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

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

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

6.2.1.72 operator==()

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

Matrix equality check operator.

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

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

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

6.2.1.73 operator^()

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

Matrix Hadamard product.

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

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

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

6.2.1.74 operator^=()

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

Matrix Hadamard product.

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

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

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

6.2.1.75 permute_cols()

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

Permute columns of the matrix.

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

Parameters

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

Returns

output matrix created by column permutation of A

Exceptions

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

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

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

6.2.1.76 permute_rows()

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

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

Permute rows of the matrix.

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

Parameters

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

Returns

output matrix created by row permutation of *A*

Exceptions

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

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

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

6.2.1.77 pinv()

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

Moore-Penrose pseudoinverse.

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

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

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

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

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

6.2.1.78 qr()

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

QR decomposition.

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

Currently, this function is a wrapper around `qr_householder()`. Refer to `qr_red_gs()` for alternative implementation.

Parameters

A	input matrix to be decomposed
$\text{calculate_}Q$	indicates if Q to be calculated

Returns

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

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

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

6.2.1.79 qr_householder()

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

QR decomposition based on Householder method.

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

This function implements QR decomposition based on Householder reflections method.

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

Parameters

A	input matrix to be decomposed, size $n \times m$
$\text{calculate_}Q$	indicates if Q to be calculated

Returns

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

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

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

6.2.1.80 qr_red_gs()

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

Reduced QR decomposition based on Gram-Schmidt method.

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

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

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

Parameters

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

Returns

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

Exceptions

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

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

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

6.2.1.81 real()

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

Get real part of complex matrix.

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

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

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

6.2.1.82 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

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

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

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

6.2.1.83 solve_posdef()

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

Solves the positive definite (Hermitian) system.

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

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

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

Parameters

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

Returns

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

Exceptions

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

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

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

6.2.1.84 solve_square()

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

```
const Matrix< T > & A,
const Matrix< T > & B )
```

Solves the square system.

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

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

Parameters

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

Returns

solution matrix of size $N \times M$.

Exceptions

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

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

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

6.2.1.85 solve_tril()

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

Solves the lower triangular system.

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

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

Parameters

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

Returns

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

Exceptions

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

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

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

6.2.1.86 solve_triu()

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

Solves the upper triangular system.

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

Parameters

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

Returns

solution matrix of size $N \times M$.

Exceptions

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

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

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

6.2.1.87 subtract() [1/2]

```
template<typename T , bool transpose_first = false, bool transpose_second = false>
```



```
Matrix< T > Mtx::subtract (
    const Matrix< T > & A,
    const Matrix< T > & B )
```

Matrix subtraction.

Performs subtraction of two matrices.

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

Template Parameters

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

Parameters

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

Returns

output matrix of size $N \times M$

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

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

6.2.1.88 subtract() [2/2]

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

Subtraction of scalar from matrix.

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

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

6.2.1.89 trace()

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

Matrix trace.

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

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

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

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

6.2.1.90 transpose()

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

Transpose a matrix.

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

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

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

6.2.1.91 tril()

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

Extract triangular lower part.

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

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

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

6.2.1.92 triu()

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

Extract triangular upper part.

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

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

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

6.2.1.93 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

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

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

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

6.2.1.94 zeros() [1/2]

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

Square matrix of zeros.

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

Parameters

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

Returns

zeros matrix

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

6.2.1.95 zeros() [2/2]

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

Matrix of zeros.

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

Parameters

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

Returns

zeros matrix

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

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

6.3 matrix.hpp

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

```

00001
00002
00003 /* MIT License
00004 *
00005 * Copyright (c) 2024 gc1905
00006 *
00007 * Permission is hereby granted, free of charge, to any person obtaining a copy
00008 * of this software and associated documentation files (the "Software"), to deal
00009 * in the Software without restriction, including without limitation the rights
00010 * to use, copy, modify, merge, publish, distribute, sublicense, and/or sell
00011 * copies of the Software, and to permit persons to whom the Software is
00012 * furnished to do so, subject to the following conditions:
00013 *
00014 * The above copyright notice and this permission notice shall be included in all
00015 * copies or substantial portions of the Software.
00016 *
00017 * THE SOFTWARE IS PROVIDED "AS IS", WITHOUT WARRANTY OF ANY KIND, EXPRESS OR
00018 * IMPLIED, INCLUDING BUT NOT LIMITED TO THE WARRANTIES OF MERCHANTABILITY,
00019 * FITNESS FOR A PARTICULAR PURPOSE AND NONINFRINGEMENT. IN NO EVENT SHALL THE
00020 * AUTHORS OR COPYRIGHT HOLDERS BE LIABLE FOR ANY CLAIM, DAMAGES OR OTHER
00021 * LIABILITY, WHETHER IN AN ACTION OF CONTRACT, TORT OR OTHERWISE, ARISING FROM,
00022 * OUT OF OR IN CONNECTION WITH THE SOFTWARE OR THE USE OR OTHER DEALINGS IN THE
00023 * SOFTWARE.
00024 */
00025
00026 #ifndef __MATRIX_HPP__
00027 #define __MATRIX_HPP__
00028
00029 #include <ostream>
00030 #include <complex>
00031 #include <vector>
00032 #include <initializer_list>
00033 #include <limits>
00034 #include <functional>
00035 #include <algorithm>
00036
00037 namespace Mtx {
00038
00039 template<typename T> class Matrix;
00040
00041 template<class T> struct is_complex : std::false_type {};
00042 template<class T> struct is_complex<std::complex<T> > : std::true_type {};
00043
00044 template<typename T, typename std::enable_if<!is_complex<T>::value, int>::type = 0>
00045 inline T cconj(T x) {
00046     return x;
00047 }
00048
00049 template<typename T, typename std::enable_if<is_complex<T>::value, int>::type = 0>
00050 inline T cconj(T x) {
00051     return std::conj(x);
00052 }
00053
00054 template<typename T, typename std::enable_if<!is_complex<T>::value, int>::type = 0>
00055 inline T csign(T x) {
00056     return (x > static_cast<T>(0)) ? static_cast<T>(1) : static_cast<T>(-1);
00057 }
00058
00059 template<typename T, typename std::enable_if<is_complex<T>::value, int>::type = 0>
00060 inline T csign(T x) {
00061     auto x_arg = std::arg(x);
00062     T y(0, x_arg);
00063     return std::exp(y);
00064 }
00065
00066 class singular_matrix_exception : public std::domain_error {
00067 public:
00068     singular_matrix_exception(const std::string& message) : std::domain_error(message) {}
00069 };
00070
00071 template<typename T>
00072 struct LU_result {
00073     Matrix<T> L;
00074     Matrix<T> U;
00075 };
00076
00077 template<typename T>
00078 struct LUP_result {
00079     Matrix<T> L;
00080     Matrix<T> U;
00081 
```

```

00118
00121     std::vector<unsigned> P;
00122 };
00123
00129 template<typename T>
00130 struct QR_result {
00133     Matrix<T> Q;
00134
00137     Matrix<T> R;
00138 };
00139
00144 template<typename T>
00145 struct Hessenberg_result {
00148     Matrix<T> H;
00149
00152     Matrix<T> Q;
00153 };
00154
00159 template<typename T>
00160 struct LDL_result {
00163     Matrix<T> L;
00164
00167     std::vector<T> d;
00168 };
00169
00174 template<typename T>
00175 struct Eigenvalues_result {
00178     std::vector<std::complex<T>> eig;
00179
00182     bool converged;
00183
00186     T err;
00187 };
00188
00189
00197 template<typename T>
00198 inline Matrix<T> zeros(unsigned nrows, unsigned ncols) {
00199     return Matrix<T>(static_cast<T>(0), nrows, ncols);
00200 }
00201
00208 template<typename T>
00209 inline Matrix<T> zeros(unsigned n) {
00210     return zeros<T>(n,n);
00211 }
00212
00221 template<typename T>
00222 inline Matrix<T> ones(unsigned nrows, unsigned ncols) {
00223     return Matrix<T>(static_cast<T>(1), nrows, ncols);
00224 }
00225
00233 template<typename T>
00234 inline Matrix<T> ones(unsigned n) {
00235     return ones<T>(n,n);
00236 }
00237
00245 template<typename T>
00246 Matrix<T> eye(unsigned n) {
00247     Matrix<T> A(static_cast<T>(0), n, n);
00248     for (unsigned i = 0; i < n; i++)
00249         A(i,i) = static_cast<T>(1);
00250     return A;
00251 }
00252
00260 template<typename T>
00261 Matrix<T> diag(const T* array, size_t n) {
00262     Matrix<T> A(static_cast<T>(0), n, n);
00263     for (unsigned i = 0; i < n; i++) {
00264         A(i,i) = array[i];
00265     }
00266     return A;
00267 }
00268
00276 template<typename T>
00277 inline Matrix<T> diag(const std::vector<T>& v) {
00278     return diag(v.data(), v.size());
00279 }
00280
00289 template<typename T>
00290 std::vector<T> diag(const Matrix<T>& A) {
00291     if (! A.issquare()) throw std::runtime_error("Input matrix is not square");
00292
00293     std::vector<T> v;
00294     v.resize(A.rows());
00295
00296     for (unsigned i = 0; i < A.rows(); i++)
00297         v[i] = A(i,i);
00298     return v;

```

```

00299 }
00300
00308 template<typename T>
00309 Matrix<T> circulant(const T* array, unsigned n) {
00310     Matrix<T> A(n, n);
00311     for (unsigned j = 0; j < n; j++)
00312         for (unsigned i = 0; i < n; i++)
00313             A((i+j) % n, j) = array[i];
00314     return A;
00315 }
00316
00327 template<typename T>
00328 Matrix<std::complex<T>> make_complex(const Matrix<T>& Re, const Matrix<T>& Im) {
00329     if (Re.rows() != Im.rows() || Re.cols() != Im.cols()) throw std::runtime_error("Size of input
matrices does not match");
00330
00331     Matrix<std::complex<T>> C(Re.rows(), Re.cols());
00332     for (unsigned n = 0; n < Re.numel(); n++) {
00333         C(n).real(Re(n));
00334         C(n).imag(Im(n));
00335     }
00336
00337     return C;
00338 }
00339
00346 template<typename T>
00347 Matrix<std::complex<T>> make_complex(const Matrix<T>& Re) {
00348     Matrix<std::complex<T>> C(Re.rows(), Re.cols());
00349
00350     for (unsigned n = 0; n < Re.numel(); n++) {
00351         C(n).real(Re(n));
00352         C(n).imag(static_cast<T>(0));
00353     }
00354
00355     return C;
00356 }
00357
00362 template<typename T>
00363 Matrix<T> real(const Matrix<std::complex<T>& C) {
00364     Matrix<T> Re(C.rows(), C.cols());
00365
00366     for (unsigned n = 0; n < C.numel(); n++)
00367         Re(n) = C(n).real();
00368
00369     return Re;
00370 }
00371
00376 template<typename T>
00377 Matrix<T> imag(const Matrix<std::complex<T>& C) {
00378     Matrix<T> Re(C.rows(), C.cols());
00379
00380     for (unsigned n = 0; n < C.numel(); n++)
00381         Re(n) = C(n).imag();
00382
00383     return Re;
00384 }
00385
00393 template<typename T>
00394 inline Matrix<T> circulant(const std::vector<T>& v) {
00395     return circulant(v.data(), v.size());
00396 }
00397
00402 template<typename T>
00403 inline Matrix<T> transpose(const Matrix<T>& A) {
00404     return A.transpose();
00405 }
00406
00412 template<typename T>
00413 inline Matrix<T> ctranspose(const Matrix<T>& A) {
00414     return A.ctranspose();
00415 }
00416
00427 template<typename T>
00428 Matrix<T> circshift(const Matrix<T>& A, int row_shift, int col_shift) {
00429     Matrix<T> B(A.rows(), A.cols());
00430     for (int i = 0; i < A.rows(); i++) {
00431         int ii = (i + row_shift) % A.rows();
00432         for (int j = 0; j < A.cols(); j++) {
00433             int jj = (j + col_shift) % A.cols();
00434             B(ii, jj) = A(i, j);
00435         }
00436     }
00437     return B;
00438 }
00439
00447 template<typename T>
00448 Matrix<T> repmat(const Matrix<T>& A, unsigned m, unsigned n) {

```

```

00449     Matrix<T> B(m * A.rows(), n * A.cols());
00450
00451     for (unsigned cb = 0; cb < n; cb++)
00452         for (unsigned rb = 0; rb < m; rb++)
00453             for (unsigned c = 0; c < A.cols(); c++)
00454                 for (unsigned r = 0; r < A.rows(); r++)
00455                     B(rb*A.rows() + r, cb*A.cols() + c) = A(r, c);
00456
00457     return B;
00458 }
00459
00460 template<typename T>
00461 double norm_fro(const Matrix<T>& A) {
00462     double sum = 0;
00463
00464     for (unsigned i = 0; i < A.numel(); i++)
00465         sum += A(i) * A(i);
00466
00467     return std::sqrt(sum);
00468 }
00469
00470 template<typename T>
00471 double norm_fro(const Matrix<std::complex<T> >& A) {
00472     double sum = 0;
00473
00474     for (unsigned i = 0; i < A.numel(); i++) {
00475         T x = std::abs(A(i));
00476         sum += x * x;
00477     }
00478
00479     return std::sqrt(sum);
00480 }
00481
00482 template<typename T>
00483 Matrix<T> tril(const Matrix<T>& A) {
00484     Matrix<T> B(A);
00485
00486     for (unsigned row = 0; row < B.rows(); row++)
00487         for (unsigned col = row+1; col < B.cols(); col++)
00488             B(row,col) = 0;
00489
00490     return B;
00491 }
00492
00493 template<typename T>
00494 Matrix<T> triu(const Matrix<T>& A) {
00495     Matrix<T> B(A);
00496
00497     for (unsigned col = 0; col < B.cols(); col++)
00498         for (unsigned row = col+1; row < B.rows(); row++)
00499             B(row,col) = 0;
00500
00501     return B;
00502 }
00503
00504 template<typename T>
00505 bool istril(const Matrix<T>& A) {
00506     for (unsigned row = 0; row < A.rows(); row++)
00507         for (unsigned col = row+1; col < A.cols(); col++)
00508             if (A(row,col) != static_cast<T>(0)) return false;
00509     return true;
00510 }
00511
00512 template<typename T>
00513 bool istriu(const Matrix<T>& A) {
00514     for (unsigned col = 0; col < A.cols(); col++)
00515         for (unsigned row = col+1; row < A.rows(); row++)
00516             if (A(row,col) != static_cast<T>(0)) return false;
00517     return true;
00518 }
00519
00520 template<typename T>
00521 bool ishess(const Matrix<T>& A) {
00522     if (!A.issquare())
00523         return false;
00524     for (unsigned row = 2; row < A.rows(); row++)
00525         for (unsigned col = 0; col < row-2; col++)
00526             if (A(row,col) != static_cast<T>(0)) return false;
00527     return true;
00528 }
00529
00530 template<typename T>
00531 inline void foreach_elem(Matrix<T>& A, std::function<T(T)> func) {
00532     for (unsigned i = 0; i < A.numel(); i++)
00533         A(i) = func(A(i));
00534 }
00535
00536

```

```

00585 template<typename T>
00586 inline Matrix<T> foreach_elem_copy(const Matrix<T>& A, std::function<T(T)> func) {
00587     Matrix<T> B(A);
00588     foreach_elem(B, func);
00589     return B;
00590 }
00591
00604 template<typename T>
00605 Matrix<T> permute_rows(const Matrix<T>& A, const std::vector<unsigned> perm) {
00606     if (perm.empty()) throw std::runtime_error("Permutation vector is empty");
00607
00608     Matrix<T> B(perm.size(), A.cols());
00609
00610     for (unsigned p = 0; p < perm.size(); p++) {
00611         if (!perm[p] < A.rows()) throw std::out_of_range("Index in permutation vector out of range");
00612
00613         for (unsigned c = 0; c < A.cols(); c++)
00614             B(p,c) = A(perm[p],c);
00615     }
00616
00617     return B;
00618 }
00619
00632 template<typename T>
00633 Matrix<T> permute_cols(const Matrix<T>& A, const std::vector<unsigned> perm) {
00634     if (perm.empty()) throw std::runtime_error("Permutation vector is empty");
00635
00636     Matrix<T> B(A.rows(), perm.size());
00637
00638     for (unsigned p = 0; p < perm.size(); p++) {
00639         if (!perm[p] < A.cols()) throw std::out_of_range("Index in permutation vector out of range");
00640
00641         for (unsigned r = 0; r < A.rows(); r++)
00642             B(r,p) = A(r,perm[p]);
00643     }
00644
00645     return B;
00646 }
00647
00662 template<typename T, bool transpose_first = false, bool transpose_second = false>
00663 Matrix<T> mult(const Matrix<T>& A, const Matrix<T>& B) {
00664     // Adjust dimensions based on transpositions
00665     unsigned rows_A = transpose_first ? A.cols() : A.rows();
00666     unsigned cols_A = transpose_first ? A.rows() : A.cols();
00667     unsigned rows_B = transpose_second ? B.cols() : B.rows();
00668     unsigned cols_B = transpose_second ? B.rows() : B.cols();
00669
00670     if (cols_A != rows_B) throw std::runtime_error("Unmatching matrix dimensions for mult");
00671
00672     Matrix<T> C(static_cast<T>(0), rows_A, cols_B);
00673
00674     for (unsigned i = 0; i < rows_A; i++)
00675         for (unsigned j = 0; j < cols_B; j++)
00676             for (unsigned k = 0; k < cols_A; k++)
00677                 C(i,j) += (transpose_first ? cconj(A(k,i)) : A(i,k)) *
00678                     (transpose_second ? cconj(B(j,k)) : B(k,j));
00679
00680     return C;
00681 }
00682
00697 template<typename T, bool transpose_first = false, bool transpose_second = false>
00698 Matrix<T> mult_hadamard(const Matrix<T>& A, const Matrix<T>& B) {
00699     // Adjust dimensions based on transpositions
00700     unsigned rows_A = transpose_first ? A.cols() : A.rows();
00701     unsigned cols_A = transpose_first ? A.rows() : A.cols();
00702     unsigned rows_B = transpose_second ? B.cols() : B.rows();
00703     unsigned cols_B = transpose_second ? B.rows() : B.cols();
00704
00705     if ((rows_A != rows_B) || (cols_A != cols_B)) throw std::runtime_error("Unmatching matrix dimensions
for mult_hadamard");
00706
00707     Matrix<T> C(static_cast<T>(0), rows_A, cols_A);
00708
00709     for (unsigned i = 0; i < rows_A; i++)
00710         for (unsigned j = 0; j < cols_A; j++)
00711             C(i,j) += (transpose_first ? cconj(A(j,i)) : A(i,j)) *
00712                 (transpose_second ? cconj(B(j,i)) : B(i,j));
00713
00714     return C;
00715 }
00716
00731 template<typename T, bool transpose_first = false, bool transpose_second = false>
00732 Matrix<T> add(const Matrix<T>& A, const Matrix<T>& B) {
00733     // Adjust dimensions based on transpositions
00734     unsigned rows_A = transpose_first ? A.cols() : A.rows();
00735     unsigned cols_A = transpose_first ? A.rows() : A.cols();
00736     unsigned rows_B = transpose_second ? B.cols() : B.rows();

```



```

00737     unsigned cols_B = transpose_second ? B.rows() : B.cols();
00738
00739     if ((rows_A != rows_B) || (cols_A != cols_B)) throw std::runtime_error("Unmatching matrix dimensions
for add");
00740
00741     Matrix<T> C(static_cast<T>(0), rows_A, cols_A);
00742
00743     for (unsigned i = 0; i < rows_A; i++)
00744         for (unsigned j = 0; j < cols_A; j++)
00745             C(i,j) += (transpose_first ? cconj(A(j,i)) : A(i,j)) +
00746                       (transpose_second ? cconj(B(j,i)) : B(i,j));
00747
00748     return C;
00749 }
00750
00765 template<typename T, bool transpose_first = false, bool transpose_second = false>
00766 Matrix<T> subtract(const Matrix<T>& A, const Matrix<T>& B) {
00767     // Adjust dimensions based on transpositions
00768     unsigned rows_A = transpose_first ? A.cols() : A.rows();
00769     unsigned cols_A = transpose_first ? A.rows() : A.cols();
00770     unsigned rows_B = transpose_second ? B.cols() : B.rows();
00771     unsigned cols_B = transpose_second ? B.rows() : B.cols();
00772
00773     if ((rows_A != rows_B) || (cols_A != cols_B)) throw std::runtime_error("Unmatching matrix dimensions
for subtract");
00774
00775     Matrix<T> C(static_cast<T>(0), rows_A, cols_A);
00776
00777     for (unsigned i = 0; i < rows_A; i++)
00778         for (unsigned j = 0; j < cols_A; j++)
00779             C(i,j) += (transpose_first ? cconj(A(j,i)) : A(i,j)) -
00780                       (transpose_second ? cconj(B(j,i)) : B(i,j));
00781
00782     return C;
00783 }
00784
00798 template<typename T, bool transpose_matrix = false>
00799 std::vector<T> mult(const Matrix<T>& A, const std::vector<T>& v) {
00800     // Adjust dimensions based on transpositions
00801     unsigned rows_A = transpose_matrix ? A.cols() : A.rows();
00802     unsigned cols_A = transpose_matrix ? A.rows() : A.cols();
00803
00804     if (cols_A != v.size()) throw std::runtime_error("Unmatching matrix dimensions for mult");
00805
00806     std::vector<T> u(rows_A, static_cast<T>(0));
00807     for (unsigned r = 0; r < rows_A; r++)
00808         for (unsigned c = 0; c < cols_A; c++)
00809             u[r] += v[c] * (transpose_matrix ? cconj(A(c,r)) : A(r,c));
00810
00811     return u;
00812 }
00813
00827 template<typename T, bool transpose_matrix = false>
00828 std::vector<T> mult(const std::vector<T>& v, const Matrix<T>& A) {
00829     // Adjust dimensions based on transpositions
00830     unsigned rows_A = transpose_matrix ? A.cols() : A.rows();
00831     unsigned cols_A = transpose_matrix ? A.rows() : A.cols();
00832
00833     if (rows_A != v.size()) throw std::runtime_error("Unmatching matrix dimensions for mult");
00834
00835     std::vector<T> u(cols_A, static_cast<T>(0));
00836     for (unsigned c = 0; c < cols_A; c++)
00837         for (unsigned r = 0; r < rows_A; r++)
00838             u[c] += v[r] * (transpose_matrix ? cconj(A(c,r)) : A(r,c));
00839
00840     return u;
00841 }
00842
00848 template<typename T>
00849 Matrix<T> add(const Matrix<T>& A, T s) {
00850     Matrix<T> B(A.rows(), A.cols());
00851     for (unsigned i = 0; i < A.numel(); i++)
00852         B(i) = A(i) + s;
00853     return B;
00854 }
00855
00861 template<typename T>
00862 Matrix<T> subtract(const Matrix<T>& A, T s) {
00863     Matrix<T> B(A.rows(), A.cols());
00864     for (unsigned i = 0; i < A.numel(); i++)
00865         B(i) = A(i) - s;
00866     return B;
00867 }
00868
00874 template<typename T>
00875 Matrix<T> mult(const Matrix<T>& A, T s) {
00876     Matrix<T> B(A.rows(), A.cols());

```

```

00877     for (unsigned i = 0; i < A.numel(); i++)
00878         B(i) = A(i) * s;
00879     return B;
00880 }
00881
00882 template<typename T>
00883 Matrix<T> div(const Matrix<T>& A, T s) {
00884     Matrix<T> B(A.rows(), A.cols());
00885     for (unsigned i = 0; i < A.numel(); i++)
00886         B(i) = A(i) / s;
00887     return B;
00888 }
00889
00890 template<typename T>
00891 std::ostream& operator<<(std::ostream& os, const Matrix<T>& A) {
00892     for (unsigned row = 0; row < A.rows(); row++) {
00893         for (unsigned col = 0; col < A.cols(); col++)
00894             os << A(row,col) << " ";
00895         if (row < static_cast<unsigned>(A.rows()-1)) os << std::endl;
00896     }
00897     return os;
00898 }
00899
00900 template<typename T>
00901 inline Matrix<T> operator+(const Matrix<T>& A, const Matrix<T>& B) {
00902     return add(A,B);
00903 }
00904
00905 template<typename T>
00906 inline Matrix<T> operator-(const Matrix<T>& A, const Matrix<T>& B) {
00907     return subtract(A,B);
00908 }
00909
00910 template<typename T>
00911 inline Matrix<T> operator^(const Matrix<T>& A, const Matrix<T>& B) {
00912     return mult_hadamard(A,B);
00913 }
00914
00915 template<typename T>
00916 inline Matrix<T> operator*(const Matrix<T>& A, const Matrix<T>& B) {
00917     return mult(A,B);
00918 }
00919
00920 template<typename T>
00921 inline std::vector<T> operator*(const Matrix<T>& A, const std::vector<T>& v) {
00922     return mult(A,v);
00923 }
00924
00925 template<typename T>
00926 inline std::vector<T> operator*(const std::vector<T>& v, const Matrix<T>& A) {
00927     return mult(v,A);
00928 }
00929
00930 template<typename T>
00931 inline Matrix<T> operator+(const Matrix<T>& A, T s) {
00932     return add(A,s);
00933 }
00934
00935 template<typename T>
00936 inline Matrix<T> operator-(const Matrix<T>& A, T s) {
00937     return subtract(A,s);
00938 }
00939
00940 template<typename T>
00941 inline Matrix<T> operator*(const Matrix<T>& A, T s) {
00942     return mult(A,s);
00943 }
00944
00945 template<typename T>
00946 inline Matrix<T> operator/(const Matrix<T>& A, T s) {
00947     return div(A,s);
00948 }
00949
00950 template<typename T>
00951 inline Matrix<T> operator+(T s, const Matrix<T>& A) {
00952     return add(A,s);
00953 }
00954
00955 template<typename T>
00956 inline Matrix<T> operator*(T s, const Matrix<T>& A) {
00957     return mult(A,s);
00958 }
00959
00960 template<typename T>
00961 inline Matrix<T>& operator+=(Matrix<T>& A, const Matrix<T>& B) {
00962     return A.add(B);
00963 }
00964

```

```

01026
01031 template<typename T>
01032 inline Matrix<T>& operator==(Matrix<T>& A, const Matrix<T>& B) {
01033     return A.subtract(B);
01034 }
01035
01040 template<typename T>
01041 inline Matrix<T>& operator*=(Matrix<T>& A, const Matrix<T>& B) {
01042     A = mult(A,B);
01043     return A;
01044 }
01045
01051 template<typename T>
01052 inline Matrix<T>& operator^=(Matrix<T>& A, const Matrix<T>& B) {
01053     return A.mult_hadamard(B);
01054 }
01055
01060 template<typename T>
01061 inline Matrix<T>& operator+=(Matrix<T>& A, T s) {
01062     return A.add(s);
01063 }
01064
01069 template<typename T>
01070 inline Matrix<T>& operator-=(Matrix<T>& A, T s) {
01071     return A.subtract(s);
01072 }
01073
01078 template<typename T>
01079 inline Matrix<T>& operator*=(Matrix<T>& A, T s) {
01080     return A.mult(s);
01081 }
01082
01087 template<typename T>
01088 inline Matrix<T>& operator/=(Matrix<T>& A, T s) {
01089     return A.div(s);
01090 }
01091
01096 template<typename T>
01097 inline bool operator==(const Matrix<T>& A, const Matrix<T>& b) {
01098     return A.isequal(b);
01099 }
01100
01105 template<typename T>
01106 inline bool operator!=(const Matrix<T>& A, const Matrix<T>& b) {
01107     return !(A.isequal(b));
01108 }
01109
01115 template<typename T>
01116 Matrix<T> kron(const Matrix<T>& A, const Matrix<T>& B) {
01117     const unsigned rows_A = A.rows();
01118     const unsigned cols_A = A.cols();
01119     const unsigned rows_B = B.rows();
01120     const unsigned cols_B = B.cols();
01121
01122     unsigned rows_C = rows_A * rows_B;
01123     unsigned cols_C = cols_A * cols_B;
01124
01125     Matrix<T> C(rows_C, cols_C);
01126
01127     for (unsigned i = 0; i < rows_A; i++)
01128         for (unsigned j = 0; j < cols_A; j++)
01129             for (unsigned k = 0; k < rows_B; k++)
01130                 for (unsigned l = 0; l < cols_B; l++)
01131                     C(i*rows_B + k, j*cols_B + l) = A(i, j) * B(k, l);
01132
01133     return C;
01134 }
01135
01143 template<typename T>
01144 Matrix<T> adj(const Matrix<T>& A) {
01145     if (!A.issquare()) throw std::runtime_error("Input matrix is not square");
01146
01147     Matrix<T> B(A.rows(), A.cols());
01148     if (A.rows() == 1) {
01149         B(0) = 1.0;
01150     } else {
01151         for (unsigned i = 0; i < A.rows(); i++) {
01152             for (unsigned j = 0; j < A.cols(); j++) {
01153                 T sgn = ((i + j) % 2 == 0) ? 1.0 : -1.0;
01154                 B(j, i) = sgn * det(cofactor(A, i, j));
01155             }
01156         }
01157     }
01158     return B;
01159 }
01160
01173 template<typename T>

```

```

01174 Matrix<T> cofactor(const Matrix<T>& A, unsigned p, unsigned q) {
01175     if (! A.issquare()) throw std::runtime_error("Input matrix is not square");
01176     if (!(p < A.rows())) throw std::out_of_range("Row index out of range");
01177     if (!(q < A.cols())) throw std::out_of_range("Column index out of range");
01178     if (A.cols() < 2) throw std::runtime_error("Cofactor calculation requested for matrix with less than
2 rows");
01179
01180     Matrix<T> c(A.rows()-1,A.cols()-1);
01181     unsigned i = 0;
01182     unsigned j = 0;
01183
01184     for (unsigned row = 0; row < A.rows(); row++) {
01185         if (row != p) {
01186             for (unsigned col = 0; col < A.cols(); col++)
01187                 if (col != q) c(i,j++) = A(row,col);
01188             j = 0;
01189             i++;
01190         }
01191     }
01192
01193     return c;
01194 }
01195
01207 template<typename T>
01208 T det_lu(const Matrix<T>& A) {
01209     if (! A.issquare()) throw std::runtime_error("Input matrix is not square");
01210
01211     // LU decomposition with pivoting
01212     auto res = lup(A);
01213
01214     // Determinants of LU
01215     T detLU = static_cast<T>(1);
01216
01217     for (unsigned i = 0; i < res.L.rows(); i++)
01218         detLU *= res.L(i,i) * res.U(i,i);
01219
01220     // Determinant of P
01221     unsigned len = res.P.size();
01222     T detP = 1;
01223
01224     std::vector<unsigned> p(res.P);
01225     std::vector<unsigned> q;
01226     q.resize(len);
01227
01228     for (unsigned i = 0; i < len; i++)
01229         q[p[i]] = i;
01230
01231     for (unsigned i = 0; i < len; i++) {
01232         unsigned j = p[i];
01233         unsigned k = q[i];
01234         if (j != i) {
01235             p[k] = p[i];
01236             q[j] = q[i];
01237             detP = - detP;
01238         }
01239     }
01240
01241     return detLU * detP;
01242 }
01243
01252 template<typename T>
01253 T det(const Matrix<T>& A) {
01254     if (! A.issquare()) throw std::runtime_error("Input matrix is not square");
01255
01256     if (A.rows() == 1)
01257         return A(0,0);
01258     else if (A.rows() == 2)
01259         return A(0,0)*A(1,1) - A(0,1)*A(1,0);
01260     else if (A.rows() == 3)
01261         return A(0,0)*(A(1,1)*A(2,2) - A(1,2)*A(2,1)) -
01262             A(0,1)*(A(1,0)*A(2,2) - A(1,2)*A(2,0)) +
01263             A(0,2)*(A(1,0)*A(2,1) - A(1,1)*A(2,0));
01264     else
01265         return det_lu(A);
01266 }
01267
01276 template<typename T>
01277 LU_result<T> lu(const Matrix<T>& A) {
01278     const unsigned M = A.rows();
01279     const unsigned N = A.cols();
01280
01281     LU_result<T> res;
01282     res.L = eye<T>(M);
01283     res.U = Matrix<T>(A);
01284
01285     // aliases
01286     auto& L = res.L;

```

```

01287     auto& U = res.U;
01288
01289     if (A.numel() == 0)
01290         return res;
01291
01292     for (unsigned k = 0; k < M-1; k++) {
01293         for (unsigned i = k+1; i < M; i++) {
01294             L(i,k) = U(i,k) / U(k,k);
01295             for (unsigned l = k+1; l < N; l++) {
01296                 U(i,l) -= L(i,k) * U(k,l);
01297             }
01298         }
01299     }
01300
01301     for (unsigned col = 0; col < N; col++)
01302         for (unsigned row = col+1; row < M; row++)
01303             U(row,col) = 0;
01304
01305     return res;
01306 }
01307
01321 template<typename T>
01322 LUP_result<T> lup(const Matrix<T>& A) {
01323     const unsigned M = A.rows();
01324     const unsigned N = A.cols();
01325
01326     // Initialize L, U, and PP
01327     LUP_result<T> res;
01328
01329     if (A.numel() == 0)
01330         return res;
01331
01332     res.L = eye<T>(M);
01333     res.U = Matrix<T>(A);
01334     std::vector<unsigned> PP;
01335
01336     // aliases
01337     auto& L = res.L;
01338     auto& U = res.U;
01339
01340     PP.resize(N);
01341     for (unsigned i = 0; i < N; i++)
01342         PP[i] = i;
01343
01344     for (unsigned k = 0; k < M-1; k++) {
01345         // Find the column with the largest absolute value in the current row
01346         auto max_col_value = std::abs(U(k,k));
01347         unsigned max_col_index = k;
01348         for (unsigned l = k+1; l < N; l++) {
01349             auto val = std::abs(U(k,l));
01350             if (val > max_col_value) {
01351                 max_col_value = val;
01352                 max_col_index = l;
01353             }
01354         }
01355
01356         // Swap columns k and max_col_index in U and update P
01357         if (max_col_index != k) {
01358             U.swap_cols(k, max_col_index); // TODO: This could be reworked to avoid column swap in U during
every iteration by:
01359                                     // 1. using PP[k] for column indexing across iterations
01360                                     // 2. doing just one permutation of U at the end
01361             std::swap(PP[k], PP[max_col_index]);
01362         }
01363
01364         // Update L and U
01365         for (unsigned i = k+1; i < M; i++) {
01366             L(i,k) = U(i,k) / U(k,k);
01367             for (unsigned l = k+1; l < N; l++) {
01368                 U(i,l) -= L(i,k) * U(k,l);
01369             }
01370         }
01371     }
01372
01373     // Set elements in lower triangular part of U to zero
01374     for (unsigned col = 0; col < N; col++)
01375         for (unsigned row = col+1; row < M; row++)
01376             U(row,col) = 0;
01377
01378     // Transpose indices in permutation vector
01379     res.P.resize(N);
01380     for (unsigned i = 0; i < N; i++)
01381         res.P[PP[i]] = i;
01382
01383     return res;
01384 }
01385

```

```

01396 template<typename T>
01397 Matrix<T> inv_gauss_jordan(const Matrix<T>& A) {
01398     if (! A.issquare()) throw std::runtime_error("Input matrix is not square");
01399
01400     const unsigned N = A.rows();
01401     Matrix<T> AA(A);
01402     auto IA = eye<T>(N);
01403
01404     bool found_nonzero;
01405     for (unsigned j = 0; j < N; j++) {
01406         found_nonzero = false;
01407         for (unsigned i = j; i < N; i++) {
01408             if (AA(i,j) != static_cast<T>(0)) {
01409                 found_nonzero = true;
01410                 for (unsigned k = 0; k < N; k++) {
01411                     std::swap(AA(j,k), AA(i,k));
01412                     std::swap(IA(j,k), IA(i,k));
01413                 }
01414                 if (AA(j,j) != static_cast<T>(1)) {
01415                     T s = static_cast<T>(1) / AA(j,j);
01416                     for (unsigned k = 0; k < N; k++) {
01417                         AA(j,k) *= s;
01418                         IA(j,k) *= s;
01419                     }
01420                 }
01421                 for (unsigned l = 0; l < N; l++) {
01422                     if (l != j) {
01423                         T s = AA(l,j);
01424                         for (unsigned k = 0; k < N; k++) {
01425                             AA(l,k) -= s * AA(j,k);
01426                             IA(l,k) -= s * IA(j,k);
01427                         }
01428                     }
01429                 }
01430             }
01431             break;
01432         }
01433         // if a row full of zeros is found, the input matrix was singular
01434         if (!found_nonzero) throw singular_matrix_exception("Singular matrix in inv_gauss_jordan");
01435     }
01436     return IA;
01437 }
01438
01449 template<typename T>
01450 Matrix<T> inv_tril(const Matrix<T>& A) {
01451     if (! A.issquare()) throw std::runtime_error("Input matrix is not square");
01452
01453     const unsigned N = A.rows();
01454
01455     auto IA = zeros<T>(N);
01456
01457     for (unsigned i = 0; i < N; i++) {
01458         if (A(i,i) == 0.0) throw singular_matrix_exception("Division by zero in inv_tril");
01459
01460         IA(i,i) = static_cast<T>(1.0) / A(i,i);
01461         for (unsigned j = 0; j < i; j++) {
01462             T s = 0.0;
01463             for (unsigned k = j; k < i; k++)
01464                 s += A(i,k) * IA(k,j);
01465             IA(i,j) = -s * IA(i,i);
01466         }
01467     }
01468
01469     return IA;
01470 }
01471
01482 template<typename T>
01483 Matrix<T> inv_triu(const Matrix<T>& A) {
01484     if (!A.issquare()) throw std::runtime_error("Input matrix is not square");
01485
01486     const unsigned N = A.rows();
01487
01488     auto IA = zeros<T>(N);
01489
01490     for (int i = N - 1; i >= 0; i--) {
01491         if (A(i,i) == 0.0) throw singular_matrix_exception("Division by zero in inv_triu");
01492
01493         IA(i, i) = static_cast<T>(1.0) / A(i,i);
01494         for (int j = N - 1; j > i; j--) {
01495             T s = 0.0;
01496             for (int k = i + 1; k <= j; k++)
01497                 s += A(i,k) * IA(k,j);
01498             IA(i,j) = -s * IA(i,i);
01499         }
01500     }
01501
01502     return IA;

```

```

01503 }
01504
01517 template<typename T>
01518 Matrix<T> inv_posdef(const Matrix<T>& A) {
01519     auto L = cholinv(A);
01520     return mult<T,true,false>(L,L);
01521 }
01522
01533 template<typename T>
01534 Matrix<T> inv_square(const Matrix<T>& A) {
01535     if (! A.issquare()) throw std::runtime_error("Input matrix is not square");
01536
01537     // LU decomposition with pivoting
01538     auto LU = lup(A);
01539     auto IL = inv_tril(LU.L);
01540     auto IU = inv_triu(LU.U);
01541
01542     return permute_rows(IU * IL, LU.P);
01543 }
01544
01555 template<typename T>
01556 Matrix<T> inv(const Matrix<T>& A) {
01557     if (! A.issquare()) throw std::runtime_error("Input matrix is not square");
01558
01559     if (A.numel() == 0) {
01560         return Matrix<T>();
01561     } else if (A.rows() < 4) {
01562         T d = det(A);
01563
01564         if (d == 0.0) throw singular_matrix_exception("Singular matrix in inv");
01565
01566         Matrix<T> IA(A.rows(), A.rows());
01567         T invdet = static_cast<T>(1.0) / d;
01568
01569         if (A.rows() == 1) {
01570             IA(0,0) = invdet;
01571         } else if (A.rows() == 2) {
01572             IA(0,0) = A(1,1) * invdet;
01573             IA(0,1) = - A(0,1) * invdet;
01574             IA(1,0) = - A(1,0) * invdet;
01575             IA(1,1) = A(0,0) * invdet;
01576         } else if (A.rows() == 3) {
01577             IA(0,0) = (A(1,1)*A(2,2) - A(2,1)*A(1,2)) * invdet;
01578             IA(0,1) = (A(0,2)*A(2,1) - A(0,1)*A(2,2)) * invdet;
01579             IA(0,2) = (A(0,1)*A(1,2) - A(0,2)*A(1,1)) * invdet;
01580             IA(1,0) = (A(1,2)*A(2,0) - A(1,0)*A(2,2)) * invdet;
01581             IA(1,1) = (A(0,0)*A(2,2) - A(0,2)*A(2,0)) * invdet;
01582             IA(1,2) = (A(1,0)*A(0,2) - A(0,0)*A(1,2)) * invdet;
01583             IA(2,0) = (A(1,0)*A(2,1) - A(2,0)*A(1,1)) * invdet;
01584             IA(2,1) = (A(2,0)*A(0,1) - A(0,0)*A(2,1)) * invdet;
01585             IA(2,2) = (A(0,0)*A(1,1) - A(1,0)*A(0,1)) * invdet;
01586         }
01587
01588         return IA;
01589     } else {
01590         return inv_square(A);
01591     }
01592 }
01593
01601 template<typename T>
01602 Matrix<T> pinv(const Matrix<T>& A) {
01603     if (A.rows() > A.cols()) {
01604         auto AH_A = mult<T,true,false>(A, A);
01605         auto Linv = inv_posdef(AH_A);
01606         return mult<T,false,true>(Linv, A);
01607     } else {
01608         auto AA_H = mult<T,false,true>(A, A);
01609         auto Linv = inv_posdef(AA_H);
01610         return mult<T,true,false>(A, Linv);
01611     }
01612 }
01613
01619 template<typename T>
01620 T trace(const Matrix<T>& A) {
01621     T t = static_cast<T>(0);
01622     for (int i = 0; i < A.rows(); i++)
01623         t += A(i,i);
01624     return t;
01625 }
01626
01634 template<typename T>
01635 double cond(const Matrix<T>& A) {
01636     try {
01637         auto A_inv = inv(A);
01638         return norm_fro(A) * norm_fro(A_inv);
01639     } catch (singular_matrix_exception& e) {
01640         return std::numeric_limits<double>::max();
01641     }
01642 }

```

```

01641     }
01642 }
01643
01655 template<typename T>
01656 Matrix<T> chol(const Matrix<T>& A) {
01657     if (! A.issquare()) throw std::runtime_error("Input matrix is not square");
01658
01659     const unsigned N = A.rows();
01660     Matrix<T> L = tril(A);
01661
01662     for (unsigned j = 0; j < N; j++) {
01663         if (L(j,j) == 0.0) throw singular_matrix_exception("Singular matrix in chol");
01664
01665         L(j,j) = std::sqrt(L(j,j));
01666
01667         for (unsigned k = j+1; k < N; k++)
01668             L(k,j) = L(k,j) / L(j,j);
01669
01670         for (unsigned k = j+1; k < N; k++)
01671             for (unsigned i = k; i < N; i++)
01672                 L(i,k) = L(i,k) - L(i,j) * cconj(L(k,j));
01673     }
01674
01675     return L;
01676 }
01677
01688 template<typename T>
01689 Matrix<T> cholinv(const Matrix<T>& A) {
01690     if (! A.issquare()) throw std::runtime_error("Input matrix is not square");
01691
01692     const unsigned N = A.rows();
01693     Matrix<T> L(A);
01694     auto Linv = eye<T>(N);
01695
01696     for (unsigned j = 0; j < N; j++) {
01697         if (L(j,j) == 0.0) throw singular_matrix_exception("Singular matrix in cholinv");
01698
01699         L(j,j) = 1.0 / std::sqrt(L(j,j));
01700
01701         for (unsigned k = j+1; k < N; k++)
01702             L(k,j) = L(k,j) * L(j,j);
01703
01704         for (unsigned k = j+1; k < N; k++)
01705             for (unsigned i = k; i < N; i++)
01706                 L(i,k) = L(i,k) - L(i,j) * cconj(L(k,j));
01707     }
01708
01709     for (unsigned k = 0; k < N; k++) {
01710         for (unsigned i = k; i < N; i++) {
01711             Linv(i,k) = Linv(i,k) * L(i,i);
01712             for (unsigned j = i+1; j < N; j++)
01713                 Linv(j,k) = Linv(j,k) - L(j,i) * Linv(i,k);
01714         }
01715     }
01716
01717     return Linv;
01718 }
01719
01734 template<typename T>
01735 LDL_result<T> ldl(const Matrix<T>& A) {
01736     if (! A.issquare()) throw std::runtime_error("Input matrix is not square");
01737
01738     const unsigned N = A.rows();
01739
01740     LDL_result<T> res;
01741
01742     // aliases
01743     auto& L = res.L;
01744     auto& d = res.d;
01745
01746     L = eye<T>(N);
01747     d.resize(N);
01748
01749     for (unsigned m = 0; m < N; m++) {
01750         d[m] = A(m,m);
01751
01752         for (unsigned k = 0; k < m; k++)
01753             d[m] -= L(m,k) * cconj(L(m,k)) * d[k];
01754
01755         if (d[m] == 0.0) throw singular_matrix_exception("Singular matrix in ldl");
01756
01757         for (unsigned n = m+1; n < N; n++) {
01758             L(n,m) = A(n,m);
01759             for (unsigned k = 0; k < m; k++)
01760                 L(n,m) -= L(n,k) * cconj(L(m,k)) * d[k];
01761             L(n,m) /= d[m];
01762         }
01763     }

```



```

01763     }
01764
01765     return res;
01766 }
01767
01779 template<typename T>
01780 QR_result<T> qr_red_gs(const Matrix<T>& A) {
01781     const int rows = A.rows();
01782     const int cols = A.cols();
01783
01784     QR_result<T> res;
01785
01786     //aliases
01787     auto& Q = res.Q;
01788     auto& R = res.R;
01789
01790     Q = zeros<T>(rows, cols);
01791     R = zeros<T>(cols, cols);
01792
01793     for (int c = 0; c < cols; c++) {
01794         Matrix<T> v = A.get_submatrix(0, rows-1, c, c);
01795         for (int r = 0; r < c; r++) {
01796             for (int k = 0; k < rows; k++)
01797                 R(r,c) = R(r,c) + cconj(Q(k,r)) * A(k,c);
01798             for (int k = 0; k < rows; k++)
01799                 v(k) = v(k) - R(r,c) * Q(k,r);
01800         }
01801
01802         R(c,c) = static_cast<T>(norm_fro(v));
01803
01804         if (R(c,c) == 0.0) throw singular_matrix_exception("Division by 0 in QR GS");
01805
01806         for (int k = 0; k < rows; k++)
01807             Q(k,c) = v(k) / R(c,c);
01808     }
01809
01810     return res;
01811 }
01812
01820 template<typename T>
01821 Matrix<T> householder_reflection(const Matrix<T>& a) {
01822     if (a.cols() != 1) throw std::runtime_error("Input not a column vector");
01823
01824     static const T ISQRT2 = static_cast<T>(0.707106781186547);
01825
01826     Matrix<T> v(a);
01827     v(0) += csign(v(0)) * norm_fro(v);
01828     auto vn = norm_fro(v) * ISQRT2;
01829     for (unsigned i = 0; i < v.numel(); i++)
01830         v(i) /= vn;
01831     return v;
01832 }
01833
01845 template<typename T>
01846 QR_result<T> qr_householder(const Matrix<T>& A, bool calculate_Q = true) {
01847     const unsigned rows = A.rows();
01848     const unsigned cols = A.cols();
01849
01850     QR_result<T> res;
01851
01852     //aliases
01853     auto& Q = res.Q;
01854     auto& R = res.R;
01855
01856     R = Matrix<T>(A);
01857
01858     if (calculate_Q)
01859         Q = eye<T>(rows);
01860
01861     const unsigned N = (rows > cols) ? cols : rows;
01862
01863     for (unsigned j = 0; j < N; j++) {
01864         auto v = householder_reflection(R.get_submatrix(j, rows-1, j, j));
01865
01866         auto R1 = R.get_submatrix(j, rows-1, j, cols-1);
01867         auto WR = v * mult<T,true,false>(v, R1);
01868         for (unsigned c = j; c < cols; c++)
01869             for (unsigned r = j; r < rows; r++)
01870                 R(r,c) -= WR(r-j,c-j);
01871
01872         if (calculate_Q) {
01873             auto Q1 = Q.get_submatrix(0, rows-1, j, rows-1);
01874             auto WQ = mult<T,false,true>(Q1 * v, v);
01875             for (unsigned c = j; c < rows; c++)
01876                 for (unsigned r = 0; r < rows; r++)
01877                     Q(r,c) -= WQ(r,c-j);
01878         }
01879     }

```

```

01879     }
01880
01881     for (unsigned col = 0; col < R.cols(); col++)
01882         for (unsigned row = col+1; row < R.rows(); row++)
01883             R(row,col) = 0;
01884
01885     return res;
01886 }
01887
01888 template<typename T>
01889 inline QR_result<T> qr(const Matrix<T>& A, bool calculate_Q = true) {
01900     return qr_householder(A, calculate_Q);
01901 }
01902
01913 template<typename T>
01914 Hessenberg_result<T> hessenberg(const Matrix<T>& A, bool calculate_Q = true) {
01915     if (! A.issquare()) throw std::runtime_error("Input matrix is not square");
01916
01917     Hessenberg_result<T> res;
01918
01919     // aliases
01920     auto& H = res.H;
01921     auto& Q = res.Q;
01922
01923     const unsigned N = A.rows();
01924     H = Matrix<T>(A);
01925
01926     if (calculate_Q)
01927         Q = eye<T>(N);
01928
01929     for (unsigned k = 1; k < N-1; k++) {
01930         auto v = householder_reflection(H.get_submatrix(k, N-1, k-1, k-1));
01931
01932         auto H1 = H.get_submatrix(k, N-1, 0, N-1);
01933         auto W1 = v * mult<T,true,false>(v, H1);
01934         for (unsigned c = 0; c < N; c++)
01935             for (unsigned r = k; r < N; r++)
01936                 H(r,c) -= W1(r-k,c);
01937
01938         auto H2 = H.get_submatrix(0, N-1, k, N-1);
01939         auto W2 = mult<T,false,true>(H2 * v, v);
01940         for (unsigned c = k; c < N; c++)
01941             for (unsigned r = 0; r < N; r++)
01942                 H(r,c) -= W2(r,c-k);
01943
01944         if (calculate_Q) {
01945             auto Q1 = Q.get_submatrix(0, N-1, k, N-1);
01946             auto W3 = mult<T,false,true>(Q1 * v, v);
01947             for (unsigned c = k; c < N; c++)
01948                 for (unsigned r = 0; r < N; r++)
01949                     Q(r,c) -= W3(r,c-k);
01950         }
01951     }
01952
01953     for (unsigned row = 2; row < N; row++)
01954         for (unsigned col = 0; col < row-2; col++)
01955             H(row,col) = static_cast<T>(0);
01956
01957     return res;
01958 }
01959
01960 template<typename T>
01961 std::complex<T> wilkinson_shift(const Matrix<std::complex<T>& H, T tol = 1e-10) {
01970     if (! H.issquare()) throw std::runtime_error("Input matrix is not square");
01971
01972     const unsigned n = H.rows();
01973     std::complex<T> mu;
01974
01975     if (std::abs(H(n-1,n-2)) < tol) {
01976         mu = H(n-2,n-2);
01977     } else {
01978         auto trA = H(n-2,n-2) + H(n-1,n-1);
01979         auto detA = H(n-2,n-2) * H(n-1,n-1) - H(n-2, n-1) * H(n-1, n-2);
01980         mu = (trA + std::sqrt(trA*trA - 4.0*detA)) / 2.0;
01981     }
01982
01983     return mu;
01984 }
01985
01997 template<typename T>
01998 Eigenvalues_result<T> eigenvalues(const Matrix<std::complex<T>& A, T tol = 1e-12, unsigned max_iter =
100) {
01999     if (! A.issquare()) throw std::runtime_error("Input matrix is not square");
02000
02001     const unsigned N = A.rows();
02002     Matrix<std::complex<T>> H;
02003     bool success = false;

```

```

02004
02005     QR_result<std::complex<T>> QR;
02006
02007     // aliases
02008     auto& Q = QR.Q;
02009     auto& R = QR.R;
02010
02011     // Transfer A to Hessenberg form to improve convergence (skip calculation of Q)
02012     H = hessenberg(A, false).H;
02013
02014     for (unsigned iter = 0; iter < max_iter; iter++) {
02015         auto mu = wilkinson_shift(H, tol);
02016
02017         // subtract mu from diagonal
02018         for (unsigned n = 0; n < N; n++)
02019             H(n,n) -= mu;
02020
02021         // QR factorization with shifted H
02022         QR = qr(H);
02023         H = R * Q;
02024
02025         // add back mu to diagonal
02026         for (unsigned n = 0; n < N; n++)
02027             H(n,n) += mu;
02028
02029         // Check for convergence
02030         if (std::abs(H(N-2,N-1)) <= tol) {
02031             success = true;
02032             break;
02033         }
02034     }
02035
02036     Eigenvalues_result<T> res;
02037     res.eig = diag(H);
02038     res.err = std::abs(H(N-2,N-1));
02039     res.converged = success;
02040
02041     return res;
02042 }
02043
02053 template<typename T>
02054 Eigenvalues_result<T> eigenvalues(const Matrix<T>& A, T tol = 1e-12, unsigned max_iter = 100) {
02055     auto A_cplx = make_complex(A);
02056     return eigenvalues(A_cplx, tol, max_iter);
02057 }
02058
02073 template<typename T>
02074 Matrix<T> solve_triu(const Matrix<T>& U, const Matrix<T>& B) {
02075     if (! U.issquare()) throw std::runtime_error("Input matrix is not square");
02076     if (U.rows() != B.rows()) throw std::runtime_error("Unmatching matrix dimensions for solve");
02077
02078     const unsigned N = U.rows();
02079     const unsigned M = B.cols();
02080
02081     if (U.numel() == 0)
02082         return Matrix<T>();
02083
02084     Matrix<T> X(B);
02085
02086     for (unsigned m = 0; m < M; m++) {
02087         // backwards substitution for each column of B
02088         for (int n = N-1; n >= 0; n--) {
02089             for (unsigned j = n + 1; j < N; j++)
02090                 X(n,m) -= U(n,j) * X(j,m);
02091
02092             if (U(n,n) == 0.0) throw singular_matrix_exception("Singular matrix in solve_triu");
02093
02094             X(n,m) /= U(n,n);
02095         }
02096     }
02097
02098     return X;
02099 }
02100
02115 template<typename T>
02116 Matrix<T> solve_tril(const Matrix<T>& L, const Matrix<T>& B) {
02117     if (! L.issquare()) throw std::runtime_error("Input matrix is not square");
02118     if (L.rows() != B.rows()) throw std::runtime_error("Unmatching matrix dimensions for solve");
02119
02120     const unsigned N = L.rows();
02121     const unsigned M = B.cols();
02122
02123     if (L.numel() == 0)
02124         return Matrix<T>();
02125
02126     Matrix<T> X(B);
02127

```

```

02128     for (unsigned m = 0; m < M; m++) {
02129         // forwards substitution for each column of B
02130         for (unsigned n = 0; n < N; n++) {
02131             for (unsigned j = 0; j < n; j++)
02132                 X(n,m) -= L(n,j) * X(j,m);
02133
02134             if (L(n,n) == 0.0) throw singular_matrix_exception("Singular matrix in solve_tril");
02135
02136             X(n,m) /= L(n,n);
02137         }
02138     }
02139
02140     return X;
02141 }
02142
02157 template<typename T>
02158 Matrix<T> solve_square(const Matrix<T>& A, const Matrix<T>& B) {
02159     if (! A.issquare()) throw std::runtime_error("Input matrix is not square");
02160     if (A.rows() != B.rows()) throw std::runtime_error("Unmatching matrix dimensions for solve");
02161
02162     if (A.numel() == 0)
02163         return Matrix<T>();
02164
02165     Matrix<T> L;
02166     Matrix<T> U;
02167     std::vector<unsigned> P;
02168
02169     // LU decomposition with pivoting
02170     auto lup_res = lup(A);
02171
02172     auto y = solve_tril(lup_res.L, B);
02173     auto x = solve_triu(lup_res.U, y);
02174
02175     return permute_rows(x, lup_res.P);
02176 }
02177
02192 template<typename T>
02193 Matrix<T> solve_posdef(const Matrix<T>& A, const Matrix<T>& B) {
02194     if (! A.issquare()) throw std::runtime_error("Input matrix is not square");
02195     if (A.rows() != B.rows()) throw std::runtime_error("Unmatching matrix dimensions for solve");
02196
02197     if (A.numel() == 0)
02198         return Matrix<T>();
02199
02200     // LU decomposition with pivoting
02201     auto L = chol(A);
02202
02203     auto Y = solve_tril(L, B);
02204     return solve_triu(L.ctranspose(), Y);
02205 }
02206
02211 template<typename T>
02212 class Matrix {
02213 public:
02214     Matrix();
02215
02216     Matrix(unsigned size);
02217
02218     Matrix(unsigned nrow, unsigned ncol);
02219
02220     Matrix(T x, unsigned nrow, unsigned ncol);
02221
02222     Matrix(const T* array, unsigned nrow, unsigned ncol);
02223
02224     Matrix(const std::vector<T>& vec, unsigned nrow, unsigned ncol);
02225
02226     Matrix(std::initializer_list<T> init_list, unsigned nrow, unsigned ncol);
02227
02228     Matrix(const Matrix &);
02229
02230     virtual ~Matrix();
02231
02232     Matrix<T> get_submatrix(unsigned row_first, unsigned row_last, unsigned col_first, unsigned
02233     col_last) const;
02234
02235     void set_submatrix(const Matrix<T>& smtx, unsigned row_first, unsigned col_first);
02236
02237     void clear();
02238
02239     void reshape(unsigned rows, unsigned cols);
02240
02241     void resize(unsigned rows, unsigned cols);
02242
02243     bool exists(unsigned row, unsigned col) const;
02244
02245     T* ptr(unsigned row, unsigned col);
02246 }

```

```

02331     T* ptr();
02332
02336     void fill(T value);
02337
02344     void fill_col(T value, unsigned col);
02345
02352     void fill_row(T value, unsigned row);
02353
02358     bool isempty() const;
02359
02363     bool issquare() const;
02364
02369     bool isequal(const Matrix<T>&) const;
02370
02376     bool isequal(const Matrix<T>&, T) const;
02377
02382     unsigned numel() const;
02383
02388     unsigned rows() const;
02389
02394     unsigned cols() const;
02395
02400     Matrix<T> transpose() const;
02401
02407     Matrix<T> ctranspose() const;
02408
02416     Matrix<T>& add(const Matrix<T>&);
02417
02425     Matrix<T>& subtract(const Matrix<T>&);
02426
02435     Matrix<T>& mult_hadamard(const Matrix<T>&);
02436
02442     Matrix<T>& add(T);
02443
02449     Matrix<T>& subtract(T);
02450
02456     Matrix<T>& mult(T);
02457
02463     Matrix<T>& div(T);
02464
02469     Matrix<T>& operator=(const Matrix<T>&);
02470
02475     Matrix<T>& operator=(T);
02476
02481     explicit operator std::vector<T>() const;
02482     std::vector<T> to_vector() const;
02483
02490     T& operator()(unsigned nel);
02491     T operator()(unsigned nel) const;
02492     T& at(unsigned nel);
02493     T at(unsigned nel) const;
02494
02501     T& operator()(unsigned row, unsigned col);
02502     T operator()(unsigned row, unsigned col) const;
02503     T& at(unsigned row, unsigned col);
02504     T at(unsigned row, unsigned col) const;
02505
02513     void add_row_to_another(unsigned to, unsigned from);
02514
02522     void add_col_to_another(unsigned to, unsigned from);
02523
02531     void mult_row_by_another(unsigned to, unsigned from);
02532
02540     void mult_col_by_another(unsigned to, unsigned from);
02541
02548     void swap_rows(unsigned i, unsigned j);
02549
02556     void swap_cols(unsigned i, unsigned j);
02557
02564     std::vector<T> col_to_vector(unsigned col) const;
02565
02572     std::vector<T> row_to_vector(unsigned row) const;
02573
02581     void col_from_vector(const std::vector<T>&, unsigned col);
02582
02590     void row_from_vector(const std::vector<T>&, unsigned row);
02591
02592 private:
02593     unsigned nrows;
02594     unsigned ncols;
02595     std::vector<T> data;
02596 };
02597
02598 /*
02599  * Implementation of Matrix class methods
02600  */
02601

```

```

02602 template<typename T>
02603 Matrix<T>::Matrix() : nrows(0), ncols(0), data() { }
02604
02605 template<typename T>
02606 Matrix<T>::Matrix(unsigned size) : Matrix(size, size) { }
02607
02608 template<typename T>
02609 Matrix<T>::Matrix(unsigned rows, unsigned cols) : nrows(rows), ncols(cols) {
02610     data.resize(numel());
02611 }
02612
02613 template<typename T>
02614 Matrix<T>::Matrix(T x, unsigned rows, unsigned cols) : Matrix(rows, cols) {
02615     fill(x);
02616 }
02617
02618 template<typename T>
02619 Matrix<T>::Matrix(const T* array, unsigned rows, unsigned cols) : Matrix(rows, cols) {
02620     data.assign(array, array + numel());
02621 }
02622
02623 template<typename T>
02624 Matrix<T>::Matrix(const std::vector<T>& vec, unsigned rows, unsigned cols) : Matrix(rows, cols) {
02625     if (vec.size() != numel()) throw std::runtime_error("Size of initialization vector not consistent
02626         with matrix dimensions");
02627     data.assign(vec.begin(), vec.end());
02628 }
02629
02630 template<typename T>
02631 Matrix<T>::Matrix(std::initializer_list<T> init_list, unsigned rows, unsigned cols) : Matrix(rows,
02632     cols) {
02633     if (init_list.size() != numel()) throw std::runtime_error("Size of initialization list not
02634         consistent with matrix dimensions");
02635     auto it = init_list.begin();
02636     for (unsigned row = 0; row < this->nrows; row++)
02637         for (unsigned col = 0; col < this->ncols; col++)
02638             this->at(row,col) = *(it++);
02639 }
02640
02641 template<typename T>
02642 Matrix<T>::Matrix(const Matrix & other) : Matrix(other.nrows, other.ncols) {
02643     this->data.assign(other.data.begin(), other.data.end());
02644 }
02645
02646 template<typename T>
02647 Matrix<T>& Matrix<T>::operator=(const Matrix<T>& other) {
02648     this->nrows = other.nrows;
02649     this->ncols = other.ncols;
02650     this->data.assign(other.data.begin(), other.data.end());
02651     return *this;
02652 }
02653
02654 template<typename T>
02655 Matrix<T>& Matrix<T>::operator=(T s) {
02656     fill(s);
02657     return *this;
02658 }
02659
02660 template<typename T>
02661 inline Matrix<T>::operator std::vector<T>() const {
02662     return data;
02663 }
02664
02665 template<typename T>
02666 inline void Matrix<T>::clear() {
02667     this->nrows = 0;
02668     this->ncols = 0;
02669     data.resize(0);
02670 }
02671
02672 template<typename T>
02673 void Matrix<T>::reshape(unsigned rows, unsigned cols) {
02674     if (this->numel() != rows * cols) throw std::runtime_error("Illegal attempt to change number of
02675         elements via reshape");
02676     this->nrows = rows;
02677     this->ncols = cols;
02678 }
02679
02680 template<typename T>
02681 void Matrix<T>::resize(unsigned rows, unsigned cols) {
02682     this->nrows = rows;
02683     this->ncols = cols;
02684     data.resize(nrows*ncols);

```

```

02685 }
02686
02687 template<typename T>
02688 Matrix<T> Matrix<T>::get_submatrix(unsigned row_base, unsigned row_lim, unsigned col_base, unsigned
col_lim) const {
02689     if (row_base > row_lim) throw std::out_of_range("Row index of submatrix out of range");
02690     if (col_base > col_lim) throw std::out_of_range("Column index of submatrix out of range");
02691     if (row_lim >= this->rows()) throw std::out_of_range("Row index of submatrix out of range");
02692     if (col_lim >= this->cols()) throw std::out_of_range("Column index of submatrix out of range");
02693
02694     unsigned num_rows = row_lim - row_base + 1;
02695     unsigned num_cols = col_lim - col_base + 1;
02696     Matrix<T> S(num_rows, num_cols);
02697     for (unsigned i = 0; i < num_rows; i++) {
02698         for (unsigned j = 0; j < num_cols; j++) {
02699             S(i,j) = at(row_base + i, col_base + j);
02700         }
02701     }
02702     return S;
02703 }
02704
02705 template<typename T>
02706 void Matrix<T>::set_submatrix(const Matrix<T>& S, unsigned row_base, unsigned col_base) {
02707     if (this->isempty()) throw std::runtime_error("Invalid attempt to set submatrix in empty matrix");
02708
02709     const unsigned row_lim = row_base + S.rows() - 1;
02710     const unsigned col_lim = col_base + S.cols() - 1;
02711
02712     if (row_base > row_lim) throw std::out_of_range("Row index of submatrix out of range");
02713     if (col_base > col_lim) throw std::out_of_range("Column index of submatrix out of range");
02714     if (row_lim >= this->rows()) throw std::out_of_range("Row index of submatrix out of range");
02715     if (col_lim >= this->cols()) throw std::out_of_range("Column index of submatrix out of range");
02716
02717     unsigned num_rows = row_lim - row_base + 1;
02718     unsigned num_cols = col_lim - col_base + 1;
02719     for (unsigned i = 0; i < num_rows; i++)
02720         for (unsigned j = 0; j < num_cols; j++)
02721             at(row_base + i, col_base + j) = S(i,j);
02722 }
02723
02724 template<typename T>
02725 inline T & Matrix<T>::operator()(unsigned nel) {
02726     return at(nel);
02727 }
02728
02729 template<typename T>
02730 inline T & Matrix<T>::operator()(unsigned row, unsigned col) {
02731     return at(row, col);
02732 }
02733
02734 template<typename T>
02735 inline T Matrix<T>::operator()(unsigned nel) const {
02736     return at(nel);
02737 }
02738
02739 template<typename T>
02740 inline T Matrix<T>::operator()(unsigned row, unsigned col) const {
02741     return at(row, col);
02742 }
02743
02744 template<typename T>
02745 inline T & Matrix<T>::at(unsigned nel) {
02746     if (!(nel < numel())) throw std::out_of_range("Element index out of range");
02747
02748     return data[nel];
02749 }
02750
02751 template<typename T>
02752 inline T & Matrix<T>::at(unsigned row, unsigned col) {
02753     if (!(row < rows() && col < cols())) throw std::out_of_range("Element index out of range");
02754
02755     return data[nrows * col + row];
02756 }
02757
02758 template<typename T>
02759 inline T Matrix<T>::at(unsigned nel) const {
02760     if (!(nel < numel())) throw std::out_of_range("Element index out of range");
02761
02762     return data[nel];
02763 }
02764
02765 template<typename T>
02766 inline T Matrix<T>::at(unsigned row, unsigned col) const {
02767     if (!(row < rows())) throw std::out_of_range("Row index out of range");
02768     if (!(col < cols())) throw std::out_of_range("Column index out of range");
02769
02770     return data[nrows * col + row];

```

```

02771 }
02772
02773 template<typename T>
02774 inline void Matrix<T>::fill(T value) {
02775     for (unsigned i = 0; i < numel(); i++)
02776         data[i] = value;
02777 }
02778
02779 template<typename T>
02780 inline void Matrix<T>::fill_col(T value, unsigned col) {
02781     if (!(col < cols())) throw std::out_of_range("Column index out of range");
02782
02783     for (unsigned i = col * nrows; i < (col+1) * nrows; i++)
02784         data[i] = value;
02785 }
02786
02787 template<typename T>
02788 inline void Matrix<T>::fill_row(T value, unsigned row) {
02789     if (!(row < rows())) throw std::out_of_range("Row index out of range");
02790
02791     for (unsigned i = 0; i < ncols; i++)
02792         data[row + i * nrows] = value;
02793 }
02794
02795 template<typename T>
02796 inline bool Matrix<T>::exists(unsigned row, unsigned col) const {
02797     return (row < nrows && col < ncols);
02798 }
02799
02800 template<typename T>
02801 inline T* Matrix<T>::ptr(unsigned row, unsigned col) {
02802     if (!(row < rows())) throw std::out_of_range("Row index out of range");
02803     if (!(col < cols())) throw std::out_of_range("Column index out of range");
02804
02805     return data.data() + nrows * col + row;
02806 }
02807
02808 template<typename T>
02809 inline T* Matrix<T>::ptr() {
02810     return data.data();
02811 }
02812
02813 template<typename T>
02814 inline bool Matrix<T>::isempty() const {
02815     return (nrows == 0) || (ncols == 0);
02816 }
02817
02818 template<typename T>
02819 inline bool Matrix<T>::issquare() const {
02820     return (nrows == ncols) && !isempty();
02821 }
02822
02823 template<typename T>
02824 bool Matrix<T>::isequal(const Matrix<T>& A) const {
02825     bool ret = true;
02826     if (nrows != A.rows() || ncols != A.cols()) {
02827         ret = false;
02828     } else {
02829         for (unsigned i = 0; i < numel(); i++) {
02830             if (at(i) != A(i)) {
02831                 ret = false;
02832                 break;
02833             }
02834         }
02835     }
02836     return ret;
02837 }
02838
02839 template<typename T>
02840 bool Matrix<T>::isequal(const Matrix<T>& A, T tol) const {
02841     bool ret = true;
02842     if (rows() != A.rows() || cols() != A.cols()) {
02843         ret = false;
02844     } else {
02845         auto abs_tol = std::abs(tol); // workaround for complex
02846         for (unsigned i = 0; i < A.numel(); i++) {
02847             if (abs_tol < std::abs(at(i) - A(i))) {
02848                 ret = false;
02849                 break;
02850             }
02851         }
02852     }
02853     return ret;
02854 }
02855
02856 template<typename T>
02857 inline unsigned Matrix<T>::numel() const {

```



```

02858     return nrows * ncols;
02859 }
02860
02861 template<typename T>
02862 inline unsigned Matrix<T>::rows() const {
02863     return nrows;
02864 }
02865
02866 template<typename T>
02867 inline unsigned Matrix<T>::cols() const {
02868     return ncols;
02869 }
02870
02871 template<typename T>
02872 inline Matrix<T> Matrix<T>::transpose() const {
02873     Matrix<T> res(ncols, nrows);
02874     for (unsigned c = 0; c < ncols; c++)
02875         for (unsigned r = 0; r < nrows; r++)
02876             res(c,r) = at(r,c);
02877     return res;
02878 }
02879
02880 template<typename T>
02881 inline Matrix<T> Matrix<T>::ctranspose() const {
02882     Matrix<T> res(ncols, nrows);
02883     for (unsigned c = 0; c < ncols; c++)
02884         for (unsigned r = 0; r < nrows; r++)
02885             res(c,r) = cconj(at(r,c));
02886     return res;
02887 }
02888
02889 template<typename T>
02890 Matrix<T>& Matrix<T>::add(const Matrix<T>& m) {
02891     if (!(m.rows() == rows() && m.cols() == cols())) throw std::runtime_error("Unmatching matrix
dimensions for iadd");
02892
02893     for (unsigned i = 0; i < numel(); i++)
02894         data[i] += m(i);
02895     return *this;
02896 }
02897
02898 template<typename T>
02899 Matrix<T>& Matrix<T>::subtract(const Matrix<T>& m) {
02900     if (!(m.rows() == rows() && m.cols() == cols())) throw std::runtime_error("Unmatching matrix
dimensions for isubtract");
02901
02902     for (unsigned i = 0; i < numel(); i++)
02903         data[i] -= m(i);
02904     return *this;
02905 }
02906
02907 template<typename T>
02908 Matrix<T>& Matrix<T>::mult_hadamard(const Matrix<T>& m) {
02909     if (!(m.rows() == rows() && m.cols() == cols())) throw std::runtime_error("Unmatching matrix
dimensions for ihprod");
02910
02911     for (unsigned i = 0; i < numel(); i++)
02912         data[i] *= m(i);
02913     return *this;
02914 }
02915
02916 template<typename T>
02917 Matrix<T>& Matrix<T>::add(T s) {
02918     for (auto& x : data)
02919         x += s;
02920     return *this;
02921 }
02922
02923 template<typename T>
02924 Matrix<T>& Matrix<T>::subtract(T s) {
02925     for (auto& x : data)
02926         x -= s;
02927     return *this;
02928 }
02929
02930 template<typename T>
02931 Matrix<T>& Matrix<T>::mult(T s) {
02932     for (auto& x : data)
02933         x *= s;
02934     return *this;
02935 }
02936
02937 template<typename T>
02938 Matrix<T>& Matrix<T>::div(T s) {
02939     for (auto& x : data)
02940         x /= s;
02941     return *this;

```

```

02942 }
02943
02944 template<typename T>
02945 void Matrix<T>::add_row_to_another(unsigned to, unsigned from) {
02946     if (!(to < rows() && from < rows())) throw std::out_of_range("Row index out of range");
02947
02948     for (unsigned k = 0; k < cols(); k++)
02949         at(to, k) += at(from, k);
02950 }
02951
02952 template<typename T>
02953 void Matrix<T>::add_col_to_another(unsigned to, unsigned from) {
02954     if (!(to < cols() && from < cols())) throw std::out_of_range("Column index out of range");
02955
02956     for (unsigned k = 0; k < rows(); k++)
02957         at(k, to) += at(k, from);
02958 }
02959
02960 template<typename T>
02961 void Matrix<T>::mult_row_by_another(unsigned to, unsigned from) {
02962     if (!(to < rows() && from < rows())) throw std::out_of_range("Row index out of range");
02963
02964     for (unsigned k = 0; k < cols(); k++)
02965         at(to, k) *= at(from, k);
02966 }
02967
02968 template<typename T>
02969 void Matrix<T>::mult_col_by_another(unsigned to, unsigned from) {
02970     if (!(to < cols() && from < cols())) throw std::out_of_range("Column index out of range");
02971
02972     for (unsigned k = 0; k < rows(); k++)
02973         at(k, to) *= at(k, from);
02974 }
02975
02976 template<typename T>
02977 void Matrix<T>::swap_rows(unsigned i, unsigned j) {
02978     if (!(i < rows() && j < rows())) throw std::out_of_range("Row index out of range");
02979
02980     for (unsigned k = 0; k < cols(); k++) {
02981         T tmp = at(i, k);
02982         at(i, k) = at(j, k);
02983         at(j, k) = tmp;
02984     }
02985 }
02986
02987 template<typename T>
02988 void Matrix<T>::swap_cols(unsigned i, unsigned j) {
02989     if (!(i < cols() && j < cols())) throw std::out_of_range("Column index out of range");
02990
02991     for (unsigned k = 0; k < rows(); k++) {
02992         T tmp = at(k, i);
02993         at(k, i) = at(k, j);
02994         at(k, j) = tmp;
02995     }
02996 }
02997
02998 template<typename T>
02999 inline std::vector<T> Matrix<T>::to_vector() const {
03000     return data;
03001 }
03002
03003 template<typename T>
03004 inline std::vector<T> Matrix<T>::col_to_vector(unsigned col) const {
03005     std::vector<T> vec(rows());
03006     for (unsigned i = 0; i < rows(); i++)
03007         vec[i] = at(i, col);
03008     return vec;
03009 }
03010
03011 template<typename T>
03012 inline std::vector<T> Matrix<T>::row_to_vector(unsigned row) const {
03013     std::vector<T> vec(cols());
03014     for (unsigned i = 0; i < cols(); i++)
03015         vec[i] = at(row, i);
03016     return vec;
03017 }
03018
03019 template<typename T>
03020 inline void Matrix<T>::col_from_vector(const std::vector<T>& vec, unsigned col) {
03021     if (vec.size() != rows()) throw std::runtime_error("Vector size is not equal to number of rows");
03022     if (col >= cols()) throw std::out_of_range("Column index out of range");
03023
03024     for (unsigned i = 0; i < rows(); i++)
03025         data[col*rows() + i] = vec[i];
03026 }
03027
03028 template<typename T>

```

```
03029 inline void Matrix<T>::row_from_vector(const std::vector<T>& vec, unsigned row) {
03030     if (vec.size() != rows()) throw std::runtime_error("Vector size is not equal to number of columns");
03031     if (row >= rows()) throw std::out_of_range("Row index out of range");
03032
03033     for (unsigned i = 0; i < cols(); i++)
03034         data[row + i*rows()] = vec[i];
03035 }
03036
03037 template<typename T>
03038 Matrix<T>::~~Matrix() { }
03039
03040 } // namespace Matrix_hpp
03041
03042 #endif // __MATRIX_HPP__
```

