

matOps

Generated by Doxygen 1.9.1

<b>1 Class Index</b>	<b>1</b>
1.1 Class List	1
<b>2 File Index</b>	<b>2</b>
2.1 File List	2
<b>3 Class Documentation</b>	<b>3</b>
3.1 Matrix Class Reference	3
3.1.1 Detailed Description	5
3.1.2 Constructor & Destructor Documentation	5
3.1.2.1 Matrix()	5
3.1.3 Member Function Documentation	6
3.1.3.1 constValMatrix()	6
3.1.3.2 determinant()	6
3.1.3.3 extractCol()	7
3.1.3.4 extractMatrix()	7
3.1.3.5 extractRow()	8
3.1.3.6 hStack()	8
3.1.3.7 identity()	9
3.1.3.8 insertCol() [1/2]	9
3.1.3.9 insertCol() [2/2]	10
3.1.3.10 insertRow() [1/2]	10
3.1.3.11 insertRow() [2/2]	11
3.1.3.12 inverse()	11
3.1.3.13 mean()	12
3.1.3.14 operator"!="()	12
3.1.3.15 operator>()	12
3.1.3.16 operator*() [1/2]	13
3.1.3.17 operator*() [2/2]	13
3.1.3.18 operator+() [1/2]	14
3.1.3.19 operator+() [2/2]	14
3.1.3.20 operator-() [1/2]	15
3.1.3.21 operator-() [2/2]	15
3.1.3.22 operator/()	16
3.1.3.23 operator==()	16
3.1.3.24 operator^()	17
3.1.3.25 shape()	17
3.1.3.26 shuffleRows() [1/2]	17
3.1.3.27 shuffleRows() [2/2]	17
3.1.3.28 sum() [1/2]	18
3.1.3.29 sum() [2/2]	18
3.1.3.30 toVector()	19
3.1.3.31 trace()	19

3.1.3.32 transpose()	19
3.1.3.33 vStack()	20
3.1.4 Friends And Related Function Documentation	20
3.1.4.1 operator*	20
3.1.4.2 operator+	21
3.1.4.3 operator-	21
3.1.4.4 operator<<	22
<b>4 File Documentation</b>	<b>23</b>
4.1 src/matOps.hpp File Reference	23
4.1.1 Macro Definition Documentation	23
4.1.1.1 EPS	23
4.1.1.2 OPENMP_THRESHOLD	24
4.1.2 Function Documentation	24
4.1.2.1 operator<<() [1/2]	24
4.1.2.2 operator<<() [2/2]	24
4.2 test/testMatrix.cpp File Reference	24
4.2.1 Detailed Description	26
4.2.2 Macro Definition Documentation	26
4.2.2.1 DOCTEST_CONFIG_IMPLEMENT_WITH_MAIN	26
4.2.3 Function Documentation	26
4.2.3.1 TEST_CASE() [1/35]	26
4.2.3.2 TEST_CASE() [2/35]	26
4.2.3.3 TEST_CASE() [3/35]	26
4.2.3.4 TEST_CASE() [4/35]	26
4.2.3.5 TEST_CASE() [5/35]	27
4.2.3.6 TEST_CASE() [6/35]	27
4.2.3.7 TEST_CASE() [7/35]	27
4.2.3.8 TEST_CASE() [8/35]	27
4.2.3.9 TEST_CASE() [9/35]	27
4.2.3.10 TEST_CASE() [10/35]	28
4.2.3.11 TEST_CASE() [11/35]	28
4.2.3.12 TEST_CASE() [12/35]	28
4.2.3.13 TEST_CASE() [13/35]	28
4.2.3.14 TEST_CASE() [14/35]	29
4.2.3.15 TEST_CASE() [15/35]	29
4.2.3.16 TEST_CASE() [16/35]	29
4.2.3.17 TEST_CASE() [17/35]	29
4.2.3.18 TEST_CASE() [18/35]	29
4.2.3.19 TEST_CASE() [19/35]	29
4.2.3.20 TEST_CASE() [20/35]	30
4.2.3.21 TEST_CASE() [21/35]	30

---

4.2.3.22 TEST_CASE() [22/35] . . . . .	30
4.2.3.23 TEST_CASE() [23/35] . . . . .	30
4.2.3.24 TEST_CASE() [24/35] . . . . .	30
4.2.3.25 TEST_CASE() [25/35] . . . . .	31
4.2.3.26 TEST_CASE() [26/35] . . . . .	31
4.2.3.27 TEST_CASE() [27/35] . . . . .	31
4.2.3.28 TEST_CASE() [28/35] . . . . .	31
4.2.3.29 TEST_CASE() [29/35] . . . . .	31
4.2.3.30 TEST_CASE() [30/35] . . . . .	31
4.2.3.31 TEST_CASE() [31/35] . . . . .	31
4.2.3.32 TEST_CASE() [32/35] . . . . .	32
4.2.3.33 TEST_CASE() [33/35] . . . . .	32
4.2.3.34 TEST_CASE() [34/35] . . . . .	32
4.2.3.35 TEST_CASE() [35/35] . . . . .	32

# Chapter 1

## Class Index

### 1.1 Class List

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

<a href="#">Matrix</a>	A simple linear algebra library for matrix operations . . . . .	<a href="#">3</a>
------------------------	---	-------------------

## Chapter 2

# File Index

### 2.1 File List

Here is a list of all files with brief descriptions:

src/ <a href="#">matOps.hpp</a> . . . . .	23
test/ <a href="#">testMatrix.cpp</a>	
Unit tests for the <a href="#">Matrix</a> class using doctest . . . . .	24

## Chapter 3

# Class Documentation

### 3.1 Matrix Class Reference

A simple linear algebra library for matrix operations.

```
#include <matOps.hpp>
```

#### Public Member Functions

- `std::pair< size_t, size_t > shape () const`  
*Returns the dimensions of the matrix.*
- `Matrix (const std::vector< std::vector< double >> &container)`  
*Constructs a [Matrix](#) from a given 2D vector container.*
- `std::vector< std::vector< double > > toVector () const`  
*Returns a copy of the matrix data.*
- `Matrix operator+ (const Matrix &other) const`  
*Adds two matrices element-wise.*
- `Matrix operator+ (double scalar) const`  
*Adds a scalar value to each element of the matrix. ( $MATRIX + K$ )*
- `Matrix operator- (const Matrix &other) const`  
*Subtracts one matrix from another element-wise.*
- `Matrix operator- (double scalar) const`  
*Subtracts a scalar from each element of the matrix. ( $MATRIX - K$ )*
- `bool operator== (const Matrix &other) const`  
*Compares two matrices for equality.*
- `bool operator!= (const Matrix &other) const`  
*Compares two matrices for inequality.*
- `Matrix operator\* (const Matrix &other) const`  
*Multiplies two matrices.*
- `Matrix operator\* (double scalar) const`  
*Multiplies each element of the matrix by a scalar. ( $MATRIX * K$ )*
- `Matrix operator/ (double scalar) const`  
*Divides each element of the matrix by a scalar.*
- `Matrix operator^ (double scalar) const`
- `double & operator\(\) (size_t row, size_t col)`

- Accesses an element of the matrix at a specified row and column.*

  - [Matrix transpose](#) () const  
*Transposes the matrix.*
  - double [determinant](#) () const  
*Computes the determinant of the matrix.*
  - [Matrix inverse](#) () const  
*Computes the inverse of the matrix.*
  - [Matrix insertRow](#) (std::vector< double > row, size\_t idx) const  
*Inserts a new row into the matrix.*
  - [Matrix insertRow](#) (double rowVal, size\_t idx) const  
*Inserts a new row filled with a constant value into the matrix.*
  - [Matrix insertCol](#) (std::vector< double > col, size\_t idx) const  
*Inserts a new column into the matrix.*
  - [Matrix insertCol](#) (double colVal, size\_t idx) const  
*Inserts a new column filled with a constant value into the matrix.*
  - [Matrix hStack](#) (const [Matrix](#) &other) const  
*Horizontally concatenates two matrices.*
  - [Matrix vStack](#) (const [Matrix](#) &other) const  
*Vertically concatenates two matrices.*
  - void [shuffleRows](#) ()  
*Shuffles the rows of the matrix using a random seed.*
  - void [shuffleRows](#) (size\_t random\_state)  
*Shuffles the rows of the matrix using a specified seed.*
  - [Matrix extractMatrix](#) (std::pair< size\_t, size\_t > rowSlice, std::pair< size\_t, size\_t > colSlice) const  
*Extracts a submatrix from the current [Matrix](#).*
  - [Matrix extractRow](#) (size\_t rowIdx) const  
*Extracts a specific row from the current [Matrix](#).*
  - [Matrix extractCol](#) (size\_t colIdx) const  
*Extracts a specific column from the current [Matrix](#).*
  - double [sum](#) () const  
*Computes the sum of the elements of a vector.*
  - double [sum](#) (double power) const  
*Computes the sum of all elements raised to a specified power.*
  - double [mean](#) () const  
*Computes the mean (average) of the elements of a vector.*
  - double [trace](#) () const  
*Computes the trace of an nxn square matrix.*

## Static Public Member Functions

- static [Matrix identity](#) (size\_t dim)  
*Constructs an Identity [Matrix](#) of specified dimensions.*
- static [Matrix constValMatrix](#) (size\_t rows, size\_t cols, double val)  
*Creates a matrix with constant values.*



## Friends

- **Matrix operator+** (double scalar, const **Matrix** &other)  
*Adds a scalar to each element of a matrix. ( $K + MATRIX$ )*
- **Matrix operator-** (double scalar, const **Matrix** &other)  
*Subtracts each element of the matrix from a scalar. ( $K - MATRIX$ )*
- **Matrix operator\*** (double scalar, const **Matrix** &other)  
*Multiplies a scalar by a matrix. ( $K * MATRIX$ )*
- `std::ostream & operator<<` (`std::ostream &os`, const **Matrix** &m)  
*Outputs the matrix to an output stream.*

### 3.1.1 Detailed Description

A simple linear algebra library for matrix operations.

#### Version

0.6.1

This class provides basic matrix operations such as addition, subtraction, multiplication, transposition, determinant calculation, inversion, and row/column insertion.

#### Example Usage:

```
Matrix A({{1, 2}, {3, 4}});
Matrix B({{5, 6}, {7, 8}});
Matrix C = A + B; // Matrix addition
Matrix D = A * B; // Matrix multiplication
double detA = A.determinant(); // Determinant calculation
```

### 3.1.2 Constructor & Destructor Documentation

#### 3.1.2.1 Matrix()

```
Matrix::Matrix (
    const std::vector< std::vector< double >> & container ) [inline]
```

Constructs a **Matrix** from a given 2D vector container.

#### Parameters

<i>container</i>	A 2D vector of doubles representing the matrix.
------------------	---

#### Exceptions

<i>std::invalid_argument</i>	if the container is empty or row sizes are inconsistent.
------------------------------	--

**Note**

The shape is determined by the size of the container.

**3.1.3 Member Function Documentation****3.1.3.1 constValMatrix()**

```
static Matrix Matrix::constValMatrix (
    size_t rows,
    size_t cols,
    double val ) [inline], [static]
```

Creates a matrix with constant values.

Function constructs and returns a [Matrix](#) object with the specified dimensions, initializing every element to the given constant value.

Example usage:

```
// Create a 3x3 matrix filled with 5.0
Matrix A = Matrix::constValMatrix(3, 3, 5.0);
// Expected Output:
// 5.0 5.0 5.0
// 5.0 5.0 5.0
// 5.0 5.0 5.0
```

**Parameters**

<i>rows</i>	The number of rows in the matrix.
<i>cols</i>	The number of columns in the matrix.
<i>val</i>	The constant value to initialize each element of the matrix.

**Returns**

A [Matrix](#) object of dimensions (rows x cols) where each element is set to *val*.

**Exceptions**

<i>std::invalid_argument</i>	if either <i>rows</i> or <i>cols</i> is zero.
------------------------------	---

**3.1.3.2 determinant()**

```
double Matrix::determinant ( ) const [inline]
```

Computes the determinant of the matrix.

**Returns**

The determinant as a double.

## Exceptions

<code>std::invalid_argument</code>	if the matrix is not square.
------------------------------------	------------------------------

## Note

The shape of the matrix remains unchanged.

3.1.3.3 `extractCol()`

```
Matrix Matrix::extractCol (
    size_t colIdx ) const [inline]
```

Extracts a specific column from the current [Matrix](#).

This function calls [Matrix::extractMatrix\(\)](#) under the hood as extracting a column will require the exact same logic.

## Parameters

<code>colIdx</code>	The zero-based index of the column to extract.
---------------------	--

## Returns

A new [Matrix](#) object containing the extracted column.

## Exceptions

<code>std::invalid_argument</code>	If the specified column index is out of range.
------------------------------------	--

3.1.3.4 `extractMatrix()`

```
Matrix Matrix::extractMatrix (
    std::pair< size_t, size_t > rowSlice,
    std::pair< size_t, size_t > colSlice ) const [inline]
```

Extracts a submatrix from the current [Matrix](#).

Given a pair of row indices and a pair of column indices, this function creates and returns a new [Matrix](#) containing the submatrix defined by the specified ranges [start, end).

## Parameters

<code>rowSlice</code>	A <code>std::pair&lt;size_t, size_t&gt;</code> representing the start and end row indices.
<code>colSlice</code>	A <code>std::pair&lt;size_t, size_t&gt;</code> representing the start and end column indices.

**Returns**

A new [Matrix](#) object containing the extracted submatrix.

**Exceptions**

<code>std::out_of_range</code>	If any indices are out of bounds or if the slice ranges are invalid.
--------------------------------	--

**3.1.3.5 extractRow()**

```
Matrix Matrix::extractRow (
    size_t rowIdx ) const [inline]
```

Extracts a specific row from the current [Matrix](#).

This function creates and returns a new [Matrix](#) object containing the row at the specified index. The row is extracted as a single-row submatrix.

**Parameters**

<code>rowIdx</code>	The zero-based index of the row to extract.
---------------------	---

**Returns**

A new [Matrix](#) object containing the extracted row.

**Exceptions**

<code>std::invalid_argument</code>	If the specified row index is out of range.
------------------------------------	---

**3.1.3.6 hStack()**

```
Matrix Matrix::hStack (
    const Matrix & other ) const [inline]
```

Horizontally concatenates two matrices.

This function creates and returns a new [Matrix](#) by appending the columns of the given matrix to the right of the calling matrix. Both matrices must have the same number of rows.

**Parameters**

<code>other</code>	The <a href="#">Matrix</a> whose columns will be appended to the calling matrix.
--------------------	--

**Returns**

A new [Matrix](#) representing the horizontal concatenation of the two matrices.

**Exceptions**

<code>std::invalid_argument</code>	if the two matrices do not have the same number of rows.
------------------------------------	--

**Note**

This implementation reserves the necessary capacity before inserting to minimize reallocations.

**3.1.3.7 identity()**

```
static Matrix Matrix::identity (
    size_t dim ) [inline], [static]
```

Constructs an Identity [Matrix](#) of specified dimensions.

```
Matrix A = Matrix::identity(3);
```

```
A = [
    [1, 0, 0]
    [0, 1, 0]
    [0, 0, 1]
]
```

**Parameters**

<i>dim</i>	Dimensions of matrix (dim x dim).
------------	-----------------------------------

**3.1.3.8 insertCol() [1/2]**

```
Matrix Matrix::insertCol (
    double colVal,
    size_t idx ) const [inline]
```

Inserts a new column filled with a constant value into the matrix.

**Parameters**

<i>colVal</i>	The constant value to fill the new column.
<i>idx</i>	The index at which to insert the column.

**Returns**

A new [Matrix](#) with the column inserted.

## Exceptions

<code>std::invalid_argument</code>	if <code>idx</code> is out of range.
------------------------------------	--------------------------------------

**3.1.3.9 insertCol()** [2/2]

```
Matrix Matrix::insertCol (  
    std::vector< double > col,  
    size_t idx ) const [inline]
```

Inserts a new column into the matrix.

## Parameters

<code>col</code>	A vector representing the new column.
<code>idx</code>	The index at which to insert the column.

## Returns

A new [Matrix](#) with the column inserted.

## Exceptions

<code>std::invalid_argument</code>	if the column size is inconsistent or if <code>idx</code> is out of range.
------------------------------------	--

**3.1.3.10 insertRow()** [1/2]

```
Matrix Matrix::insertRow (  
    double rowVal,  
    size_t idx ) const [inline]
```

Inserts a new row filled with a constant value into the matrix.

## Parameters

<code>rowVal</code>	The constant value to fill the new row.
<code>idx</code>	The index at which to insert the row.

## Returns

A new [Matrix](#) with the row inserted.

## Exceptions

<code>std::invalid_argument</code>	if <code>idx</code> is out of range.
------------------------------------	--------------------------------------

**3.1.3.11 insertRow()** [2/2]

```
Matrix Matrix::insertRow (  
    std::vector< double > row,  
    size_t idx ) const [inline]
```

Inserts a new row into the matrix.

## Parameters

<i>row</i>	A vector representing the new row.
<i>idx</i>	The index at which to insert the row.

## Returns

A new [Matrix](#) with the row inserted.

## Exceptions

<code>std::invalid_argument</code>	if the row size is inconsistent or if <code>idx</code> is out of range.
------------------------------------	---

**3.1.3.12 inverse()**

```
Matrix Matrix::inverse ( ) const [inline]
```

Computes the inverse of the matrix.

## Returns

A new [Matrix](#) representing the inverse.

## Exceptions

<code>std::runtime_error</code>	if the matrix is singular (non-invertible).
---------------------------------	---

## Note

The shape remains unchanged.

**3.1.3.13 mean()**

```
double Matrix::mean ( ) const [inline]
```

Computes the mean (average) of the elements of a vector.

Calculates the mean value for matrices that are considered as vectors. It supports both row vectors (1 x K) and column vectors (K x 1) by dividing the sum of the elements by the number of elements in the vector.

**Returns**

The mean (average) value of the vector elements.

**Exceptions**

<code>std::invalid_argument</code>	If the matrix is not a one-dimensional vector.
------------------------------------	--

**3.1.3.14 operator!=(=)**

```
bool Matrix::operator!= (
    const Matrix & other ) const [inline]
```

Compares two matrices for inequality.

This operator checks if two matrices are not equal by comparing their dimensions and individual elements. Two matrices are considered not equal if:

- Their dimensions differ, or
- At least one pair of corresponding elements differs by more than EPS.

**Parameters**

<code>other</code>	The matrix to compare with.
--------------------	-----------------------------

**Returns**

true if the matrices differ by at least one element more than EPS; false otherwise.

**3.1.3.15 operator()(=)**

```
double& Matrix::operator() (
    size_t row,
    size_t col ) [inline]
```

Accesses an element of the matrix at a specified row and column.



## Parameters

<i>row</i>	The row index.
<i>col</i>	The column index.

## Returns

Reference to the value at the specified position. (modifiable)

## Exceptions

<i>std::out_of_range</i>	if the indices are out of bounds.
--------------------------	-----------------------------------

**3.1.3.16 operator\*() [1/2]**

```
Matrix Matrix::operator* (
    const Matrix & other ) const [inline]
```

Multiplies two matrices.

## Parameters

<i>other</i>	The <a href="#">Matrix</a> to multiply with.
--------------	--

## Returns

A new [Matrix](#) resulting from matrix multiplication.

## Exceptions

<i>std::invalid_argument</i>	if the number of columns of the first matrix does not match the number of rows of the second.
------------------------------	---

## Note

The shape of the result is (nrows of first, ncols of second).

**3.1.3.17 operator\*() [2/2]**

```
Matrix Matrix::operator* (
    double scalar ) const [inline]
```

Multiplies each element of the matrix by a scalar. (MATRIX \* K)

## Parameters

<i>scalar</i>	The scalar value.
---------------	-------------------

## Returns

A new [Matrix](#) with each element multiplied by the scalar.

## Note

The shape remains unchanged.

**3.1.3.18 operator+()** [1/2]

```
Matrix Matrix::operator+ (
    const Matrix & other ) const [inline]
```

Adds two matrices element-wise.

## Parameters

<i>other</i>	The <a href="#">Matrix</a> to add.
--------------	------------------------------------

## Returns

A new [Matrix](#) representing the element-wise sum.

## Exceptions

<i>std::invalid_argument</i>	if the dimensions of the two matrices do not match.
------------------------------	---

## Note

The shape remains unchanged.

**3.1.3.19 operator+()** [2/2]

```
Matrix Matrix::operator+ (
    double scalar ) const [inline]
```

Adds a scalar value to each element of the matrix. (MATRIX + K)

## Parameters

<i>scalar</i>	A double value to add.
---------------	------------------------

## Returns

A new [Matrix](#) with the scalar added to each element.

## Note

The shape remains unchanged.

**3.1.3.20 operator-() [1/2]**

```
Matrix Matrix::operator- (  
    const Matrix & other ) const [inline]
```

Subtracts one matrix from another element-wise.

## Parameters

<i>other</i>	The <a href="#">Matrix</a> to subtract.
--------------	---

## Returns

A new [Matrix](#) representing the element-wise difference.

## Exceptions

<code>std::invalid_argument</code>	if the dimensions of the two matrices do not match.
------------------------------------	---

## Note

The shape remains unchanged.

**3.1.3.21 operator-() [2/2]**

```
Matrix Matrix::operator- (  
    double scalar ) const [inline]
```

Subtracts a scalar from each element of the matrix. (MATRIX - K)

**Parameters**

<i>scalar</i>	The scalar value to subtract.
---------------	-------------------------------

**Returns**

A new [Matrix](#) with each element reduced by the scalar.

**Note**

The shape remains unchanged.

**3.1.3.22 operator/()**

```
Matrix Matrix::operator/ (
    double scalar ) const [inline]
```

Divides each element of the matrix by a scalar.

**Parameters**

<i>scalar</i>	The scalar value.
---------------	-------------------

**Returns**

A new [Matrix](#) with each element divided by the scalar.

**Exceptions**

<i>std::runtime_error</i>	if scalar is zero.
---------------------------	--------------------

**Note**

The shape remains unchanged.

**3.1.3.23 operator==()**

```
bool Matrix::operator== (
    const Matrix & other ) const [inline]
```

Compares two matrices for equality.

## Parameters

<i>other</i>	The <a href="#">Matrix</a> to compare with.
--------------	---

## Returns

True if the matrices are equal (within a tolerance), false otherwise.

**3.1.3.24 operator^()**

```
Matrix Matrix::operator^ (  
    double scalar ) const [inline]
```

**3.1.3.25 shape()**

```
std::pair<size_t, size_t> Matrix::shape ( ) const [inline]
```

Returns the dimensions of the matrix.

## Returns

A `std::pair` where first is the number of rows and second is the number of columns.

**3.1.3.26 shuffleRows() [1/2]**

```
void Matrix::shuffleRows ( ) [inline]
```

Shuffles the rows of the matrix using a random seed.

This function uses a random seed generated by `std::random_device` to initialize the random number generator, ensuring that the shuffle is non-deterministic.

**3.1.3.27 shuffleRows() [2/2]**

```
void Matrix::shuffleRows (  
    size_t random_state ) [inline]
```

Shuffles the rows of the matrix using a specified seed.

This function initializes the random number generator with the provided seed (`random_state`), allowing for reproducible shuffling.

**Parameters**

<i>random_state</i>	The seed value used to initialize the random number generator.
---------------------	--

**3.1.3.28 sum() [1/2]**

```
double Matrix::sum ( ) const [inline]
```

Computes the sum of the elements of a vector.

Calculates the sum of elements for matrices that are considered as vectors. It supports both column vectors ( $K \times 1$ ) and row vectors ( $1 \times K$ ).

**Returns**

The sum of all elements in the vector.

**Exceptions**

<i>std::invalid_argument</i>	If the matrix is not a one-dimensional vector.
------------------------------	--

**3.1.3.29 sum() [2/2]**

```
double Matrix::sum (
    double power ) const [inline]
```

Computes the sum of all elements raised to a specified power.

This function calculates the sum of each element in the matrix after raising it to the given `power`. The function only works for matrices that are either a row matrix ( $1 \times K$ ) or a column matrix ( $K \times 1$ ). If the matrix has any other dimensions, an `std::invalid_argument` exception is thrown.

Additionally, if an element is zero and the `power` is less than or equal to zero, the function will throw an `std::runtime_error` to indicate a division by zero scenario, since 0 raised to a non-positive power is undefined.

**Parameters**

<i>power</i>	The exponent to which each element in the matrix is raised.
--------------	---

**Returns**

The sum of all elements raised to the specified power.

## Exceptions

<code>std::invalid_argument</code>	if the matrix dimensions are not (1, K) or (K, 1).
<code>std::runtime_error</code>	if any element is zero and <code>power</code> is less than or equal to zero.

**3.1.3.30 toVector()**

```
std::vector<std::vector<double> > Matrix::toVector ( ) const [inline]
```

Returns a copy of the matrix data.

This function returns a new 2D vector containing the matrix elements. Modifications to the returned vector do not affect the original matrix.

## Returns

A copy of the 2D vector representing the matrix.

**3.1.3.31 trace()**

```
double Matrix::trace ( ) const [inline]
```

Computes the trace of an nxn square matrix.

## Returns

Trace of calling [Matrix](#) object.

## Exceptions

<code>std::invalid_argument</code>	If the matrix is not square.
------------------------------------	------------------------------

**3.1.3.32 transpose()**

```
Matrix Matrix::transpose ( ) const [inline]
```

Transposes the matrix.

### Returns

A new matrix of dim (mxn) for a calling matrix of dim (nxm) Example:

```
Matrix A({{1, 2}, {3, 4}});
A = A.transpose();
// A becomes:
// [
//   [1, 3],
//   [2, 4]
// ]
```

#### 3.1.3.33 vStack()

```
Matrix Matrix::vStack (
    const Matrix & other ) const [inline]
```

Vertically concatenates two matrices.

This function creates and returns a new [Matrix](#) by appending the rows of the given matrix below the rows of the calling matrix. Both matrices must have the same number of columns.

### Parameters

<i>other</i>	The <a href="#">Matrix</a> whose rows will be appended to the calling matrix.
--------------	---

### Returns

A new [Matrix](#) representing the vertical concatenation of the two matrices.

### Exceptions

<i>std::invalid_argument</i>	if the two matrices do not have the same number of columns.
------------------------------	---

### Note

The function appends each row of the second matrix to the container of the first, and updates the total row count accordingly.

## 3.1.4 Friends And Related Function Documentation

#### 3.1.4.1 operator\*

```
Matrix operator* (
    double scalar,
    const Matrix & other ) [friend]
```

Multiplies a scalar by a matrix. ( $K * MATRIX$ )



## Parameters

<i>scalar</i>	The scalar value.
<i>other</i>	The <a href="#">Matrix</a> to multiply.

## Returns

A new [Matrix](#) with each element multiplied by the scalar.

## Note

The shape remains unchanged.

### 3.1.4.2 operator+

```
Matrix operator+ (  
    double scalar,  
    const Matrix & other ) [friend]
```

Adds a scalar to each element of a matrix. ( $K + \text{MATRIX}$ )

## Parameters

<i>scalar</i>	The scalar value.
<i>other</i>	The <a href="#">Matrix</a> to add the scalar to.

## Returns

A new [Matrix](#) with the result.

## Note

The shape remains unchanged.

### 3.1.4.3 operator-

```
Matrix operator- (  
    double scalar,  
    const Matrix & other ) [friend]
```

Subtracts each element of the matrix from a scalar. ( $K - \text{MATRIX}$ )

## Parameters

<i>scalar</i>	The scalar value.
<i>other</i>	The <a href="#">Matrix</a> whose elements are subtracted from the scalar.

**Returns**

A new [Matrix](#) with the result.

**Note**

The shape remains unchanged.

**3.1.4.4 operator<<**

```
std::ostream& operator<< (  
    std::ostream & os,  
    const Matrix & m ) [friend]
```

Outputs the matrix to an output stream.

**Parameters**

<i>os</i>	The output stream.
<i>m</i>	The <a href="#">Matrix</a> to output.

**Returns**

A reference to the output stream.

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

- [src/matOps.hpp](#)

## Chapter 4

# File Documentation

### 4.1 src/matOps.hpp File Reference

```
#include <iostream>
#include <vector>
#include <cmath>
#include <random>
#include <algorithm>
```

Include dependency graph for matOps.hpp: This graph shows which files directly or indirectly include this file:

#### Classes

- class [Matrix](#)  
*A simple linear algebra library for matrix operations.*

#### Macros

- #define [EPS](#) 1e-12
- #define [OPENMP\\_THRESHOLD](#) 10000

#### Functions

- std::ostream & [operator<<](#) (std::ostream &os, const std::pair< size\_t, size\_t > &shape)
- std::ostream & [operator<<](#) (std::ostream &os, const std::vector< std::vector< double >> &container)

#### 4.1.1 Macro Definition Documentation

##### 4.1.1.1 EPS

```
#define EPS 1e-12
```

#### 4.1.1.2 OPENMP\_THRESHOLD

```
#define OPENMP_THRESHOLD 10000
```

### 4.1.2 Function Documentation

#### 4.1.2.1 operator<<() [1/2]

```
std::ostream& operator<< (
    std::ostream & os,
    const std::pair< size_t, size_t > & shape ) [inline]
```

#### 4.1.2.2 operator<<() [2/2]

```
std::ostream& operator<< (
    std::ostream & os,
    const std::vector< std::vector< double >> & container ) [inline]
```

## 4.2 test/testMatrix.cpp File Reference

Unit tests for the [Matrix](#) class using doctest.

```
#include "doctest.h"
#include "../src/matOps.hpp"
#include <vector>
#include <stdexcept>
#include <cmath>
Include dependency graph for testMatrix.cpp:
```

### Macros

- `#define DOCTEST_CONFIG_IMPLEMENT_WITH_MAIN`

## Functions

- [TEST\\_CASE](#) ("Matrix Construction and Shape")  
*Tests for [Matrix](#) construction and shape reporting.*
- [TEST\\_CASE](#) ("Matrix Addition and Subtraction")  
*Tests for [Matrix](#) addition and subtraction operations.*
- [TEST\\_CASE](#) ("Matrix Multiplication and Division")  
*Tests for [Matrix](#) multiplication and division.*
- [TEST\\_CASE](#) ("Element Access Operator()")  
*Tests for element access using operator().*
- [TEST\\_CASE](#) ("Matrix Transpose")  
*Tests for [Matrix](#) transposition.*
- [TEST\\_CASE](#) ("Matrix Determinant")  
*Tests for [Matrix](#) determinant computation.*
- [TEST\\_CASE](#) ("Matrix Inverse")  
*Tests for [Matrix](#) inversion.*
- [TEST\\_CASE](#) ("Matrix Row and Column Insertion")  
*Tests for [Matrix](#) row and column insertion methods.*
- [TEST\\_CASE](#) ("Matrix Horizontal and Vertical Stacking")  
*Tests for [Matrix](#) horizontal and vertical stacking.*
- [TEST\\_CASE](#) ("Matrix Submatrix Extraction")  
*Tests for [Matrix](#) submatrix extraction functionality using exclusive indices.*
- [TEST\\_CASE](#) ("Equality Tolerance Test")  
*Tests for equality comparisons with tolerance.*
- [TEST\\_CASE](#) ("Matrix Inequality Operator")
- [TEST\\_CASE](#) ("Copy Constructor and Assignment Operator")  
*Tests for copy construction and assignment operator.*
- [TEST\\_CASE](#) ("3x3 [Matrix](#) Inversion and Identity Check")  
*Tests for 3x3 matrix inversion and verifying the identity.*
- [TEST\\_CASE](#) ("Testing [Matrix::identity](#)")
- [TEST\\_CASE](#) ("Single Row and Single Column Submatrix Extraction")  
*Tests for submatrix extraction of a single row and a single column.*
- [TEST\\_CASE](#) ("Chained Mixed Operations")  
*Tests for chained mixed arithmetic operations.*
- [TEST\\_CASE](#) ("Row vector: sum and mean")
- [TEST\\_CASE](#) ("Column vector: sum and mean")
- [TEST\\_CASE](#) ("Non-vector matrix: sum throws invalid\_argument")
- [TEST\\_CASE](#) ("Matrix exponentiation with scalar 1 returns the same matrix")
- [TEST\\_CASE](#) ("Matrix exponentiation with scalar 2 returns element-wise square")
- [TEST\\_CASE](#) ("Matrix exponentiation with scalar 0 returns element-wise 1 (nonzero elements)")
- [TEST\\_CASE](#) ("Matrix exponentiation when an element is 0 and exponent is  $\leq 0$ ")
- [TEST\\_CASE](#) ("Matrix shuffleRows methods")
- [TEST\\_CASE](#) ("Matrix::constValMatrix creates a constant matrix")
- [TEST\\_CASE](#) ("Matrix::constValMatrix throws for zero dimensions")
- [TEST\\_CASE](#) ("Row matrix: valid sum calculation with positive power")
- [TEST\\_CASE](#) ("Column matrix: valid sum calculation with positive power")
- [TEST\\_CASE](#) ("Invalid matrix dimensions should throw invalid\_argument")
- [TEST\\_CASE](#) ("Row matrix: zero element with non-positive power throws runtime\_error")
- [TEST\\_CASE](#) ("Column matrix: zero element with non-positive power throws runtime\_error")
- [TEST\\_CASE](#) ("Testing [Matrix::extractRow](#)")
- [TEST\\_CASE](#) ("Testing [Matrix::extractCol](#)")
- [TEST\\_CASE](#) ("trace() with vector-of-vector constructor for square matrices")

### 4.2.1 Detailed Description

Unit tests for the [Matrix](#) class using doctest.

### 4.2.2 Macro Definition Documentation

#### 4.2.2.1 DOCTEST\_CONFIG\_IMPLEMENT\_WITH\_MAIN

```
#define DOCTEST_CONFIG_IMPLEMENT_WITH_MAIN
```

### 4.2.3 Function Documentation

#### 4.2.3.1 TEST\_CASE() [1/35]

```
TEST_CASE (
    "3x3 Matrix Inversion and Identity Check" )
```

Tests for 3x3 matrix inversion and verifying the identity.

#### 4.2.3.2 TEST\_CASE() [2/35]

```
TEST_CASE (
    "Chained Mixed Operations" )
```

Tests for chained mixed arithmetic operations.

#### 4.2.3.3 TEST\_CASE() [3/35]

```
TEST_CASE (
    "Column matrix: valid sum calculation with positive power" )
```

#### 4.2.3.4 TEST\_CASE() [4/35]

```
TEST_CASE (
    "Column matrix: zero element with non-positive power throws runtime_error" )
```

#### 4.2.3.5 TEST\_CASE() [5/35]

```
TEST_CASE (
    "Column vector: sum and mean" )
```

#### 4.2.3.6 TEST\_CASE() [6/35]

```
TEST_CASE (
    "Copy Constructor and Assignment Operator" )
```

Tests for copy construction and assignment operator.

#### 4.2.3.7 TEST\_CASE() [7/35]

```
TEST_CASE (
    "Element Access Operator()" )
```

Tests for element access using operator().

Verify correct element access.

Modification via operator(). Tests that modifying an element via the accessor works correctly.

Out-of-range access. Checks that accessing an element outside the matrix bounds throws an exception.

#### 4.2.3.8 TEST\_CASE() [8/35]

```
TEST_CASE (
    "Equality Tolerance Test" )
```

Tests for equality comparisons with tolerance.

#### 4.2.3.9 TEST\_CASE() [9/35]

```
TEST_CASE (
    "Invalid matrix dimensions should throw invalid_argument" )
```

#### 4.2.3.10 TEST\_CASE() [10/35]

```
TEST_CASE (
    "Matrix Addition and Subtraction" )
```

Tests for [Matrix](#) addition and subtraction operations.

[Matrix](#) addition (matrix + matrix). Verifies that element-wise addition produces the expected result.

[Matrix](#) addition with scalar (matrix + k and k + matrix). Verifies both left and right scalar addition.

[Matrix](#) subtraction (matrix - matrix). Checks that subtracting one matrix from another yields the correct result.

[Matrix](#) subtraction with scalar (matrix - k and k - matrix). Verifies subtraction when a scalar is involved.

#### 4.2.3.11 TEST\_CASE() [11/35]

```
TEST_CASE (
    "Matrix Construction and Shape" )
```

Tests for [Matrix](#) construction and shape reporting.

Valid construction with a 2x2 matrix. Checks that the matrix correctly reports its dimensions.

Invalid construction with an empty container. Expects an `invalid_argument` exception.

Invalid construction with inconsistent row sizes. Expects an `invalid_argument` exception.

#### 4.2.3.12 TEST\_CASE() [12/35]

```
TEST_CASE (
    "Matrix Determinant" )
```

Tests for [Matrix](#) determinant computation.

Determinant for a 1x1 matrix.

Determinant for a 2x2 matrix.

Determinant for a 3x3 matrix.

Non-square matrix determinant. Expects an exception because the determinant is only defined for square matrices.

#### 4.2.3.13 TEST\_CASE() [13/35]

```
TEST_CASE (
    "Matrix exponentiation when an element is 0 and exponent is <= 0" )
```



**4.2.3.14 TEST\_CASE() [14/35]**

```
TEST_CASE (
    "Matrix exponentiation with scalar 0 returns element-wise 1 (nonzero elements)"
)
```

**4.2.3.15 TEST\_CASE() [15/35]**

```
TEST_CASE (
    "Matrix exponentiation with scalar 1 returns the same matrix" )
```

**4.2.3.16 TEST\_CASE() [16/35]**

```
TEST_CASE (
    "Matrix exponentiation with scalar 2 returns element-wise square" )
```

**4.2.3.17 TEST\_CASE() [17/35]**

```
TEST_CASE (
    "Matrix Horizontal and Vertical Stacking" )
```

Tests for [Matrix](#) horizontal and vertical stacking.

Horizontal stacking. Verifies that concatenating matrices side-by-side produces the expected result.

Vertical stacking. Verifies that concatenating matrices top-to-bottom produces the expected result.

Stacking with mismatched dimensions. Ensures that attempting to stack matrices with incompatible dimensions throws an exception.

**4.2.3.18 TEST\_CASE() [18/35]**

```
TEST_CASE (
    "Matrix Inequality Operator" )
```

**4.2.3.19 TEST\_CASE() [19/35]**

```
TEST_CASE (
    "Matrix Inverse" )
```

Tests for [Matrix](#) inversion.

Inversion of a 2x2 invertible matrix. Checks that the computed inverse matches the expected result.

Inversion of a singular matrix. Verifies that attempting to invert a non-invertible matrix throws an exception.

#### 4.2.3.20 TEST\_CASE() [20/35]

```
TEST_CASE (
    "Matrix Multiplication and Division" )
```

Tests for [Matrix](#) multiplication and division.

[Matrix](#) multiplication (matrix \* matrix). Verifies that the product of two matrices is as expected.

Scalar multiplication (matrix \* k and k \* matrix). Checks that scalar multiplication scales each element appropriately.

[Matrix](#) division by scalar. Tests division by a scalar and expects an exception when dividing by zero.

#### 4.2.3.21 TEST\_CASE() [21/35]

```
TEST_CASE (
    "Matrix Row and Column Insertion" )
```

Tests for [Matrix](#) row and column insertion methods.

Insert row with a provided vector.

Insert row filled with a scalar value.

Insert column with a provided vector.

Insert column filled with a scalar value.

#### 4.2.3.22 TEST\_CASE() [22/35]

```
TEST_CASE (
    "Matrix shuffleRows methods" )
```

#### 4.2.3.23 TEST\_CASE() [23/35]

```
TEST_CASE (
    "Matrix Submatrix Extraction" )
```

Tests for [Matrix](#) submatrix extraction functionality using exclusive indices.

#### 4.2.3.24 TEST\_CASE() [24/35]

```
TEST_CASE (
    "Matrix Transpose" )
```

Tests for [Matrix](#) transposition.

**4.2.3.25 TEST\_CASE() [25/35]**

```
TEST_CASE (
    "Matrix::constValMatrix creates a constant matrix" )
```

**4.2.3.26 TEST\_CASE() [26/35]**

```
TEST_CASE (
    "Matrix::constValMatrix throws for zero dimensions" )
```

**4.2.3.27 TEST\_CASE() [27/35]**

```
TEST_CASE (
    "Non-vector matrix: sum throws invalid_argument" )
```

**4.2.3.28 TEST\_CASE() [28/35]**

```
TEST_CASE (
    "Row matrix: valid sum calculation with positive power" )
```

**4.2.3.29 TEST\_CASE() [29/35]**

```
TEST_CASE (
    "Row matrix: zero element with non-positive power throws runtime_error" )
```

**4.2.3.30 TEST\_CASE() [30/35]**

```
TEST_CASE (
    "Row vector: sum and mean" )
```

**4.2.3.31 TEST\_CASE() [31/35]**

```
TEST_CASE (
    "Single Row and Single Column Submatrix Extraction" )
```

Tests for submatrix extraction of a single row and a single column.

Extract the second row as a 1x3 matrix. Using row indices [1,2) and column indices [0,3).

Extract the third column as a 3x1 matrix. Using row indices [0,3) and column indices [2,3).

**4.2.3.32 TEST\_CASE()** [32/35]

```
TEST_CASE (
    "Testing Matrix::extractCol" )
```

**4.2.3.33 TEST\_CASE()** [33/35]

```
TEST_CASE (
    "Testing Matrix::extractRow" )
```

**4.2.3.34 TEST\_CASE()** [34/35]

```
TEST_CASE (
    "Testing Matrix::identity" )
```

**4.2.3.35 TEST\_CASE()** [35/35]

```
TEST_CASE (
    "trace() with vector-of-vector constructor for square matrices" )
```