1 Introduction

EASYMAT++ is a C++ template class library for scientific computing. It offers a high level of abstraction, but performance which rivals Fortran. The current version supports matrices and vectors (1 and 2-dimensional matrices). It has been developed upon the Numerics package used in the Image Understanding Environment (IUE) program package. The "original" Numerics package has the following classes:

- Classes for Matrix Decompositions

  *SVD* - Compute and cache the SVD of a Matrix.
  *BaseSVD* - Compute and cache the SVD of a Matrix.
  *GeneralizedEigensystem* - Symmetric-definite generalized eigensystem
  *Eigensystem* – Unsymmetric real eigensystem.
  *SymmetricEigensystem* - Solve $A x = \lambda x$ using QR.
  *Cholesky* - Decomposition of symmetric matrix
  *QR* - QR decomposition Optimisation
LevenbergMarquardt - Levenberg Marquardt non-linear least squares
Amoeba - Nelder-Mead downhill simplex.
CostFunction - Vector->Real Function
LeastSquaresFunction - Vector->Vector Function

- Classes for Fourier Transform

FFT1D - Fourier transform (1d)
FFT2D - Fourier transform (2d)

- Classes for Fast fixed-size matrices and vectors

MatrixFixed - Fixed size matrix
MatrixFixedRef - Fixed size stack-stored IUE_matrix.
MatrixRef - IUE_matrix reference to user-supplied storage.
VectorFixed - Fixed length stack-stored IUE_vector.
VectorRef - IUE_vector using user-supplied storage
Double2 - IUE_vector of 2 doubles.
Double3 - IUE_vector of 3 doubles.
Double4 - IUE_vector of 4 doubles.
Double2x2 - 2x2 Matrix of double
Double3x3 - 3x3 Matrix of double
Double3x4 - 3x4 Matrix of double
Double4x4 - 4x4 Matrix of double
Float3 - IUE_vector of 3 floats.
LinearOperators3 - 3D linear algebra operations
Polynomials
RealPolynomial - Evaluation of real polynomials.
RPolyRoots - Roots of real polynomial.

- A Class for More speed

FastOps – Fast operations

- Classes for Additional matrix operations

MatOps - A collection of Matrix operations.
Transpose - Efficient matrix transpose.
MatrixInverse - MatrixInverse via SVD

- Classes for C++ template helpers

numeric_limits - Standard limits for numeric datatypes
Special Matrices
DiagMatrix - Time/space efficient diagonal matrix
Outer-Product
CrossProductMatrix - 3x3 cross-product matrix of vector
GaussianKernel1D - 1D discrete Gaussian.
FileMatrix - Load DoubleMatrix from file.
IntMatrix - Matrix of ints.
1.1 Why using \textit{EASYMAT++}?

The idea was to build a computationally efficient, general-purpose, abstract, and easy-to-use tool to ease the implementations of scientific computing applications, which need to solve both Linear Algebra and Matrix Manipulation problems. Using the \textit{Numerics} classes directly is problematic, because someone unfamiliar with these classes won't stand a chance of understanding what a certain attribute means or a certain member function does.

In stead of worrying about all different classes above, I mainly extended the \textit{IUE\_matrix} class to be able to ”automatically” use other needed classes. Many additional constructors, operators and methods were added to the \textit{IUE\_matrix} class. Furthermore, modifications of some old methods, of the \textit{IUE\_matrix} class, were done.

The other classes were also modified to make it easier for the \textit{IUE\_matrix} class to use them efficiently. See the next sections for more details.

Note that all ”old” class names are preserved, and that’s why most of them begin with the prefix ”\textit{IUE}”.

1.2 Compiling programs with \textit{EASYMAT++}

Currently, only the Digital C++ compiler, \textit{DECcxx}, is supported.

\textbf{1.2.1 Directories}

- \texttt{EASYMAT++/math} \textit{EASYMAT++} headers and source files
- \texttt{EASYMAT++/Numerics} More \textit{EASYMAT++} headers and source files
- \texttt{EASYMAT++/Examples} Example programs

\textbf{1.2.2 Header files}

The \textit{EASYMAT++} library follows the X-windows style convention for header files. All headers are referred to with a prefix of either ”\texttt{Numerics/}” or ”\texttt{math/}”. For example, to use the \textit{IUE\_matrix<T>} class, one needs to include \texttt{<math/matrix.h>} instead of just \texttt{<matrix.h>}. To make this work, the main \textit{EASYMAT++} directory must be in your include path. For example, if \textit{EASYMAT++} was installed in \texttt{/software/EASYMAT++}, you will need to compile with: \texttt{-I /software/EASYMAT++}

\textbf{1.2.3 Explicit instantiation}

It is not possible to do explicit instantiation of \textit{EASYMAT++} matrices. The reason is that explicit instantiation results in all members of a class template being instantiated. This is \textit{not} the case for implicit instantiation, in which only required members are instantiated. The \textit{IUE\_matrix<T>} class contains members which are not valid for all types \texttt{T}: for example, the binary \texttt{AND} operation \&= is non-sensical if \texttt{T = double}. If you attempt to explicitly instantiate a matrix class, e.g. \texttt{template class IUE\_matrix<double>}, then you will be rewarded with many compile errors, due to methods such as \&= which are non-sensical for double.
As some consolation, explicit instantiation would not be much help with \textit{EASYMAT++} matrices.

\section*{2 The IUE\textunderscore matrix\textless T\textgreater Class}

\subsection*{2.1 Getting started}

The IUE\textunderscore matrix\textless T\textgreater class provides a dynamically allocated matrix (2-dimensional array), with flexible expression handling, sub-matrices and many other useful features.

\subsection*{2.1.1 Description of IUE\textunderscore matrix\textless T\textgreater}

The IUE\textunderscore matrix\textless T\textgreater class implements two-dimensional arithmetic matrices for a user-specified numeric data type. The IUE\textunderscore matrix\textless T\textgreater class takes one template parameter T, which is the numeric type to be stored in the matrix. T can be an integral type (bool, char, unsigned char, short int, short unsigned int, int, unsigned int, long, unsigned long), floating point type (double, long double) complex type (complex\textless double\textgreater, complex\textless long double\textgreater) or any user-defined type with appropriate numeric semantics. Using the parameterized types facility of C++, it is possible, for example, for the user to create a matrix of rational numbers by parameterizing the IUE\textunderscore matrix class over the Rational class. The only requirement for the type is that it supports the basic arithmetic operators. Note that unlike the other sequence classes, the IUE\textunderscore matrix\textless T\textgreater class is fixed-size. It will not grow once the size has been specified to the constructor or changed by the assignment or multiplication operators. The IUE\textunderscore matrix\textless T\textgreater class is row-based with addresses of rows being cached, and elements accessed as m[row][col]. Indexing of the matrix is zero-based, so the top-left element is m(0,0).

\subsection*{2.2.2 Usage of IUE\textunderscore matrix\textless T\textgreater}

To use the IUE\textunderscore matrix\textless T\textgreater class, include the header <math\textunderscore matrix\textgreater.h>

\begin{verbatim}
#include <math\textunderscore matrix\textgreater.h>

IUE\textunderscore matrix\textless int\textgreater x(1, 10); // A one-dimensional (1x10) matrix of int is created

IUE\textunderscore matrix\textless double\textgreater y(4, 6); // A two-dimensional (4x6) matrix of double is created

IUE\textunderscore matrix\textless double\textgreater z(y); // A copy of y is created
\end{verbatim}

The contents of a newly-created matrix are garbage. To initialise the matrix, you can write:

\begin{verbatim}
y = 0;
\end{verbatim}

and all the elements of the matrix y will be set to zero. If the contents of the matrix are known, you can initialise it, for example, in the following way:
double values[] = {
    1, 0, 0, 0,
    0, 1, 0, 0,
    0, 0, 1, 0,
    0, 0, 0, 1};

y = IUE_matrix<double> (values, 4, 4);

this code excerpt sets y equal to a (4x4) identity matrix. Note that the dimensions of the matrix y are changed after the assignment statement.

2.2 Matrix types

The IUE_matrix<T> class supports a variety of matrices:

- Matrices of scalar types, such as IUE_matrix<int> and IUE_matrix<double>
- Complex matrices, such as IUE_matrix<complex<double>>
- Matrices of user-defined types. If you have a class called Polynomial, then IUE_matrix<Polynomial> is a matrix of Polynomial objects. Note that the used type must support the basic arithmetic operators.

2.2.1 Storage format in EASYMAT++

The storage format is row-major, C-style matrices whose indices start at zero. Each matrix contains a protected data section that has a T** slot that points to the physical memory allocated for the two-dimensional array. In addition, two integers specify the number of rows and columns for the matrix. These values are provided in the constructors.

2.2.2 A simple example

Here's an example program which creates two 3x3 matrices, initialises them, and adds them:

```cpp
#include <math/matrix.h>

int main()
{
    IUE_matrix<double> A(3,3), B(3,3), C(3,3);

    double A_values[] = {
        1, 0, 0,
        2, 2, 2,
        1, 0, 0};

    double B_values[] = {
        0, 0, 7,
        0, 0, 0,
        0, 0, 0};

    C = A + B;
}
```
0, 8, 0,
9, 9, 9};
A = IUE_matrix<double>(A_values, 3, 3);
B = IUE_matrix<double>(B_values, 3, 3);
C = A + B;
cout << "A = \n" << A << endl
<< "B = \n" << B << endl
<< "C = \n" << C << endl;
return 0;

2.3 Constructors

2.3.1 Default constructor

IUE_matrix<T>();
The default constructor creates a matrix of zero size (an empty matrix). Any attempt to access data in the matrix may result in a run-time error, because there isn't any data to access!

2.3.2 Constructors which take dimension parameters

IUE_matrix<T>(num of rows, num of columns);
Creates a matrix with given number of rows and columns. Elements are not initialized (contain garbage).

2.3.2.1 Constructors with extra arguments

IUE_matrix<T>(num of rows, num of columns, value);
IUE_matrix<T>(text, num of rows, num of columns);
The first constructor creates a matrix with given number of rows and columns, and initialize all elements to value. Usage example:

IUE_matrix<double> ONES(10, 10, 1);

This creates the (10x10) matrix, ONES, with ones.
The second one creates a matrix with given number of rows and columns, and initialise all elements to "random" values between 0.0 and 1.0. Usage example:

```cpp
IUE_matrix<double> RAND("random", 50, 100);
```

Creates the (50x100) matrix, RAND, with random numbers.

### 2.3.3 The Copy Constructor

#### 2.3.3.1 Copying a matrix, a deep copy, or referencing a sub-matrix, a shallow copy

```cpp
IUE_matrix<T>(matrix or submatrix);
```

Creates a new matrix and copies all the elements of from matrix if we don't have a "shallow copy"-matrix, i.e. a sub-matrix. Otherwise the new matrix will also be a "shallow copy"-matrix (a sub-matrix).

This constructor makes a shared view of another sub-matrix's data. After this constructor is used, both matrix objects refer to the same data. Any changes made to one matrix will appear in the other matrix. If you want to make a duplicate copy (a deep copy) of a sub-matrix, use the "=" operator, or the overloaded copy constructor:

```cpp
IUE_matrix<T>(matrix or sub-matrix, text);
```

Usage example: `IUE_matrix<int> B(A[1][2][1][2], "deep");`

The copy constructor is automatically used at initialisations and returning values (not references) from member functions or methods.

#### 2.3.3.2 Creating a matrix from an expression

```cpp
IUE_matrix<T>(matrix expression);
```

You may create a matrix from a matrix expression. For example,

```cpp
IUE_matrix<double> A(4,3), B(4,3);
IUE_matrix<double> C(A*2.0+B);
```

This is an explicit constructor (it will not be used to perform implicit type conversions). The newly constructed matrix will have the same storage format as the result of the expression.

### 2.3.4 Creating a matrix from pre-existing data

#### 2.3.4.1 From arrays

We can create a matrix using a pointer to already existing data, by e.g. using:

```cpp
IUE_matrix<T>(num of rows, num of columns, num of elements n, 1-D Array of values);
IUE_matrix<T>(pointer to data block, num of rows, num of columns);
```
The first one creates a matrix with given dimension (rows, cols) and initialise the first n elements, row-wise, to values from the specified automatic array. While the second constructor creates a matrix from a block array of data, stored row-wise.

### 2.3.4.2 FROM IPAD-IMAGES

```
IUE_matrix(IMAGE);
IUE_matrix(IMAGE, band number);
IUE_matrix(IMAGE, start band, end band);
```

- The first one creates a 2-D matrix from an N-dimensional IMAGE the rows of the Matrix point at the channels or bands of the IMAGE. Note that NO copying of data is performed.
- The second constructor creates a 2-D matrix from one channel or band of an IMAGE. The rows of Matrix point at the chosen channel'n's rows (of IMAGE).
- The third constructor creates a 2-D matrix from a piece of an N-D IMAGE from 'start_band' to 'end_band' in the IMAGE (used to be a big multi-dimensional ipad-image). The rows of the Matrix point at some bands of the IMAGE.

### 2.3.5 A note about id-numbers

`EASYMAT++` matrices use id-numbers. When you create a new matrix, a memory block is allocated, with a unique id-number. The matrix object acts like a handle for this memory block. A memory block can be shared among multiple matrix objects, when you take sub-matrices. All sub-matrices, of the same matrix, have the same id-number. A memory block can be deleted only if just one matrix object refers to it. The id-number is a great help to know if a matrix operation (a statement or an expression) uses sub-matrices of the same matrix (i.e. can be overlapped) or not (i.e. safe, no risk for overlapping).

### 2.4 Indexing and sub-matrices

This section describes how to access the elements of a matrix. There are two main ways of direct access to matrix elements:

- **Indexing** obtains a single element
- Creating a **sub-matrix** which refers to a smaller portion of a matrix

Indexing and sub-matrices use the overloaded parenthesis `operator()` and `operator[]` respectively.

As a running example, we'll consider the two-dimensional (4×4) matrix, A below. Bounded portions of the matrix show regions which have been obtained by indexing and creating a sub-matrix.
2.4.1 Indexing
The way to get a single element from a matrix is to provide a set of integer
operands to operator():

\[
A(3, 0) = 5; \\
cout << "A(3, 0) = " << A(3, 0) << endl;
\]

this will view the element with indices (3, 0), e.g. on column 1 of row 4.

\[
\begin{array}{cccc}
2 & 4 & 1 & 7 \\
12 & 5 & 11 & 9 \\
0 & 8 & 2 & 7 \\
5 & 3 & 1 & 6 \\
\end{array}
\]

2.4.2 Sub-matrices
You can obtain a sub-matrix by providing range operands to operator[]. For
example,

\[
cout << "A(1..3, 1..2) = " << A[1][3][1][2] << endl;
\]

The object \(A[1][3][1][2]\) refers to elements with indices \((1..3, 1..2)\)
of the matrix \(A\). Sub-matrices of the same matrix, \(A\), can be used on both sides of
an assignment expression even if they overlap. If the sub-matrices overlap, copies
of them are created before performing operations using these sub-matrices.

2.4.3 A note about assignment and sub-matrices
The assignment operator "=" always results in the expression on the right-hand
side (rhs) being copied to the lhs (i.e. the data on the lhs is over-written with the
result from the rhs). This is different from some matrix packages in which the
assignment operator makes the lhs a reference (or alias) to the rhs. To further
confuse the issue, the copy constructor for matrices does have reference semantics.
Here's an example which should clarify things:

```
IUE_matrix<int> A(3, 3), B(6, 6);
A = B[0][2][3][5];    // Statement 1
IUE_matrix<int> C = B[0][1][3][4]; // Statement 2
IUE_matrix<int> D = B[0][1][3][4]); // Statement 3
IUE_matrix<int> E = B; // Statement 4
IUE_matrix<int> F = B); // Statement 5
```

Statement 1 results in a portion of B's data being copied into A, and matrices A
and B have their own (non-overlapping) blocks of data. Contrast this behaviour
with that of Statements 2 and 3, which are not assignments (the copy constructor is
used here). After Statements 2 and 3 are executed, the matrices C and D are
references (or aliases) to the same portion of B's data (a sub-matrix).
If no sub-matrix is used in the statement, as in Statements 4 and 5, we get "deep" copies of the rhs. So, matrices E, F, and B have their own (non-overlapping) blocks of data (exactly with the same values, of course).
So to summarize: If you want to copy the rhs (which is a sub-matrix), use an assignment operator. If you want to reference (or alias) the rhs (which is a sub-matrix), use the copy constructor.

2.4.4 An example

```cpp
#include <math/matrix.h>

int main()
{
    IUE_matrix<int> A(6,6);
    // Set the upper left quadrant of A to 5
    A[0][2][0][2] = 5;
    // Set the upper right quadrant of A to
    // an identity matrix
    int B[] = {1, 0, 0,
              0, 1, 0,
              0, 0, 1};
    A[0][2][3][5] = IUE_matrix<int> (B, 3, 3);
    // Set the fourth row to 1
    A[3][3] = 1;
    // Set the last two rows to 0
    A[4] = 0;
    // Set the bottom right element to 8
    A(5,5) = 8;
    Cout << "A = 
" << A << endl;
    Return 0;
}
```

The output:

```
A =
  5  5  5  1  0  0
  5  5  5  0  1  0
  5  5  5  0  0  1
  1  1  1  1  1  1
  0  0  0  0  0  0
  0  0  0  0  0  8
```

2.5 Member functions and overloaded operators
The \texttt{IUE\_matrix<T>} class consists of many member functions and overloaded operators. Most of them were defined and added to the \texttt{IUE\_matrix<T>} class by the author, as a try to make the functionality of this class as complete as possible.

\subsection*{2.5.1 Old member functions descriptions}

\textit{Numerics} has the following useful "original" member functions:

\begin{verbatim}
inline unsigned long rows () const;
Return number of rows.

inline unsigned long columns () const { return num_cols; }  
inline unsigned long cols () const { return num_cols; }  
Return number of columns.

inline void put (unsigned long r, unsigned long c, const T&);  
Assign a value: Puts value into element at specified row and column. Checks for valid range of indices.

inline T get (unsigned long r, unsigned long c) const;
Get value: Returns the value of the element at specified row and column. Checks for valid range of indices.

void fill (const T&);  
void fill_diagonal (const T&);  
Fill with a value: Sets all (diagonal) elements of matrix to specified value.

\texttt{IUE\_matrix<T> transpose () const;}
Transpose row/column: Returns new matrix with rows and columns transposed.

void inplace_transpose();
Transpose row/column in place: transposes the THIS matrix.

T maxVal () const;
Returns max of all elements of THIS.

friend
\texttt{\texttt{IUE\_matrix<T> element\_product (const IUE\_matrix<T>&,
                     const IUE\_matrix<T>&);}  
Array multiply, element wise multiplication. Returns new matrix whose elements are the products m1[i][j]*m2[i][j].  
(Return: new matrix = A .* B , (Matlab syntax))}

friend
\texttt{\texttt{IUE\_matrix<T> element\_quotient (const IUE\_matrix<T>&,
                       const IUE\_matrix<T>&);}  
Right array divide, element wise division. Returns new matrix whose elements are the quotients m1[i][j]/m2[i][j].  
(Return: new matrix = A ./ B , (Matlab syntax))}

friend
\texttt{\texttt{T dot\_product (const IUE\_matrix<T>& m1,
                       const IUE\_matrix<T>& m2);}  
Returns the dot product of the two matrices, which is the sum of all pair-wise product of the elements m1[i,j]*m2[i,j].}
\end{verbatim}
2.5.1 Old Overloaded Operators descriptions

*Numerics* has the following useful "original" operators:

```cpp
inline T& operator() (unsigned long r);
inline T& operator() (unsigned long r, unsigned long c);
inline const T& operator() (unsigned long r) const;
inline const T& operator() (unsigned long r,
                           unsigned long c) const;
```

Overloaded operators `operator()` with (constant) matrix to return a (constant) reference to element at specified indexes. No range check is performed.

```cpp
IUE_matrix<T>& operator= (const T&);
Assigns value to all elements of a matrix. Assignment: m = value;
```

```cpp
IUE_matrix<T>& operator= (const IUE_matrix<T>&);
Copies all elements of rhs matrix into lhs matrix. If needed, the arrays in lhs matrix are freed up, and new arrays are allocated to match the dimensions of the rhs matrix. Assignment: mat1 = mat2;
```

```cpp
IUE_matrix<T>& operator+= (const T&);
IUE_matrix<T>& operator*= (const T&);
IUE_matrix<T>& operator/= (const T&);
inline IUE_matrix<T>& operator-= (const T&);
```

Binary operation and assignment. Mutates lhs matrix to add/multiply/divide/subtract in place, all its elements with value.

```cpp
IUE_matrix<T>& operator+= (const IUE_matrix<T>&);
IUE_matrix<T>& operator-= (const IUE_matrix<T>&);
inline IUE_matrix<T>& operator*= (const IUE_matrix<T>&);
```

Matrix operation and assignment. Adds/Subtracts/Multiplies lhs matrix with rhs matrix, then assigns the product to lhs matrix. The dimensions of the two matrices must match.

```cpp
IUE_matrix<T> operator- () const;
IUE_matrix<T> operator+ (const T&) const;
IUE_matrix<T> operator* (const T&) const;
IUE_matrix<T> operator/ (const T&) const;
inline IUE_matrix<T> operator- (const T&) const;
```

Negation (of THIS matrix) and all binary operations return by values. All binary operations return new matrix with elements of lhs matrix added with/multiplied by/divided by/subtracted value.

```cpp
friend
IUE_matrix<T> operator- (const T&, const IUE_matrix<T>&);
friend
inline IUE_matrix<T> operator+ (const T&,
                                const IUE_matrix<T>&);

friend
inline IUE_matrix<T> operator* (const T&,
                                const IUE_matrix<T>&);
```

Friend functions. Return: new matrix = value <op> old matrix, where <op> is "+", "-" or "*". The dimensions of the two matrices (lhs and rhs) must match.

```cpp
inline IUE_matrix<T> operator+ (const IUE_matrix<T>& rhs) const;
inline IUE_matrix<T> operator- (const IUE_matrix<T>& rhs) const;
```

Returns: new matrix = THIS <op> old matrix, where <op> is "+" or "-". The dimensions of the two matrices (lhs and rhs) must be identical.
IUE_matrix<T> operator* (const IUE_matrix<T>& rhs) const;
Returns new matrix which is the product of THIS with rhs, THIS * rhs. Number of columns of first matrix must match number of rows of second matrix.

bool operator== (const IUE_matrix<T>&) const;
Perform equality comparison test. Two matrices are equal if and only if they have the same dimensions and the same values. Elements are compared with operator== as default.

inline bool operator!= (const IUE_matrix<T>&) const;
Perform not equal comparison test.

friend
ostream& operator<< (ostream& os, const IUE_matrix<T>& m);
Prints the 2-D matrix of elements of a matrix out to a stream.

friend
istream& operator>> (istream& os, IUE_matrix<T>& m);
Reads an IUE_matrix from an ascii istream, automatically determining file size if the input matrix has zero size.

2.5.3 New member functions descriptions

• The following methods were modified (corrected?) by the author:

  IUE_numeric_traits<T>::abs_big_t one_norm() const;
  Return the maximum sum of absolute value of M's columns,
  (the 1 norm)

  IUE_numeric_traits<T>::abs_big_t inf_norm() const;
  Return the maximum sum of absolute values of M's rows, 
  (the infinity norm)

  IUE_numeric_traits<T>::abs_big_t fro_norm() const;
  Returns the root sum of squares of THIS's components. 
  (the Frobenius norm)

• The following methods (member functions) were added by the author:

  T minVal () const;
  Returns min of all elements of THIS.

  void inplace_flipud();
  Reverses in place order of rows. Name is from Matlab, meaning
  "flip upside down".

  void inplace_fliplr();
  Reverses in place order of columns. Name is from Matlab, meaning
  "flip left-side right".

  IUE_matrix<T> flipud();
  Returns a new matrix with reversed order of rows. Name is from
  Matlab, meaning "flip up-side down".

  IUE_matrix<T> fliplr();
  Returns a new matrix with reversed order of columns. Name is from
  Matlab, meaning "flip left-side right".

  IUE_matrix<T> sort(IUE_matrix<long>& index_vector,
int order = 0, int flag = 0) const;
Returns a new sorted matrix.
The corresponding indices of the original matrix, are placed in
the 1-D row `IUE_matrix"index_vector"`.
We have the following parameters:

order = 0  =>  Ascending order, default
order = 1  =>  Descending order

flag = 0  =>  Order columns by columns' sums, default
flag = 1  =>  Order rows by rows' sums

bool is_complex() const;
Return true if THIS is of complex type.

IUE_matrix<double> abs() const;
Returns a new Matrix of absolute values of THIS’ elements.

IUE_matrix<double> angle() const;
Returns a new Matrix of the angles of THIS’ elements.

IUE_matrix<double> real() const;
Returns a new Matrix of the real parts of THIS’ elements.

IUE_matrix<double> imag() const;
Returns a new Matrix of the imaginary parts of THIS’ elements.

IUE_matrix<T> conj() const;
Returns a new matrix with the conjugates of THIS’ elements.

bool is_symmetric();
Returns true if the matrix is symmetric.

IUE_matrix<T> make_symmetric();
Returns a new symmetric matrix.

void inplace_make_symmetric();
Convert into symmetric matrix.

IUE_numeric_traits<T>::big_t sum() const;
Return the sum of the elements of THIS.

IUE_matrix<IUE_numeric_traits<T>::big_t> cols_sum() const;
Return the sum of the columns of THIS. (return a row-vector)

IUE_matrix<IUE_numeric_traits<T>::big_t> rows_sum() const;
Return the sum of the rows of THIS. (return a column-vector)

IUE_matrix<T> normalize_to_zero_mean_unit_var() const;
Normalize to zero mean and unit variance: Returns a normalized
new matrix with zero mean and unit variance.

T IUE_matrix<T>::mean() const;
Return the mean of the elements of THIS.

IUE_matrix<T> remove_mean() const;
Return: (new matrix) = (old matrix) - (mean of old matrix
elements)

IUE_matrix<T> remove_rows_mean() const;
Return: (new matrix) = (old matrix rows) - (mean of old matrix rows)

    IUE_matrix<T> remove_cols_mean() const;
Return: (new matrix) = (old matrix cols) - (mean of old matrix cols)

    void remove_mean_inplace();
(old matrix) = (old matrix) - (mean of old matrix elements)

    void remove_rows_mean_inplace();
( old matrix) = ( old matrix rows) - (mean of old matrix rows)

    void remove_cols_mean_inplace();
( old matrix) = ( old matrix cols) - (mean of old matrix cols)

    IUE_matrix<T> covariance(int);
Returns the Covariance Matrix (a new matrix).
if THIS is a vector, returns the variance. For matrices, where
each row is an observation, and each column a variable, the
covariance matrix is returned.
covariance() normalizes by (N-1) where N is the number of
observations. This makes it the best unbiased estimate of the
covariance matrix if the observations are from a normal
distribution.
covariance(1) normalizes by N and produces the second moment
matrix of the observations about their mean.
covariance(0) is the same as covariance().

    IUE_matrix<double> std_deviation(int);
Returns the Standard deviation (a new matrix).
For vectors, std_deviation() returns the standard deviation. For
matrices, std_deviation() is a row vector containing the standard
deivation of each column.
std_deviation() normalizes by (N-1) where N is the sequence
length. This makes std_deviation() .^ 2 the best unbiased
estimate of the variance if THIS is a sample from a normal
distribution.
std_deviation(1) normalizes by N and produces the second moment of
the sample about its mean. std_deviation(0) is the same as
std_deviation().

    IUE_matrix<double> variance(int);
Returns the Variance (a new matrix).
For vectors, variance() returns the variance of THIS. For
matrices, variance() is a row vector containing the variance of
each column of THIS.
variance() normalizes by N-1 where N is the sequence length. This
makes variance() the best unbiased estimate of the variance if
THIS is a sample from a normal distribution.
variance(1) normalizes by N and produces the second moment of the
sample about its mean.

    IUE_matrix<T> a_x_a() const;
Return: new matrix = A * A , (Matlab syntax)

    IUE_matrix<T> a_x_aT() const;
Return: new matrix = A * A' , (Matlab syntax)

    IUE_matrix<T> aT_x_a() const;
Return: new matrix = A' * A , (Matlab syntax)
IUE_matrix<T> a_x_aT_conj() const;
Return: new matrix = A * conj(A') , (Matlab syntax)

IUE_matrix<T> aT_conj_x_a() const;
Return: new matrix = conj(A') .* A , (Matlab syntax)

IUE_matrix<T> a_conj_dot_x_a() const;
Return: new matrix = conj(A) .* A , (Matlab syntax)

IUE_matrix<T> sub_matrix (long start_r, long end_r,
   long start_c, long end_c) const;
Returns a sub-matrix (the same result as operator[]).
Usage examples:
  sub_mat = MAT.sub_mat(1, 3, 2, 5);   // rows[1..3], cols[2..5]
  sub_mat = MAT.sub_mat(-1, 3, 2, -1); // rows[0..3], cols[2..end]
  sub_mat = MAT.sub_mat(2, -1, -1, 3); // rows[2..end], cols[0..3]
  sub_mat = MAT.sub_mat(-1, -1, 2, 4); // rows[0..end], cols[2..4]
Range check is performed.

inline unsigned long id () const;
Returns the id-number of THIS.

inline bool shallowCopy () const;
Returns true if THIS is a shallow copy (i.e. a sub-matrix).

IUE_matrix<T> GT (const IUE_matrix<T>& M) const;
IUE_matrix<T> GE (const IUE_matrix<T>& M) const;
IUE_matrix<T> LT (const IUE_matrix<T>& M) const;
IUE_matrix<T> LE (const IUE_matrix<T>& M) const;
IUE_matrix<T> EQ (const IUE_matrix<T>& M) const;
IUE_matrix<T> NE (const IUE_matrix<T>& M) const;
Returns a new matrix = element wise comparison between THIS’s and
M’s elements. Here:
  GT = greater than, >, GE = greater or equal to, >=,
  LT = less than, <, LE = less or equal to, <=,
  EQ = equal to, ==, NE = not equal to, !=
If THIS’s element >, >=, <, <=, ==, or != the corresponding
element in M then the resulting element will be equal to the
corresponding element in THIS, otherwise it will be = 0.
(Lhs and rhs matrices must have the same dimensions!)

IUE_matrix<T> max (const IUE_matrix<T>& M) const;
IUE_matrix<T> min (const IUE_matrix<T>& M) const;
Returns a new matrix = element wise comparison between THIS’s and
M’s elements. The resulting element will be equal to the maximum /
minimum of the corresponding two elements in THIS and M.
(Lhs and rhs matrices must have the same dimensions!)

IUE_matrix<T> GT (const T& value) const;
IUE_matrix<T> GE (const T& value) const;
IUE_matrix<T> LT (const T& value) const;
IUE_matrix<T> LE (const T& value) const;
IUE_matrix<T> EQ (const T& value) const;
IUE_matrix<T> NE (const T& value) const;
Returns a new matrix = element wise comparison between THIS’s
elements and value. Here:
  GT = greater than, >, GE = greater or equal to, >=,
  LT = less than, <, LE = less or equal to, <=,
  EQ = equal to, ==, NE = not equal to, !=
If THIS's element >, >=, <, <=, ==, or != value then the resulting element will be equal to the corresponding element in THIS, otherwise it will be = 0.

\[
\text{IUE\_matrix<T> max (const T& value) const;}
\]
\[
\text{IUE\_matrix<T> min (const T& value) const;}
\]

Returns a new matrix = element wise comparison between THIS's elements and value. The resulting element will be equal to the maximum / minimum of the corresponding element in THIS and value.

\[
\text{inline void fast\_put (unsigned long r, unsigned long c, const T&);}
\]

Assign a value: Puts value into element at specified row and column. No range check of indices is performed.

\[
\text{inline T fast\_get (unsigned long r, unsigned long c) const;}
\]

Get value: Returns the value of the element at specified row and column. No range check of indices is performed.

\[
\text{friend IUE\_matrix<T> powN (const IUE\_matrix<T>& M, unsigned int power);} \\
\text{(The same as "operator^")}
\]

\[
\text{friend IUE\_matrix<double> element\_product (const IUE\_matrix<short int>&, const IUE\_matrix<double>&);} \\
\text{(Return: new matrix = A.*B, (Matlab syntax))}
\]

\[
\text{IUE\_matrix<T> select\_rows (const IUE\_matrix<long>& mask\_vector) const;}
\]

Returns a new matrix with the rows of THIS selected by the mask vector, which contains the needed rows' indices. The "mask_vector" must be a 1-D IUE_matrix.

\[
\text{IUE\_matrix<T> select\_cols (const IUE\_matrix<long>& mask\_vector) const;}
\]

Returns a new matrix with the columns of THIS selected by the mask vector, which contains the needed cols' indices. The "mask_vector" must be a 1-D IUE_matrix.

\[
\text{IUE\_matrix<T> orth();}
\]

Orthogonalization: \(Q = A.\text{orth}()\) Returns an orthonormal basis for the range of A. That is, \(Q'\cdot Q = I\), the columns of Q span the same space as the columns of A, and the number of columns of Q is the rank of A.

\[
\text{IUE\_matrix<T> inverse(int flag = 0);} \\
\text{Computes and returns the inverse of THIS, computed by using SVD (singular value decomposition).}
\]

flag != 0 => different method is used if THIS is symmetric. Default is : flag = 0
Part 4  EASYMAT++

IUE_matrix<T> t_inverse(int flag = 0);
Computes and returns the inverse of the transpose of THIS,
computed by using SVD (singular value decomposition).
flag != 0 => different method is used if THIS is symmetric.
Default is : flag = 0

IUE_matrix<double_complex> eigenvectors();
Computes and returns the eigenvectors (in the columns of the
double_complex resulting IUE_matrix).
Note: can handle double IUE matrices only.

DiagMatrix<double_complex> eigenvalues();
Computes and returns the eigenvalues (on the diagonal of the
double_complex resulting DiagMatrix).
Note: can handle double IUE matrices only.

double residual();
Returns the residual = infinity norm of ( X*V - V*D )
where : [V, D] = eig(X) , (Matlab syntax).
V contains the eigenvectors
D contains the eigenvalues

double norm();
Returns the norm of THIS:
if THIS is a vector -> Return the Frobenius norm.
if THIS is a matrix -> Return the largest singular value of THIS.

int rank();
Returns the rank of THIS.

double abs_det();
Returns the absolute value of the determinant of THIS.

IUE_matrix<T> enhance_contrast(double percent = 0.05,
bool cut = true) const;
Return: new matrix = old matrix without 5% of it's extreme values
We have the following parameters:

percent = 0.05 (5%) by default

cut = true  =>  (default) cut the extreme values
cut = false  =>  zeroing the extreme values

2.4.3.1 Mathemtical Functions, Element wise functions on THIS

IUE_matrix<T> abs_fcn () const;
Returns the absolute values (magnitudes) of THIS's elements.

IUE_matrix<T> sqr_mag_fcn () const;
Returns the squared magnitudes of THIS's elements.

IUE_matrix<T> exp_fcn () const;
Returns "e" to the power of the elements of THIS.

IUE_matrix<T> tanh_fcn () const;
Returns the hyperbolic tangent of THIS's elements.

IUE_matrix<T> sin_fcn () const;
Returns the SIN of THIS's elements.
IUE_matrix<T> cos_fcn () const;
Returns the COS of THIS's elements.

IUE_matrix<T> log_fcn () const;
Returns the natural (base e) logarithm, ln, of THIS's elements.

IUE_matrix<T> log10_fcn () const;
Returns the common (base 10) logarithm, LOG, of THIS's elements.

IUE_matrix<T> pow_fcn (const double &x) const;
Returns THIS's elements raised to a floating-point exponent x.

IUE_matrix<T> sqr_fcn () const;
Returns THIS's elements raised to 2 (i.e. the squares of THIS's elements).

IUE_matrix<T> sqrt_fcn () const;
Returns the square roots (sqrt) of THIS's elements.

2.5.4 New Overloaded Operators descriptions

The following overloaded operators were added by the author:

IUE_matrix<T> operator- () const;
Returns new matrix with rows and columns transposed.

IUE_matrix<T> operator[] (long index);
Returns a sub-matrix.
Usage: sub_mat = MAT[r1][r2][c1][c2]
Usage examples: sub_mat = MAT[_][r2][c1][_]
( the same as: MAT[_][r2][c1] )
sub_mat = MAT[_][r2][_][c2]
sub_mat = MAT[_][r2][][]
( the same as: MAT[_][r2] )
sub_mat = MAT[r1][_][_][]
( the same as: MAT[r1] )
Range check is performed.

IUE_matrix<T> operator[] (char* chr);
Returns a sub-matrix.
Usage: sub_mat = MAT[r1][r2][c1][c2]
Usage examples: sub_mat = MAT["":""] [r2][c1][":"]
( the same as: MAT["":""][r2][c1] )
sub_mat = MAT["":""][r2]["":""]
( the same as: MAT["":""][r2] )
sub_mat = MAT[r1]["":""]["":""]
( the same as: MAT[r1] )
No range check is needed here.
( it is done in the other overloaded operator[] )

IUE_matrix<T> operator^ (unsigned int power) const;
Return: new matrix = THIS .^ power
EASYMAT++

(Array power, element wise) , The same as "powN()".

\textbf{IUE\_matrix\textless T\textgreater } \textbf{operator!} () const;
Returns "the negative" of the matrix values.

\textbf{IUE\_matrix\textless short\ int\textgreater } \textbf{operator&&} (\textbf{const IUE\_matrix\textless T\&} \textbf{&} \textbf{M}) \textbf{const};
Returns a "short int" matrix = element wise && (logical AND) of
THIS and M. (The result consists of 0’s and 1’s)

\textbf{IUE\_matrix\textless short\ int\textgreater } \textbf{operator||} (\textbf{const IUE\_matrix\textless T\&} \textbf{&} \textbf{M}) \textbf{const};
Returns a "short int" matrix = element wise || (logical OR) of
THIS and M. (The result consists of 0’s and 1’s)

\textbf{IUE\_matrix\textless short\ int\textgreater } \textbf{operator>} (\textbf{const IUE\_matrix\textless T\&} \textbf{&} \textbf{M}) \textbf{const};
Returns a "short int" matrix = element wise > comparison of THIS
and M elements. (The result consists of 0’s and 1’s; 1 if THIS’s
element > the corresponding element in M, 0 otherwise)
Lhs and rhs matrices must have the same dimensions!

\textbf{IUE\_matrix\textless short\ int\textgreater } \textbf{operator>} (\textbf{const \ T\&} \textbf{value}) \textbf{const};
Returns a "short int" matrix = element wise > comparison of THIS
elements and value. (The result consists of 0’s and 1’s; 1 if
THIS’s element > value, 0 otherwise)

\textbf{IUE\_matrix\textless short\ int\textgreater } \textbf{operator<} (\textbf{const IUE\_matrix\textless T\&} \textbf{&} \textbf{M}) \textbf{const};
Returns a "short int" matrix = element wise < comparison of THIS
and M elements. (The result consists of 0’s and 1’s; 1 if THIS’s
element < the corresponding element in M, 0 otherwise)
Lhs and rhs matrices must have the same dimensions!

\textbf{IUE\_matrix\textless short\ int\textgreater } \textbf{operator<} (\textbf{const \ T\&} \textbf{value}) \textbf{const};
Returns a "short int" matrix = element wise < comparison of THIS
elements and value. (The result consists of 0’s and 1’s; 1 if
THIS’s element < value, 0 otherwise)

\textbf{IUE\_matrix\textless short\ int\textgreater } \textbf{operator>=} (\textbf{const IUE\_matrix\textless T\&} \textbf{&} \textbf{M}) \textbf{const};
Returns a "short int" matrix = element wise \textasciitilde{>=} comparison of
THIS and M elements. (The result consists of 0’s and 1’s; 1 if
THIS’s element \textasciitilde{>=} the corresponding element in M, 0 otherwise)
Lhs and rhs matrices must have the same dimensions!

\textbf{IUE\_matrix\textless short\ int\textgreater } \textbf{operator>=} (\textbf{const \ T\&} \textbf{value}) \textbf{const};
Returns a "short int" matrix = element wise \textasciitilde{>=} comparison of
THIS elements and value. (The result consists of 0’s and 1’s; 1 if
THIS’s element \textasciitilde{>=} value, 0 otherwise)

\textbf{IUE\_matrix\textless short\ int\textgreater } \textbf{operator<=} (\textbf{const IUE\_matrix\textless T\&} \textbf{&} \textbf{M}) \textbf{const};
Returns a "short int" matrix = element wise "\textasciitilde{<=}" comparison of
THIS and M elements. (The result consists of 0’s and 1’s; 1 if
THIS’s element \textasciitilde{<=} the corresponding element in M, 0 otherwise)
Lhs and rhs matrices must have the same dimensions!

\textbf{IUE\_matrix\textless short\ int\textgreater } \textbf{operator<=} (\textbf{const \ T\&} \textbf{value}) \textbf{const};
Returns a "short int" matrix = element wise "\textasciitilde{<=}" comparison of
THIS elements and value. (The result consists of 0’s and 1’s; 1 if
THIS’s element \textasciitilde{<=} value, 0 otherwise)

\textbf{IUE\_matrix\textless short\ int\textgreater } \textbf{operator!=} (\textbf{const \ T\&} \textbf{value}) \textbf{const};
Returns a "short int" matrix = element wise "!=" (not equal)
comparison of THIS elements and value. (The result consists of 0’s
and 1’s; 1 if THIS’s element != value, 0 otherwise)
IUE_matrix<short int> operator== (const T& value) const;
Returns a “short int” matrix = element wise “==” (equal)
comparison of THIS elements and value. (The result consists of 0’s
and 1’s; 1 if THIS’s element == value, 0 otherwise)

IUE_matrix<T>& operator^= (const IUE_matrix<short int>&);
Assigning operator: (*this)<T> = IUE_matrix<short int>
( Matrices must be of the same dimensions )

IUE_matrix<T> operator/= (const IUE_matrix<T>& rhs);
Return: old matrix = old matrix ./ rhs , (Matlab syntax)
( element wise division )

IUE_matrix<T> operator/ (const IUE_matrix<T>& rhs);
Return: new matrix = old matrix ./ rhs , (Matlab syntax)
( element wise division )

3 Matrix Expressions

3.1 Expression operators

These binary operators are supported:
+  -  *  /  >  <  >=  <=  ==  !=  &&  ||  ^
Note: operator << and >> are reserved for use in input/output (are not bit-shift
operations).
These unary operators are supported:
-  ~  !
The operators >  <  >=  <=  ==  !=  && and || result in a "bool-valued expression", 1’s and 0’s of type short int.
All operators are applied element-wise.
You can only use operators which are well-defined for the number type stored in
the matrices. For example, bit-wise OR , ||, is meaningful for integers, so this code
is all right:

IUE_matrix<int> A(3,4), B(3,4), C(3,4);
C = B || C;

Bit-wise OR is not meaningful on floating point types, e.g. double, so this code
will generate a compiler error:

IUE_matrix<double> A(3,4), B(3,4), C(3,4);
C = B || C;

If you are creating matrices using a type you have created yourself, you will need
to overload whatever operators you want to use on matrices. For example, if you
create a class Polynomial, and want to write code such as:

IUE_matrix<Polynomial> A(3,4), B(3,4), C(3,4);
C = A * B;

You would have to provide operator* for Polynomial by implementing:
Polynomial Polynomial::operator*(Polynomial);

or

Polynomial operator*(Polynomial, Polynomial);

3.2 Assignment operators

These assignment operators are supported:

=  +=  -=  *=  /=  ^=

A matrix object should appear on the left side of the operator. The right side can be:

- A constant (or literal) of type T
- A matrix of appropriate rank, possibly of a different numeric type
- A matrix expression, with appropriate rank

3.3 Mathematical functions

All of the mathematical functions described in Section 2.4.3.1 are element-wise. For example, this code

```cpp
IUE_matrix<double> A(3,4), B(3,4);
A = sin(B);
```

results in: \( A(i,j) = \sin(B(i,j)) \) for all \((i,j)\)

3.4 Matrix reductions

Currently, EASYMAT++ matrices support two forms of reduction:

- Reductions which transform a matrix (or a vector) into a scalar (for example, summing the elements). These are referred to as complete reductions.
- Reducing a matrix (or matrix expression) to a vector expression. These are called partial reductions.
- Extracting certain parts of a matrix or a vector. These are called block reductions.

3.4.1 Complete reductions

Complete reductions transform a matrix (or matrix expression) or a vector into a scalar. Here are some examples:

```cpp
double values[] = {0, 1, 2,
    3, 4, 5,
    6, 7, 8};
IUE_matrix<double> A(values, 3, 3);
cout << A.sum() << endl // 36
    << A.minVal() << endl // 0
    << A.maxVal() << endl; // 8
```

Here are the available complete reductions for both matrices and vectors:

- **sum()** Sum of all elements
- **mean()** Arithmetic mean of all elements
Part 4  EASYMAT++

- **minVal()** Minimum value of all elements
- **maxVal()** Maximum value of all elements
- **norm()** The norm of the matrix
- **one_norm()** The one norm of the matrix
- **inf_norm()** The infinity norm of the matrix
- **fro_norm()** The Frobenius norm of the matrix
- **rank()** The rank of the matrix
- **abs_det()** Absolute value of the determinant of the matrix

Some complete reductions are for only vectors:
- **variance()** The variance of the vector
- **std_deviation()** The standard deviation
- **cols_sum()** and **rows_sum()** for column and row vectors respectively.

### 3.4.2 Partial reductions

Here's an example which computes the sum of each row of a matrix:

```cpp
double values[] = {0, 1, 2,
                   3, 4, 5,
                   6, 7, 8};
IUE_matrix<double> A(values, 3, 3);
cout << A.cols_sum() << endl  // 9 11 15
   << A.rows_sum() << endl; // 3
       // 12
       // 21
```

Here's a list of the partial reduction operations:
- **cols_sum()** Sums of each column’s elements. Gives a row vector.
- **rows_sum()** Sums of each row’s elements. Gives a column vector.
- **variance()** The variances of the columns of the matrix. Gives a row vector.
- **std_deviation()** The standard deviations of the columns of the matrix. Gives a row vector.

### 3.4.3 Block reductions

Here's an example which extracts a rectangular block of a matrix:

```cpp
double values[] = {0, 1, 2, 3, 4
                   5, 6, 7, 8, 9
                   10, 11, 12, 13, 14};
IUE_matrix<double> A(values, 3, 5);
cout << A.sub_matrix(1, 2, 2, 4) << endl; // 7 8 9
       // 12 13 14
```

Here's a list of the block reduction operations:
- **sub_matrix(start row, end row, start col, end col)** Extracting a rectangular block of the matrix.
- **select_rows(vector of indices)** Extracting certain rows from the matrix.
• **select_cols(vector of indices)** Extracting certain columns from the matrix.

The difference between the above operations and sub-matrices (see Section 2.4.2, and `operator[]`) is that the results here are independent matrices (a deep copies), while in sub-matrices we get shallow copies that refer to certain parts of the original matrix.

## 4 Advanced Features in EASYMAT++

### 4.1 Special support for Complex Matrices

Since complex matrices are used frequently, **EASYMAT++** provides the following special element-wise methods to be used with complex-valued matrices:

- **real()** Getting the real components of the elements
- **imag()** Getting the imaginary components of the elements
- **conj()** Getting the conjugates of the elements
- **angle()** Getting the angles of the elements
- **abs()** Getting the absolute values (the magnitudes) of the elements
- **is_complex()** Returns true if the matrix is complex-valued

All member functions and operators are able to fully handle complex-valued matrices. The absolute values are used for comparisons between complex numbers.

Note: **EASYMAT++** provides numerous math functions defined over complex-valued matrices, such as `cos_fcn, exp_fcn, tanh_fcn` etc. See the section on math functions in Section 2.4.3.1 for details.

### 4.2 Never return non-trivial objects (matrices or vectors)

For big matrices, the cost of multiple constructing, copying, destroying of objects may be very large. This is why additional member functions were added to perform operations on matrices, such as sorting, flipping up-side-down, etc, without much fuss and without moving a lot of stuff around.

In this category, the following member functions are available:

- **flipud() , inplace_flipud()** Reversing order of rows
- **fliplr() , inplace_fliplr()** Reversing order of columns
- **remove_mean() , remove_mean_inplace()** Removing the mean of all elements
- **remove_rows_mean() , remove_rows_mean_inplace()** Removing the mean of each row’s elements
- **remove_cols_mean() , remove_cols_mean_inplace()** Removing the mean of each column’s elements
- **a_x_a()** Matrix Multiply: A * A
- **a_x_aT()** Matrix Multiply: A * A’
- **aT_x_a()** Matrix Multiply: A’ * A
• a_x_aT_conj() Matrix Multiply: $A \times \text{conj}(A')$
• aT_conj_x_a() Matrix Multiply: $\text{conj}(A') \times A$
• t_inverse() Giving the inverse of $A'$

where: $A' = \text{the transpose of } A$

Functions called "inplace" don't return any new matrix, the action is performed on
the matrix itself.

4.3 Advanced Matrix Comparisons

The binary overloaded operators: $>\;<\;>=\;<=\;==\;!=\;&&\;\text{and}\;||$ are all element-wise, and they return binary-valued matrices, with 1's and
0's of type $\text{short int}$. We get a $\text{one}$ when the comparison results in $\text{true}$, otherwise
we get a $\text{zero}$.

These operations are very useful in generating masks to exclude certain parts of a
matrix.

In addition to those overloaded operators, corresponding member functions
perform both the above comparisons and the masking operations; i.e. we get a
copy of the matrix with zeroes placed where the comparison results in $\text{false}$.
Here's a list of these functions:

• GT() Greater than
• GE() Greater or equal
• LT() Less than
• LE() Less or equal
• EQ() Equal to
• NE() Not equal to

The following member functions also perform element-wise comparisons on
matrices:

• max() Returning the maximum of each two compared elements
• min() Returning the minimum of each two compared elements