


Universidad Politécnica de Madrid

Escuela Técnica Superior de Ingenieros de Caminos,
Canales y Puertos

Linked MatriX methods collection

Users Guide
Version 1.0

The LMX development team

Madrid, 2008

Linked MatriX Methods User's Guide

Universidad Politécnica de Madrid
Madrid, 2006-2008

Composition of this document has been made using LaTeX
and related aplications in GNU/Linux under GPL licenses.

LMX - Team members:

- Team Monitor: Juan Carlos García Orden
- Parallel Extensions Developer: Juan J. Arribas
- Sparse Matrices and Methods Developer: Roberto Ortega Aguilera
- Main Developer: Daniel Iglesias Ibáñez

Escuela Técnica Superior de Ingenieros de Caminos, Canales y Puertos
Departamento de Mecánica de Medios Continuos y Teoría de Estructuras

Universidad Politécnica de Madrid
Profesor Aranguren s/n
Madrid 28040

Contents

Chapter 1

Getting started

1.1 Overview

LMX is a library for numerical computations, including matrix and vector algebra, linear and nonlinear solvers, ordinary differential systems manipulation and both explicit and implicit integrators.

The programming language used is *standard C++* and code design features *generic programming* by the intense use of *templates*, allowing the use of different numeric types without any change in the way functions are called or operators are used. Top level class design is not full generic as it letting switch between different matrix types after compiling the code.

The goals that where sought during design and development – and hopefully reached – are mainly the following:

- Intuitive (math-like) code is generated, not requiring specific training and reducing maintenance time in LMX-based applications. Efficient functions are also available –replacing overloaded operators– for faster execution.

- As a consequence of the use of templates, different numerical types are treated the same way.

- Once the code of a LMX-based application has been developed, changing the type of matrices and vectors used doesn't require any modification and can be implemented as a run-time option. The same works for linear and non-linear solvers.

- Any other algebra and numerical solvers library can be linked easily and then used with LMX-like code.

- All the facilities of object oriented programming (OOP) have been deeply studied and applied since design stage reducing maintenance, support and future development efforts.

- Code is thorough documented and a (developers) reference manual is available.

1.2 Installation

LMX uses automatic configuring and compiling from GNU autotools so installing in any supported platform should be straight forward. Just follow the following steps:

1. Run configure script. For example, in bash shell:
   `$ ./configure`
2. Automatic generation and execution of Makefiles by running either
   `$ make`
   or
   `$ gmake`
   In version 0.4.1 and above, no library is precompiled and only header files are used, so `make` does not create any compiled file.
3. Optionally, you can execute some general tests by running
   `$ make check`
   from the base directory.
4. Also optional, we provide some linked libraries specific tests. Before you can run them, if the libs or headers are in non-path places, you may need to edit manually each `Makefile.am`. Then, just repeat step 2 and execute the tests by a simple
   `$ make test`
   from each of the specific directories (below the `tests/` dir).
5. As version 0.4.1 and above are instalable, running `# make install` will place all the headers in your default dir (`/usr/local/include/LMX`) or whatever dir defined in `configure` using the `--prefix` option.

If tou want to use any linked library features, you must add the corresponding define statement when compiling and include their headers directories when linking. The way to do so is the following:

SuperLU option is active when the compiler finds a `#define HAVE_SUPERLU` in your code. If so, headers must be included at compile time (-I option in GNU's gcc). Also, SuperLU and BLAS (".a") libraries should be available for link time.

GMM++ option is active when the compiler finds a `#define HAVE_GMM` in your code. Also, headers also have to be available when you compile your LMX-based application.

Lapack option is active when the compiler finds a `#define HAVE_LAPACK` in your code. Also, BLAS static (".a") library should be available for link time.

Most compilers let add define rules when compiling (`-D` flag in gcc) which is preferable when working on a medium or large project.

1.3 Quick start

If you followed the above steps you should have a nice and powerful new collection of algebra and numerical solving methods called LMX so, how can you start using it? The header files that you should include in your program can be:

- `#include "LMX/lmx.h"` is the basic functionality and implements algebra methods and linear solvers.
- `#include "LMX/lmx_nlsolvers.h"` adds the NLSolver class.
- `#include "LMX/lmx_diff_problem_first.h"` lets first order ODE manipulation and integrators.
- `#include "LMX/lmx_diff_problem_second.h"` when a second order ODE system wants to be integrated.

Some basic examples of use are shown in table 1.1. Table 1.2 contains more specific examples the extended functionality.

Selection of dense matrix type (`gmm::dense_matrix`): `lmx::setMatrixType(0);`
Selection of dense vector type (`std::vector`): `lmx::setVectorType(0);`
Selection of direct linear solver type (LU solver): `lmx::setLinSolverType(0);`
Defining a 2x2 matrix: `lmx::Matrix<double> my_matrix(2,2);`
Defining a two dimension vector: `lmx::Vector<double> my_vector(2);`
Matrix addition: `A = B + C;`
Multiple operations in one instruction: `A = B*x - C*D*y;`
Defining a linear system: `lmx::LinearSystem<double> my_system(A,x,b);`
Solving previous system: `my_system.solveYourself();`

Table 1.1: Basic LMX examples

1.4 Examples

1.4.1 Basic algebra operations

The next code selects Matrix and Vector types, creates some instances of these classes, and operates with the objects. Follow the commented code for usage details.

```
1  unsigned int size = 4;
2
3  // (optional) Make an alias for matrix and vector with
4  // double precision data:
5  typedef lmx::Matrix<double> dMatrix;
6  typedef lmx::DenseMatrix<double> dDenseMatrix;
7  typedef lmx::Vector<double> dVector;
8
9  // Select CSC matrix storage (Harwell-boeing) and dense
10 // STL vector:
11 lmx::setMatrixType(3);
12 lmx::setVectorType(1);
13
14 // Create a vector using the alias defined above.
```

Differential system definition:
`dif::Mec_ode<double>`
`my_mechanism(mass, q_q, q_t, jacq_q, jacq_qdot);`

Integrator's definition: `dif::Integrator_am<double> my_integrator(2);`

Initial value problem definition:
`dif::Init_val_problem<double>`
`my_problem(my_mechanism, my_integrator, q_init);`

Start solving (integration) [interval, t=(5,10)], [time step, h=0.1]: `my_problem.solveYourself(5. , 10. , 0.1);`

Table 1.2: Advanced LMX examples

```
15   dVector vec1(size);
16  // The next vector is the same type and size as vec1:
17   lmx::Vector<double> vec2(size);
18
19  // Manually fill first vector.
20   vec1(0) = 1.;
21   vec1(1) = 1.;
22   vec1(2) = 1.;
23   vec1(3) = 1.;
24
25  // We could call a function for automatic filling.
26  // This would do the same thing as the previous input:
27   vec1.fillIdentity();
28
29  // And the same function can be used with a scale factor:
30   vec2.fillIdentity( 3.5 );
31
32
33  // Create a non-squared Matrix object...
34   dMatrix mat1(10,5);
35  // ...and an empty DenseMatrix:
36   dDenseMatrix mat2;
37
38  // Resize mat2 to mat1 size:
39   mat2.resize( mat1.rows(), mat1.cols() );
40
41  // Fill with random numbers between 0. and 1. mat1:
42 //   mat1.fillRandom();
43   for (int i = 0; i<mat1.rows(); ++i){
44     for (int j = 0; j<mat1.cols(); ++j){
45       mat1(i,j) = 2*i+j;
```

```
46      }
47    }
48
49    // And fill mat2 with random between 0. and 10.
50    mat2.fillRandom(10.);
51
52    // Output mat1:
53    cout << mat1;
54
55  // Resize mat1 to make it square:
56    mat1.resize(size,size);
57    // And change it's contents to identity:
58    mat1.fillIdentity();
59
60    // Output mat1:
61    cout << mat1;
62
63    // Output mat2 with name:
64    cout << "mat2: " << endl << mat2;
65
66    // Multiply mat2 and vec1 and save result in vec2:
67    vec2.resize(mat2.rows() );
68    vec1.resize(mat2.cols() );
69    vec2.mult(mat2,vec1);
70
71    // Output vec1 with name:
72    cout << "vec1: " << endl << vec1 << endl;
73
74    // Output vec2 with name:
75    cout << "vec2: " << endl << vec2 << endl;
76
77    // Output internal product of vec2 by itself with name:
78    cout << "vec2(i) * vec2(i): " << endl
79         << vec2.multElements( vec2 ) << endl;
```

1.4.2 Solving linear systems

The next code selects Matrix and Vector types, creates some instances of these classes, and creates a Linear Solver object. After setting the detail of output, the solver is launched. Follow the commented code for usage details.

```
1   typedef lmx::Matrix<double> dMatrix;
2   typedef lmx::Vector<double> dVector;
3
4   // Select dense matrix storage and STL vector:
5   setMatrixType(0);
6   setVectorType(1);
```

```
7  // Select direct solver for symmetric systems
8   setLinSolverType(0);
9
10  dVector vec1(4);
11  dVector vec2(4);
12
13  dMatrix mat1(4,4);
14
15 // Fill Vector vec1 and Matrix mat1
16  vec1.fillIdentity();
17
18  mat1(0,0) = 121830.;
19  mat1(0,1) = mat1(1,0) = -44400.6;
20  mat1(1,1) = 121830.;
21  mat1(0,2) = mat1(2,0) = -74832.1;
22  mat1(1,2) = mat1(2,1) = -3352.78;
23  mat1(2,2) = 121830.;
24  mat1(0,3) = mat1(3,0) = 3352.78;
25  mat1(1,3) = mat1(3,1) = 12340.3;
26  mat1(2,3) = mat1(3,2) = 44400.6;
27  mat1(3,3) = 121830.;
28
29 // Create a linear system object A*x = b -- mat1*vec2 = vec1
30  LinearSystem<double> lin1(mat1,vec2,vec1);
31
32 // Set detailed output information level
33  lin1.setInfo(2);
34 // Launch the solver
35  lin1.solveYourself();
36
37 // Output the solution:
38  cout << "Solution: " << endl
39       << vec2 << endl;
```

Chapter 2

Matrix and Vector classes

The template classes `lmx::Matrix<T>`, `lmx::DenseMatrix<T>` and `lmx::Vector<T>` allow the creation of the basic objects used by LMX. Template parameter, `T`, can be any of the types

Data type	Description
`int`	Integer
`long`	Long integer
`float`	Simple precision real
`double`	Double precision real
`lmx::Complex(double r, double i)`	Double precision complex[1]
`cofe::TensorRank1<int Dim, T>`	1st order tensors (section 3.1)
`cofe::TensorRank2<int Dim, T>`	2nd order tensors (section 3.2)
`cofe::TensorRank2Sym<int Dim, T>`	Symmetric tensors (section 3.3)

Table 2.1: Matrix and vector value types, `T`

The type of data used for indexing vectors and matrices can be any of the table 2.2, although in this manual the `int` type is used for clarity's sake.

Size type	Description
`int`	Integer
`unsigned int`	Positive integer
`long`	Long integer
`unsigned long`	Positive long integer

Table 2.2: Matrix and vector size types

Any of these classes can be instantiated using any of their constructors[2]. Once an object has been created, it can be resized, filled, assigned to another

[1] To be implemented.

[2] See the following specific sections.

object and, of course, be placed as a parameter in functions and overloaded operators. Each of these facilities are described in detail in the following sections.

You can choose between different types of matrix and vector types by calling their specific function, either `lmx::setMatrixType(int type)` or `lmx::setVectorType(int type)`, where each type value are shown in table 2.3.

Type number	Matrix type	Vector type
0	`Type_dense`	`Type_stdvector`
1	`Type_hb`	`Type_gmm_sparse1`
2	`Type_gmm`	`Type_cvector`
3	`Type_gmm_sparse`	

Table 2.3: Matrix and vector types

2.1 Matrix class

2.1.1 Function Summary

Constructors: (p. 12)

`Matrix<T>();` (p. 12)

`Matrix<T>(int rows, int columns);` (p. 12)

`Matrix<T>(Matrix<T>& another_matrix);` (p. 13)

Member functions:

`int rows() const;` (p. 20)

`int cols() const;` (p. 20)

`void matrixMarketLoad(char* "file_name");` (p. 18)

`void harwellBoeingLoad(char* "file_name");` (p. 18)

`void fillIdentity(T factor);` (p. 26)

`void fillRandom(T factor);` (p. 27)

`T readElement(int row, int column) const;` (p. 21)

```
void writeElement(T theValue,
                  int row, int column) const; (p. 24)
```

```
void addElement(T theValue,
                int row, int column) const; (p. 25)
```

`void resize(int new_rows, int new_columns);` (p. 27)

`void clean(T factor);` (p. 27)

`Matrix& add(const Matrix& A, const Matrix& B);` (p. 35)

`Matrix& subs(const Matrix& A, const Matrix& B);` (p. 35)

`Matrix& mult(const Matrix& A, const Matrix& B);` (p. 36)

`Matrix& multElements(const Matrix& B);` (p. 36)

`Matrix& multElements(const Matrix& A, const Matrix& B);` (p. 36)

`Matrix& transpose();` (p. 38)

Overloaded left value operators:

`T&`[3] `operator () (int row, int column);` (p. 24)

`Matrix& operator = (const Matrix& rmatrix);` (p. 29)

`Matrix& operator = (const Matrix<C>& rmatrix);` (p. 29)

`Matrix& operator += (const Matrix& rmatrix);` (p. 30)

[3]Return type is not really a value type but an intermediate LMX class that can be used in the same way. It cannot be used with pointers, though.

```
Matrix& operator -= (const Matrix& rmatrix);
```
(p. 30)

```
Matrix& operator *= (const T& scalar);
```
(p. 31)

Overloaded left value const operators:

```
Matrix<T> operator + (const Matrix& rmatrix) const;
```
(p. 32)

```
Matrix<T> operator - (const Matrix& rmatrix) const;
```
(p. 32)

```
Matrix<T> operator * (const Matrix& rmatrix) const;
```
(p. 33)

```
Matrix<T> operator * (const T& scalar) const;
```
(p. 33)

Friend functions:

```
Matrix trasposed(const Matrix& mat_in);
```
(p. 38)

```
void latexPrint(std::ofstream& os, char* mat_name,
                        Matrix& mat, int precision);
```
(p. 22)

Overloaded right value operators:

```
Matrix<T> operator * (const T& scalar,
                      const Matrix<T>& rmatrix);
```
(p. 33)

```
std::ostream& operator << (std::ostream& os,
                    const Matrix<T>& rmatrix);
```
(p. 21)

2.1.2 Constructors

All of this constructors belong to the `lmx` namespace, so the user can choose between defining a `using namespace lmx` (or other `using` statements) or simply adding a `lmx::` prefix when defining a new `lmx::Matrix<>`.

Empty constructor:

Call: `Matrix<T>();`

Description: Creates a Matrix object with default values, that are rows and columns equal to zero and an empty data structure.

Example:

```
Matrix<double> my_empty_matrix;
```
creates an empty matrix object called *my_empty_matrix*.

Standard constructor:

Call: `Matrix<T>(rows, columns);`

Parameters: `T` is a value type and `rows, columns` are of any positive integer type (size type).

Description: Creates an object with the specified size and every element equal to zero.

Example:

```
Matrix<double> my_non_squared_matrix(2, 3);
```
creates a 2×3 matrix object called *my_non_squared_matrix* filled up with zeros.

Copy constructor:

Call: `Matrix<T>(another_matrix);`

Parameters: `another_matrix` a reference of type `lmx::Matrix<T>`.

Description: Makes a copy of every one of the `another_matrix` parameters into the new Matrix.

Example:

```
Matrix<double> my_copy_matrix(my_matrix_copied);
```
creates a copy matrix called *my_copy_matrix* from another one with name *my_matrix_copied*.

2.2 DenseMatrix class

This class inherits from `class Matrix<T>` so all the functions implemented for the general type are available for this specific class. In fact, inner structure is the same as the general `Matrix` class when the selector function:

```
lmx::setMatrixType(int type)
```

is called with an argument `type = 0`.

2.3 Vector class

2.3.1 Function Summary

Constructors: (p. 16)

`Vector<T>();` (p. 16)

`Vector<T>(int rows);` (p. 16)

`Vector<T>(Vector<T>& another_vector);` (p. 17)

Member functions:

`int size() const;` (p. 20)

`void load(char* input_file);` (p. 19)

`void fillIdentity(T factor);` (p. 26)

`void fillRandom(T factor);` (p. 27)

`T readElement(int row) const;` (p. 21)

`void writeElement(T theValue, int row) const;` (p. 24)

`void addElement(T theValue, int row) const;` (p. 25)

`void resize(int new_rows);` (p. 27)

`void clean(T factor);` (p. 27)

`Vector& add(const Vector& A, const Vector& B);` (p. 35)

`Vector& subs(const Vector& A, const Vector& B);` (p. 35)

`Vector& mult(const Matrix& A, const Vector& B);` (p. 36)

`Vector& mult(const DenseMatrix& A, const Vector& B);` (p. 36)

`Vector& mult(const T& scalar);` (p. 36)

`Vector& mult(const Vector& B, const T& scalar);` (p. 36)

`Vector& mult(const T& scalar, const Vector& B);` (p. 36)

`Vector& multElements(const Vector& B);` (p. 36)

`Vector& multElements(const Vector& A, const Vector& B);` (p. 36)

`inline T norm1 () const;` (p. 38)

`inline T norm2 () const;` (p. 38)

Overloaded left value operators:

`T&`[4] `operator () (int row, int column);` (p. 24)

`Vector& operator = (const Vector& rvector);` (p. 29)

`Vector& operator = (const Matrix& rmatrix);` (p. 29)

[4]Return type is not really a value type but an intermediate LMX class that can be used in the same way. It cannot be used with pointers, though.

`Vector& operator = (const Vector<C>& rvector);` (p. 29)

`Vector& operator += (const Vector& rvector);` (p. 30)

`Vector& operator -= (const Vector& rvector);` (p. 30)

`Vector& operator *= (const T& scalar);` (p. 31)

Overloaded left value const operators:

`Vector<T> operator + (const Vector& rvector) const;` (p. 32)

`Vector<T> operator - (const Vector& rvector) const;` (p. 32)

`Vector<T> operator * (const Vector& rvector) const;` (p. 33)

`Vector<T> operator * (const T& scalar) const;` (p. 33)

Friend functions:

```
void latexPrint(std::ofstream& os, char* mat_name,
                        Vector& mat, int precision);
```
(p. 22)

Overloaded right value operators:

```
Vector<T> operator * (const T& scalar,
                      const Vector<T>& rvector);
```
(p. 33)

```
std::ostream& operator << (std::ostream& os,
                    const Vector<T>& rvector);
```
(p. 21)

2.3.2 Constructors

As in the Matrix case, all of this constructors belong to the `lmx` namespace, so the user can choose between defining a `using namespace lmx` (or other `using` statements) or simply adding a `lmx::` prefix when defining a new `lmx::Matrix<>`.

Empty constructor:

Call: `Vector<T>();`

Description: Creates a Vector object with default values, that are rows equal to zero and an empty data structure.

Example:

```
Vector<double> my_empty_vector;
```
creates an empty vector object called *my_empty_vector*.

Standard constructor:

Call: `Vector<T>(rows);`

Parameters: `T` is a value type and `rows` is of any positive integer type (size type).

Description: Creates an object with the specified size and every element equal to zero.

Example:

```
Vector<double> my_vector(5);
```
creates a 5-element vector object called *my_vector* filled up with zeros.

Copy constructor:

Call: `Vector<T>(another_vector);`

Parameters: `another_vector` a reference of type `lmx::Vector<T>`.

Description: Makes a copy of every one of the `another_vector` parameters into the new Vector.

Example:

```
Vector<double> my_copy_vector(my_vector_copied);
```
creates a copy vector called *my_copy_vector* from another one with name *my_vector_copied*.

2.4 Matrix and Vector input and output formats

2.4.1 Matrix I/O functions

Matrix Market reading:

Call: `void matrixMarketLoad("file_name");`

Parameters: `"file_name"` of type `char*`.

Description: Reads the Matrix-Market format matrix from the specified file.

Example:

```
my_matrix.matrixMarketLoad("my_mm_file.mtx");
```
loads the matrix that is contained in file *my_mm_file.mtx* into the object *my_matrix*.

Harwell-Boeing reading:

Call: `void harwellBoeingLoad("file_name");`

Parameters: `"file_name"` of type `char*`.

Description: Reads the Harwell-Boeing format matrix from the specified file.

Example:

```
my_matrix.harwellBoeingLoad("my_mm_file.rsa");
```
loads the matrix that is contained in file *my_hb_file.rsa* into the object *my_matrix*.

2.4.2 Vector I/O functions

File reading (non-standard):

Call: `void load("file_name");`

Parameters: `"file_name"` of type `char*`.

Description: Reads the Vector from the specified file.

Example:

```
my_vector.load("my_file.vec");
```

loads the vector that is contained in file *my_file.vec* into the object *my_vector*.

2.5 Matrix and Vector general functions

2.5.1 Read-only functions

Size of vector:

Call: `int size() const;`

Description: size of the Vector object (read-only access). The vector is not column or row oriented.

Returns: any positive integer type (size type) with the size value.

Example:

```
int m = my_vector.rows();
```
saves the size of *my_vector* into the variable *m*.

Number of rows:

Call: `int rows() const;`

Description: number of rows in the Matrix object (read-only access).

Returns: any positive integer type (size type) with the number of rows value.

Example:

```
int m = my_matrix.rows();
```
saves the number of rows of *my_matrix* into the variable *m*.

Number of columns:

Call: `int cols();`

Description: number of columns in Matrix (read-only access).

Returns: any positive integer type (size type) with the number of columns value.

Example:

```
int n = my_matrix.rows();
```
saves the number of columns of *my_matrix* into the variable *n*.

Element reading:

Call:

For Matrix: `T readElement(row, column);`

For Vector: `T readElement(row);`

Parameters: `row, column` are any of the integer size types (i.e. `int`).

Description: Read only access to element in specified position.

Returns: A copy of element of value type `T`.

Example:

```
double aValue = my_matrix.readElement(2,3);
```
saves the element in position (2,3)[5] of *my_matrix* into the variable *aValue*. Note that in this case, (rows ≥ 3) and (cols ≥ 4) or an error will be thrown.

Streams:

Call: `std::ostream& operator << (os, object);`

Parameters: `os` is an output stream `std::ostream` used as reference.

`object` is the object output, can be a `Matrix` or a `Vector`.

Description: Outputs the object to the stream. First written line describes the kind of object's type and its dimensions.

Returns: A copy of element of value type `T`.

Examples:

```
std::cout << my_matrix;
```
puts *my_matrix* into the standard output.

LaTeX output:

Call: `void latexPrint(os, obj_name, obj, precision);`

Parameters: `os` is an output file stream `std::ofstream` used as reference.

`obj_name` is a `char*` that defines the name with LaTeX syntax.

`obj` is the object output, can be a `Matrix` or a `Vector`.

`precision` of type `int` is the maximum number of digits used to output each element in object.

Description: Outputs to the file stream the object in LaTeX format. If object is a Matrix, brackets (parenthesis) are used. If it is a Vector, braces are used instead.

Example:

`latexPrint(outfstream, name, my_matrix, 3);`
outputs *my_matrix* to file stream *outfstream* with three digit of maximum precision. If name string is `"\bf\overlineB"`, once typeset is done, the output will look like the matrix shown below, where elements $A(i,j)$ would be the numerical values of the matrix elements.

$$\overline{\mathbf{B}} = \begin{pmatrix} A(0,0) & A(0,1) & \cdots & A(0,m) \\ A(1,0) & A(1,1) & \cdots & A(1,m) \\ \vdots & \vdots & \ddots & \vdots \\ A(n,0) & A(n,1) & \cdots & A(n,m) \end{pmatrix}$$

2.5.2 Manipulation functions

Element read/write operator:

Call:

for Matrix: `A(row, column);`

for Vector: `b(row);`

Parameters: `row, column` are any of the integer size types (i.e. `int`).

`A` is any kind of matrix.

`b` is any Vector type.

Description: Read / write access to element in specified position.

Returns: An intermediate class object that can be treated as if it was a value type `T` as the cast is automaticly done for interfaced value types. Just be carefull to not try to get its memory direction or assign it to a pointer.

Examples:

```
double aValue = my_matrix(2,3);
my_matrix(0,0) = aValue;
```

Reads element in position (2,3) of *my_matrix* and stores it in the variable *aValue* and then writes it in position (0,0). Note that in this case, (rows ≥ 3) and (cols ≥ 4) or an error will be thrown.

```
my_vector(0) = my_vector(1);
```

Reads element in position (1) of *my_vector* and then writes it in position (0). Note that in this case, (rows ≥ 2) or an error will be thrown.

Write element function:

Call:

for Matrix: `A.writeElement(aValue, row, column);`

for Vector: `b.writeElement(aValue, row);`

Parameters: `aValue` is of any value type `T` and `row, column` are any of the integer size types (i.e. `int`).

`A` is any kind of matrix.

`b` is any Vector type.

Description: Write access to element in specified position.

Examples:

`my_matrix.writeElement( 14.5, 1, 2 );`
Writes the value `14.5` in position (1,2). Note that (rows $\geq$ 2) and (cols $\geq$ 3) or an error will be thrown.

`my_vector.writeElement(3) = my_matrix(1,2);`
Reads element in position (1,2) of *my_matrix* and then writes it in position (3) of `my_vector`. Note that (`my_vector.rows()` $\geq$ 4) or an error will be thrown.

Add element function:

Call:

for Matrix: `A.addElement(aValue, row, column);`

for Vector: `b.addElement(aValue, row);`

Parameters: `aValue` is of any value type `T` and `row, column` are any of the integer size types (i.e. `int`).

`A` is any kind of matrix.

`b` is any Vector type.

Description: Add efficiently element in specified position.

Examples:

`my_matrix.addElement( 14.5, 1, 2 );`
Adds the value `14.5` in position (1,2). Note that (rows $\geq$ 2) and (cols $\geq$ 3) or an error will be thrown.

`my_vector.addElement(3) = my_matrix(1,2);`
Reads element in position (1,2) of *my_matrix* and then adds it in position (3) of `my_vector`. Note that (`my_vector.rows()` $\geq$ 4) or an error will be thrown.

Identity Matrix and Vector:

Call: `A.fillIdentity(factor=1);`

Parameters: `factor` is of any value type `T`. Default value is unity if it is not provided in the call.

`A` is any kind of Matrix or Vector. Matrix objects must be squared or an error will be thrown.

Description: Makes matrices and vectors identity, erasing all data in them. If a factor value is given, a scaled Matrix or Vector is built.

Examples:

`my_matrix.fillIdentity();`
If `my_matrix` is squared, this transforms it into an identity matrix.

`my_vector.fillIdentity(4.3);`
All of the elements in `my_vector` are given the value `4.3`.

Random data fill:

Call: `A.fillRandom(factor=1);`

Parameters: `factor` is of any value type `T`. Default value is unity if it is not provided in the call.

`A` is any kind of Matrix or Vector.

Description: Fills up matrices and vectors with random data between `0` and `1`, erasing all data in them. If a factor value is given, random data will be generated between 0 and `factor`.

Examples:

```
my_matrix.fillRandom();
```
Fills `my_matrix` with random data between `0` and `1`.

```
my_vector.fillRandom(100);
```
Fills `my_vector` with random data between `0` and `100`.

Matrix resizing:

Call:

for Matrix: `void resize(new_rows, new_columns);`

for Vector: `void resize(new_rows);`

Parameters: `new_rows, new_columns` are any of the integer size types (i.e. `int`).

Description: Resizes the object to specified size. Creates new zero elements if the new size is bigger and eliminates the element with higher indexes if it is smaller.

Examples:

```
my_matrix.resize(10,20);
```
resizes the existing object *my_matrix*.

```
my_vector.resize(8);
```
resizes the existing object *my_vector*.

Cleaning round off errors:

Call: `void clean(factor);`

Parameters: `factor` is a value type `T`.

Description: Makes zero any value in a Matrix or Vector object below the given factor.

Example:

```
my_matrix.clean(1E-12);
```
Switches to `0.` any value below `1E-12` in object *my_matrix*.

2.6 Linear algebra operations

2.6.1 Efficient overloaded operators

Assignment operator:

operator =

Call: `A = B;`

Parameters: `A,B` can be of either (equal) type `Matrix` and `Vector`. `B` is used as a constant reference. It is possible to assign a Matrix `B` to a Vector `A` if it has only one column.

Description: Copies each parameter in object `B` to the object `A`. Resizing of `A` is done if necessary.

If different data value types are used in each object, the operator casts the values of `B` to the value type used in `A`.

Returns: A reference to the object making possible functions and operators linkage.

Examples:

`my_lmatrix = my_rmatrix;`
copies *my_rmatrix* into the first object *my_lmatrix*.

`my_lvector = my_rvector;`
copies *my_rvector* into the first object *my_lvector*.

Addition and substraction operators:

operator +=

Call: `A += B;`

Parameters: `A,B` can be of either (equal) type `Matrix<T>` and `Vector<T>`. `B` is used as a constant reference.

Description: Adds the two objects and saves the result into the first one (left right side).

Returns: A reference to the object making possible functions and operators linkage.

Examples:

`my_lmatrix += my_rmatrix;`
saves the result of *(my_lmatrix + my_rmatrix)* into the first object *my_lmatrix*.

`my_lvector += my_rvector;`
saves the result of *(my_lvector + my_rvector)* into the first object *my_lvector*.

operator -=

Call: `A -= B;`

Parameters: `A,B` can be of either (equal) type `Matrix<T>` and `Vector<T>`. `B` is used as a constant reference.

Description: Substracts the two objects and saves the result into the first one (left right side).

Returns: A reference to the object making possible functions and operators linkage.

Examples:

`my_lmatrix -= my_rmatrix;`
saves the result of *(my_lmatrix - my_rmatrix)* into the first object *my_lmatrix*.

`my_lvector -= my_rvector;`
saves the result of *(my_lvector - my_rvector)* into the first object *my_lvector*.

Scale operator:

operator *=

Call: `A *= b;`

Parameters: `A` can be of either (equal) type `Matrix<T>` and `Vector<T>`. `b` is of value type `T` and is used as a constant reference.

Description: Scales the object's elements by `b`.

Returns: A reference to the object making possible functions and operators linkage.

Examples:

`my_lmatrix *= aValue;`
saves the result of *(my_lmatrix(i,j) * aValue)* into the first object's elements *my_lmatrix(i,j)*.

`my_lvector *= aValue;`
saves the result of *(my_lvector(i) * aValue)* into the first object's elements *my_lvector(i)*.

2.6.2 Infficient overloaded operators

Usage of these operators is not recommended if efficiency is a requirement as a temporary copy of the result is used, thus creating an destroying an object in every call. They are provided as an easy way of implementing each operation and generating an easy to read and maintain code.

Addition and substraction operators:

operator +

Call: `A + B;`

Parameters: `A,B` can be of either (equal) type `Matrix<T>` and `Vector<T>`. Both are used as a constant reference.

Description: Adds the two objects and returns the result.

Returns: A new object with the result.

Examples:

`my_lmatrix + my_rmatrix;`
returns the result of *(my_lmatrix + my_rmatrix)*.

`my_lvector + my_rvector;`
returns the result of *(my_lvector + my_rvector)*.

operator -

Call: `A - B;`

Parameters: `A,B` can be of either (equal) type `Matrix<T>` and `Vector<T>`. Both are used as a constant reference.

Description: Substract the two objects and returns the result.

Returns: A new object with the result.

Examples:

`my_lmatrix - my_rmatrix;`
returns the result of *(my_lmatrix - my_rmatrix)*.

`my_lvector + my_rvector;`
returns the result of *(my_lvector - my_rvector)*.

Multiplying operator:

operator *

Call: `A * B;`

Parameters: `A,B` can be of either type `Matrix<T>`, `Vector<T>` or value type `T` independently. All of them are used as a constant reference.

Description: Multiplies the two objects and returns the result.

If both `A` and `B` are matrices (i.e. $\boldsymbol{A}, \boldsymbol{B}$), matrix multiplication is done ($\boldsymbol{AB}$).

If `A` is a Matrix (i.e. $\boldsymbol{A}$) and `B` is a Vector (i.e. $\boldsymbol{b}$), matrix-vector multiplication is done ($\boldsymbol{Ab}$).

If `A` is a Vector(i.e. $\boldsymbol{a}$) , and `B` (i.e. $\boldsymbol{B}$) is a Matrix, vector-matrix multiplication is done ($\boldsymbol{a^T B}$).

If both `A` and `B` are vectors (i.e. $\boldsymbol{a}, \boldsymbol{b}$), scalar product is done ($\boldsymbol{a^T b}$).

If one of them (either `A` or `B`) is a value type `T`, each element of the other one (Matrix or Vector) is multiplied by it.

Returns: A new object with the result.

If both `A` and `B` are matrices, the return type is a `Matrix<T>`.

If both `A` and `B` are vectors, the return type is a value type `T`.

If one of the parameters is a value type, the return type will be the same type as the other parameter.

In other cases, the return type is a `Vector<T>`.

Examples:

`my_lmatrix * my_rmatrix;`
returns the result matrix of *(my_lmatrix * my_rmatrix).*

`my_lmatrix * my_rvector;`
returns the result vector of *(my_lmatrix * my_rvector).*

`my_lvector * my_rmatrix;`
returns the result vector of *(my_lvector * my_rmatrix).*

`my_lvector * my_rvector;`
returns the result scalar value of *(my_lvector · my_rvector).*

operator * (continued)

Examples (cont.):

`my_lmatrix * aValue;`
returns a new Matrix with elements *(i,j)* computed as *(my_lmatrix(i,j) * aValue)*.

`my_lvector * aValue;`
returns a new Vector with elements *(i)* computed as *(my_lmatrix(i) * aValue)*.

2.6.3 Linear algebra functions

Addition:

Call: `add(A, B);`

Parameters: `A,B` are both of type `Matrix<T>` or `Vector<T>` and are used as constant references.

Description: Computes `A + B` and saves the result into the object.

Returns: A reference of the object in witch result is stored so linking functions is possible.

Examples:

`my_matrix.add(my_lmatrix, my_rmatrix);`
saves the result of *(my_lmatrix + my_rmatrix)* in the object *my_matrix*.

`my_vector.add(my_lvector, my_rvector);`
saves the result of *(my_lvector + my_rvector)* in the object *my_vector*.

Substraction:

Call: `subs(A, B);`

Parameters: `A,B` are both of type `Matrix<T>` or `Vector<T>` and are used as constant references.

Description: Computes `A - B` and saves the result into the object.

Returns: A reference of the object in witch result is stored so linking functions is possible.

Examples:

`my_matrix.subs(my_lmatrix, my_rmatrix);`
saves the result of *(my_lmatrix - my_rmatrix)* in the object *my_matrix*.

`my_vector.add(my_lvector, my_rvector);`
saves the result of *(my_lvector - my_rvector)* in the object *my_vector*.

Multiplication:

Matrix-Matrix multiplication:

Call: `Matrix& mult(A, B);`

Parameters: `A,B` are of type `Matrix<T>` and are used as constant references.

Description: Multiplies the two matrices and saves the result into the object.

Returns: A reference of the Matrix object so linking functions is possible.

Example:

`my_matrix.mult(my_lmatrix, my_rmatrix);`
saves the result of *(my_lmatrix * my_rmatrix)* in the object *my_matrix*.

Matrix-Vector multiplication:

Call: `Vector& mult(A, b);`

Parameters: `A` is a `Matrix<T>` and `b` is a Vector. Both are used as constant references.

Description: Multiplies the Matrix and Vector and saves the result into the object.

Returns: A reference of the Vector object so linking functions is possible.

Example:

`my_vector.mult(my_matrix, my_vector);`
saves the result of *(my_matrix * my_vector)* in the object *my_vector*.

For Vector-Vector dot product see overloaded operator * (p. 33)

Element-by-Element multiplication:

Call: `Matrix& multElements(A, B);`
`Matrix& multElements(A);`
`Vector& multElements(a, b);`
`Vector& multElements(a);`

Parameters: `A, B` are of type `Matrix<T>` and `a, b` are of type `Vector<T>`. All the parameters are used as constant references.

Description: Multiplies element by element so arguments and object must have the same size.

Returns: A reference of the Matrix or Vector object so linking functions is possible.

Example:

`my_matrix.multElements(my_lmatrix, my_rmatrix);`
saves the result of *(my_lmatrix(i,j) * my_rmatrix(i,j))* in the object *my_matrix(i,j)*.

Matrix transposition:

Member function (modifies object):

Transpose Matrix

Call: `void transpose();`

Description: Transposes (modifying) the matrix object.

Example:

```
my_matrix.transpose();
```
transposes the existing object *my_matrix*.

Friend function (without modifying the object):

Matrix transposed

Call: `void transposed(A);`

Parameters: `A` of type `Matrix<T>` is used as a constant reference.

Description: Calculates the transpose of matrix `A`.

Returns: A new `Matrix<T>` object.

Example:

```
my_transposed_matrix = transpose(my_matrix);
```
saves in *my_transposed_matrix* the object *my_matrix* transposed.

Vector norms:

Call:

First norm: `T norm1();`

Second (Euclidean) norm: `T norm2();`

Description: Calculates the norm specified by the name funcion.

Returns: A value type `T`.

Example:

```
double n = my_vector.norm1();
```
saves the result of the first norm of *my_vector* in the variable *n*.

Chapter 3

Tensors

LMX interfaces Cofe library (Continuum-oriented finite elements) from Jaime Planas and Jose M Sancho for tensor manipulation capabilities. Any of the tensor classes can be used as a value type `T` in Matrix and Vector template classes.

The template parameters of any of these classes define the space dimension in which they are defined, `Dim`, and the value type used, `T`, which can be either simple (`float`) or `double` precision real numbers.

3.1 First order tensors

The first order tensor has the following declaration:

```
template <int Dim, class T = double> class cofe::TensorRank1;
```

The namespace scope corresponds to `cofe::`, so implementing a typedef is recommended for generating clear code.

3.1.1 Function Summary

Constructors: (p. 42)

`TensorRank1();` (p. 42)

`TensorRank1(const T* ptr);` (p. 42)

`TensorRank1(const T& aValue);` (p. 42)

`TensorRank1(const std::string& stringOfComponents);` (p. 43)

`TensorRank1(const char* c_s);` (p. 43)

`TensorRank1(const own_type& inV);` (p. 43)

Member functions:

`int size();` (p. 46)

```
void zero(); (p. 47)
void copyFrom(const scalar_type* ptr); (p. 47)
scalar_type dot(const vector_type & aVector)const; (p. 48)
void beProductOf(const tensor_type & aR2T,
                 const vector_type & aV); (p. 49)
void beSolutionOf(const tensor_type & aR2T,
                 const vector_type & aV); (p. 51)
```

Overloaded left value operators:

```
scalar_type& operator()(int i); (p. 57)
own_type&  operator = (const scalar_type& aValue); (p. 57)
own_type&  operator =
 (const std::string& stringOfComponents); (p. 57)
own_type&  operator = (const char* c_s); (p. 57)
own_type&  operator = (const own_type& inV); (p. 57)
void  operator *= (const scalar_type& aValue) (p. 58)
void  operator += (const vector_type aV); (p. 58)
void  operator -= (const vector_type aV); (p. 58)
```

Overloaded left value const operators:

```
const scalar_type& operator()(int i)const; (p. 57)
```

Related functions:

```
template <int Dim,class T>
  std::ostream & operator << (std::ostream & os,
              const cofe::TensorRank1<Dim,T> & a); (p. 59)
```

3.1.2 Constructors

Following constructors are available for creating new TensorRank1 objects:

Empty constructor:

```
TensorRank1();
```

Constructor from array of values:

```
TensorRank1(const T* ptr);
```

Constructor with one value:

```
TensorRank1(const T& aValue);
```

Constructor from a STL string of values:

```
TensorRank1(const std::string& stringOfComponents);
```

Constructor from a c-string of values:

```
TensorRank1(const char* c_s);
```

Copy constructor:

```
TensorRank1(const own_type& inV);
```

3.2 Second order tensors

3.2.1 Function Summary

Constructors: (p. 44)

```
TensorRank2(); (p. 44)
TensorRank2(const T ptr[Dim][Dim]); (p. 44)
TensorRank2(const T& aValue); (p. 44)
TensorRank2(const std::string& stringOfComponents); (p. 45)
TensorRank2(const char* c_stringOfComponents); (p. 45)
TensorRank2(const own_type& aTen); (p. 45)
```

Member functions:

```
int size(); (p. 46)
scalar_type trace()const; (p. 46)
void zero(); (p. 47)
void copyFrom(const scalar_type* ptr); (p. 47)
scalar_type dot(const tensor_type & aTensor)const; (p. 48)
void operateOn(const vector_type& operand,
                       vector_type& answer)const; (p. 49)
void beTransposeOf(const tensor_type& Ten); (p. 47)
void beProductOf(const vector_type & leftVec,
                 const ar4tensor_type & r4T,
                 const vector_type & rightVec); (p. 49)
void beProductOf(const vector_type& leftVec,
                 const vector_type& rightVec); (p. 49)
void beProductOf(const tensor_type& leftTen,
                 const tensor_type& rightTen); (p. 49)
void beTraProductOf(const tensor_type& leftTen,
                    const tensor_type& rightTen); (p. 50)
```

```
void beProductTraOf(const tensor_type& leftTen,
                    const tensor_type& rightTen); (p. 50)
void beUnityTensor(); (p. 47)
void solve(const vector_type& operand,
                  vector_type & answer)const; (p. 50)
void beInverseOf(const tensor_type& atensor); (p. 51)
scalar_type determinant() const; (p. 52)
scalar_type secondInvariant() const; (p. 52)
```

Overloaded left value operators:

```
scalar_type& operator()(int i, int j); (p. 57)
own_type&  operator = (const scalar_type& aValue); (p. 57)
own_type&  operator =
 (const std::string& stringOfComponents); (p. 57)
own_type&  operator = (const char* c_s); (p. 57)
own_type&  operator = (const tensor_type & aTen); (p. 57)
void  operator *= (const scalar_type& aValue) (p. 58)
void  operator += (const TensorRank2<Dim, T> a); (p. 58)
void  operator -= (const TensorRank2<Dim, T> a); (p. 58)
```

Overloaded left value const operators:

```
const scalar_type& operator()(int i)const; (p. 57)
```

Related functions:

```
template <int Dim,class T>
  std::ostream & operator << (std::ostream & os,
              const cofe::TensorRank2<Dim,T> & a); (p. 59)
```

3.2.2 Constructors

Following constructors are available for creating new TensorRank2 objects:

Empty constructor:

```
TensorRank2();
```

Constructor from array of values:

```
TensorRank2(const T* ptr);
```

Constructor with one value:

```
TensorRank2(const T& aValue);
```

Constructor from a STL string of values:

```
TensorRank2(const std::string& stringOfComponents);
```

Constructor from a c-string of values:

```
TensorRank2(const char* c_s);
```

Copy constructor:

```
TensorRank2(const own_type& inV);
```

3.3 Second order symmetric tensors

As it is derived from template class

```
TensorRank2<int Dim, T = double>
```

all its funcions and overloaded operators work the same way. The symmetry is checked in object construction and element definition.

3.3.1 Function Summary

Constructors: (p. 46)

```
TensorRank2Sym(); (p. 46)
TensorRank2Sym(const T ptr[Dim][Dim]); (p. 46)
TensorRank2Sym(const T& aValue); (p. 46)
TensorRank2Sym(const TensorRank2Sym<Dim,T> & aTen); (p. 46)
```

Member functions:

```
void beSymmetricPartOf
     (const cofe::TensorRank2<Dim, T>& r2T); (p. 53)
void beHydrostaticPartOf
     (const cofe::TensorRank2Sym<Dim, T>& r2T); (p. 53)
void beDeviatoricPartOf
     (const cofe::TensorRank2Sym<Dim, T>& r2T) (p. 54)
void beProductOf
     (const cofe::AbstractTensorRank4SS<Dim, T>&  r4T,
      const cofe::TensorRank2Sym<Dim, T>& r2ST); (p. 49)
void beSymProductOf
     (const cofe::TensorRank1<Dim, T>&  lv,
      const cofe::TensorRank1<Dim, T>& rv); (p. 54)
bool checkSymmetry(); (p. 55)
double maxPrincipalValue()const; (p. 55)
```

```
void computeMaxPrincipalDirection
     (TensorRank1<Dim,T>&)const; (p. 56)
```

Overloaded left value operators:

```
own_type&  operator =
           (const TensorRank2Sym<Dim,T> & aTen); (p. 57)
```

3.3.2 Constructors

Following constructors are available for creating new TensorRank2Sym objects:

Empty constructor:

```
TensorRank2Sym();
```

Constructor from array of values:

```
TensorRank2Sym(const T* ptr);
```

Constructor with one value:

```
TensorRank2Sym(const T& aValue);
```

Copy constructor:

```
TensorRank2Sym(const own_type& inV);
```

3.4 Fourth order tensors

```
class AbstractTensorRank4SS
```

To be done.

3.5 Tensor functions

Except when it is specified, the following functions are valid for both first and second rank tensors.

Object's size:

Call: `int size();`

Returns: a type `int` with the size of the tensor.

Example:

```
int theSize = myTensorRank2.size();
```

Trace of a TensorRank2:

Call: `scalar_type trace()const;`

Returns: a `scalar_type` with the trace of the tensor, calculated as:

$$tr(\boldsymbol{T}) = T_{ii}$$

Example:

```
double theTrace = myTensorRank2.trace();
```

Zero a Tensor:

Call: `void zero()const;`

Example:

```
myTensorRank1.zero();
```

Copy an array of values:

Call: `void copyFrom(ptr);`

Parameters: ptr of type `scalar_type*` and used as a `const` reference.

Example:

```
myTensorRank2.copyFrom(myScalarArray);
```

Unit rank two tensors:

Call: `void beUnityTensor();`

Description: Converts the tensor into a unit tensor.

Example:

```
myTenR2.beUnitVector();
```

Rank two Tensor transposition:

Call: `void beTransposeOf(Ten);`

Parameters: Ten is of type `TensorRank2`.

Description: Saves the transposed argument into the object.

Example:

```
myTransposedTenR2.beTransposeOf(myNotTransposedTenR2);
```

Dot product (and double contraction):

Call:

For TensorRank1: `scalar_type dot(aVector)const;`

For TensorRank2: `scalar_type dot(aTensor)const;`

Parameters: must be of same type as objects from with their invoked (either `TensorRank1` for `aVector` and `TensorRank2` for `aTensor`).

Description: Calculates the dot product and saves it into the object.

For TensorRank1, the operation is defined as the dot product of vectors:

$$\boldsymbol{a} \cdot \boldsymbol{b} = a_p b_p$$

For TensorRank2, the operation is defined as the double contraction:

$$\boldsymbol{A} : \boldsymbol{B} = a_{pq} b_{pq}$$

Returns: The result of a value type `T`.

Example:

```
double aValue = myTensorRank1.dot(otherTensorRank1)
+ myTensorRank2.dot(otherTensorRank2);
```

TensorRank2-TensorRank1 product:

Call: `void operateOn(operand, answer)const;`

Parameters: `operand, answer` are TensorRank1 tensors. `operand` is a const reference and `answer` is a reference.

Description: Common tensor-vector product invoked from the tensor `TensorRank2` object (i.e. $\boldsymbol{A}$). Multiplies it to a vector `TensorRank1 operand` (i.e. $\boldsymbol{x}$) and saves the result into vector `TensorRank1 answer` (i.e. $\boldsymbol{b} = \boldsymbol{A} \cdot \boldsymbol{x}$): $b_i = A_{ip} \cdot x_p$

Example:

```
myTenR2.operateOn(myTenR1, myTenR1Solution);
```

Varied tensorial products:

Call:

```
void beProductOf(lTensor, rTensor);
void beProductOf(lTensor,cTensor, rTensor);
```

Parameters: `lTensor` and `rTensor` are any first or second rank tensors. `cTensor` is a fourth order tensor class, `AbstractTensorRank4SS`. All of them are used as constant references.

Description: Depending on parameters types, different operations are done. If $\boldsymbol{a}, \boldsymbol{b}$ are `TensorRank1` objects, $\boldsymbol{R}, \boldsymbol{S}, \boldsymbol{T}$ are `TensorRank2` objects and $\mathcal{C}$ is a fourth order tensor `AbstractTensorRank4SS`, the following operations are allowed:

$$\boldsymbol{a} = \boldsymbol{T} \cdot \boldsymbol{b}$$

$$\boldsymbol{T} = \boldsymbol{a} \cdot \mathcal{C} \cdot \boldsymbol{b}$$

$$\boldsymbol{T} = \boldsymbol{a} \otimes \boldsymbol{b}$$

$$\boldsymbol{T} = \boldsymbol{R} \cdot \boldsymbol{S}$$

Example:

```
myTenR2.beProductOf(mylTenR1, myTenR4, myrTenR1);
```

Transposed rank two tensor product:

TensorRank2-TensorRank1 product:

Call:

```
void beTraProductOf(leftTen, rightTen)const;
void beProductTraOf(leftTen, rightTen)const;
```

Parameters: `leftTen, rightTen` are TensorRank2 tensors. Both are used as const references.

Description: Rank two tensor multiplication (dot product) after transposing one of the operands.

For `beTraProductOf`: $\boldsymbol{T} = \boldsymbol{R}^T \cdot \boldsymbol{S}$

For `beProductTraOf`: $\boldsymbol{T} = \boldsymbol{R} \cdot \boldsymbol{S}^T$

Example:

```
myTenR2.beTraProductOf(myLTenR2, myRTenR2);
```

Linear problem solution:

Call from a TensorRank2 object:

Call:

```
void solve(operand, answer);
```

Parameters: `operand, answer` are TensorRank1 tensors. `operand` is a const reference and `answer` is a reference.

Description: Solves linear tensor system and saves the result into the `answer` object ($\boldsymbol{x}$).

$$\boldsymbol{A} \cdot \boldsymbol{x} = \boldsymbol{b}$$

Example:

```
myConstTenR2.solve(myRhsTenR1, myTenR1Solution);
```

Linear problem solution:

Call from a TensorRank1 object:

Call:

```
void beSolutionOf(aR2T, aV);
```

Parameters: `AR2T` is a TensorRank2 and `answer` is a TensorRank1. Both arguments are const references.

Description: Solves linear tensor system and saves the result into the object from witch the function is called($\boldsymbol{x}$).

$$\boldsymbol{A} \cdot \boldsymbol{x} = \boldsymbol{b}$$

Example:

```
myTenR1.solve(myConstLhsTenR2, myConstRhsTenR1);
```

Tensor inverse:

Call:

```
void beInverseOf(aTensor);
```

Parameters: `aTensor` is a constant reference to a TensorRank2.

Description: Computes the inverse of the second order tensor ($\boldsymbol{T}$) and stores the result into the object from witch it is invoked ($\boldsymbol{X}$).

$$\boldsymbol{X} = \boldsymbol{T}^{-1}$$

Example:

```
myTenR2.beInverseOf(myConstLhsTenR2);
```

Determinant of a tensor:

Call:

```
scalar_type determinant() const;
```

Description: Computes the determinant of the second order tensor object from witch it is invoked.

Returns: The result of a value type `T`.

Example:

```
double det = myDoubleTenR2.determinant();
```

Second invariant of a tensor:

Call:

```
scalar_type secondInvariant() const;
```

Description: Computes the second invariant of the second order tensor object ($\mathbf{S}$) from witch it is invoked as a function of the first invariant ($\boldsymbol{I}_\mathbf{s} = tr(\mathbf{S})$).

$$\boldsymbol{II}_\mathbf{S} = \frac{1}{2}\boldsymbol{I}_\mathbf{S} - \mathbf{S} : \mathbf{S}$$

Returns: The result of a value type `T`.

Example:

```
double det = myDoubleTenR2.determinant();
```

3.5.1 Symmetric rank two tensors specific functions:

Symmetric decomposition:

Call:

```
void beSymmetricPartOf(aTensor);
```

Parameters: `aTensor` is a constant reference to a TensorRank2.

Description: Computes the symmetric part of the second order tensor ($\boldsymbol{T}$) and stores the result into the object from witch it is invoked ($\boldsymbol{X}$).

$$\boldsymbol{X} = \boldsymbol{T}^S \longleftrightarrow X_{ij} = \frac{1}{2}(T_{ij} + Tji)$$

Example:

```
myTenR2.beSymmetricPartOf(myConstLhsTenR2);
```

Hydrostatic part:

Call:

```
void beHydrostaticPartOf(aTensor);
```

Parameters: `aTensor` is a constant reference to a TensorRank2.

Description: Computes the hydrostatic part of the second order tensor ($\boldsymbol{T}$) and stores the result into the object from witch it is invoked ($\boldsymbol{X}$).

$$\boldsymbol{X} = \frac{1}{3}\ \mathbf{1}\ tr(\boldsymbol{T})$$

Example:

```
myTenR2.beHydrostaticPartOf(myConstLhsTenR2);
```

Deviatoric part:

Call:

```
void beDeviatoricPartOf(aTensor);
```

Parameters: `aTensor` is a constant reference to a TensorRank2.

Description: Computes the deviatoric part of the second order tensor ($\boldsymbol{T}$) and stores the result into the object from witch it is invoked ($\boldsymbol{X}$).

$$\boldsymbol{X} = \mathbf{T} - \frac{1}{3}\,\mathbf{1}\,tr(\boldsymbol{T})$$

Example:

```
myTenR2.beDeviatoricPartOf(myConstLhsTenR2);
```

Symmetric external product:

Call:

```
void beSymProductOf(lVector, rVector);
```

Parameters: `lVector` and `rVector` are first rank tensors used as constant references.

Description: Computes the symmetric external product of two vectors, defined as:

$$\boldsymbol{T} = \boldsymbol{a} \otimes^S \boldsymbol{b} \longleftrightarrow T_{ij} = \frac{1}{2}(a_i bj + a_j bi)$$

Example:

```
myTenR2.beSymProductOf(mylTenR1, myrTenR1);
```

Check symmetry:

Call:

```
bool checkSymmetry() const;
```

Description: Checks the symmetry element-by-element in a second order tensor object (**T**) from witch it is invoked.

$$\mathbf{T} \text{ is symmetric } \longleftrightarrow T_{ij} = T_{ji}$$

Returns: `1` (true) or `0` (false).

Example:

```
if( myTenR2Sym.checkSymmetry() ){ ... };
```

Compute the maximum principal value:

Call:

```
double maxPrincipalValue() const;
```

Description: Computes the first eigenvalue of the second order symmetric tensor. Implemented for dimension 2 and 3 tensors.

Returns: The maximum principal value as a `double` data type.

Example:

```
double firstEigenV = myTenR2Sym.maxPrincipalValue();
```

Compute the maximum principal direction:

Call:

```
void computeMaxPrincipalDirection(aVector) const;
```

Parameters: `aVector` is a first rank tensor used as a reference.

Description: Computes the direction of the first eigenvalue of the second order symmetric tensor (first eigenvector). Implemented for dimension 2 and 3 tensors.

Example:

```
myTenR2Sym.computeMaxPrincipalDirection(
aTensorRank1 );
```

3.6 Tensor overloaded operators

Element read/write operator:

Call:

for TensorRank2: `A(row, column);`

for TensorRank1: `b(row);`

Parameters: `row, column` are any of the integer size types (i.e. `int`). Must be bounded between tensor dimensions.

`A` is a TensorRank2 or TensorRank2Symm.

`b` is a TensorRank1 type.

Description: Read / write access to element in specified position.

Returns: a reference or constant reference to element in specified position.

Examples:

```
double aValue = aTensorRank2(2,3);
aTensorRank2(0,0) = aValue;
aTensorRank1(0) = anotherTensorRank1(1);
```

Assignment operator

Call: `A = B;`

Parameters: A is any tensor type. B can be a scalar, a std::string or char* of components (separated by spaces) or a tensor of the same type as A. B is used as a constant reference.

Description: Copies each parameter in object `B` to the object `A`.

Returns: A reference to the object making possible functions and operators linkage.

Examples:

```
char* data1 = ‘‘1 2 3’’;
aTensorRank1Dim3 = data1;
std::string data2(‘‘1 2 1 2’’);
aTensorRank2Dim2 = data2;
anotherTensorRank2Dim2 = aTensorRank2Dim2;
```

Scale operator

Call: `A *= z;`

Parameters: A is any tensor type. z is a scalar type used as constant reference.

Description: Scales A by multiplying every element by z.

Example:

```
double scalar = 2.5;
aTensorRank2 *= scalar;
```

Add operator

Call: `A += B;`

Parameters: A and B are any tensor type but of the same type. B is used as constant reference.

Description: Adds the tensor B to A.

Example:

```
aTensorRank2 += anotherTensorRank2;
```

Substract operator

Call: `A -= B;`

Parameters: A and B are any tensor type but of the same type. B is used as constant reference.

Description: Substracts the tensor B to A.

Example:

```
aTensorRank2 -= anotherTensorRank2;
```

Streaming operator

Call: `aStream << A;`

Parameters: A is any tensor type used as constant reference.

Description: passes the element of A to the stream.

Returns: a reference to the stream to be able to nest outputs.

Example:

```
cout << A << endl << B;
```

Chapter 4

Linear systems

The LinearSystem class implements a typical system with a rhs vector:

$$\mathbf{A}\ \boldsymbol{x} = \boldsymbol{b} \tag{4.1}$$

or a multiple linear system scheme with a shared LHS matrix and multiple RHS vectors. This is formulated in a matrix form as:

$$\mathbf{A}\ \boldsymbol{X} = \boldsymbol{B} \tag{4.2}$$

4.1 Linear solver types

The different solvers available (shown in table 4.1) are switched by the `Selector` class using the function `setLinSolverType(int)`. Depending on Matrix and linear solver types selected, the combinations shown in table 4.1 are possible.

When a solver is not available, an error will be thrown.

lin_solver_type	**Description**
0	Direct for symmetric matrices
1	Direct for non-symmetric matrices
2	Iterative for symmetric matrices
3	Iterative for non-symmetric matrices

Table 4.1: Linear solvers types.

IMPORTANT LIMITATIONS:

1. `gmm::lu_solve` is not implemented when the vector type is set to code 1 (Type_gmmVector_sparse). *To Be Done.*

2. `lin_solver_type` switch does nothing when solving multiple systems (i.e. $\boldsymbol{AX} = \boldsymbol{B}$) as at the only solver supported for this systems is the `lmx::LU` solver.

lin_solver_type	matrix_type	Solver used:
0	0	lmx::LU
0	1	SuperLU
0	2	gmm::lu_solve
0	3	gmm::lu_solve
1	0	lmx::LU
1	1	SuperLU
1	2	gmm::lu_solve
1	3	gmm::lu_solve
2	0	lmx::Cg
2	1	lmx::Cg
2	2	lmx::Cg
2	3	lmx::Cg
3	0	-
3	1	-
3	2	-
3	3	gmm::gmres

Table 4.2: Linear solvers used for each Matrix type.

4.2 Linear systems functions

4.2.1 Constructors

All of this constructors belong to the `lmx` namespace, so the user can choose between defining a `using namespace lmx` (or other `using` statements) or simply adding a `lmx::` prefix when defining a new `lmx::LinearSystem<>`.

Empty constructor:

Call: `LinearSystem<T>();`

Description: Creates an empty LinearSystem object.

Example:

```
LinearSystem<float> my_empty_linSys;
```
creates an empty LinearSystem object called *my_empty_linSys*.

Constructors with two parameters:

Call: `LinearSystem(A, b);`
`LinearSystem(A, B);`

Parameters: `A` and `B` are Matrix or DenseMatrix objects and `b` is a Vector. Both data types must be the same but can differ from the data type of the linear system (see the example provided).

Description: Creates a linear system object specifying the LHS matrix and the RHS vector or matrix. The system is defined by $\mathbf{Ax} = \mathbf{b}$ or $\mathbf{AX} = \mathbf{B}$ in each case.

Example:

```
Matrix<float> A(3,3);
Vector<float> b(3);
LinearSystem<double> my_lin_sys(A, b);
```

creates a linear system from a 3x3 matrix and a 3-sized vector.

Constructors with three parameters:

Call: `LinearSystem(A, x, b);`
`LinearSystem(A, X, B);`

Parameters: `A`, `X` and `B` are Matrix or DenseMatrix objects and `x` and `b` are Vectors. `A`, `B` and `b` data must be of the same type but can differ from the data type of the linear system (see the example provided). Matrix `X` and vector `x` data must be of the same type of the linear system.

Description: Creates a linear system object specifying the LHS matrix and the RHS matrix or vector. The system is defined by $\mathbf{Ax} = \mathbf{b}$ or $\mathbf{AX} = \mathbf{B}$ in each case.

Example:

```
Matrix<float> A(3,3);
Vector<float> b(3);
Vector<double> x(3);
LinearSystem<double> my_lin_sys(A, x, b);
```

creates a linear system from a 3x3 matrix and two 3-sized vector of different types..

4.2.2 Member functions

Information output:

Call: `void setInfo(level);`

Parameters: `level` is of type `int`.

Description: Sets the level of information printed to standard output while solving the system. Values of `level`: 0 for no output, 1 for standard output.

Example:

```
NLSolver<MyExternalSystem, double> my_system();
my_system.setInfo(0);
```
Defines no output for the solver.

Solving linear systems:

Call: `void solveYourself( recalc = 0 );`

Parameters: `recalc` is of type `int`.

Description: Solves the linear system with the linear solver selected. *Only for direct methods:* The parameter `recalc` can be set to `1` if a previous solution was calculated in order to use the same factorization.

Examples:

```
my_system.solveYourself();
```
solves the linear system `my_system`.

```
my_solution_vector = my_system.solveYourself();
```
solves the linear system *my_system* and saves the result in *my_solution_vector*.

Solution access for single system:

Call: `Vector& getSolutionVector();`

Returns: A reference to the solution Vector $\boldsymbol{x} = \boldsymbol{A}^{-1}\ \boldsymbol{b}$.

Examples:

```
my_system.solveYourself();
my_solution_vector = my_system.getSolutionVector();
```

solves the linear system *my_system* and then saves the result in *my_solution_vector* afterwards.

Solution access for multiple linear systems:

Call: `Matrix& getSolutionMatrix();`

Returns: A reference to the solution Matrix $\boldsymbol{X} = \boldsymbol{A}^{-1}\ \boldsymbol{B}$.

Examples:

```
my_system.solveYourself();
my_solution_matrix = my_system.getSolutionMatrix();
```

solves the linear system *my_system* and then saves the result in *my_solution_matrix* afterwards.

Chapter 5

Nonlinear systems

The numerical solvers that can solve a nonlinear system in the general residual form,

$$\mathbf{R}(\mathbf{x}) = \mathbf{0}$$

are wrapped in the `lmx::NLSolver` class. As this class is not included beneath the general include file *"lmx.h"*, the header file *"lmx_nlsolvers.h"* must be included explicitly in order to have the nonlinear solvers functionality.

Note the difference in the name of the `lmx::NLSolver<Sys, T>` class compared to the `lmx::LinearSystem<T>`. While this last one implements the system from a matrix and at least one *lhs* vector, the `lmx::NLSolver<Sys, T>` class needs the nonlinear system to be implemented outside of the LMX library. The way this class can know which external class will define the nonlinear functions is by the first parameter of the template list `Sys`.

5.1 Nonlinear solver types

As in the case of linear solvers, the structure of the `NLSolver` class is prepared to implement multiple solvers. Although at this time only the Newton-Rhapson Method is working, there are plans to also introduce quasi-Newton methods such as the Broyden-Fletcher-Goldfarb-Shanno (BFGS) method.

At the moment, as there is only one solver available, no solver's selection function is implemented.

5.2 Nonlinear solver functions

5.2.1 Constructors

All of this constructors belong to the `lmx` namespace, so the user can choose between defining a `using namespace lmx` (or other `using` statements) or simply adding a `lmx::` prefix when defining a new `lmx::NLSolver<>`.

Empty constructor:

Call: `NLSolver<Sys, T>();`

Description: Creates an nonlinear solver object.

Example:

```
NLSolver<MyExternalSystem, double> my_nlSolver;
```
creates a NLSolver object called *my_nlSolver*.

5.2.2 Member functions

Information output:

Call: `void setInfo(level);`

Parameters: `level` is of type `int`.

Description: Sets the level of information printed to standard output while solving the system. Values of `level`: 0 for no output, 1 for standard output of each iteration.

Example:

```
NLSolver<MyExternalSystem, double> my_system();
my_system.setInfo(0);
```
Defines no output for the solver.

Initial configuration and system dimension:

Call: `setInitialConfiguration( b );`

Parameters: *b* is a Vector. Note that it can have different data type than the nonlinear solver object but **its size must be consistent with what the external system expects**.

Description: Defines a Vector as the initial guess of the iterative solver procedure. In fact, it represents the only way to tell the solver what the nonlinear system dimension is, so its definition is mandatory. The iterations start with a copy of this vector, so its values are not modified.

Example:

```
Vector<float> b(3);
b.fillIdentity();
NLSolver<MyExternalSystem, double> my_nlSolver();
my_nlsolver.setInitialConfiguration( b );
```
Creates a nonlinear solver with a 3-sized identity vector as an initial guess.

Nonlinear system setting:

Call: `void setSystem( theSystemObject );`

Parameters: `theSystemObject` is of type `Sys`.

Description: Sets which Sys object is going to be used for function calls such as the residue or jacobian.

Example:

```
MyExternalSystemClass aSystemObject;
NLSolver<MyExternalSystemClass, double> my_nlsolver;
my_nlsolver.setSystem(aSystemObject);
```

Sets *aSystemObject* as the target of the *NLSolver* object, *my_nlsolver*, for function calls.

Working with increments:

Call: `void setDeltaInResidue( state );`

Parameters: `state` is of type `bool`. TRUE (default) if the variable's increment is going to be passed.

Description: Sets whether the parameter in Residue function corresponds to the actual variables configuration or indicates the increment of those variables.

Example:

```
my_nlsolver.setIncrement(1);
```

Residual function setting:

Call: `void setResidue( residueMemberFunction );`

Parameters: `residueMemberFunction` is a pointer to a member function of class `Sys`.

Description: Defines the member function that computes the residue.

Example:

```
MyExternalSystemClass aSystemObject;
NLSolver<MyExternalSystemClass, double> my_nlsolver;
my_nlsolver.setResidue(
&MyExternalSystemClass::externalClassResidue);
```

Sets *externalClassResidue* as the residue function for the *NLSolver* object algorithm.

Jacobian function setting:

Call: `void setJacobian( jacobianMemberFunction );`

Parameters: `jacobianMemberFunction` is a pointer to a member function of class `Sys`.

Description: Defines the member function that computes the jacobian.

Example:

```
MyExternalSystemClass aSystemObject;
NLSolver<MyExternalSystemClass, double> my_nlsolver;
my_nlsolver.setJacobian(
&MyExternalSystemClass::externalClassJacobian);
```

Sets *externalClassJacobian* as the residue function for the *NLSolver* object algorithm.

Convergence norm tolerance setting:

Call: `void setConvergence( epsilon );`

Parameters: `epsilon` is a `double` precision value. Default value is 1E-5.

Description: Defines the **optional** value of the precision in terms of the L2 norm of the residue.

Example:

```
MyExternalSystemClass aSystemObject;
NLSolver<MyExternalSystemClass, double> my_nlsolver;
my_nlsolver.setResidue( 1E-6 );
```

Sets a tolerance value for the residual L2 norm.

Convergence function setting:

Call: `void setConvergence( convergenceMemberFunction );`

Parameters: `convergenceMemberFunction( 1[, 2, 3] )` is a pointer to a member function of class `Sys`. It can have 1, 2 or 3 parameters, representing in this order:

1. Residue Vector,
2. Configuration Vector, and
3. Increments Vector.

Description: Defines the **optional** member function for convergence evaluation.

Example:

```
MyExternalSystemClass aSystemObject;
NLSolver<MyExternalSystemClass, double> my_nlsolver;
my_nlsolver.setResidue(
&MyExternalSystemClass::externalClassConvergence);
```

Sets *externalClassConvergence* as the residue function for the *NLSolver* object algorithm.

Solve function:

Call: `void solve( max_iter );`

Parameters: `max_iter` is an `int` type.

Description: Initiates the nl-solver loop.

Example:

```
my_nlsolver.solve( 1000 );
```
Solves the system with a maximum of 1000 iterations.

Solution access:

Call: `lmx::Vector<T>& getSolution();`

Returns: A reference to the solution Vector.

Description: R-W solution Vector access.

Example:

```
cout << my_nlsolver.getSolution( ) << endl;
```
Prints the solution Vector.

5.3 Example of a nonlinear system implementation

The sample code next shows how to use the nonlinear solver class to solve a system declared and implemented outside the library. First, MyExternalSystem class is declared. Afterwards, in the main function, an object of that class is solved using the LMX nonlinear solver.

```
class MyExternalSystem{
public:
  MyExternalSystem(){}
  ~MyExternalSystem(){}
  void myResidue(lmx::Vector<double>& res_in,
                 lmx::Vector<double>& q_in);
  void myJacobian(lmx::Matrix<double>& jac_in,
                  lmx::Vector<double>& q_in);
  bool myConvergence( lmx::Vector<double>& res_in );

  private:
     ...
};
// Definition of MyExternalSystem member functions
// should be here:
// ...
//

int main(int argc, char* argv){

  lmx::setMatrixType(0);
  lmx::setVectorType(0);
  lmx::setLinSolverType(0);

  lmx::Vector<double> b(3);

  b.fillIdentity();

  MyExternalSystem theSystem;
  lmx::NLSolver<MyExternalSystem> theSolver;

  theSolver.setInitialConfiguration( b );
  theSolver.setSystem( theSystem );
  theSolver.setResidue( &MyExternalSystem::myResidue );
  theSolver.setJacobian( &MyExternalSystem::myJacobian );
  theSolver.setConvergence( &MyExternalSystem::myConvergence );
  theSolver.solve( 10 ); // max. iterations = 10
  cout << theSolver.getSolution() << endl;
}
```

Chapter 6

Differential systems

This chapter explains the way to solve first and second order ODE systems using LMX. For this purpose two basic classes have been implemented: `DiffProblemFirst<Sys,T>` and `DiffProblemSecond<Sys,T>`, respectively.

For using either of both classes, their header file must be included explicitly. So, `lmx_diffproblem_first.h` or `lmx_diffproblem_second.h` should be included as well as the global `lmx.h` header file.

As occurs with the `NLSolver` class, the problem classes are templatized for the definition of the external functions within an (also external) class, represented by parameter `Sys`. The functions needed by the problems depend on the integrator used and nonlinear solver selected—in case of using an implicit integrator scheme—.

The next sections present a classification of the available integrators and a detailed description of the classes and functions that constitute the differential systems API of LMX.

6.1 Integrator types

The integrator types that are already working or planned to be implemented (marked with*) are the following:

Multistep linear integrators: Multiple order backward differential formulae (BDF) methods and multiple orderAdams-Bashford (explicit) and Adams-Moulton (implicit).

Runge-Kutta methods*: forth-fifth order Runge-Kutta-Fehlberg.

Structural integrators[1]**:** β-Newmark family integrators (explicit, central differences, trapezoidal rule, etc.), HHT-alpha method and Exact energy-momentum conserving algorithm*.

[1]Not available for first order ODEs

The way to select an integrator differs from the strategy used in the selection of matrices or linear solvers. For switching the integrators, a member function `setIntegrator` is defined and overloaded for the different differential problems. The possible arguments of this function are shown in next tables.

When an implicit integrator scheme is selected, the `NLSolver` object is automatically created.

Integrator	**Argument 1**	**Arg 2**	**Arg 3**	**Arg 4**
Adams-Bashford, first order	"AB-1"			
Adams-Bashford, second order	"AB-2"			
Adams-Bashford, third order	"AB-3"			
Adams-Bashford, fourth order	"AB-4"			
Adams-Bashford, fifth order	"AB-5"			
Adams-Moulton, first order	"AM-1"			
Adams-Moulton, second order	"AM-2"			
Adams-Moulton, third order	"AM-3"			
Adams-Moulton, fourth order	"AM-4"			
Adams-Moulton, fifth order	"AM-5"			
Backward Differences, first order	"BDF-1"			
Backward Differences, second order	"BDF-2"			
Backward Differences, third order	"BDF-3"			
Backward Differences, fourth order	"BDF-4"			
Backward Differences, fifth order	"BDF-5"			
Central differences	"CD"			
β-Newmark	"Newmark"	β	γ	
Fox & Goodwin	"Newmark"	1/12	1/2	
Linear acceleration	"Newmark"	1/6	1/2	
Trapezoidal rule	"Newmark"	1/4	1/2	
β-Newmark	"Newmark"	β	γ	α
HHT-α	"Alpha"	α		
HHT-α	"Alpha"	β	γ	α

Table 6.1: `setIntegrator` function arguments description

6.2 Differential problem classes

6.2.1 Constructors

Empty constructors:

Call: `DiffProblemFirst<Sys,T>();` &
`DiffProblemSecond<Sys,T>();`

Description: Creates a differential problem object of first or second order, respectively.

Example:

```
DiffProblemFirst<ExternalDiffSystemClass, double>
my_diffproblem;
```
creates a first order differential object called *my_diffproblem*.

6.2.2 General member functions

Differential system setting:

Call: `void setDiffystem( theSystemObject );`

Parameters: `theSystemObject` is of type `Sys`.

Description: Sets which differential Sys object is going to be used for function calls such as the residue or jacobian.

Example:

```
MyExternalSystemClass aSystemObject;
DiffProblemFirst<MyExternalSystemClass, double>
my_problem;
my_problem.setSystem(aSystemObject);
```

Sets *aSystemObject* as the target of the *DiffProblemFirst* object, *my_problem*, for function calls.

Integrator setting:

Call: `void setIntegrator(type, opt1=0, opt2=0, opt3=0);`

Parameters: `type` is a `char*`. `opt1`, `opt2` & `opt 3` are `int` or `double` with default zero value. See table 6.1 for detailed values for each option.

Description: Sets the integrator used to solve the problem and possible optional values.

Example:

```
DiffProblemFirst<MyExternalSystemClass, double>
my_problem;
my_problem.setIntegrator(''Newmark'', 0.25, 0.5);
```

Sets the integrator as a Newmark scheme with β = and $\gamma = 0.5$.

Output to file:

Call: `void setOutputFile( filename, diffOrder )`

Parameters: `filename` is a `char*` and `diffOrder` is `int`, from 0 to 2, depending on the differential problem order.

Description: Defines the name of the file for storing all the results for a specific order of the results.

Example:

```
DiffProblemFirst<MyExternalSystemClass, double>
my_problem;
my_problem.setOutputFile(‘‘displacements.dat’’, 0);
```

Stores the 0-order results in a file named *displacements.dat*.

Time parameters setting:

Call: `void setTimeParameters( to, tf, stepSize )`

Parameters: `to`, *tf* and *stepSize* are `double` type and indicate the start time, the end time and the suggested size of time increments, respectively.

Description: Defines the basic time controls.

Example:

```
DiffProblemFirst<MyExternalSystemClass, double>
my_problem;
my_problem.setTimeParameters(0, 10, 0.1);
```

Defines the integration, beginning at $t_o = 0$, ending at $t_f = 10$ and with increments of $t_{n+1} - t_n = 0.1$.

Solving the problem:

Call: `void solve( )`

Description: Begins the integration of the differential system, from the beginning to the end so everything must be set up before if you want no errors.

Example:

```
DiffProblemFirst<MyExternalSystemClass, double>
my_problem;
// set everything up here...  then
my_problem.solve();
```

6.2.3 First order specific functions

Residue setting:

Call: `void setResidue( *residueFunction )`

Parameters: `*residueFunction` is a pointer to member function that can have any name but must have the next parameters:

```
void residueFunction(lmx::Vector<T>& residue,
                     const lmx::Vector<T>& q,
                     const lmx::Vector<T>& qdot,
                     double time);
```

Description: Sets the residual member function.

Example:

```
DiffProblemFirst<MyExtSysClass, double> my_problem;
my_problem.setResidue(&MyExtSysClass::resFunction);
```

Jacobian setting:

Call: `void setJacobian( *jacobianFunction )`

Parameters: `*jacobianFunction` is a pointer to member function that can have any name but must have the next parameters:

```
void jacobianFunction(lmx::Matrix<T>& jacobian,
                      const lmx::Vector<T>& q,
                      const lmx::Vector<T>& qdot,
                      double partialqdot);
```

where `partialqdot` is the $\frac{\partial \dot{q}}{\partial q}$ computed by the integrator.

Description: Sets the tangent member function. This is needed when an implicit integrator scheme is used along with the (default) Newton-Rhapson nonlinear solver.

Example:

```
DiffProblemFirst<MyExtSysClass, double> my_problem;
my_problem.setJacobian(&MyExtSysClass::jacFunction);
```

Evaluation setting:

Call: `void setEvaluation( *evaluationFunction )`

Parameters: `*evaluationFunction` is a pointer to member function that can have any name but must have the next parameters:

```
void evaluationFunction(const lmx::Vector<T>& q,
                        lmx::Vector<T>& qdot,
                        double time);
```

Description: Sets the function that computes the first derivative. Needed for explicit integrators and for the first iteration in the implicit integrator schemes.

Example:

```
DiffProblemFirst<ExtSysClass, double> my_problem;
my_problem.setEvaluation(&ExtSysClass::evalFunction);
```

Convergence setting:

Call: `void setConvergence( *convergenceFunction )`

Parameters: `*convergenceFunction` is a pointer to member function that can have any name but must have the next parameters and return type:

```
bool convergenceFunction(const lmx::Vector<T>& q,
                         const lmx::Vector<T>& qdot,
                         double time);
```

Description: (Optional) Sets a function as an convergence criteria. If no function is set, the L_2 euclidean norm of the residue is used.

Example:

```
DiffProblemFirst<ExtSysClass, double> my_problem;
my_problem.setConvergence(&ExtSysClass::convFunction);
```

6.2.4 Second order specific functions

Residue setting:

Call: `void setResidue( *residueFunction )`

Parameters: `*residueFunction` is a pointer to member function that can have any name but must have the next parameters:

```
void residueFunction(lmx::Vector<T>& residue,
                     const lmx::Vector<T>& q,
                     const lmx::Vector<T>& qdot,
                     const lmx::Vector<T>& qddot,
                     double time);
```

Description: Sets the residual member function.

Example:

```
DiffProblemFirst<MyExtSysClass, double> my_problem;
my_problem.setResidue(&MyExtSysClass::resFunction);
```

Jacobian setting:

Call: `void setJacobian( *jacobianFunction )`

Parameters: `*jacobianFunction` is a pointer to member function that can have any name but must have the next parameters:

```
void jacobianFunction(lmx::Matrix<T>& jacobian,
                      const lmx::Vector<T>& q,
                      const lmx::Vector<T>& qdot,
                      const lmx::Vector<T>& qddot,
                      double partialqdot,
                      double partialqddot);
```

where `partialqdot` and `partialqddot` are the $\frac{\partial \dot{q}}{\partial q}$ $\frac{\partial \ddot{q}}{\partial q}$, respectively, computed by the integrator.

Description: Sets the tangent member function. This is needed when an implicit integrator scheme is used along with the (default) Newton-Rhapson nonlinear solver.

Example:

```
DiffProblemFirst<MyExtSysClass, double> my_problem;
my_problem.setJacobian(&MyExtSysClass::jacFunction);
```

Residue by parts setting:

Call: `void setResidueByParts( *residue_q_qdot, *residue_qddot, *residue_time )`

Parameters: `*residue_q_qdot`, `*residue_qddot` and `*residue_time` are pointer to member functions that can have any name but must have the next parameters each:

```
void residue_q_qdot(lmx::Vector<T>& residue,
                    const lmx::Vector<T>& q,
                    const lmx::Vector<T>& qdot);
void residue_qddot(lmx::Vector<T>& residue,
                   const lmx::Vector<T>& qddot);
void residue_time(lmx::Vector<T>& residue,
                  double time);
```

Description: Sets the residual as a sum of the contributions of different parts. Needed for the HHT-α integrator.

Example:

```
DiffProblemFirst<MyExtSysClass, double> my_problem;
my_problem.setResidueByParts(
&MyExtSysClass::resQQdot,
                          &MyExtSysClass::resQddot,
                          &MyExtSysClass::resTime );
```

Jacobian by parts setting:

Call: `void setJacobianByParts( *jacobian_q_qdot, *jacobian_qddot)`

Parameters: `*jacobian_q_qdot` and `*jacobian_qddot` are pointer to member functions that can have any name but must have the next parameters each:

```
void jacobian_q_qdot(lmx::Matrix<T>& jacobian,
                     const lmx::Vector<T>& q,
                     const lmx::Vector<T>& qdot,
                     double partial_qdot);
void jacobian_qddot(lmx::Matrix<T>& jacobian,
                     const lmx::Vector<T>& qddot,
                     double partial_qddot);
```

and `partial_qdot`, `partial_qddot` are the $\frac{\partial \dot{q}}{\partial q}$ $\frac{\partial \ddot{q}}{\partial q}$, respectively, computed by the integrator.

Description: Sets the jacobian as a sum of the contributions of different parts. Needed for the HHT-α integrator.

Example:

```
DiffProblemFirst<MyExtSysClass, double> my_problem;
my_problem.setJacobianByParts
                    ( &MyExtSysClass::jacQQdot,
                      &MyExtSysClass::jacQddot );
```

Evaluation setting:

Call: `void setEvaluation( *evaluationFunction )`

Parameters: `*evaluationFunction` is a pointer to member function that can have any name but must have the next parameters:

```
void evaluationFunction(const lmx::Vector<T>& q,
                        const lmx::Vector<T>& qdot,
                        lmx::Vector<T>& qddot,
                        double time);
```

Description: Sets the function that computes the second derivative. Needed for explicit integrators and for the first iteration in the implicit integrator schemes.

Example:

```
DiffProblemFirst<ExtSysClass, double> my_problem;
my_problem.setEvaluation(&ExtSysClass::evalFunction);
```

Convergence setting:

Call: `void setConvergence( *convergenceFunction )`

Parameters: `*convergenceFunction` is a pointer to member function that can have any name but must have the next parameters and return type:

```
bool convergenceFunction(const lmx::Vector<T>& q,
                         const lmx::Vector<T>& qdot,
                         const lmx::Vector<T>& qddot,
                         double time);
```

Description: (Optional) Sets a function as an convergence criteria. If no function is set, the L_2 euclidean norm of the residue is used.

Example:

```
DiffProblemFirst<ExtSysClass, double> my_problem;
my_problem.setConvergence(&ExtSysClass::convFunction);
```

6.3 Differential systems example

```
#include"LMX/lmx.h"
#include"LMX/lmx_diff_problem_first.h"

using namespace std;

// System: qdot + K*q = f(t)
class MyDiffSystem{
  public:
    MyDiffSystem()
    {
      K.resize(2,2);
      K(0,0) = 2.;
      K(1,1) = 3.;
    }

    ~MyDiffSystem(){}

    void myEvaluation( const lmx::Vector<double>& q,
                       lmx::Vector<double>& qdot,
                       double time
                     )
    {
      qdot(0) += time;
      qdot(1) += 2*time;
      qdot -= K*q;
    }

    void myResidue( lmx::Vector<double>& residue,
                    const lmx::Vector<double>& q,
                    const lmx::Vector<double>& qdot,
                    double time
                  )
    {
      residue = qdot + K*q;
      residue(0) -= time;
      residue(1) -= 2.*time;
    }

    void myTangent( lmx::Matrix<double>& tangent,
                    const lmx::Vector<double>& q,
                    const lmx::Vector<double>& qdot,
                    double partial_qdot
                  )
    {
      tangent.fillIdentity( partial_qdot );
```

```
46      tangent += K;
47    }
48
49  private:
50    lmx::Matrix<double> K;
51};
52
53int main(int argc, char *argv[])
54{
55
56  lmx::setMatrixType( 0 );
57  lmx::setVectorType( 0 );
58  lmx::setLinSolverType( 0 );
59
60  lmx::DiffProblemFirst< MyDiffSystem > theProblem;
61  MyDiffSystem theSystem;
62  // Initial conditions:
63  lmx::Vector<double> q0(2);
64  q0(0) = 0.2;
65  q0(1) = 0.;
66
67  theProblem.setDiffSystem( theSystem );
68  theProblem.setIntegrator( "BDF-2" );
69  theProblem.setInitialConfiguration( q0 );
70  theProblem.setTimeParameters( 0, 5, 0.04 );
71  theProblem.setOutputFile("disp.dat", 0);
72  theProblem.setOutputFile("vel.dat", 1);
73  theProblem.setEvaluation( &MyDiffSystem::myEvaluation );
74  theProblem.setResidue( &MyDiffSystem::myResidue );
75  theProblem.setJacobian( &MyDiffSystem::myTangent );
76  theProblem.solve();
77
78  return 1;
79}
```