



Developer documentation

Nicolas KIELBASIEWICZ, Eric LUNÉVILLE, Manh
Ha NGUYEN, Yvon LAFRANCHE

April 28, 2018



Contents

1	General organisation	8
1.1	Licence	8
1.2	Credits	8
1.3	Installation	8
1.3.1	Under MacOS and Linux	9
1.3.2	Under Windows	9
1.4	Compilation	10
1.4.1	Under MacOS and Linux	10
1.4.2	Under Windows	11
1.4.3	Useful general options to CMAKE	12
1.4.4	Configuring XLIFE++ with GMSH and/or PARAVIEW	13
1.4.5	Configuring XLIFE++ with OPENMP	13
1.4.6	Configuring XLIFE++ with ARPACK	13
1.4.7	Configuring XLIFE++ with UMFPACK	13
1.5	Folder organisation	14
1.6	Namespace	14
1.7	Headers	15
1.8	Development principles	16
2	Initialisation / finalization and global definitions	18
2.1	About the <code>init</code> function	18
2.2	Global declarations and definitions	19
2.2.1	<code>setup.hpp</code>	19
2.2.2	<code>config.hpp</code>	19
2.2.3	<code>globalScopeData.hpp</code>	21
2.3	About the <code>finalize</code> function	21
3	The <i>utils</i> library	22
3.1	The <code>Environment</code> class	23
3.1.1	Using the dictionary	24
3.2	Messages management	24
3.2.1	The <code>MsgData</code> class	25
3.2.2	The <code>MsgFormat</code> class	26
3.2.3	The <code>Messages</code> class	26
3.2.4	External functions	27
3.2.5	Throwing messages	28
3.3	Managing traces	29
3.3.1	The <code>Trace</code> class	29
3.3.2	How to use the verbose level ?	30
3.3.3	How to know where does messages come ?	30
3.4	The <code>Timer</code> class	31
3.5	The <code>Memory</code> class	32
3.6	The <code>String</code> class	33

3.7	The <code>Point</code> class	34
3.8	The <code>Vector</code> class	36
3.9	The <code>Matrix</code> class	38
3.10	The collection classes	42
3.10.1	The <code>Collection</code> class	42
3.10.2	The <code>PCollection</code> class	43
3.11	Parameters management	44
3.11.1	The <code>Parameter</code> class	44
3.11.2	The <code>Parameters</code> class: list of parameters	46
3.12	The <code>Function</code> class	47
3.13	The <code>Kernel</code> class	51
3.14	The <code>TensorKernel</code> class	54
3.15	The <code>ThreadData</code> class	56
3.16	The <code>SymbolicFunction</code> class	58
3.17	The <code>Graph</code> class	61
3.18	The <code>Value</code> class	61
3.19	The <code>KdTree</code> class	63
3.20	The polynomials classes	66
3.20.1	The <code>MonomialT</code> class	67
3.20.2	The <code>PolynomialT</code> class	67
3.20.3	The <code>PolyNodeT</code> class	69
3.20.4	The <code>PolynomialBasisT</code> class	70
3.20.5	The <code>PolynomialsBasisT</code> class	72
4	The <i>geometry</i> library	75
4.1	Defining geometries	75
4.1.1	<code>BoundingBox</code> and <code>MinimalBox</code> classes	75
4.1.2	The <code>Geometry</code> class	78
4.1.3	<code>Curve</code> , <code>Surface</code> and <code>Volume</code> classes	82
4.1.4	Definition of canonical geometries	84
4.2	Mesh management	111
4.2.1	The <code>Mesh</code> class	112
4.2.2	Meshing tools	115
4.3	Domain management	121
4.3.1	The <code>GeomDomain</code> base class	121
4.3.2	The <code>MeshDomain</code> child class	123
4.3.3	The <code>CompositeDomain</code> child class	125
4.3.4	The <code>PointsDomain</code> child class	126
4.3.5	The <code>AnalyticalDomain</code> child class	126
4.3.6	The <code>Domains</code> class	126
4.4	Geometric element	127
4.4.1	The <code>GeomElement</code> class	128
4.4.2	The <code>MeshElement</code> class	130
4.5	Geometric map data	131
4.5.1	Differential calculus	131
4.5.2	The <code>GeomMapData</code> class	134
5	The <i>finiteElements</i> library	135
5.1	The <code>Interpolation</code> class	135
5.2	The <code>RefDof</code> class	137
5.3	Geometric data of a reference element	139
5.3.1	The <code>GeomRefElement</code> class	139

5.3.2	Child classes of <code>GeomRefElement</code>	141
5.4	Reference elements	142
5.4.1	The <code>ShapeValues</code> class	142
5.4.2	The <code>RefElement</code> class	143
5.4.3	Child classes for segment	145
5.4.4	Child classes for triangle	146
5.4.5	Child classes for quadrangle	148
5.4.6	Child classes for tetrahedron	149
5.4.7	Child classes for hexahedron	151
5.4.8	Child classes for prism	151
5.4.9	Child classes for pyramid	152
5.5	Examples of elements	152
5.5.1	Examples of triangles	152
5.5.2	Examples of quadrangles	153
5.5.3	Examples of tetrahedra	154
5.5.4	Examples of hexahedra	155
5.5.5	Examples of prisms	156
5.5.6	Examples of pyramids	157
5.6	Integration methods	158
5.6.1	The <code>IntegrationMethod</code> , <code>SingleIM</code> , <code>DoubleIM</code> classes	159
5.6.2	The <code>QuadratureIM</code> class	160
5.6.3	The <code>ProductIM</code> class	160
5.6.4	The <code>Quadrature</code> , <code>QuadratureRule</code> classes	160
5.6.5	How to add a new quadrature formula ?	167
5.6.6	Integration methods for singular kernels	167
5.6.7	The <code>IntegrationMethods</code> class	169
6	The <i>space</i> library	171
6.1	Space management	171
6.1.1	The <code>Space</code> and <code>SpaceInfo</code> classes	171
6.1.2	The <code>FeSpace</code> class	174
6.1.3	The <code>SpSpace</code> class	176
6.1.4	The <code>SubSpace</code> and <code>FeSubSpace</code> classes	176
6.1.5	The <code>ProdSpace</code> class	178
6.1.6	The <code>Spaces</code> class	178
6.2	DoF management	179
6.2.1	The <code>Dof</code> class	179
6.2.2	<code>FeDof</code> and <code>SpDof</code> classes	179
6.2.3	The <code>DofComponent</code> class	180
6.3	Unknown management	181
6.3.1	The <code>Unknown</code> class	181
6.3.2	The <code>ComponentOfUnknown</code> class	182
6.3.3	The <code>Unknowns</code> class	182
6.4	The <code>Element</code> class	183
6.5	Spectral basis management	185
6.5.1	The <code>SpectralBasis</code> class	185
6.5.2	The <code>SpectralBasisFun</code> class	186
6.5.3	The <code>SpectralBasisInt</code> class	186

7	The <i>operator</i> library	187
7.1	The <code>DifferentialOperator</code> class	187
7.2	The <code>Operand</code> class	189
7.3	The <code>OperatorOnUnknown</code> class	190
7.3.1	Operations between <code>Unknown</code> and <code>Function</code>	192
7.3.2	Operations between <code>Unknown</code> and <code>Value</code>	193
7.3.3	Operations between <code>OperatorOnUnknown</code> and <code>Function</code>	193
7.3.4	Operations between <code>OperatorOnUnknown</code> and <code>Value</code>	193
7.4	The <code>OperatorOnUnknowns</code> class	194
7.5	The <code>LcOperatorOnUnknown</code> class	195
7.6	The <code>OperatorOnKernel</code> class	198
7.7	The <code>KernelOperatorOnUnknowns</code> class	199
7.8	The <code>TensorKernel</code> class	201
8	The <i>form</i> library	203
8.1	Linear forms	205
8.1.1	The <code>BasicLinearForm</code> class	205
8.1.2	The <code>IntgLinearForm</code> class	205
8.1.3	The <code>DoubleIntgLinearForm</code> class	206
8.1.4	The <code>SuLinearForm</code> class	206
8.1.5	The <code>LinearForm</code> class	207
8.2	Bilinear forms	209
8.2.1	The <code>BasicBilinearForm</code> class	209
8.2.2	The <code>IntgBilinearForm</code> class	209
8.2.3	The <code>DoubleIntgBilinearForm</code> class	210
8.2.4	The <code>SuBilinearForm</code> class	211
8.2.5	The <code>BilinearForm</code> class	212
9	The <i>largeMatrix</i> library	214
9.1	The <code>LargeMatrix</code> class	215
9.2	The <code>MatrixStorage</code> class	222
9.3	Dense storages	225
9.3.1	The <code>DenseStorage</code> class	226
9.3.2	<code>RowDenseStorage</code> , <code>ColDenseStorage</code> , <code>DualDenseStorage</code> and <code>SymDenseStorage</code> classes	227
9.4	Compressed sparse storages	228
9.4.1	The <code>CsStorage</code> class	230
9.4.2	<code>RowCsStorage</code> and <code>ColCsStorage</code> classes	232
9.4.3	<code>DualCsStorage</code> and <code>SymCsStorage</code> classes	233
9.5	Skyline storages	235
9.5.1	The <code>SkylineStorage</code> class	235
9.5.2	<code>DualSkylineStorage</code> and <code>SymSkylineStorage</code> classes	237
10	The <i>hierarchicalMatrix</i> library	239
10.1	Cluster tree	239
10.1.1	The <code>ClusterNode</code> class	240
10.1.2	The <code>ClusterTree</code> class	242
10.2	Approximate matrix	244
10.2.1	The <code>ApproximateMatrix</code> class	244
10.2.2	The <code>LowRankMatrix</code> class	245
10.3	HMatrix	247
10.3.1	The <code>HMatrixNode</code> class	247

10.3.2	The HMatrix class	249
10.3.3	The HMatrixEntry class	251
10.3.4	The HMatrixIM class	252
11	The <i>term</i> library	254
11.1	The Term class	255
11.2	The TermVector class	256
11.3	The SuTermVector class	263
11.4	The TermVectors class	267
11.5	The KernelOperatorOnTermVector class	267
11.6	The TermMatrix class	268
11.7	The SuTermMatrix class	277
11.8	The LcTerm class	279
11.9	The SymbolicTermMatrix class	281
11.10	The computation algorithms	283
11.10.1	Matrix computation	283
11.10.2	Understanding TermMatrix operations	284
11.10.3	Vector computation	285
11.11	The Projector class	286
12	The <i>essentialConditions</i> library	289
12.1	A general process	289
12.2	The EssentialCondition class	291
12.3	The EssentialConditions class	293
12.4	The Constraints class	294
13	The <i>solvers</i> library	298
13.1	The IterativeSolver class	299
13.2	The Preconditioner class	299
13.3	Solvers	301
13.3.1	The BicgSolver class	301
13.3.2	The BicgStabSolver class	302
13.3.3	The CgSolver class	302
13.3.4	The CgsSolver class	303
13.3.5	The GmresSolver class	303
13.3.6	The QmrSolver class	304
13.3.7	The SorSolver class	304
13.3.8	The SsorSolver class	305
14	The <i>umfpackSupport</i> library	306
14.1	The UMFPACK wrappers	306
14.2	The UMFPACK class	307
14.3	The UmfPackSolver class	309
14.4	The UmfPackLU class	311
15	The <i>eigenSolvers</i> library	314
15.1	<i>eigenCore</i> sub-library	315
15.1.1	<i>utils</i> part	315
15.1.2	<i>decomposition</i> part	318
15.1.3	<i>eigenSolver</i> main part	322
15.2	<i>eigenSparse</i> sub-library	328
15.2.1	The Operator/Multivector Interface	328

15.2.2	<i>eigenSparse</i> framework	332
15.2.3	<i>eigenSparse</i> classes	334
15.2.4	Examples	342
16	The <i>arpackppSupport</i> library	344
16.1	ARPACK++ interface layer	344
16.1.1	The <code>LargeMatrixArpackppStdReg</code> class	346
16.1.2	The <code>LargeMatrixArpackppStdShf</code> class	346
16.1.3	The <code>LargeMatrixArpackppGen</code> class	347
16.1.4	The <code>LargeMatrixArpackppGenReg</code> class	348
16.1.5	The <code>LargeMatrixArpackppSymGenReg</code> class	348
16.1.6	The <code>LargeMatrixArpackppGenShf</code> class	349
16.1.7	The <code>LargeMatrixArpackppGenBuc</code> class	349
16.1.8	The <code>LargeMatrixArpackppGenCay</code> class	350
16.1.9	The <code>LargeMatrixArpackppGenCsh</code> class	350
16.2	Users interface layer	351
16.2.1	The <code>EigenSolverBaseArpackpp</code> class	351
16.2.2	The <code>EigenSolverStdArpackpp</code> class	352
16.2.3	The <code>EigenSolverGenArpackpp</code> class	353
16.2.4	Examples	355
17	Multi-threading with OpenMP	359
17.1	Sparse matrix-vector multiplication	359
17.1.1	SpMV of CSR	359
17.1.2	SpMV of CSC	361
17.2	Sparse matrix factorization	363
A	External libraries	365
A.1	Installation and use of BLAS and LAPACK libraries	365
A.2	Installation and use of UMFPACK library	365
A.3	Installation and use of ARPACK++ library	366
B	Documentation policy	367
B.1	Inline documentation	367
B.1.1	General points	367
B.1.2	Documentation of a file	367
B.1.3	Documentation of a class, a typedef, an enum or a struct	368
B.1.4	Documentation of attributes	369
B.1.5	Documentation of functions and constructors	369
B.1.6	Documentation of arguments	370
B.2	User and developer documentation	370
C	Tests policy	374
C.1	Test families and test functions	374
C.1.1	About test families	374
C.1.2	Definition of a test function	374
C.1.3	Development of a test function	375
C.1.4	Includes	376
C.2	Test of a new version of the library	377
C.2.1	Compilation of tests	377
C.2.2	Running tests	377

D	Compiling and linking policy	378
D.1	General purpose	378
D.2	CMAKE and IDE generators	378
D.2.1	CMAKE and Windows	378
D.2.2	CMAKE and Unix or Mac OS	378
D.2.3	New files and IDE generators	378
D.3	CMAKE and XLIFE++	378
D.3.1	Building libraries	378
D.3.2	Building and linking test executables	379

1

General organisation

This chapter concerns some important information about the organisation of the library: license consideration, folder organisation, installation process, header and namespace consideration, coding rules. Documentation policy, test policy and compiling/linking considerations are the subject of separate chapters.

1.1 Licence

XLiFE++ is copyright (C) 2014 by E. Lunéville and N. Kielbasiewicz and is distributed under the terms of the GNU General Public License (GPL) (Version 3 or later, see <https://www.gnu.org/licenses/gpl-3.0.en.html>). This means that everyone is free to use XLiFE++ and to redistribute it on a free basis. XLiFE++ is not in the public domain; it is copyrighted and there are restrictions on its distribution. You cannot integrate XLiFE++ (in full or in parts) in any closed-source software you plan to distribute (commercially or not). If you want to integrate parts of XLiFE++ into a closed-source software, or want to sell a modified closed-source version of XLiFE++, you will need to obtain a different license. Please contact us directly for more information.

The developers do not assume any responsibility in the numerical results obtained using the XLiFE++ library and are not responsible of bugs.

1.2 Credits

The XLiFE++ library has been mainly developped by E. Lunéville and N. Kielbasiewicz of POEMS lab (UMR 7231, CNRS-ENSTA ParisTech-INRIA). Some parts are inherited from Melina++ library developped by D. Martin (IRMAR lab, Rennes University, now retired) and E. Lunéville. Other contributors are :

- Y. Lafranche (IRMAR lab), mesh tools using subdivision algorithms and wrapper to ARPACK++
- C. Chambeyron (POEMS lab), iterative solvers
- M.H N'Guyen (POEMS lab), eigen solvers and OpenMP implementation

1.3 Installation

XLiFE++ is under a git repository on the INRIA Gforge. So access to source code needs to log to the forge and to define a ssh-key for authentication. The first part is to be registered to INRIA GForge and to be accepted as member of XLiFE++ project. To do so, please do as follows:

1. Please go to <https://gforge.inria.fr/projects/xlifep/>.
2. If you have already had a user account, please go to step #4
3. If you don't have a user account, please create it.
4. If you have already been member of the XLiFE++ project on INRIA GForge, please go to step #6

5. If you are not member of the XLiFE++ project on INRIA GForge, please contact one of the XLiFE++ administrators to join the XLiFE++ development team on <http://uma.ensta-paristech.fr/soft/XLiFE++/> and wait until your registration to XLiFE++ project on INRIA GForge is done to go to next section.
6. For further documentation, you may go to the XLiFE++ main page : <http://uma.ensta-paristech.fr/soft/XLiFE++/>

1.3.1 Under MacOS and Linux

To install XLiFE++ in order to take part to its development, you may use the following steps:

1. Please install GIT (<http://www.git-scm.com>) and CMAKE (<http://cmake.org>).
2. If you have already generated a ssh key and want to use it for XLiFE++ project, please go to step #4
3. If you don't have generated a ssh key, please use the command `ssh-keygen` to generate one.
4. To add the public ssh key you want to use, please go to your account settings on INRIA GForge and add the key by clicking on "Edit keys" at the bottom. Copy the public key where it is needed, validate and register changes to your account. This may take some time so that the key is truly activated.
5. On <https://gforge.inria.fr/projects/xlifepp/>, please go to the "Sources" tab to see the `git clone ...` command-line you have to run in a terminal to create your local XLiFE++ repository.
6. In the root directory of XLiFE++, you have a shell script `pre-commit.sh` and a hidden directory `.git`. Copy the script to `.git/hooks/pre-commit` (without extension) and make the copy executable. When done, you will update file `VERSION.txt` automatically.
7. You can now compile and develop. See next section.

1.3.2 Under Windows

To install XLiFE++ in order to take part to its development, you may use the following steps:

1. Please install GIT (<http://www.git-scm.com>) and CMAKE (<http://cmake.org>). Under Windows, git is usable with the contextual menu showed with clicks on the right mouse button.
2. Please open Git GUI in the contextual menu.
3. Please click on "Cloning existing repository". A window opens.
4. Please open the "Help" tab and select "Show SSH key"
5. Click on "Generate key". The public key will be shown. Copy it.
6. To add the public ssh key, please go to your account settings on INRIA GForge and add the key by clicking on "Edit keys" at the bottom. Copy the public key where it is needed, validate and register changes to your account. This may take some time so that the key is truly activated.
7. To set the "Source location" in the Git GUI window, please go to the "Sources" tab on <https://gforge.inria.fr/projects/xlifepp/> to see the `git clone ...` command-line. Copy the url (beginning by `git+ssh//`). Define a non-existing directory for the "Target' Directory". And click on "Clone" to create your local repository.

8. In the root directory of XLiFE++, you have a shell script *pre-commit.sh* and a hidden directory *.git*. Copy the script to *.git/hooks/pre-commit* (without extension) and make the copy executable. When done, you will update file *VERSION.txt* automatically.
9. You can now compile and develop. See next section.

1.4 Compilation

Compilation of XLiFE++ is managed by CMAKE, a high-level cross-platform compilation tool, running on MacOS, Windows, and Linux. This application only needs a configuration file named *CMakeLists.txt*, at the root directory of the XLiFE++ archive. Whatever the OS, cmake also asks for another directory where to put generated files for compilation, called build directory hereafter. This directory can be everywhere. It will contains compilation files (objects files, ...), a Makefile or an IDE project file named XLiFE++ (for Eclipse, CodeBlocks, Visual Studio, XCode, ...). So we suggest you to set this directory as a subdirectory of XLiFE++ install directory, with the name "build" for instance.

1.4.1 Under MacOS and Linux

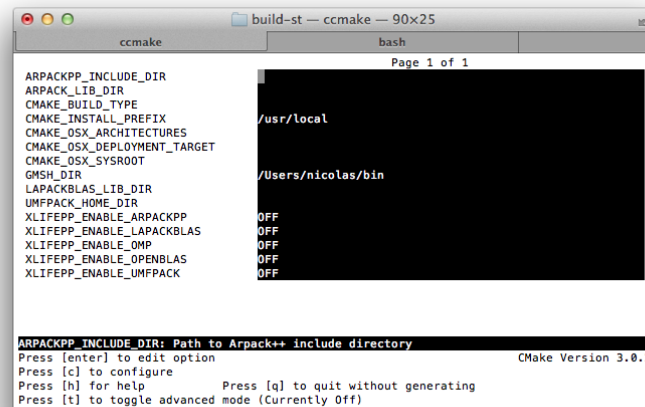


Figure 1.1: ccmake (MacOS, Unix)

When running **ccmake**, the default CMAKE GUI under MacOS or Linux, the build directory is given in argument.

```
ccmake [options] path/to/CMakeLists.txt path/to/buildDirectory
```

When running CMAKE GUI application, you have to set the directory "Where is the source code" containing *CMakeLists.txt* you want to run CMAKE on, and to set the build directory: "Where to build the binaries". Then, you click the "Configure" button. It will ask ou the generator you want. Then, you click the "Generate" button, to generate your IDE project file or your Makefile.

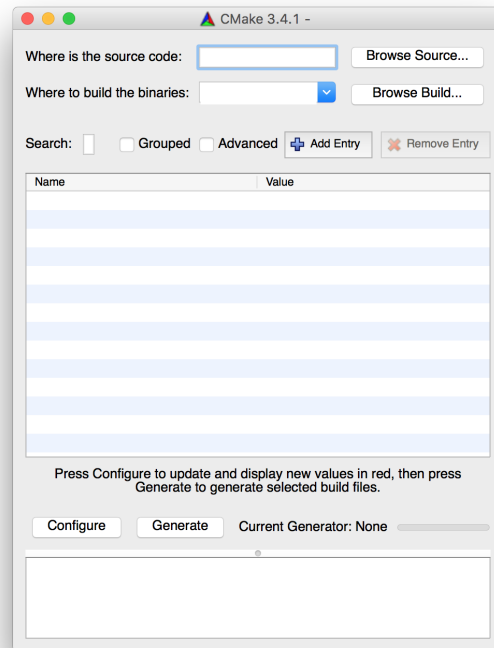


Figure 1.2: CMAKE application (MacOS)

When running CMAKE in command-line mode, the build directory is generally the directory in which you are when calling the CMAKE command. If you want to know the general case, please take a look at CMAKE option `-b`.

To compile XLIFF++, you just have to run CMAKE on the CMakeLists.txt file:

```
cmake path/to/CMakeLists.txt
```

If you generated a project file, you launch your IDE on this file and can compile and run tests. The following command chooses for instance to use codeblocks on unix platform:

```
cmake path/to/CMakeLists.txt -G "CodeBlocks - Unix Makefiles"
```

If you generated a Makefile, you have to run the `make` command, to compile XLIFF++ sources and tests. If you want to compile XLIFF++ sources only, the target is "libs":

```
make libs
make
```

If you generated an IDE project file, you have to open it and compile target all to compile everything, sources and tests, or target libs, to compile sources only.

1.4.2 Under Windows

When running CMAKE GUI application, you have to set the directory "Where is the source code" containing CMakeLists.txt you want to run CMAKE on, and to set the build directory: "Where to build the binaries". Then, you click the "Configure" button. It will ask you the generator you want. Then, you click the "Generate" button, to generate your IDE project file or your Makefile.

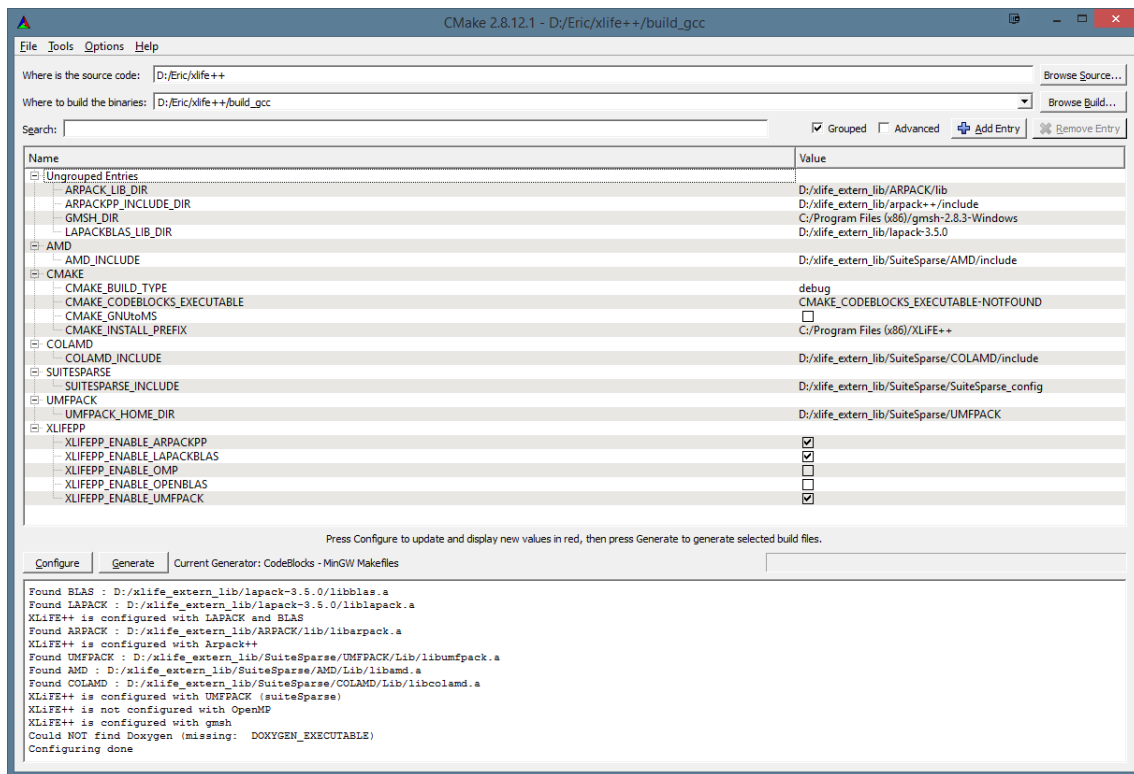


Figure 1.3: **cmake** (Windows), after you clicked the configure button.



You can have the list of available generators inside CMAKE help. It does not tell that eclipse, codeblocks, ... are installed, but it tells that they can be installed on your computer and how you can tell CMAKE to use them as generators.

1.4.3 Useful general options to CMAKE

Four options may be very useful:

CMAKE_CXX_COMPILER This option is used to set the compiler CMAKE will use to compile your code:

```
cmake path/to/CMakeLists.txt -DCMAKE_CXX_COMPILER=g++-48
```

CMAKE_BUILD_TYPE This option is used to set the type of build (debug, release, ...). Possible values are essentially Debug and Release. The default value is Release.

```
cmake path/to/CMakeLists.txt -DCMAKE_BUILD_TYPE=Debug
```

XLIFEPP_PRECISION This option is used to set the precision of numerical types (int, ...). Possible values are STD, LONG and LONGLONG. The default value is LONG.

```
cmake path/to/CMakeLists.txt -DXLIFEPP_PRECISION=LONG
```

Except if you have a very good reason, you are not supposed to use this option.

XLIFEPP_STRING_TYPE This option is used to set string type: STD (for std::string) and WIDE (for std::wstring). The default value is STD.

```
cmake path/to/CMakeLists.txt -DXLIFEPP_STRING_TYPE=STD
```

Except if you have a very good reason, you are not supposed to use this option.

1.4.4 Configuring XLiFE++ with GMSH and/or PARAVIEW

XLIFEPP_GMSH_EXECUTABLE to specify the GMSH binary with full path. If GMSH is in your PATH, it will be automatically detected, else you can use this option:

```
cmake path/to/CMakeLists.txt -DXLIFEPP_GMSH_EXECUTABLE=gmshtmp.exe/with/full/path
```

XLIFEPP_PARAVIEW_EXECUTABLE to specify the PARAVIEW binary with full path. If PARAVIEW is in your PATH, it will be automatically detected, else you can use this option:

```
cmake path/to/CMakeLists.txt  
-DXLIFEPP_PARAVIEW_EXECUTABLE=paraview.exe/with/full/path
```

1.4.5 Configuring XLiFE++ with OPENMP

XLIFEPP_ENABLE_OMP Activates/Deactivates use of OpenMP

```
cmake path/to/CMakeLists.txt -DXLIFEPP_ENABLE_OMP=ON
```

1.4.6 Configuring XLiFE++ with ARPACK

Configuring with ARPACK also means configuring with BLAS and LAPACK. When asked, these 3 libraries are automatically searched in standard paths. If these libraries are installed in other paths, you can give it directly. Otherwise, sources files of ARPACK inside XLiFE++ sources will be compiled and used.

XLIFEPP_ENABLE_ARPACK Activates/Deactivates configuration with ARPACK

XLIFEPP_BLAS_LIB_DIR to specify the directory containing BLAS library

XLIFEPP_LAPACK_LIB_DIR to specify the directory containing LAPACK library

XLIFEPP_ARPACK_LIB_DIR to specify the directory containing ARPACK library

See section A.3 for more details

1.4.7 Configuring XLiFE++ with UMFPACK

Configuring with UMFPACK also means configuring with BLAS and LAPACK. It can also mean configuring with other libraries provided by SUITESPARSE, insofar as UMFPACK is now distributed inside SUITESPARSE.

XLIFEPP_ENABLE_UMFPACK Activates/Deactivates configuration with UMFPACK

XLIFEPP_BLAS_LIB_DIR to specify the directory containing BLAS library

XLIFEPP_LAPACK_LIB_DIR to specify the directory containing LAPACK library

XLIFEPP_UMFPACK_INCLUDE_DIR to specify the directory containing UMFPACK header

XLIFEPP_UMFPACK_LIB_DIR to specify the directory containing UMFPACK header

XLIFEPP_SUITESPARSE_HOME_DIR to specify the home directory of SUITESPARSE, containing UMFPACK. This option is to be used if you compiled SUITESPARSE by yourself. In this case, UMFPACK will be searched in the UMFPACK subdirectory.

There are specific options to each library provided by SUITESPARSEUMFPACK may depend on. See section A.2 for more details



Using CMAKE generates a cache file (CMakeCache.txt) containing value of every option defined during execution of CMAKE, so that when reusing cmake on the same file resumes CMakeCache.txt, unless you change some options. This is why XLIFEPP_ENABLE_XXX options can take the value OFF.

1.5 Folder organisation

The XLIFE++ main directory is organised as follows :

bin the binary directory containing the test executable and the output files for test execution

CMakeLists.txt this is the parameters file for CMAKE

doc the documentation directory with the source documentation (the **api** subdirectory), generated by DOXYGEN, and the TeX/pdf documentation (the **tex** subdirectory)

etc this directory contains templates files for installation, parameters files for DOXYGEN, dictionaries and messages formats

include this directory contains the header aliases for each sublibrary of XLIFE++ and config files

lib this directory contains the static sublibraries of XLIFE++

README the README file

src this directory contains all source and header files. Each subdirectory represents a sublibrary of XLIFE++

tests this directory contains source files for unitary and non-regressive tests

usr this directory contains all files needed by the user to write a program using XLIFE++ and compile it.

1.6 Namespace

The whole XLIFE++ library is defined in the namespace *xlfepp*. Then, the developers have to encapsulate all their developments in this namespace either in the header files (.hpp) or in the implementation files (.cpp). For instance, a header file looks like :

```
#ifndef FILE_HPP
#define FILE_HPP

#include "otherfile.hpp"
#include <sysinclude>

namespace xlfepp{
    ...
} //end of namespace
#endif
```

1.7 Headers

All the header files of the library end by *.hpp* and are located in the folder related to the topics of the header file. There are four particular header files located in the *init* folder :

- *setup.hpp* defining the pre-compiler macro related to the install path, the operating system, the precision type, the string type, the language and debug feature. Be cautious, changing one of this macro implies the recompilation of all the library.
- *config.hpp* declaring all the general stuff : scalar definitions, string definition, enumeration, numerical constants and global variables used by general tools. It includes the header *setup.hpp*. Note that this file may be (and must be) included everywhere.
- *globalScopeData.hpp* initializing global variables declared in *config.hpp*. This file has to be included once.

Some header interface files are defined in the *include* folder, their name end with *.h*. The header interface files are

- *lib.h* for each library folder (e.g *init.h*, *utils.h*, ...). This header file included all the header files (*.hpp*) of the library folder and is used by other libraries that require objects of this library folder.
- *config.h* is an alias to the *config.hpp* located in the *init* folder.
- *globalScopeData.h* is an alias to the *globalScopeData.hpp* located in the *init* folder.
- *xlife++.h* is the global header file. As this file includes the *globalScopeData.hpp* initializing all the global variables it must be included only in the main program.

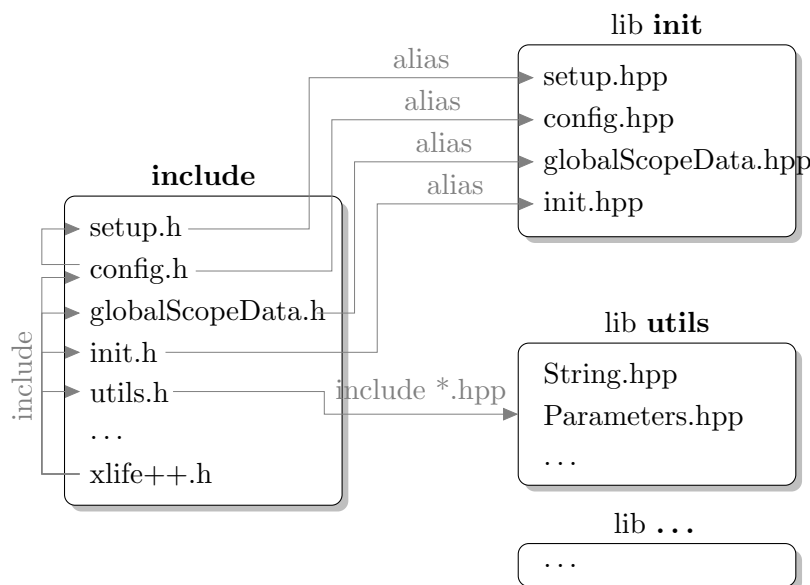


Figure 1.4: Headers organisation

From a practical point of view :

- when you develop a new class or utilities in the *mystuff.cpp* file in the library *lib*, create the header file *mystuff.hpp* and add the alias `#include "../src/lib/mystuff.hpp"` in the *lib.h* located in the *include* folder.

- include the *config.h* in your files to access to all general definitions, local header files (.hpp) if you refer to some stuff of the library and global header files (.h) if you refer to some stuff of other libraries
- when you develop a new library (*mylib*), you have to create the *mylib.h* header file in the *include* folder and add *#include "mylib.h"* in the *xlife++.h*.

1.8 Development principles

This section gives some "philosophical" principles to develop the library XLiFE++. These principles can be sum up in the following commandments :

- **Code has to be efficient**
It means that time consuming functions and memory consuming object has to be designed to be the most efficient as possible. It is not required else.
- **Be friendly with end users**
no templated and not too much end user's classes, no pointers, friendly syntaxes, ...
- **Be friendly with not advanced developers**
it means that the intermediate code has to be not too intricate
- **Make code documentation**
Write inline documentation along DOXYGEN syntax and outline documentation in Latex: the developer documentation gives principles, useful comments and a description of classes and functions, user documentation describes how to use the end user's classes and functions.
- **Write test functions**
It is mandatory to perform low level tests and to archive it (see chapter Test Policy)

As usual, there may be exceptions to the rules but they have to be approved.

Dealing with multithreading

Up to now, XLiFE++ supports OMP if library is compiled with the OMP flag. It is up to the developer to use it or not but when heavy computation it is advised to. As XLiFE++ already uses OMP in all FE computations or matrix computations, be care when using it at a high level of the code.

How to encapsulate a class hierarchy in an end user class

The problem is to propose to end user a single class (*User*) masking a class hierarchy (*Child1*, *Child2*, ... that inherits from *User*):

```
class Child1 : public User { ... };
class Child2 : public User { ... };
```

There are at least three solutions :

- using child pointers of child classes inherited from *BaseChild* base class :

```
class BaseChild { ... }
class Child1 : public BaseChild { ... };
class Child2 : public BaseChild { ... };

class User
{
  ChildType childType;           //type of child
  Child1 * ch1_p;                //pointer to Child1 object
}
```

```
Child2 * ch2_p;           //pointer to Child2 object
...                       //common attributes and functions
};
```

All the member functions of **User** class analyse the child type to address the right child functions.

- using pointer to **BaseChild** base class :

```
class BaseChild {...}
class Child1 : public BaseChild {...};
class Child2 : public BaseChild {...};

class User
{BaseChild * bch_p;       //pointer to BaseChild object
...
};
```

All the member functions of **User** class call their equivalent **BaseChild** member functions.

- using a contraction of the previous model named "abstract/non abstract pattern" :

```
class User
{User* us_p;      //pointer to child or itself if it is a child
virtual Child1* child1()
{if(us_p!=this) us_p->child1();
return 0;}
};

class Child1 : public User
{Child1* child1(){return this;}
...
};
class Child2 : public User
{Child2* child2(){return this;}
...
};
```

This model is named "abstract/non abstract" because **User** class is a non abstract class but is used as an abstract class. In this model, **User** class attributes are duplicated. If they take a large amount of memory, encapsulate it in a structure and use a pointer to this structure :

```
class Attributes
{...
};

class User
{User* us_p;           //pointer to child or itself if it is a child
Attributes* at_p;      //pointer to User attributes
};
```

Each of them has their advantages and defaults. Use the one you prefer.

2

Initialisation / finalization and global definitions

2.1 About the `init` function

Initialization is the aim of the `init` library and more precisely the `init.cpp` and `init.hpp` files and the definition of the `init` functions.

Each user program needing XLIFE++ has to begin with a call to the `init` routine with up to 4 key/value arguments:

`_lang` enum to set the language for print and log messages. Possible values are `en`, `fr`, or `de`. Default value is `en`.

`_verbose` integer to set the verbose level. Default value is 1.

`_trackingMode` boolean to set if in the log file, you have a backtrace of every call to a XLIFE++ routine. Default value is false.

`_isLogged` boolean to activate log. Default value is false.

Available languages are English, French and German, but XLIFE++ is easily extendible for other languages.

```
init(_lang=en)
```

What is the `init` function doing ?

1. If the `init` function was previously called, it stops, or does the following steps otherwise.
2. It builds the global `Environment` object to initialize dictionary and paths to messages formats, according to the given language.
The path to the english dictionary file is `XLIFEPP_DIR/etc/Messages/en/dictionary.txt`, the path to the french messages file is `XLIFEPP_DIR/etc/Messages/fr/messages.txt`, ...
3. It builds the global `Trace object` to initialize logs
4. It prints the execution header (logo + execution date, host machine ...)
5. It opens the print file and write the header in it.
6. It builds the global `Messages` object to load messages formats
7. It initializes RTI names for `Function` and `Value` objects.

Global objects are declared in `config.hpp` and defined in `globalScopeData.hpp`. In the next section, we will see what is the content of these two files and of `setup.hpp`

2.2 Global declarations and definitions

2.2.1 *setup.hpp*

This file is automatically generated when you execute CMAKE. Its aim is to set compilation macros :

- **OS_IS_UNIX**, **OS_IS_WIN**, to set the OS. It is useful for instance to load specific source code for time management (*Timer.*pp*)
- **XLIFEPP_WITH_GMSH** and **XLIFEPP_WITHOUT_GMSH**, to activate/deactivate compilation of XLiFE++ routines using a system call to GMSH
- **XLIFEPP_WITH_PARAVIEW** and **XLIFEPP_WITHOUT_PARAVIEW**, to activate/deactivate compilation of XLiFE++ routines using a system call to PARAVIEW
- **XLIFEPP_WITH_ARPACKPP** and **XLIFEPP_WITHOUT_ARPACKPP**, to activate/deactivate compilation of XLiFE++ interface of ARPACK++ library.
- **XLIFEPP_WITH_UMFPACK** and **XLIFEPP_WITHOUT_UMFPACK**, to activate/deactivate compilation of XLiFE++ interface of UMFPACK library.
- **XLIFEPP_WITH_OMP** and **XLIFEPP_WITHOUT_OMP**, to activate/deactivate compilation of XLiFE++ using OpenMP library.
- **INSTALL_PATH**, to set the install path of XLiFE++ and build paths for dictionary or messages formats.
- **GMSH_PATH**, to set the path of GMSH.
- **PARAVIEW_PATH**, to set the path of PARAVIEW.
- **STD_TYPES**, **LONG_TYPES**, **LONGLONG_TYPES** to set the precision of scalar non integer values. The default is **LONG_TYPES**
- **STD_STRING**, **WIDE_STRING**, to set the encoding of strings, by loading either the STL string classes, or . The default is **STD_STRING**.
- **COMPILER_IS_32_BITS**, **COMPILER_IS_64_BITS** to set the precision of integer values

The last five macros are used in *config.hpp* to define some typedefs.

2.2.2 *config.hpp*

This file defines some scalar types, objects and parameters, and also enums.

- **Scalar types** : XLiFE++ uses scalar types that depend on the precision given in *setup.hpp* : `int_t` , `real_t`, `complex_t` are typedefs for respectively `int`, `float`, `complex<float>` and their families (`long int`, `double`,...) according to the corresponding macro. Two additional typedefs are `number_t` and `dimen_t` respectively for `size_t` and `short unsigned int`, independent on precision. These scalar types can be called by users. According to the XLiFE++ code convention, they have user typedefs : `Int`, `Real`, `Complex`, `Number`, `Dimen`. For the same reason, there is the `String` typedef for STL classes `string` or `wstring` according to the corresponding macro, and the `Strings` typedef for `std::vector<String>`
- **Global parameters** : This is about output format parameters and extremal values :

```

extern const number_t entriesPerRow;
extern const number_t entryWidth;
extern const number_t entryPrec;
extern const number_t addrPerRow;
extern const number_t addrWidth;
extern const number_t numberPerRow;
extern const number_t numberWidth;
extern const number_t theNumberMax;
extern const size_t theSizeMax;
extern const dimen_t theDimMax;
extern const real_t theRealMax;
extern const real_t theEpsilon;
extern const real_t theTolerance;
extern const real_t theZeroThreshold;
extern const number_t theLanguage;
extern unsigned int theGlobalVerboseLevel;
extern unsigned int theVerboseLevel;

```

There is also some mathematical constants :

```

extern const real_t over3;
extern const real_t over6;
extern const real_t pi;
extern const real_t overpi;
extern const real_t over2pi;
extern const real_t over4pi;
extern const real_t sqrtOf2_;
extern const real_t sqrtOf3_;
extern const real_t logOf2_;
extern const real_t theEulerConst;

```

- **Global objects** : Global Trace, Messages, Parameters and Environment are also declared here
- **enums** : This is the list of enums defined here. For convenience, we give only their declaration :

```

enum Language;
enum ShapeType;
enum ShapesType;
enum StrucType;
enum SymType;
enum ValueType;
enum FunctType;
enum FuncFormType;
enum SobolevType;
enum SpaceType;
enum DofType;
enum SupportType;
enum UnknownType;
enum FEType;
enum FESubType;
enum PolynomType;
enum QolynomType;
enum LinearFormType;
enum QuadRule

//for matrix storages
enum StorageType;
enum AccessType;
enum MatrixPart;

// for solver purpose
enum FactorizationType;

```

```

enum PreconditionerType;
enum EigenComputationalMode;
enum EigenSolverMode;
enum MsgEigenType;

enum ComputationInfo;
enum IOFormat;
enum TestStatus;
enum DataAccess;

```

Each enum has a dictionary counter-part for output.

2.2.3 *globalScopeData.hpp*

This file is the last file include by main programs. Its aim is to define the global parameters and objects declared in *config.hpp*, and also the static class attributes :

```

std::vector<Interpolation*> Interpolation::theInterpolations;
std::vector<Space*> Space::theSpaces;
Number FeSpace::lastEltIndex = 0;
std::vector<Unknown*> Unknown::theUnknowns;
std::vector<DifferentialOperator*> DifferentialOperator::theDifferentialOperators;
std::map<String, std::pair<ValueType, StrucType>> Function::returnTypes;
std::map<String, std::pair<ValueType, StrucType>> Function::returnArgs;
std::map<String, std::pair<ValueType, StrucType>> Value::theValueTypeRTInames;
std::vector<const GeomDomain*> GeomDomain::theDomains;
std::vector<GeomRefElement*> GeomRefElement::theGeomRefElements;
std::vector<RefElement*> RefElement::theRefElements;
std::vector<Quadrature*> Quadrature::theQuadratures;
std::vector<Term*> Term::theTerms;
std::vector<MatrixStorage*> MatrixStorage::theMatrixStorages;

```

You may read their function in the chapter dedicated to the corresponding classes.

2.3 About the finalize function

The **finalize** function is defined in the *finalize* library. Its aim is to clean every global static list defined in *globalScopeData.hpp*. Why a library with a single external function ? For dependencies purpose. As the *init* library and *utils* library depend on each other through headers, the **finalize** function depends on libraries concerned by those global static lists, namely *finiteElements*, *operator*, *space*, *geometry*, *largeMatrix* and *term*. To preserve independence of the *utils* library, the **finalize** function cannot be placed in the *init* library.

Here is the implementation of this function :

```

void finalize()
{
    // clean all non user run-time lists
    Interpolation::clearGlobalVector();
    GeomRefElement::clearGlobalVector();
    RefElement::clearGlobalVector();
    DifferentialOperator::clearGlobalVector();
    Domain::clearGlobalVector();
    Space::clearGlobalVector();
    Unknown::clearGlobalVector();
    Quadrature::clearGlobalVector();
    MatrixStorage::clearGlobalVector();
    Term::clearGlobalVector();
}

```

It is used to preserve independence of tests when running all tests.

3

The *utils* library

The *utils* library collects all the general useful classes and functionalities of the XLiFE++ library. It is an independent library. It addresses:

- **String** capabilities (mainly additional functions to the `std::string` class),
- **Point** class to deal with point of any dimension,
- **Vector** class to deal with numerical vectors (say real or complex vectors),
- **Matrix** class to deal with numerical matrices with dense storage (small matrices),
- **Value** class to handle any real/complex scalar/vector/matrix value,
- **VectorEntry** class to deal with any scalar/vector real/complex vector,
- **Collection** class to deal with collection of anything, inheriting from `std::vector`,
- **Parameter** classes to deal with general set of user parameters,
- **Function** class encapsulating user functions,
- **SymbolicFunction** class handling symbolic functions,
- **Kernel** class encapsulating user kernels,
- **ThreadData** class handling FE data computation (normal vectors, ...),
- **Transformation** class to handle geometric transformations,
- **Messages** classes to handle error, warning and info messages,
- **Trace** class to handle log messages,
- **Timer** class to handle timer utilities,
- **Memory** class to handle memory utilities,
- **PrintStream** class that handles the output stream in multithread environment
- **Environment** class to handle XLiFE++ runtime variables,
- **Graph** class to deal with graph,
- **KdTree** class to handle kd-tree (binary separation of kd points with a fast search of nearest point),
- **EigenInterface** class that handles the interface to Eigen library

3.1 The Environment class

The `Environment` class collects all stuff related to the XLife++ execution context : machine, operating system, processors-thread, some paths, message language, ...

```
class Environment
{
private:
const int theLanguageAsAnInt_; // language as an int
string_t theXlifeppVersion_; // version number of XLife++
string_t theXlifeppDate_; // version date of XLife++
string_t theOS_; // OS name
string_t theProcessor_; // processor or machine type
string_t theMachineName_; // machine name
string_t thePathToMessageFiles_; // path to error files
string_t thePathToGeoMacroFile_; // path to geo macro file

static const string_t theInstallPath_; // xlife++ installation path
static const string_t theGmshPath_; // gmsh binary path
static const string_t theParaviewPath_; // paraview binary path
static bool running_; // true when running
static bool parallelOn_; // parallel flag (default on)
static const int theNumberOfLanguages = 2; // number of languages
static const string_t theLanguageNames_[]; // language [ "en", "fr" ]
static std::map<string_t, std::vector<string_t> > enumWords_; // global enums
static std::map<string_t, string_t> words_; // global words
```

At initialization (see `init.cpp`), the `Environment` pointer `theEnvironment_p` is allocated and initialized.

Only one constructor from language is allowed :

```
Environment(int lang = _en );
```

Copy constructor and assign operator are not available.

As all member data are private, there exists their corresponding accessors:

```
static bool running();
static bool parallelOn();
static string_t installPath();
static string_t gmshPath();
static string_t paraviewPath();
static int numberOfLanguages();
int language() const;
string_t osName() const;
string_t processorName() const;
string_t machineName() const;
string_t msgFilePath() const;
string_t geoMacroFilePath() const;
bool known() const;
static void parallel(bool);
void names();
void processor();
void setMsgFilePath();
void setGeoMacroFilePath();
void localizedStrings();
void version();
string_t languageString() const;
```

Besides some utilities are provided:


```

int returnedFunctionType(const string_t& s);
int returnedKernelType(const string_t& s);
void printHeader(const Timer&);
void printHeader(std::ostream&);
void printDictionary(std::ostream&);

```

Other utilities are provided as non member functions:

```

bool is32bits();
bool is64bits();
bool parallelOn();
void parallel(bool);
number_t numberOfThreads(int n=-1); //if omp is available
number_t currentThread(); //if omp is available

```

3.1.1 Using the dictionary

The `Environment` class provides 4 external functions to deal with localized strings:

```

String words(const String&);
String words(const char*);
String words(const bool b);
String words(const String&, const int);

```

The last function deals with enumeration items. You give the family and the enum item and you get the localized string. The two first functions deals with dictionary words. You give the dictionary keyword :

```

std::cout << words("shape", _segment) << std::endl;
std::cout << words("access type", _sym) << std::endl;
std::cout << words("constructors for real matrix") << std::endl;

```

library :	utils
header :	Environment.hpp
implementation :	Environment.cpp
unitary tests :	
header dependences :	config.h, String.hpp, Timer.hpp, Messages.hpp, logo.hpp

3.2 Messages management

Handling error, warning and info messages is a essential part of a code. In C++, a choice could be using exceptions. As XLiFE++ is a very large code, with a high number of levels (routines called by routines themselves called by routines ...), using exceptions has a lot of constraints. The other way is to develop our own classes. The message machinery works as follows:

- A message is a string of characters including free text and particular codes (as much as you want) of the form %s, %i, %r, %c and %b which will be substituted respectively by a string, an integer, a real, a complex and a boolean.
- For each message is defined a unique string identifier, a type (ERROR, WARNING or INFO), a runtime behaviour (STOP or CONT) and an output console flag (STDOUT or NOSTDOUT).
- All the messages are listed in a file named *messages.txt* in the directory *etc/Messages/lang* where *lang* is the language code; for instance *en* for the English, *fr* for the French, ...
- By translating the English reference *messages.txt* file in an other language it is easy to internationalize the XLiFE++ code.

For instance, the beginning of the English *messages.txt* is

```
#!/
===== internal messages =====

# logon INFO CONT STDOUT Log file "%s" activated.
# logoff INFO CONT STDOUT Log file "%s" deactivated.
# errorundef ERROR STOP STDOUT Error message of type %s and id %i undefined (%s)!
This might occur with a double string message Id in a message.txt file
or the forgetting of the message type, the behaviour type or the output type
...
```

Note that a message format may be defined on several lines (the next character `#` is the end delimiter of a message) and a comment begins by `#!/`.

The message management is based on three classes: `MsgData` (storage of the values to be included), `MsgFormat` (multilingual message format) and `Messages`, the main class collecting all formats and methods to produce and output the formatted messages.

3.2.1 The `MsgData` class

The aim of the `MsgData` class is to store in the same structure, all sorts of variables to use to build a message, of various types: integer, real, complex, string and bool. Thus, this class proposes as private members:

```
class MsgData {
private :
    std::vector<int> i_;           // to store int type data
    std::vector<real_t> r_;       // to store real_t type data
    std::vector<complex_t> c_;    // to store complex_t type data
    std::vector<String> s_;       // to store string data
    std::vector<bool> b_;         // to store boolean data
    bool read_;                  // flag to specify that the structure has been read
    ...
};
```

It offers:

- a default constructor:

```
MsgData() : i_(0), r_(0), c_(0), s_(0), b_(0), read_(true) {}
```

- some public access functions to data

```
yyy xxxParameter(const unsigned int n);
// where (xxx,yyy)=(int,int), (real,real_t), (complex,complex_t), (string,string),
// (boolean,bool)
bool read();
```

- a public method to specify data was read

```
void readData();
```

- some public append methods for each managed data type or variant (unsigned int or char* for example)

```
void push(const T t); // T is int, real_t, string, ...
```

- some public streaming operators, appending data of various types

```
MsgData& operator<<(const T i) // T is int, real_t, string, ...
```

- a private reset method, to reset every list of data

```
void reset_();
```

3.2.2 The MsgFormat class

The `MsgFormat` Object contains a message format for output. In addition, it contains information on message such as the message type, the message behaviour, the message output and the message id. Thus, the `MsgFormat` class proposes as private members:

```
enum MsgType {_error=0,_warning,_info};

class MsgFormat {
private :
    String format_;           // format of message
    MsgType type_;           // flag for warning/error
    bool stop_;              // flag for stop/continue
    bool consoleOut_;        // flag for output to console
    String ids_;             // string id of format
    ...
};
```

It offers:

- two constructors, the default one and the full one:

```
MsgFormat(const String &ms, const String &ids, MsgType t, bool s, bool c);
```

- some accessors, one for each attribute:

```
String format() const {return format_;}
MsgType type() const {return type_;}
bool stop() const {return stop_;}
bool console() const {return consoleOut_;}
String stringId() const {return ids_;}
```

3.2.3 The Messages class

The `Messages` class manages the list of messages format and the display of every message. Thus, this class proposes as private members:

```
class Messages {
private :
    String msgType_;           // message type
    std::map<String,MsgFormat*> stringIndex_; // index of messages by string id
    std::ofstream* msgStream_p; // file stream where are sent messages
    String msgFile_;           // file where are sent messages
    ...
};
```

It offers:

- two constructors, the default one and the full one:

```
Messages(const std::string& file, std::ofstream& out, const std::string& msgFile,
         const std::string& msgType = std::string("XLiFE++"));
```

- some accessors:

```
String msgType() const;
String msgFile();
std::ofstream* msgStream() const;
```

- some utilities:

```
int numberOfMessages() const;
void loadFormat(const String&); // load format from a file
void printList(std::ofstream&); // print the list of all messages
MsgFormat* find(const String&); // find an error format
void append(MsgFormat&); // append a new message format in list
void appendFromFile(const String&); // append a new message format in list
```

3.2.4 External functions

To throw messages, some external functions with useful aliases and shortcuts functions are provided. The general functions are:

```
String message(const String& msgIds, MsgData& msgData=theMessageData,
              Messages* msgSrc=theMessages_p);
void msg(const String& msgIds, MsgData& msgData, MsgType& msgType, Messages*
        msgSrc);
```

The first one (*message*) reads the message format and replaces the strings %i,%r,%c,%s and %b by their values given by the *MsgData* object. It produces a *String* containing the formatted message. This function is called by the *msg* function which outputs the formatted message to the console or to a print file. Note that this function may be also called by any function to retrieve the formatted message without output.

The *msg* function is called by the three useful functions:

```
void error(const String& msgIds, MsgData& msgData=theMessageData,
          Messages* msgSrc=theMessages_p);
void info(const String& msgIds, MsgData& msgData=theMessageData,
          Messages* msgSrc=theMessages_p);
void warning(const String& msgIds, MsgData& msgData=theMessageData,
            Messages* msgSrc=theMessages_p);
```

which are aliases for the *msg* function, giving the value of the *msgType* argument. These are the functions to be used when you need to throw messages in other classes of XLiFE++ library.

Besides, for sake of simplicity, some shortcut functions are also provided. They avoid to manage explicitly the *MsgData* object storing the message values. These shortcut functions are templated functions (one, two or three templated parameters), so they only work up to three message values!

```
String message(const String& msgIds, const T& v, Messages* msgSrc=theMessages_p);
String message(const String& msgIds, const T1& v1, const T2& v2,
              Messages* msgSrc=theMessages_p);
```

```
String message(const String& msgIds, const T1& v1, const T2& v2, const T3& v3,
               Messages* msgSrc=theMessages_p);
void error(const String& msgIds, const T& v, Messages* msgSrc=theMessages_p);
void error(const String& msgIds, const T1& v1, const T2& v2,
           Messages* msgSrc=theMessages_p);
void error(const String& msgIds, const T1& v1, const T2& v2, const T3& v3,
           Messages* msgSrc=theMessages_p);
//the same for warning and info functions
```

3.2.5 Throwing messages

There is one global `MsgData` object: *theMessageData* which is used as default argument of message functions. Using it, it is very easy to throw messages. For instance, these lines show how to throw an error message:

```
if(x.size()!=y.size())
{theMessageData<<name()<<x.size()<<y.size(); //store the message values
  error("ker_bad_dim"); //throw the error message
  ker_bad_dim
}
```

The message format *ker_bad_dim* has the following definition:

```
# ker_bad_dim ERROR STOP STDOUT Call of kernel %s
with vectors of incompatible dimensions : (%i,%i)
```

If you want to use international messages in your own output stream, you cannot use the function *info* because it outputs the message on the default print stream. You have to use the function *message* which returns the message as a `String` without printing output. For instance:

```
std::ostream& operator<<(std::ostream& out, const Space & sp)
{
  theMessageData<<sp.name()<<sp.typeOfSpace()<<sp.domain().name();
  out<<message("space_def");
  return out;
}
```

with the message format *space_def*:

```
# space_def INFO CONT STDOUT space %s of type %s on the geometrical domain %s
```

Using shortcut functions, the previous example reads:

```
std::ostream& operator<<(std::ostream& out, const Space & sp)
{
  out<<message("space_def", sp.name(), sp.typeOfSpace(), sp.domain().name());
  return out;
}
```

library :	<code>utils</code>
header :	<code>Messages.hpp</code>
implementation :	<code>Messages.cpp</code>
unitary tests :	<code>test_Messages.cpp</code>
header dependences :	<code>config.h, String.hpp</code>

3.3 Managing traces

3.3.1 The Trace class

The `Trace` class is devoted to keep trace of runtime of the hierarchy of function calls. Thus, we can have access to a stack of called functions, the next position to be used and the last one. Thus, this class proposes as members:

```
class Trace {
private:
    number_t pos_; //!< next position in trace object list member
    static const number_t maxPos_=36; //!< last position in trace object list member
    String fList [maxPos_]; //!< stack of called functions

    static const String theLogFile_; //!< name of log file
    static bool isLogged_;

public:
    static std::ofstream theLogStream_;
    ...
};
```

It offers:

- a default constructor that initializes `pos_` to 0, and private copy constructor and assignment operator
- some public access functions to data:

```
static const String logFile() { return theLogFile_; }
static bool isLogged() { return isLogged_; }
static void logOn() { isLogged_ = true; }
static void logOff() { isLogged_ = false; }
number_t length() const; //!< function length returns the number of items of f-list
```

- some public utilities methods for adding trace or having info on current position:

```
void push(const String&); //!< "pushes" name into Trace function list member
void pop(); //!< "pops" current input out of a Trace function
    list member
String current(const number_t); //!< returns string of last l to current inputs of
    a Trace object list
String current(); //!< returns last input string of a Trace object
    list
String list(); //!< returns function list of a Trace object list
```

- some public display management methods:

```
void indent(); //!< function indent prints context-depending
    indentation to log file
void print(std::ofstream&); //!< prints function list to opened ofstream or to
    default print file
void print();
```

- some public methods to build log messages:

```
void log(); //!< outputs indentation to log file
template<typename T_0> void log(const T_0& s);
template<typename T_0, typename T_1> void log(const T_0& s, const T_1& t1)
```

```
template<typename T_0, typename T_1, typename T_2> void log(const T_0& s, const
    T_1& t1, const T_2& t2);
template<typename T_0, typename T_1, typename T_2, typename T_3> void log(const
    T_0& s, const T_1& t1, const T_2& t2, const T_3& t3);
template<typename T_0, typename T_1, typename T_2, typename T_3, typename T_4> void
    log(const T_0& s, const T_1& t1, const T_2& t2, const T_3& t3, const T_4& t4);
```

- some external functions to manage trace errors:

```
void incompleteFunction(const String& s = "");
void invalidFunction(const String& s = "");
void constructorError();
set setNewHandler();
```

- some external functions to manage verbose level:

```
void setGlobalVerboseLevel(const unsigned int);
unsigned int verboseLevel(const unsigned int);
```

- one external functions to manage the message location :

```
String & where(const String& s);
```

3.3.2 How to use the verbose level ?

The verbose level is managed through 2 global variables : `theVerboseLevel` and `theGlobalVerboseLevel`, and 2 functions. The behavior is the following :

- To set globally the maximum verbose level to an int value *i*, you can use the following syntax : `setGlobalVerboseLevel(i)`; This sets the variable `theGlobalVerboseLevel` and overrides every local definition of the verbose level to higher values. This function is to be used once at the beginning of a main program.
- To set locally the maximum verbose level to a value *i*, you can use the following syntax :

```
int j=verboseLevel(i);
if (i < theGlobalVerboseLevel) {
    j=theVerboseLevel=i;
} else {
    j=theGlobalVerboseLevel;
}
```

- To get the current maximum verbose level, you can use directly the variable `theVerboseLevel`

3.3.3 How to know where does messages come ?

To see how to throw messages, please read the previous section about it. Here, we will discuss how a message knows where it is and how to give is more information about it.

The main mechanism is to use the functions `push` and `pop` respectively at the beginning and the end of whatever function. This piece of information is dealt with by messages handlers automatically. But we cannot do it for every function in the code. We have to reserve it for the top-level functions in order not to slow execution.

For the deepest functions, there is another way : using the `where` function. This function is to be used just before a message call where `push` and `pop` are not used. Messages will deal with this additional

information automatically. As a result, there is no need to define message format with the routine name as message data.

```

        library :   utils
        header :   Trace.hpp
implementation :   Trace.cpp
        unitary tests :
header dependences : config.h, String.hpp

```

3.4 The Timer class

The `Timer` class is a utility class to perform computational time analysis (cpu time and elapsed time) and manage dates. It uses the following data members which store time, cpu time and system time in various format.

```

class Timer
{private:
    time_t t; //time type provided by ctime
    tm localtime; //time structure provided by ctime
    long int sec_, microSec_; //time in seconds and microseconds
                                // since 01/01/1970 00:00:00
    long int usrSec_, usrMicroSec_; // cputime in seconds and microseconds
                                // since beginning of current process
    long int sysSec_, sysMicroSec_; // system time in seconds and microseconds
                                // since beginning of current process
    ...
};

```

The `Timer` class has only one constructor (no argument) which initializes the start time. It provides a few basic functionalities:

```

void update() // converts time to tm struct
int year() // year as an int
unsigned short int month() // month as an int [0-11]
unsigned short int day() // day of month as an int [0-31]
unsigned short int hour() // hour of day (24h clock)
unsigned short int minutes() // minutes of hour [0-59]
unsigned short int seconds() // seconds of minute [0-59]
void getCpuTime() // update cputime interval since beginning of
    process
double deltaCpuTime(Timer*) // returns elapsed time interval since "ot" time
void getTime() // update time in s and ms since 01/01/1970
    00:00:00
double deltaTime(Timer*) // returns elapsed time interval since "ot" time

```

Associated to these member functions, there are the following external functions (user functions):

```

String theTime() // returns current time
String theDate() // returns current date as dd.mmm.yyyy
String theShortDate() // returns current date as mm/dd/yyyy or dd/mm/yyyy
String theLongDate() // returns current date as Month Day, Year or Day Month
    Year
String theIsoDate() // returns ISO8601 format of current date (yyyy-mm-dd)
String theIsoTime() // returns ISO8601 format of current time (hh-mi-ss)

double cpuTime() // returns cputime in sec. since last call
double cpuTime(const String&) // same and prints it with comment
double totalCpuTime() // returns cputime in sec. since first call

```



```
double totalCpuTime(const String&) //same and prints it with comment
double elapsedTime() // returns elapsed time in sec. since last call
double elapsedTime(const String&) // same and prints it with comment
double totalElapsedTime() // returns elapsed time in sec. since first call
double totalElapsedTime(const String&) // returns elapsed time interval in sec.
// same and prints it with comment
```

Note that this class is OS dependant because Unix and Windows do not provide the same system time functions. The macro `OS_IS_WIN32` set in *setup.hpp* is used to select the right version. In particular, for Unix platform the class uses some functions provided by the header *sys/time.h* and for Windows platform, some functions provided by the header *sys/timeb.h*, this header requiring the headers *windef.h*, *stdio.h*, *cstdarg*, *winbase.h*.

The function *timerInit()* in the *init.cpp* file creates two `Timer` objects (*theStartTime_p* and *theLastTime_p*) at the beginning of the execution. These objects have a global scope and are used by the `Timer` class to make easier the time computation analysis for the user. There is no reason to create other instances of `Timer`.

So, it is easy to the user to perform time computation analysis. For instance:

```
#include "xlife++.h"
using namespace xlifepp;
int main()
{
    init(fr); //initializes timers
    //task 1
    ...
    cpuTime("cpu time for task 1");
    elapsedTime("ellapsed time for task 1");
    //task 2
    ...
    cpuTime("cpu time for task 2");
    elapsedTime("ellapsed time for task 2");
    //end of tasks
    totalCpuTime("total cpu time");
    totalElapsedTime("total ellapsed time");
}
```

library :	utils
header :	Timer.hpp
implementation :	Timer.cpp
unitary tests :	
header dependences :	config.h

3.5 The Memory class

The `Memory` class is a small utility class giving some informations about the memory usage in different format.

```
enum MemoryUnit { _byte , _kilobyte , _megabyte , _gigabyte , _terabyte };
class Memory
{
public:
    static real_t physicalMem(MemoryUnit mu=_megabyte);
    static real_t physicalFreeMem(MemoryUnit mu=_megabyte);
```

```
static real_t virtualMem(MemoryUnit mu=_megabyte);
static real_t virtualFreeMem(MemoryUnit mu=_megabyte);
static real_t processPhysicalMem(MemoryUnit mu=_megabyte);
static real_t processVirtualMem(MemoryUnit mu=_megabyte);
};

real_t byteTo(number_t mem, MemoryUnit mu=_megabyte);
```

All the member functions are static. Default unit is the MegaByte. Note that virtual memory is not available for unix/linux systems!

library :	utils
header :	Memory.hpp
implementation :	Memory.cpp
unitary tests :	
header dependences :	config.h

3.6 The String class

String is no more than an alias (typedef) to the String class of the STL. By using a macro variable it is possible to choose either standard string (utf8) or wide string (utf16):

```
#ifndef WIDESTRING
    typedef std::wstring String;
#else
    typedef std::string String;
#endif
```

The *WIDE_STRING* macro variable is currently set in the *Config.hpp* header file.

As the string or wstring class of STL, String proposes all the functionalities of std::string (see the STL documentation) and new additional ones:

```
//conversion utilities
template<typename T> String toString(const T& t); // 'anything' to String
template<typename T> T_ stringto(const String& s); //String to 'anything'

//transformation utilities
String lowercase(const String&); // String converted to lowercase
String uppercase(const String&); // String converted to uppercase
String capitalize(const String&); // String with initial converted to uppercase
String trimLeading(const String& s, const char* delim = " \t\n\f\r"); // trims
    leading white spaces
String trimTrailing(const String& s, const char* delim = " \t\n\f\r"); // trims
    trailing white spaces
String trim(const String& s, const char* delim = " \t\n\f\r"); // trims leading and
    trailing white spaces
String delSpace(const String& s); //delete all white space
String& replaceString(String& s, const String& s1, const String& s2); // replace
    string s1 by string s2 in string s
String& replaceChar(String& s, char c1, char c2); // replace char c1 by char c2 in
    string s
String fileExtension(const String& f); // return file name extension using last
    point as delimiter
std::pair<String, String> fileRootExtension(const String& f); // return rootname
    and extension of a file
void blanks(String&,int n); // add or remove n blanks at end of string

//search capabilities
```

```
int findString(const String, const std::vector<String>&); ///returns index
```

Be cautious with templated conversion functions. The function *tostring* (resp. *stringto*) works when the operator << (resp. >>) on stringstream is defined for the template type $T_$. The function *tostring* is more flexible than *stringto*. Note that the template $T_$ type has to be clarified when invoking *stringto*.

```
//examples of conversion stringto
String s="1 2 3";
int i=stringto<int>(s);           //i=1
real_t r=stringto<real_t>(s);     //r=1.
complex_t c=stringto<complex_t>(s); //c=(1.,0)
void * p=stringto<void*>(s);       //p=0x1
String ss=stringto<String>(s);     //ss="1"
s="(0,1)";
c=stringto<complex_t>(s);          //c=(0.,1.)
```

```
library :   utils
header :   String.hpp
implementation : String.cpp
unitary tests : test_String.cpp
header dependences :
```

3.7 The Point class

The purpose of the `Point` class is to deal with point in any dimension. It is derived from the `std::vector<Real>`.

```
typedef std::vector<Real>::iterator ivr_t;
typedef std::vector<Real>::const_iterator civr_t;

class Point : public std::vector<Real>
{public:
    static Real tolerance;
    ...
}
```

The static member *tolerance* is used in comparison operations. By default, its value is 0; As a child class of `std::vector`, the `Point` class does not have copy constructor and assignment = operator. It offers some basic constructors:

```
Point(const int, const Real v = 0.); // dimension and constant value
Point(const Dimen, const Real v = 0.); // dimension and constant value
Point(const Real x1); // 1D constructor by coordinate
Point(const Real x1, const Real x2); // 2D constructor by coordinates
Point(const Real x1, const Real x2, const Real x3); // 3D constructor by coordinates
Point(const Dimen d, Real* pt); // constructor by dimension and Real array
Point(const std::vector<Real>& pt); // constructor by stl vector
Point(const std::vector<Real>::const_iterator, Dimen); // constructor by stl vector
iterator
~Point(); // destructor
```

One can access (both read and read/write access) to the coordinates either by index (from 1 to dim) or by explicit name `x()`, `y()` and `z()` (restricted to 3D point):

```
Real operator()(const Dimen d) const; //the d-th coordinates (r)
Real& operator()(const Dimen d); //d-th coordinates (r/w)
```

```

Real x() const; //the first coordinate (r)
Real& x();       //the first coordinate (r/w)
Real y() const; //the second coordinate (r)
Real& y();       //the second coordinate (r/w)
Real z() const; //the third coordinate (r)
Real& z();       //the third coordinate (r/w)
std::vector<Real> toVect(const Point &) //!<convert to a std::vector

```

Through overloaded operator, the Point class provides some algebraic operations and distance computation functions:

```

Point& operator+=(const Point &); //add a point to the current point
Point& operator-=(const Point &); //subtract a point to the current point
Point& operator+=(const Real);    //add a constant point to the current point
Point& operator-=(const Real);    //subtract a constant point to the current point
Point& operator*=(const Real);    //scale the current point
Point& operator/=(const Real);    //scale the current point
Real squareDistance(const Point&) const; //square distance to an other point
Real distance(const Point&) const; //distance to an other point

```

Note that size compatibility tests are performed on such computations.

Algebraic operations on points may be also be processed by external function to the class:

```

Point operator+(const Point &); //same point (completeness)
Point operator-(const Point &); //opposite point
Point operator+(const Point &,const Point &); //sum of two points
Point operator-(const Point &,const Point &); //difference of two points
Point operator+(const Point &,const Real ); //sum of two points
Point operator-(const Point &,const Real ); //difference of two points
Point operator+(const Real ,const Point &); //sum of two points
Point operator-(const Real ,const Point &); //difference of two points
Point operator*(const Real ,const Point &); //scale a point
Point operator*(const Point &,const Real ); //scale a point
Point operator/(const Point &,const Real ); //scale a point
Real pointDistance(const Point&,const Point&); //distance between two points
Real squareDistance(const Point&,const Point&); //square distance between two points

```

`distance()` is a function of the `std::iterator` class, it is the reason why the distance between two points is named `pointDistance`.

Finally, to order (partially) a couple of points, the comparison operators (`==`, `!=`, `<`, `>`, `<=` and `>=`) are overloaded:

```

bool operator==(const Point &,const Point &); //equality of two points
bool operator!=(const Point &,const Point &); //not equality of two points
bool operator< (const Point &,const Point &); //leather than
bool operator<=(const Point &,const Point &); //leather than or equal
bool operator> (const Point &,const Point &); //greater than
bool operator>=(const Point &,const Point &); //greater between or equal

```

The comparison between points are performed using the tolerance value τ (`Point::tolerance`) which means for points p and q in \mathbb{R}^n :

$$\begin{aligned}
p == q & \text{ if } |p - q| \leq \tau \\
p < q & \text{ if } \exists i \leq n, \forall j < i, |p_j - q_j| \leq \tau \text{ and } p_j < q_j - \tau.
\end{aligned}$$

```

library :   utils
header :   Point.hpp
implementation : Point.cpp
unitary tests : test_Point.cpp
header dependencies : config.h

```

3.8 The Vector class

The purpose of the `Vector` class is mainly to deal with complex or real vector. It is derived from the `std::vector<K>` and templated by the vector type:

```

template<typename K>
class Vector : public std::vector<K>
{
public:
    typedef K type_t;
    typedef typename std::vector<K>::iterator ivk_t;
    typedef typename std::vector<K>::const_iterator civk_t;
    ...

```

As it is templated, it is also possible to deal with vector of vectors and generally vector of anything. But some algebraic functions no longer work.

It offers some basic constructors and assignment function:

```

Vector(); //default constructor
Vector(const dimen_t); //constructor by length
Vector(const dimen_t, const K&); //constructor by length for constant vector
void assign(const K&); //assign const scalar value to all entries
Vector<K>& operator=(const K&); //assign const scalar value to all entries

```

The default constructor initializes a vector of length 1 with a "0" value ($K()$).

One can access (both read and read/write access) to the component either by iterators (`begin()` and `end()`) or by index from 1 to the length:

```

size_t size() const; //overloaded size
ivk_t begin(); //overloaded iterator begin()
civk_t begin() const; //overloaded const iterator begin()
ivk_t end(); //overloaded iterator end()
civk_t end() const; //overloaded const iterator end()
K operator()(int i) const; //i-th component (i=1..n) (r)
K& operator()(int i); //i-th component (i=1..n) (r/w)

```

There are also generalized accessors that allow to get or set a part of vector, indices being given either by lower and upper indices or by a vector of indices:

```

Vector<K> get(Number i1, Number i2) const; //get i1->i2 components
Vector<K> get(const std::vector<Number>& is) const; //get 'is' components
void set(Number i1, Number i2, const std::vector<K>& vec); //set i1->i2 components
void set(const std::vector<Number>& is, const std::vector<K>& vec); //set 'is' components
Vector<K> operator()(const std::vector<Number>& is) const; //get 'is' components
Vector<K> operator()(Number i1, Number i2) const; //get i1->i2 components

```

Through overloaded operators, the `Vector` class provides the following (internal) algebraic operations:

```
Vector<K>& operator+=(const K&); //add a scalar to current vector
Vector<K>& operator-=(const K&); //subtract a scalar to current vector
Vector<K>& operator*=(const K&); //multiply the current vector by a scalar
Vector<K>& operator/=(const K&); //divide the current vector by a scalar
Vector<K>& operator+=(const Vector<K>&); //add a vector to current vector
Vector<K>& operator-=(const Vector<K>&); //subtract a vector to current vector
```

Note that compatibility tests are performed on such computations:

```
void mismatchSize(const String&, const size_t) const;
void divideByZero(const String&) const;
void complexCastWarning(const String&, const dimen_t) const;
```

The class also provides norm computations:

```
Real norminf() const; //sup norm
Real norm2squared() const; //squared quadratic norm
Real norm2() const; //<quadratic norm
```

and output facilities (templated ostream <<):

```
void print(std::ostream& ) const;
template<typename K>
std::ostream& operator<<(std::ostream&, const Vector<K>&);
```

Algebraic operations are defined as external templated functions (*template <typename K>* is omitted):

```
Vector<K> operator+(const Vector<K>&); //+U
Vector<K> operator-(const Vector<K>&); //-U
Vector<K> operator+(const Vector<K>&, const Vector<K>&); //U+V
Vector<K> operator+(const K&, const Vector<K>&); //x+U
Vector<K> operator+(const Vector<K>&, const K&); //U+x
Vector<K> operator-(const Vector<K>&, const Vector<K>&); //U-V
Vector<K> operator-(const K&, const Vector<K>&); //x-U
Vector<K> operator-(const Vector<K>&, const K&); //U-x
Vector<K> operator*(const K&, const Vector<K>&); //x*U
Vector<K> operator*(const Vector<K>&, const K&); //U*x
Vector<K> operator/(const Vector<K>&, const K&); //U/x
Vector<K> conj(const Vector<K>&); //conjugate(U)
```

Not that the conjugate function is defined for every vector of type K ; it assumes that the *conj* function is also defined for K value and returns a K value.

To insure automatic cast from real to complex (not the contrary) and particular functions related to complex, some template specializations are provided:

```
std::ostream& operator<<(std::ostream&, const Vector<Real>&); // real vector
flux insertion
std::ostream& operator<<(std::ostream&, const Vector<Complex>&); //complex vector
flux insertion
Vector<Complex> operator+(const Vector<Real>&, const Vector<Complex>&); //rU+cV
Vector<Complex> operator+(const Vector<Complex>&, const Vector<Real>&); //cV+rU
Vector<Complex> operator-(const Vector<Real>&, const Vector<Complex>&); //rU-cV
Vector<Complex> operator-(const Vector<Complex>&, const Vector<Real>&); //cV-rU
Vector<Complex> operator+(const Vector<Real>&, const Complex&); //rU+cx
Vector<Complex> operator+(const Complex&, const Vector<Real>&); //cx+rU
Vector<Complex> operator+(const Vector<Complex>&, const Real&); //cU+rx
Vector<Complex> operator+(const Real&, const Vector<Complex>&); //rx+cU
```

```

Vector<Complex> operator-(const Vector<Real>&, const Complex&);           //rU-cx
Vector<Complex> operator-(const Complex&, const Vector<Real>&);           //cx-rU
Vector<Complex> operator-(const Vector<Complex>&, const Real&);           //cU-rx
Vector<Complex> operator-(const Real&, const Vector<Complex>&);           //rx-cU
Vector<Complex> operator*(const Complex&, const Vector<Real>&);           //cx*rU
Vector<Complex> operator*(const Vector<Real>&, const Complex&);           //rU*cx
Vector<Complex> operator*(const Real&, const Vector<Complex>&);           //rx*cU
Vector<Complex> operator*(const Vector<Complex>&, const Real&);           //cU*rx
Vector<Complex> operator/(const Vector<Real>&, const Complex&);           //rU/cx
Vector<Complex> operator/(const Vector<Complex>&, const Real&);           //cU/rx
Vector<Real> real(const Vector<Complex>&);                                //real(U)
Vector<Real> imag(const Vector<Complex>&);                                //imag(U)
Vector<Complex> cmplx(const Vector<Real>&);
Vector<Complex> cmplx(const Vector<Complex>&);

```

Note that the operations and their specializations allow all standard operations a user can wait from a real or complex vector. In particular, operations involving a casting from real to complex. On the other hand, be cautious when using a `Vector<Vector<K>> V`, all algebraic operations involving `V` and a scalar (of type `K`) are not supported, only those involving `V` and a `vector<K>` are working. Besides, automatic casting from real to complex of vector of vectors is not supported, use the `cmplx` function to convert to complex type.

```

Vector<Vector<Complex>> > cmplx(const Vector<Vector<Real>> >&);

```



Although the `Point` class also inherits from `std::vector` and offers similar algebraic computations facilities to the `Vector` class, it is not linked to the `Vector` class because it deals only with real vector and offers other functionalities, in particular comparison capabilities.

Useful aliases hiding template syntax, are available for end users:

```

typedef Vector<Real> Reals;
typedef Vector<Complex> Complexes;
typedef Vector<Real> RealVector;
typedef Vector<Complex> ComplexVector;
typedef Vector<Vector<Real>> > RealVectors;
typedef Vector<Vector<Complex>> > ComplexVectors;

```

They are defined in `users_typedef.hpp` (`init` library).

library :	utils
header :	Vector.hpp
implementation :	Vector.cpp
unitary tests :	test_Vector.cpp
header dependences :	config.h

3.9 The Matrix class

The purpose of the `Matrix` class is to deal with complex or real dense matrices. It is derived from the `std::vector<K>` and templated by the coefficient type; the coefficients are stored row by row.

```

template<typename K>
class Matrix : public std::vector<K>
{private:
    dimen_t rows_;           //!<number of rows>
public:
    typedef K type_t;         //!<alias on matrix coefficient type>

```



```
typedef typename std::vector<K>::iterator itvk_t;
typedef typename std::vector<K>::const_iterator citvk_t;
...
```

As it is templated, it is also possible to deal with matrix of matrices. But all functions are not supported, in particular those involving autocast.

It offers some basic constructors and assignment function:

```
Matrix(); //by default
Matrix(const dimen_t, const dimen_t); //by dimensions
Matrix(const dimen_t r, const dimen_t, const K&); //by dimensions and constant
Matrix(const Matrix<K>&); //by copy
Matrix(const vector<K>&); //diagonal square matrix from
vector
Matrix(const vector<vector<K>>&); //from a vector of vectors
Matrix(const vector<Vector<K>>&); //from a vector of Vectors
Matrix(const dimen_t, const SpecialMatrix); //for special square matrix
Matrix(const String&); //from file
Matrix(const char *); //from file
```

As the **Vector** class inherited from **std::vector**, the constructor from **std::vector** of **std::vector** works also for **Vector** of **std::vector**, for **std::vector** of **Vector** and for **Vector** of **Vector**. The special matrices are enumerated by the *enum*:

```
enum SpecialMatrix{ _zeroMatrix=0, _idMatrix, _onesMatrix, _hilbertMatrix };
```

Besides the constructors, the following assignment operators are allowed:

```
Matrix<K>& operator=(const Matrix<K>&);
Matrix<K>& operator=(const Matrix<KK>&);
Matrix<K>& operator=(const vector<vector<K>>&);
Matrix<K>& operator=(const vector<Vector<K>>&);
private: void assign(const KK&);
```

Be cautious with the generalized templated version **Matrix<K>& operator=(const Matrix<KK>&)**.

One can access to the dimensions, to the coefficients (both read and read/write access) either by iterators (**begin()** and **end()**) or by index from 1 to the length, to the rows or the columns (both read and read/write access), to the main diagonal:

```
size_t size() const; //overloaded size()
size_t vsize() const; //column length (number of rows)
dimen_t numberOfRows() const; //number of rows
size_t hsize() const; //row length (number of columns)
dimen_t numberOfColumns() const; //number of columns
itvk_t begin(); //overloaded iterator begin
citvk_t begin() const; //overloaded const_iterator begin
itvk_t end(); //overloaded iterator end
citvk_t end() const; //overloaded const_iterator begin
const K& operator()(const dimen_t, const dimen_t) const; //protected access
K& operator()(const dimen_t, const dimen_t); //read/write access
Vector<K> row(const dimen_t) const; //return r-th row (r > 0) as Vector
void row(const dimen_t, const vector<K>&); //set r-th row from a stl vector
Vector<K> column(const dimen_t) const; //return c-th column as a Vector
void column(const dimen_t, const vector<K>&); //set c-th column from a stl vector
Vector<K> diag() const; //return diagonal as a Vector
Matrix<K>& diag(const K); //reset matrix as a diag. matrix
Matrix<K>& diag(const vector<K>&); //reset matrix as a diag. matrix
```


Be care with functions on the diagonal, they change the current matrix to a diagonal matrix.

There are also generalized accesors that allow to extract or set a part of matrix. The row indices and column indices may be given either by lower and upper indices or by vector of indices:

```
Matrix<K> get(const std::vector<Dimen>&, const std::vector<Dimen>&) const;
Matrix<K> get(const std::pair<Dimen, Dimen>&, const std::pair<Dimen, Dimen>&) const;
Matrix<K> get(Dimen, const std::pair<Dimen, Dimen>&) const;
Matrix<K> get(const std::pair<Dimen, Dimen>&, Dimen) const;
Matrix<K> get(Dimen, const std::vector<Dimen>&) const;
Matrix<K> get(const std::vector<Dimen>&, Dimen) const;
Matrix<K> get(Dimen, Dimen, Dimen, Dimen) const;
Matrix<K> operator()(const std::vector<Dimen>&, const std::vector<Dimen>&) const;
Matrix<K> operator()(const std::pair<Dimen, Dimen>&, const std::pair<Dimen, Dimen>&)
    const;
Matrix<K> operator()(Dimen, const std::pair<Dimen, Dimen>&) const;
Matrix<K> operator()(const std::pair<Dimen, Dimen>&, Dimen) const;
Matrix<K> operator()(Dimen, const std::vector<Dimen>&) const;
Matrix<K> operator()(const std::vector<Dimen>&, Dimen) const;
Matrix<K> operator()(Dimen, Dimen, Dimen, Dimen) const;
void set(const std::vector<Dimen>&, const std::vector<Dimen>&, const
    std::vector<K>&);
void set(const std::pair<Dimen, Dimen>&, const std::pair<Dimen, Dimen>&, const
    std::vector<K>&);
void set(Dimen, const std::pair<Dimen, Dimen>&, const std::vector<K>&);
void set(const std::pair<Dimen, Dimen>&, Dimen, const std::vector<K>&);
void set(Dimen, const std::vector<Dimen>&, const std::vector<K>&);
void set(const std::vector<Dimen>&, Dimen, const std::vector<K>&);
void set(Dimen, Dimen, Dimen, Dimen, const std::vector<K>&);
```

Through overloaded operators, the `Matrix` class provides the following (internal) algebraic operations:

```
Matrix<K>& operator+=(const Matrix<KK>&); //add a matrix to the matrix
Matrix<K>& operator+=(const K&);         //add a scalar to the matrix
Matrix<K>& operator-=(const Matrix<KK>&); //subtract a matrix to the matrix
Matrix<K>& operator-=(const K&);         //subtract a scalar to the matrix
Matrix<K>& operator*=(const KK&);        //multiply by a scalar the matrix
Matrix<K>& operator/=(const KK&);        //divide by a scalar the matrix
```

Be careful, the operations such as `Matrix<Matrix<K> > += Matrix<K>` does not work.

Moreover, the `Matrix` class provide a lot of functionalities:

```
bool shareDims(const Matrix<KK>&) const; //true if matrices bear same dimensions
bool isSymmetric() const;               //symmetric matrix test
bool isSkewSymmetric() const;           //skew-symmetric matrix test
bool isSelfAdjoint() const;             //self adjoint matrix test
bool isSkewAdjoint() const;            //skew adjoint matrix test
Matrix<K>& transpose();                  //matrix self-transposition
Matrix<K>& adjoint();                   //matrix self-adjoint
Matrix<Real> real() const;              //real part of a matrix
Matrix<Real> imag() const;              //imaginary part of a matrix
```

and input/output and error utilities:

```
void loadFromFile(const char*);
void saveToFile(const char*);
istream& operator >> (istream&, Matrix<K>&);
ostream& operator << (ostream&, const Matrix<Real>&);
ostream& operator << (ostream&, const Matrix<Complex>&);
```

```

void print(ofstream&);
void print() const;

void nonSquare(const String&, const size_t, const size_t) const;
void mismatchDims(const String&, const size_t, const size_t) const;
void divideByZero(const String&) const;
void isSingular() const;
void complexCastWarning(const String&, const size_t, const size_t) const;

```

Finally, this class provides almost algebraic operations as external functions, with specialized versions allowing automatic cast from real to complex type:

```

bool operator==(const Matrix<K>&, const Matrix<K>&);           //matrix comparison
bool operator!=(const Matrix<K>&, const Matrix<K>&);           //matrix comparison

Matrix<K> operator+(const Matrix<K> &);                       //unary operator+
Matrix<K> operator-(const Matrix<K> &);                       //unary operator-

Matrix<K> operator+(const Matrix<K>&, const Matrix<K>&);       //matrix + matrix
Matrix<K> operator+(const KK&, const Matrix<K>&);             //scalar + matrix
Matrix<K> operator+(const Matrix<K>&, const KK&);               //matrix + scalar
Matrix<Complex> operator+(const Complex&, const Matrix<Real>&);
Matrix<Complex> operator+(const Matrix<Real>&, const Complex&);
Matrix<Complex> operator+(const Matrix<Real>&, const Matrix<Complex>&);
Matrix<Complex> operator+(const Matrix<Complex>&, const Matrix<Real>&);

Matrix<K> operator-(const Matrix<K>&, const Matrix<K>&);       //matrix - matrix
Matrix<K> operator-(const KK&, const Matrix<K>&);             //scalar - matrix
Matrix<K> operator-(const Matrix<K>&, const KK&);               //matrix - scalar
Matrix<Complex> operator-(const Complex&, const Matrix<Real>&);
Matrix<Complex> operator-(const Matrix<Real>&, const Complex&);
Matrix<Complex> operator-(const Matrix<Real>&, const Matrix<Complex>&);
Matrix<Complex> operator-(const Matrix<Complex>&, const Matrix<Real>&);

Matrix<K> operator*(const KK&, const Matrix<K>&);              //scalar * matrix
Matrix<K> operator*(const Matrix<K>&, const KK&);              //matrix * scalar
Matrix<Complex> operator*(const Complex&, const Matrix<Real>&);
Matrix<Complex> operator*(const Matrix<Real>&, const Complex&);
Matrix<K> operator/(const Matrix<K>&, const KK&);              //matrix / scalar
Matrix<Complex> operator/(const Matrix<Real>&, const Complex&);

Vector<K> operator*(const Matrix<K>&, const Vector<V>&);       //matrix * vector
Vector<Complex> operator*(const Matrix<Real>&, const Vector<Complex>&);
Vector<Complex> operator*(const Matrix<Complex>&, const Vector<Real>&);

Matrix<K> operator*(const Matrix<K>&, const Matrix<K>&);       //matrix * matrix
Matrix<Complex> operator*(const Matrix<Real>&, const Matrix<Complex>&);
Matrix<Complex> operator*(const Matrix<Complex>&, const Matrix<Real>&);

Matrix<K> transpose(const Matrix<K>&);                         //transpose matrix
void diag(Matrix<K>&);                                         //change matrix into its diagonal
Matrix<Real> real(const Matrix<Complex>&);                     //real part
Matrix<Real> imag(const Matrix<Complex>&);                     //imaginary part
Matrix<Real> conj(const Matrix<Real>&);                         //conjugate complex matrix
Matrix<Complex> cmplx(const Matrix<Real>&);                     //cast a real matrix to complex one
Matrix<Complex> conj(const Matrix<Complex>&);                   //conjugate complex matrix

```

Some of algebraic operations are based on general templated functions using iterators on vector or matrix:

```

void transpose(const dimen_t, const dimen_t, M_it, MM_it);    //transposition
void adjoint(const dimen_t, const dimen_t, M_it, MM_it);     //adjoint
void matvec(M_it, const V1_it, const V1_it, V2_it, V2_it);   //matrix * vector
V2_it matvec(const Matrix<K>&, const V1_it, const V2_it);     //matrix * vector

```

```
void vecmat(M_it, const V1_it, const V1_it, V2_it, V2_it); //vector * matrix
V2_it vecmat(const Matrix<K>&, const V1_it, const V2_it); //vector * matrix
void matmat(A_it, const dimen_t, B_it, const dimen_t, const dimen_t, R_it);
//matrix * matrix
```

Use these algebraic operations only with `Matrix<real_t>` and `Matrix<complex_t>`. They do not all work for more complicated structure such that `Matrix<Matrix<K> >`.

If the Eigen library is available (it should be), QR and SVD decomposition are available:

```
void qr(const Matrix<T>&, Matrix<T>&, Matrix<T>&);
void svdMat(const Matrix<T>&, Matrix<T>&, Vector<T>&, Matrix<T>&, real_t eps=0);
```

Useful aliases hiding template syntax, are available for end users:

```
typedef Matrix<Real> RealMatrix,
typedef Matrix<Complex> ComplexMatrix;
typedef Matrix<Matrix<Real> > RealMatrices;
typedef Matrix<Matrix<Complex> > ComplexMatrices;
```

They are defined in `users_typedef.hpp` (*init* library).

library :	utils
header :	Matrix.hpp
implementation :	Matrix.cpp
unitary tests :	test_Matrix.cpp
header dependences :	config.h, Vector.hpp, Trace.hpp, Messages.hpp, String.hpp, Algorithms.hpp, complexUtils.hpp

3.10 The collection classes

The `Collection` and `PCollection` are simple template classes to deal with with collection of objects or pointers to object. Inheriting from `vector<T>` or `vector<T*>`, they propose constructors with one, two, three, ... objects that are useful when users want to pass several numbers, several strings, ... to `Parameter` object.

3.10.1 The Collection class

The `Collection` class deals with some collections of objects. It has the following constructors :

```
template <typename T>
class Collection : public std::vector<T>
{
public:
    Collection () : std::vector<T>() {}
    Collection (const std::vector<T>& ts);
    explicit Collection (int n);
    explicit Collection (int n, const T& s)
    Collection(const T& s1);
    Collection(const T& s1, const T& s2);
    ... //constructors up to 48 arguments !
}
```

Besides, it has one accessor to the *i*-th item and some print stuff

```
const T& operator()(number_t n) const;
T& operator()(number_t n);
void print(std::ostream& out) const;
void print(PrintStream& os);
template<typename S>
ostream& operator<<(ostream& out, const Collection<S>& ns);
```

and the operator << is overloaded to mimic the insertion of an object in the collection :

```
template<typename T>
Collection<T>& operator<<(Collection<T>& ns, const T& n)
```

Two important collections are predefined in XLiFE++: the **Numbers** (`Collection<Number>`) and the **Strings** (`Collection<String>`):

```
Numbers ns(2,34,12);
Strings ss("Omega", "Gamma", "Sigma");
```

3.10.2 The PCollection class

the `PCollection` class is similar to the `Collection` class but it deals with collection of pointers to avoid copy of large objects. In order to be user friendly, it hides the pointers and its use is the same as the `Collection` class:

```
template <typename T>
class PCollection : public std::vector<T*>
{
public:
explicit PCollection (number_t n=0) : std::vector<T*>(n,0) {}
explicit PCollection (int n) : std::vector<T*>(n,0) {}
PCollection (const std::vector<T*>& ts) : std::vector<T*>(ts) {}
PCollection(const T& t1);
PCollection(const T& t1, const T& t2);
... //constructors up to 10 arguments !
const T & operator()(number_t n) const;
PCollectionItem<T> operator()(number_t n);
void clearPointers();
void print(std::ostream& out) const;
void print(PrintStream& os) const ;
template <typename S>
ostream& operator<<(ostream& out, const PCollection<S>& ts)
template <typename S>
PCollection<T>& operator<<(PCollection<T>& ts, const T& t)
}
```

`PCollectionItem` class is a small class that interfaces one item of the collection. Its purpose is to move from reference to pointer when inserting an item in the collection using the access operator ():

```
template<typename T> class PCollectionItem
{ public:
typename std::vector<T*>::iterator itp;
PCollectionItem(typename std::vector<T*>::iterator);
T& operator = (const T& t);
operator T&(); // //autocast PCollectionItem -> T&
}
```

As said before, this class may be used in a same way as the `Collection` class is used, keeping in mind that there is no copy of objects and therefore the pointers are shared!

```

PCollection<String> ss(3);
String s1="Omega";
ss(1)=s1; // OK while s1 is not deleted
ss(2)="Gamma"; // FORBIDDEN because "Gamma" is temporary

```

This class is designed for large objects of XLiFE++, such as **GeomDomain**, **Space** that will be presented later.

library :	utils
header :	Collection.hpp
implementation :	Collection.hpp
unitary tests :	test_Parameters.cpp
header dependences :	config.h

3.11 Parameters management

It may be useful to have a data structure to store freely any kind of parameters (values, string, ...) with a very simple interface for the end user. In particular, such structure could be attached to user functions (see chap. 3.12). This is achieved with two classes : the **Parameter** class which define one parameter and the **Parameters** class which manages a list of **Parameter** objects.

3.11.1 The Parameter class

The aim of the **Parameter** class is to encapsulate in the same structure, a data of integer type, of real type, of complex type, of string type, of real vector/complex vector type, of real vector/complex matrix type or of pointer type (void pointer) with the capability to name the parameter. Thus, this class proposes as private members:

```

class Parameter
{private :
    int i_; //!<to store int type data
    real_t r_; //!<to store real_t type data
    complex_t c_; //!<to store complex_t type data
    String s_; //!<to store string data
    void * p_; //!<to store pointer
    String name_; //!<parameter name
    number_t type_; //!<type of value
    ...
}

```

It offers

- a copy constructor and constructors associated to each managed type T (*int*, *real_t*, *complex_t*, *string*, *char**, *void **, *real vector*, *complex vector*, *real matrix*, *complex matrix*), with optional naming:

```

Parameter(const T, const String& nm = String(""));
Parameter(const Parameter&, const String& nm = String(""));

```

- assignment operator = with specialized cases

```

Parameter& operator=(const Parameter& p);
Parameter& operator=(const T&);

```

The real/complex vector and real/complex matrix are managed using the void pointer p_. More precisely, when a parameter is defined from a vector (or a matrix), for instance :

```
Parameter par(std::vector<Real>(5,1.);
```

a copy of the given vector is allocated and its pointer is stored as a void pointer in member p_. In assignment operator, the *deletePointer()* member function is used to free memory previously allocated.

The class provides

- some access functions to data

```
T get_s() const; // (T,s)=(int,i),(real_t,r),(complex_t,c),(String,s),(void *,p)
                // (T,s)=(std::vector<real_t>&rv),(std::vector<complex_t>&cv)
                // (T,s)=(Matrix<real_t>&rm),(Matrix<complex_t>&cm)
number_t get_value_type(const String&);
```

- access to the parameter name and its type

```
string name(); // access to name
void name(const String &); // to change the name
number_t type() const; // access to type
string_t typeName() const; // access to type name
```

- some print and read facilities

```
ostream& operator<<(ostream&, const Parameter&);
void print(const Parameter&);
istream& operator>>(istream&, Parameter&);
```

- some control functions

```
void illegalOperation(const String& t1, const String& op, const String& t2) const;
void illegalOperation2(const String& op, const String& t) const;
void undefinedParameter(const String& t) const;
void illegalDivision() const;
```

When it has meaning, the operator +=, -=, *=, /=, +, -, *, /, ==, !=, >, >=, <, <= are overloaded for parameters and it is possible to call standard mathematical functions (*abs*, *conj*, *log*, ...) with a *Parameter* object as argument. Obviously, these operations are not allowed for pointer or string type parameter.

The pointer type parameter has been introduced to give to the user the possibility to store anything in a parameter (useful in the context of user functions). **It is the responsibility of the users to well manage their void pointers !**

There are casting operators associated to the *Parameter* class. Some are explicit:

```
real_t real(const Parameter& p);
complex_t cmplx(const Parameter& p);
String str(const Parameter& p);
void * pointer(const Parameter& p);
```

and the others are implicit (cast operator):

```
operator int(); //!< cast to int
operator real_t(); //!< cast to real_t
operator complex_t(); //!< cast to complex_t
operator String(); //!< cast to String
operator void*(); //!< cast to void*
operator std::vector<real_t>(); //!< cast to real vector
operator std::vector<complex_t>(); //!< cast to complex vector
operator Matrix<real_t>(); //!< cast to real matrix
operator Matrix<complex_t>(); //!< cast to complex matrix
```


These cast operators allow to write such syntax:

```
Parameter pi(Real(3.1415926), "the real pi");
real_t x=pi; //autocast
```

Unfortunately, casting a parameter to complex does not fully work:

```
Parameter i(complex_t(0,1)); //i is a complex parameter
complex_t j=i; //is working
j=i; //does not work !!!
```

The compiler cannot resolve an ambiguity on the operator `=`: `complex_t=real_t` or `complex_t=complex_t`. The reason is that the autocast of the `Parameter` can produce either a `complex_t` or `real_t` ! It is not possible to omit the `complex_t` autocast version because then the form `complex_t j=i;` no longer works. There is probably a solution but we did not find it !

The `Parameter` class is very simple to use, for instance:

```
Parameter k(3.1415926,"k"); //named real parameter
Parameter i(complex(0.,1.),"i"); //named complex parameter
Parameter s("my string"); //no named string parameter
Parameter pv(RealVector(5,0.5),"vec 5");
Parameter pam(Matrix<Real>(3,_hilbertMatrix),"hilbert matrix");
Vector<String> mat_names(3);
mat_names(1)="iron"; mat_names(2)="aluminium"; mat_names(1)="air";
Parameter names(&mat_names,"mat_names"); //named pointer parameter
real_t vk=get_r();
Parameter ik=i*k; //operation on parameters
ik.name("i*k"); //rename parameter
```

3.11.2 The Parameters class: list of parameters

The `Parameters` class manages a list of `Parameter` objects. Using stream operator `>>`, it offers a friendly way to add a parameter in the list and an index or string index to access to the parameters of the list. It is based on the `map` structure of the STL:

```
class Parameters
{private :
    std::map<String,int> index_; //!< associative list to manage string index
    std::vector<Parameter*> list_; //!< list of parameter's pointers
}
```

The list of parameters pointers is stored in a vector thus providing a direct access to a parameter of the list. The associative map `index_` also provides a direct access to a parameter using a string index.

This classes proposes constructors with a single parameter initialization with an optional name:

```
Parameters();
Parameters(Parameter&);
Parameters(const void*, const string_t& nm = string_t(""));
template<typename T>
Parameters(const T& t, const string_t& nm = string_t(""));
```

some list insertion facilities:

```
void push(Parameter&); //!< append a parameter into list
Parameters& operator<<(Parameter*); //!< insert a parameter into list
Parameters& operator<<(Parameter&); //!< insert a parameter into list
Parameters& operator<<(const Parameter&); //!< insert a parameter into list
Parameters& operator<<(const void*); //!< insert a void pointer into list
template<typename T>
```

```
Parameters& operator<<(const T& t) // insert a T value in list
```

some direct access operators (by index, string index or parameter):

```
Parameter& operator() (unsigned int);
Parameter& operator() (const String&);
Parameter& operator() (const char *);
Parameter& operator() (const Parameter&);
```

and some print utilities:

```
void print(std::ostream&) const ;
void print() const ;
friend void print(const Parameters&);
friend std::ostream& operator<<(std::ostream&, const Parameters&);
```

This parameter list is easy to use:

```
Parameter k(3.1415926, "k"); //named real parameter
RealVector v(10, 0.); //a real vector
Parameter pv(v, "v1"); //associate vector v to a parameter pv
Parameters params; //new list of parameters
params<<k<<"a string"<<pv; //insert parameters or explicit data
RealVector x=params("v1"); //retrieve vector v1
```

When you retrieve a parameter from the list, you have to know the type of the parameter to cast it in the right type. Note that the member function *type()* of the Parameter class gives the parameter type.

There exists a *defaultParameters* global variable declared in the header *config.h* and initialized as an empty list in the header *globalScopeData.h*.

library :	utils
header :	Parameters.hpp
implementation :	Parameters.cpp
unitary tests :	test_Parameters.cpp
header dependences :	config.h, String.hpp

3.12 The Function class

The object function aims to encapsulate the user functions with an additional information, says parameter data. This encapsulation are concerned with functions of a point or a couple of points, required by finite element computations at a low level. These functions are typically some physical parameters involved in integrals or source data functions. They may require other constant parameters. It is the reason why it is necessary to attach to the function a list of parameters (constant values of real type, complex type, ...). This class is designed to deal with standard functions involved in a finite element software (24 types):

```
r      function(const Point&, Parameters&)
Vector<r> function(const Vector<Point>&, Parameters&)
r      function(const Point&, const Point&, Parameters&)
Vector<r> function(const Vector<Point>&, const Vector<Point>&, Parameters&)
```

with r of type:

```
real_t , complex_t , Vector<real_t> , Vector<complex_t> , Matrix<real_t> ,
Matrix<complex_t>
```

The functions (resp. kernels) taking a point or a couple of points as input argument are called "scalar" functions (resp. kernel) whereas those taking a vector of points or a couple of vector of points are called "vector" functions (resp. kernels). Do not confuse a "vector" function with a function returning a vector

and taking a point as input argument. For a such function, its vector form returns a vector of vectors.

The object function is based on the non templated class **Function**:

```
template<typename T_> class Function
{
public:
    Function() {}
protected:
    ValueType type_returned_; // type of returned value ( _real, _complex)
    StrucType struct_returned_; // struct of returned value (_scalar, _vector, _matrix)
    FuncType type_function_; // type of the function (_function or _kernel)
    ArgType argType_; // type of the argument (_point, _vector_point)
    mutable Parameters* params_; // pointer to parameter list
    dimPair dims_; // dimension of returned values
    String name_; // name of function and comments
    void * fun_; // store pointer to function in a void *
    //flag in OperatorOnUnknown construction (see operator lib)
    mutable bool check_type_; // to activate a checking of arguments
    mutable bool conjugate_; // temporary flag for conjugate operation
    mutable bool transpose_; // temporary flag for transpose operation
    //to manage data requirements
    bool requireNx; // true if function requires normal or x-normal vector
    bool requireNy; // true if function (kernel) requires y-normal vector
    bool requireElt; // true if function requires element
    bool requireDom; // true if requires domain
    bool requireDof; // true if requires dof
    //special attributes for kernel
    mutable bool xpar; // true if x is consider as a parameter
    mutable Point xory; // x or y as parameter of the kernel
public:
    static std::map<String, std::pair<ValueType, StrucType> > returnTypes;
    static std::map<String, std::pair<ValueType, StrucType> > returnArgs;
};
```

which manages the user function storing its pointer in the void pointer *fun_*, its parameters list in the Parameters pointer *params_* and a function name (optional) in the string *name_*. There are also members to store characteristics of the function (type and arguments type) allowing to identify quickly the return type of the function. The member *check_type_* is a flag used in safe evaluation process of the function. The two static maps are used to the management of the types.



We choose to use a non templated class for two reasons: to avoid template paradigm to the end users and to have a simpler way to transmit such object function in different part of the library. The counterpart of this choice is a more complex Function class but the complexity is only located in the class.

For each type of function is defined a useful alias of the pointer function:

```
typedef r(fun_xx_t)(const Point&, Parameters&);
typedef Vector<r>(vfun_xx_t)(const Vector<Point>&, Parameters&);
typedef r(ker_xx_t)(const Point&, const Point&, Parameters&);
typedef Vector<r>(vker_xx_t)(const Vector<Point>&, const Vector<Point>&,
    Parameters&);
```

where *r* is the return type and *xx* a code of the return type (sr,sc,vr,vc,mr,mc); s for scalar, v for vector, m for matrix, r for real and c for complex.

The class **Function** proposes:

- two constructors for each type of function (named and unnamed):

```
Function(fun_sr_t& f, Parameters& pa = def_parameters); //real scalar type
Function(fun_sr_t& f, const String& na, Parameters& pa = def_parameters);
```

```

Function(fun_sc_t& f, Parameters& pa = def_parameters); //complex scalar type
Function(fun_sc_t& f, const String& na, Parameters& pa = def_parameters);
...

```

which initialize the void pointer and compute the return types using the member function *init()*.

- evaluation capabilities (unsafe but fast) for each type of function:

```

real_t fun_sr(const Point &x)
complex_t fun_sc(const Point &x)
...

```

As these functions only recast the void pointer to the required pointer function:

```

real_t fun_sr(const Point &x)
{return reinterpret_cast<fun_sr_t*>(fun_)(x,*params_);}

```

they are unsafe !

- templated overloaded operator (), insensitive to the form of function (safe but not as fast as previous)

```

template <typename T_>
T_& operator()(const Point &x, T_ & res);
template <typename T_>
T_& operator()(const Point &x, const Point& y, T_& res);
template <typename T_>
Vector<T_>& operator()(const Vector<Point>& x, Vector<T_>& res);
template <typename T_>
Vector<T_>& operator()(const Vector<Point>& x,
                    const Vector<Point>& y, Vector<T_> & res);

```

Note that these functions return a reference, contrary to the direct straight functions which return a value. Besides, by testing the input arguments, they are able to deal with single point or vector of points independently of the scalar or vector form of the user function :

```

template<typename T_>
T_& Function::operator()(const Point &x, T_ & res)
{
    if (check_type_) checkFunctionType(res, Function_t); //check argument types
    if (type_arg_==Point_t) //case of a function of a point
    {
        typedef typename PointerF<T_>::fun_p* funT;
        funT f=reinterpret_cast<funT>(fun_);
        res=f(x,*params_);
    }
    else //case of a function of a vector of points (construct a vector)
    {
        typedef typename PointerF<T_>::vfun_p* vfunT;
        vfunT f=reinterpret_cast<vfunT>(fun_);
        Vector<Point> V;
        V.push_back(x);
        Vector<T_> R=f(V,*params_);
        res=R[0];
    }
    return res;
}

```

Moreover, using the *checkFunctionType()* they can also perform a control of the type of the function. As the checking function uses expensive RTTI capabilities, this possibility is conditioned by the *check_type* state (true or false) which is automatically reset to false by the *checkFunctionType()*.

- accessors to the function characteristics:

```

ValueType typeReturned() const;
StrucType structReturned() const;
ValueType valueType() const;
StrucType strucType() const;
FuncType typeFunction() const;
ArgType typeArg() const;
bool conjugate() const;
bool& conjugate();
void conjugate(bool v) const;
bool transpose() const;
bool& transpose();
void transpose(bool v) const;
bool isVoidFunction() const;
const void* fun_p() const;
Parameters& params();
Parameters& params() const;
Parameters*& params_p();
const Parameters* params_p() const;
dimPair dims() const;
string_t name() const;

```

- some functions to manage the function parameters and X/Y parameter in kernel:

```

void addParam(Parameter& pa;
template<typename K>
void setParam(K& v, const string_t& name) const;
Parameter& parameter(const string_t&) const;
Parameter& parameter(const char* s) const;
Parameter& parameter(const size_t) const;
void setX(const Point& P) const; // for kernel
void setY(const Point& P) const;

```

- some functions to manage the function data required during computation:

```

bool normalRequired() const;
void require(UnitaryVector uv, bool tf=true);
void require(const string_t& s, bool tf=true);

```

- a print utility:

```

void printInfo() const

```

To deal with some data that are available during FE computation such as normal vectors, current element, current dof or current domain, there is two things to do : tell to XLiFE++ that the Function will use such data

```

Function F(fun); //link a C++ function fun to a Function object
F.require(_n); //tell that function fun will use normal vector
F.require("element"); //tell that function fun will use element

```

and get data in the C++ function related to the Function object:

```

Real fun(const Point& x, Parameters& pars=theDefaultParameters)
{...
  Vector<Real>& n = getN(); //get the normal vector
  GeomElement& elt=getElement(); //get element
  ...
}

```

This machinery is based on the `ThreadData` class that manages one instance of data by thread.

library :	utils
header :	Function.hpp
implementation :	Function.cpp
unitary tests :	test_Function.cpp
header dependences :	config.h, Parameters.hpp, Point.hpp, String.hpp, Vector.hpp

3.13 The Kernel class

The **Kernel** class is intended to deal with kernel, say function $G(x, y)$ where x, y are two points. Basically, it manages the function itself using **Function** object that is able to deal with such function but it manages other functions such as $\nabla_x G$, $\nabla_y G$, $\nabla_{xy} G$, $\frac{\partial G}{\partial n_x}$, $\frac{\partial G}{\partial n_y}$:

```
public :
Function kernel;
Function gradx;
Function grady;
Function gradxy;
Function ndotgradx;
Function ndotgrady;
Parameters userData;
```

All functions defined in **Kernel** class share the same parameters given by **userData**. Besides, the class provides some additional properties:

```
SingularityType singularType;    // singularity type
real_t singularOrder;            // singularity order
complex_t singularCoefficient;   // singularity coefficient
SymType symmetry;               // kernel symmetry
string_t name;                  // kernel name
dimen_t dimPoint;               // dim of points x,y
//temporary data
mutable bool checkType_;         // to activate checking mode
mutable bool conjugate_;         // flag for conjugate operation
mutable bool transpose_;         // flag for transpose operation
mutable bool xpar;               // true if x is consider as a parameter
mutable Point xory;             // x/y as parameter of the kernel
//to manage data requirements
bool requireNx;                 // true if requires normal or x-normal vector
bool requireNy;                 // true if requires y-normal vector
bool requireElt;                // true if requires element
bool requireDom;                // true if requires domain
bool requireDof;                // true if requires dof
```

The following singularity types are currently available:

```
enum SingularityType {_notsingular , _r , _logr , _loglogr };
```

It is also possible to specify the singular and regular part using **Kernel** pointer (0 by default):

```
Kernel* singPart;
Kernel* regPart;
```

There are simple constructors from **Function** objects or from explicit C++ functions having a kernel form (see **Function** class):

```
Kernel() ;                      // default constructor
Kernel(const Kernel&);          // copy constructor
Kernel& operator=(const Kernel&); // assign operator
//from Function object
Kernel(const Function& ker , SingularityType st=_notsingular ,
```

```

    real_t so=0, SymType sy=noSymmetry);
Kernel(const Function& ker, const string_t& na, SingularityType st=_notsingular,
    real_t so=0, SymType sy=noSymmetry);
//from explicit function having a kernel form
template <typename T>
Kernel(T(fun)(const Point&, const Point&, Parameters&),
    SingularityType st=_notsingular, real_t so=0, SymType sy=noSymmetry);
template <typename T>
Kernel(T(fun)(const Point&, const Point&, Parameters&), const string_t& na,
    SingularityType st=_notsingular, real_t so=0, SymType sy=noSymmetry);
...
virtual void initParameters(); //to share user parameters

```

Note that these simple constructors do not allow to set other functions (gradx, grady, ...).

This class provides some accessors:

```

bool conjugate() const;
bool& conjugate();
void conjugate(bool v) const;
bool transpose() const;
bool& transpose();
void transpose(bool v) const;
virtual bool isSymbolic() const;
virtual ValueType valueType() const;
virtual StrucType structType() const;
virtual dimPair dims() const;

```

It provides also some tools to manage parameters:

```

void setParameters(const Parameters& par);
void setParameter(Parameter& par);
template<typename T>
void setParameter(const string_t& name, T value);
template<typename T>
T getParameter(const string_t& name, T value) const;
void setX(const Point& P) const;
void setY(const Point& P) const;

```

and some functions to manage the kernel data required during computation:

```

bool normalRequired() const;
bool xnormalRequired() const;
bool ynormalRequired() const;
void require(UnitaryVector uv, bool tf=true);
void require(const string_t& s, bool tf=true);

```

In some computations, a **Kernel** object has to be viewed as a **Function** object, x or y being considered as a parameter. the following member functions do the job:

```

//move to y->k(x,y) or x->k(x,y)
const Function& operator()(const Point& x, variableName vn) const;
const Function& operator()(variableName vn, const Point& y) const;
const Function& operator()(variableName vn) const;

```

variableName is defined by the following enumeration:

```

enum VariableName {_x, _x1=_x, _y, _x2=_y, _z, _x3=_z, _t};

```

The class provides function to evaluate kernel:

```

template<typename T>
T& operator()(const Point&, const Point&, T&) const; //compute at (x,y)

```

```

template<typename T>
T& operator()(const Point&, T&) const; //compute at (x,P) or (P,y) with P=xory
template<typename T>
Vector<T>& operator()(const Vector<Point>&,const Vector<Point>&,
                    Vector<T>&) const; //compute at a list of points

```

As the `Kernel` class is the base class of `TensorKernel` class which deals with particular kernel, there are some virtual functions:

```

virtual KernelType type() const; //generalKernel, _tensorKernel
virtual const TensorKernel* tensorKernel() const; //downcast to tensorkernel
virtual TensorKernel* tensorKernel(); //downcast to tensorkernel*

```

To deal with some data that are available during FE computation such as normal vectors, current element, current dof or current domain, there is two things to do : tell to XLiFE++ that the Kernel will use such data

```

Kernel K(ker); //link a C++ function(kernel) to a Kernel object
K.require(_nx); //tell that function ker will use normal vector

```

and get data in the C++ function related to the Kernel object:

```

Real ker(const Point& x, const Point& y, Parameters& pars=theDefaultParameters)
{...
Vector<Real>& n = getN(); //get the normal vector
...
}

```

This machinery is based on the `ThreadData` class that manages one instance of data by thread.

As an example, we give the sketch of the program constructing the Laplace kernel in 3d:

```

Kernel Laplace3dKernel(Parameters& pars)
{
    Kernel K;
    K.name="Laplace 3D kernel";
    K.singularType =_r;
    K.singularOrder = -1;
    K.singularCoefficient = over4pi;
    K.symmetry=_symmetric;
    K.userData.push(pars);
    K.kernel = Function(Laplace3d, K.userData);
    K.gradx = Function(Laplace3dGradx, K.userData);
    K.grady = Function(Laplace3dGrady, K.userData);
    K.ndotgradx = Function(Laplace3dNxdotGradx, K.userData);
    K.ndotgrady = Function(Laplace3dNydotGrady, K.userData);
    return K;
}

real_t Laplace3d(const Point& x, const Point& y, Parameters& pars)
{
    real_t r = x.distance(y);
    return over4pi / r;
}

//derivatives
Vector<real_t> Laplace3dGradx(const Point& x, const Point& y, Parameters& pars)
{
    real_t r2 = x.squareDistance(y);
    real_t r = std::sqrt(r2);
    Vector<real_t> g1(3);
    scaledVectorTpl(over4pi / (r*r2), x.begin(), x.end(), y.begin(), g1.begin());
    return g1;
}

```

```

Vector<real_t> Laplace3dGrady(const Point& x, const Point& y, Parameters& pars)
{
    real_t r2 = x.squareDistance(y);
    real_t r = std::sqrt(r2);
    Vector<real_t> g1(3);
    scaledVectorTpl(-over4pi / (r*r2), x.begin(), x.end(), y.begin(), g1.begin());
    return g1;
}

real_t Laplace3dNxdotGradx(const Point& x, const Point& y, Parameters& pars)
{
    Vector<real_t>& nxp = getNx(); //get normal at x point
    std::vector<real_t>::const_iterator itx=x.begin(), ity=y.begin(), itn=nxp.begin();
    real_t d1=(*itx++ - *ity++), d2=(*itx++ - *ity++), d3=(*itx - *ity);
    real_t r = d1* *itn++; r+= d2* *itn++; r+=d3* *itn;
    real_t r3= d1*d1+d2*d2+d3*d3; r3*=std::sqrt(r3);
    return -r*over4pi_/r3;
}
...

```

See in the *mathResources* directory, the kernels that are available.

library :	utils
header :	Kernel.hpp
implementation :	Kernel.cpp
unitary tests :	test_Function.cpp
header dependences :	config.h, Parameters.hpp, Point.hpp, String.hpp, Vector.hpp, Function.hpp

3.14 The TensorKernel class

The **TensorKernel** class, inherited from the **Kernel** class, deals with kernels of the form:

$$K(x, y) = \sum_{1 \leq m \leq M} \sum_{1 \leq n \leq N} \psi_m(x) \mathbb{A}_{mn} \phi_n(y)$$

with \mathbb{A} a $M \times N$ matrix. Such kernel is involved, for instance, in Dirichlet to Neumann method using spectral expansion. In many cases, this matrix is a diagonal one:

$$K(x, y) = \sum_{1 \leq n \leq N} \psi_n(x) \lambda_n \phi_n(y).$$

In the meaning of XLiFE++, the families $(\psi_m)_{1 \leq m \leq M}$ and $(\psi_n)_{1 \leq n \leq N}$ may be considered as some basis of spectral space, say **SpectralBasis** object. So, the **TensorKernel** class is defined as follow:

```

protected :
    const SpectralBasis* phi_p;
    const SpectralBasis* psi_p;
    VectorEntry* matrix_p;
public :
    bool isDiag;           // true if matrix is a diagonal one
    Function xmap;         // geometric transformation applied to x
    Function ymap;         // geometric transformation applied to y
    bool isConjugate;      // conjugate phi_p in computation (default=false)

```

The **xmap** and **ymap** functions allow to map the basis functions from a reference space to an other one. For instance, suppose the basis $\cos(n\pi x)$ on the segment $]0, 1[$ is given:


```

Real cosn(const Point& P, Parameters& pars)
{
  Real x=P(1);
  Int n=pa("basis index")-1; //get the index of function to compute
  return cos(n*pi*x);
}

```

and has to be used on the boundary $\Sigma = \{(0, y), 0 < y < h\}$. The basis will be map onto using the function:

```

Point sigmaTo(const Point& P, Parameters& pars)
{
  return Point(P(2)/h);
}

```

The **SpectralBasis** class manages either a basis given by an explicit C++ function (analytic form) or a basis given by a set of **TermVector** objects (numerical form). So the constructors provided by the **TensorKernel** class manages these different cases too:

```

TensorKernel(); //default constructor
template<typename T>
//from one or two SpectralBasis objects
TensorKernel(const SpectralBasis&, const vector<T>&, bool conj=false);
TensorKernel(const SpectralBasis&, const vector<T>&, const SpectralBasis&);
//from one or two Unknown objects belonging to spectral spaces
TensorKernel(const Unknown&, const vector<T>&, bool conj=false);
TensorKernel(const Unknown&, const vector<T>&, const Unknown&);
//from one or two Function objects
TensorKernel(Function, const vector<T>&, bool conj=false);
TensorKernel(Function, const vector<T>&, Function);
//from one or two set of TermVector objects
TensorKernel(const vector<TermVector>&, const vector<T>&, bool conj =false);
TensorKernel(const vector<TermVector>&, const vector<T>&,
              const vector<TermVector>&);

```

Note that these constructors does not allow to set maps! They have to be set after construction. When the user defines **TensorKernel** from an explicit function, it has to get the index from the parameters related to the function. This index is automatically set by the computation functions and its name in parameters is "basis index".

The **TensorKernel** class provides some accessors and properties:

```

const VectorEntry& vectorEntry() const;
const SpectralBasis* phip() const;
const SpectralBasis* psip() const;
const SpectralBasis& phi() const;
const SpectralBasis& psi() const;
dimen_t dimOfPhi() const;
dimen_t dimOfPsi() const;
number_t nbOfPhi() const;
number_t nbOfPsi() const;
bool phiIsAnalytic() const;
bool psiIsAnalytic() const;
virtual KernelType type() const;
virtual bool isSymbolic() const;
virtual ValueType valueType() const;
virtual StrucType structType() const;
bool sameBasis() const;
virtual const TensorKernel* tensorKernel() const;
virtual TensorKernel* tensorKernel();

```

It implements the computation of kernel:


```
template<typename T>
T kernelProduct(number_t, const T&) const;
template<typename T>
T kernelProduct(number_t, number_t, const T&) const;
```

library :	terms
header :	TensorKernel.hpp
implementation :	TensorKernel.cpp
unitary tests :	test_Function.cpp
header dependences :	config.h, Parameters.hpp, Point.hpp, String.hpp, Vector.hpp, Function.hpp, TermVector.hpp, SpectralBasis.hpp

3.15 The ThreadData class

The `ThreadData` class is a utility class that manages some FE computation data (normal vectors, current element, current dof, current domain) that may be get by user function or user kernel. When the computation is done on several threads, these data have multiple instances (one by thread), this a reason why a `ThreadData` object handles some vectors of data pointers:

```
vector<Vector<real_t>*> theCurrentNxs;    // normal pointers or x-normal pointers
vector<Vector<real_t>*> theCurrentNys;    // y-normal pointers
vector<GeomElement*> theCurrentElements; // element pointers
vector<Dof*> theCurrentDofs;             // dof pointers
GeomDomain* currentDomainx;             // domain pointer or domain_x pointer
GeomDomain* currentDomainy;             // domain_y pointer
```

It has only one a basic constructor and a resize function that is called when the number of threads changes:

```
ThreadData(number_t s=1);
void resize(number_t);
```

Besides, it offers various functions to set or get the data:

```
void setNx(Vector<real_t>*>);
void setNy(Vector<real_t>*>);
void setElement(GeomElement*);
void setDomainx(GeomDomain*);
void setDomainy(GeomDomain*);
void setDof(Dof*);
void setNx(Vector<real_t>*, number_t);
void setNy(Vector<real_t>*, number_t);
void setElement(GeomElement*, number_t);
void setDof(Dof*, number_t);

Vector<real_t>& getNx(number_t) const;
Vector<real_t>& getNx() const;
Vector<real_t>& getNy(number_t) const;
Vector<real_t>& getNy() const;
GeomElement& getElement(number_t);
GeomElement& getElement() const;
GeomDomain& getDomainx() const;
GeomDomain& getDomainy() const;
Dof& getDof(number_t) const;
Dof& getDof() const;
FeDof& getFeDof(number_t) const;
```

```
FeDof& getFeDof() const;
```

Print utility are also available

```
void print(std::ostream&) const;
void print(PrintStream& os) const {print(os.currentStream());}
```

When starting, XLiFE++ create one instance of `ThreadData`: `theThreadData` (see *globalScopeData.hpp*, *config.hpp*). To avoid the user to deal explicitly with it, some extern functions (wrappers to member functions) are also proposed:

```
inline void setNx(Vector<real_t>* p, number_t t);
inline void setNy(Vector<real_t>* p, number_t t);
inline void setElement(GeomElement* p, number_t t);
inline void setDof(Dof* p, number_t t);
inline void setNx(Vector<real_t>* p);
inline void setNy(Vector<real_t>* p);
inline void setElement(GeomElement* p);
inline void setDomain(GeomDomain* p);
inline void setDomainx(GeomDomain* p);
inline void setDomainy(GeomDomain* p);
inline void setDof(Dof* p);
inline Vector<real_t>& getN(number_t t);
inline Vector<real_t>& getNx(number_t t);
inline Vector<real_t>& getNy(number_t t);
inline GeomElement& getElement(number_t t);
inline Dof& getDof(number_t t);
inline FeDof& getFeDof(number_t t);
inline Vector<real_t>& getN();
inline Vector<real_t>& getNx();
inline Vector<real_t>& getNy();
inline GeomElement& getElement();
inline GeomDomain& getDomain();
inline GeomDomain& getDomainx();
inline GeomDomain& getDomainy();
inline Dof& getDof();
inline FeDof& getFeDof();
```

This machinery allows the user to simply deal with such computation data:

```
Real fun(const Point& x, Parameters& pars=theDefaultParameters)
{...
Vector<Real>& n = getN();           //get the normal vector
GeomElement& elt=getElement();    //get element
...}

int main()
{...
Function F(fun);                  //link a C++ function fun to a Function object
F.require(_n);                    //tell that function fun will use normal vector
F.require("element");             //tell that function fun will use element
...
}
```

library :	utils
header :	ThreadData.hpp
implementation :	ThreadData.cpp
unitary tests :	test_ThreadData.cpp
header dependences :	config.h, utils.h

3.16 The SymbolicFunction class

The `SymbolicFunction` class handles a function described in a symbolic way, that is an expression involving variables, constant and operators. In fact, a `SymbolicFunction` object is a node of a tree :

```
class SymbolicFunction
{
public :
    const SymbolicFunction* fn1 , *fn2;    //nodes involved 0, 1 or 2 (0 by default)
    VariableName var;                      //variable involved (undef by default)
    SymbolicOperation op;                  //operation involved (id by default)
    complex_t coef;                        //coefficient to apply to
    complex_t par;                          //additional parameter
    ...
}
```

A node may be

- a constant value stored in `coef` with `fn1=fn2=0` and `var=_varundef`
- a variable (`_x1`, or `_x2` or `_x3`) stored in `var`, with `fn1=fn2=0`
- a unary operation stored in `op` and acting either on the variable or the `SymbolicFunction` object `fn1`
- a binary operation stored in `op` and acting on the `SymbolicFunction` objects `fn1` and `fn2`

The unary and binary operators are given by the enumeration:

```
enum SymbolicOperation
{_idop=0, _plus, _minus, _multiply, _divide, _power,
 _equal, _different, _less, _lessequal, _greater, _greaterequal, _and, _or,
 _abs, _realPart, _imagPart, _sqrt, _squared, _sin, _cos, _tan,           //unary
 _asin, _acos, _atan, _sinh, _cosh, _tanh, _asinh, _acosh, _atanh,       //unary
 _exp, _log, _log10, _pow, _not                                         //unary
};
```

As an example, the function $f(x_1, x_2) = (x_1 + x_2) < 1 \ \&\& \ \exp(x_1) > 2$ has the following tree representation:

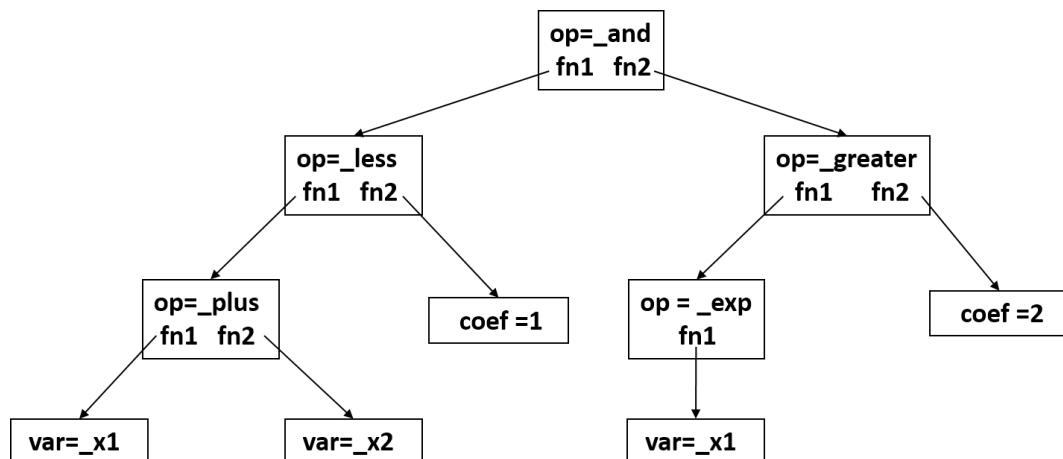


Figure 3.1: example of SymbolicFunction tree



The function `asinh`, `acosh`, `atanh` are only available when compiling with a C++ version greater than 2011.

The `SymbolicFunction` class provides some basic constructors for each kind of node:

```
explicit SymbolicFunction(VariableName v=_varUndef);
explicit SymbolicFunction(const real_t&);
explicit SymbolicFunction(const complex_t&);
SymbolicFunction(const SymbolicFunction&, SymbolicOperation,
                 complex_t c=1., complex_t p=0.);
SymbolicFunction(const SymbolicFunction&, const SymbolicFunction&,
                 SymbolicOperation, complex_t c=1., complex_t p=0.);
SymbolicFunction(const SymbolicFunction&);
~SymbolicFunction();
SymbolicFunction& operator=(const SymbolicFunction&);
```

The copy constructor and the assign operator do full copy of object referenced by pointers `fn1` and `fn2`. So, the destructor deletes them if they are assigned. Note that the variable `coef` and `par` are complex. Obviously, they may store some real scalars and the code interprets the imaginary part to know if they are real.

In order to make easier the construction of `SymbolicFunction` object, a lot of function or operator are provided, for instance :

```
//binary operation on SymbolicFunction's
SymbolicFunction& operator+ (const SymbolicFunction&, const SymbolicFunction&);
SymbolicFunction& operator- (const SymbolicFunction&, const SymbolicFunction&);
SymbolicFunction& operator* (const SymbolicFunction&, const SymbolicFunction&);
SymbolicFunction& operator/ (const SymbolicFunction&, const SymbolicFunction&);
SymbolicFunction& operator^ (const SymbolicFunction&, const SymbolicFunction&);
SymbolicFunction& operator==(const SymbolicFunction&, const SymbolicFunction&);
SymbolicFunction& operator!=(const SymbolicFunction&, const SymbolicFunction&);
SymbolicFunction& operator> (const SymbolicFunction&, const SymbolicFunction&);
SymbolicFunction& operator>=(const SymbolicFunction&, const SymbolicFunction&);
SymbolicFunction& operator< (const SymbolicFunction&, const SymbolicFunction&);
SymbolicFunction& operator<=(const SymbolicFunction&, const SymbolicFunction&);
SymbolicFunction& operator&&(const SymbolicFunction&, const SymbolicFunction&);
SymbolicFunction& operator||(const SymbolicFunction&, const SymbolicFunction&);
```

```
//unary operation on SymbolicFunction
SymbolicFunction& operator-(const SymbolicFunction&);
SymbolicFunction& operator+(const SymbolicFunction&);
SymbolicFunction& operator!(const SymbolicFunction&);
SymbolicFunction& abs(const SymbolicFunction&);
SymbolicFunction& real(const SymbolicFunction&);
SymbolicFunction& imag(const SymbolicFunction&);
SymbolicFunction& sqrt(const SymbolicFunction&);
SymbolicFunction& squared(const SymbolicFunction&);
SymbolicFunction& sin(const SymbolicFunction&);
SymbolicFunction& cos(const SymbolicFunction&);
SymbolicFunction& tan(const SymbolicFunction&);
SymbolicFunction& asin(const SymbolicFunction&);
SymbolicFunction& acos(const SymbolicFunction&);
SymbolicFunction& atan(const SymbolicFunction&);
SymbolicFunction& sinh(const SymbolicFunction&);
SymbolicFunction& cosh(const SymbolicFunction&);
SymbolicFunction& tanh(const SymbolicFunction&);
SymbolicFunction& exp(const SymbolicFunction&);
SymbolicFunction& log(const SymbolicFunction&);
SymbolicFunction& log10(const SymbolicFunction&);
SymbolicFunction& pow(const SymbolicFunction&, const double&);
#if __cplusplus > 199711L
SymbolicFunction& asinh(const SymbolicFunction&);
SymbolicFunction& acosh(const SymbolicFunction&);
```

```
SymbolicFunction& atanh(const SymbolicFunction&);
#endif
```

The binary operators are also provided for `SymbolicFunction` object and constant (real or complex).

Besides, some basic tools are proposed by the class:

```
bool isSingle() const; //true if a constant or variable
bool isConst() const; //true if a constant expression
set<VariableName> listOfVar() const; //list of involved variables
number_t numberOfVar() const; //number of involved variables
void print(ostream& = cout) const;
void printTree(ostream& =cout, int lev=0) const;
```

Finally, by overloading the operator `()`, the symbolic function can be evaluated for any set of real values or complex values:

```
complex_t operator()(const vector<complex_t>& xs) const;
complex_t operator()(const complex_t& x1) const;
complex_t operator()(const complex_t& x1, const complex_t& x2) const;
complex_t operator()(const complex_t& x1, const complex_t& x2,
                    const complex_t& x3) const;
real_t operator()(const vector<real_t>& xs) const;
real_t operator()(const real_t& x1) const;
real_t operator()(const real_t& x1, const real_t& x2) const;
real_t operator()(const real_t& x1, const real_t& x2, const real_t& x3) const;
```

These member functions use the fundamental functions that performs computations:

```
inline real_t evalOp(SymbolicOperation op, const real_t& x, const real_t& y);
inline real_t evalFun(SymbolicOperation op, const real_t& x, const real_t& p=0.);
inline complex_t evalOp(SymbolicOperation op, const complex_t& x,
                      const complex_t& y);
inline complex_t evalFun(SymbolicOperation op, const complex_t& z,
                      const complex_t& p=0.);
```

Hereafter, some examples of construction of `SymbolicFunction` objects:

```
SymbolicFunction fs;
fs = x_1+x_2-1;
fs = 2*x_1-3*x_2;
fs = 5*(2*x_1-3*x_2);
fs = 4*sqrt(2*x_1-3*x_2)/x_2;
fs = sin(cos(sqrt(abs(log(x_1*x_1+x_2*x_2+x_3*x_3)))));
fs = (1.-sin(x_1))*(cos(1.+x_1)/5 + 2*sin(3*x_1));
fs = pow(squared(x_1), 0.5);
fs = SymbolicFunction(3);
fs = -sin(x_1);
fs = +sin(-x_1);
fs = cos(x_2)-2*sqrt(x_1*x_2);
fs = x_1 < 1;
fs = (x_1+x_2)<=1 && exp(x_1)>2;
fs = (x_1+x_2)^3;
```

library :	utils
header :	SymbolicFunction.hpp
implementation :	SymbolicFunction.cpp
unitary tests :	unit_SymbolicFunction.cpp
header dependences :	config.h, Messages.hpp

3.17 The Graph class

The **Graph** class is a simple class representing a numbering graph well suited to deal with storage of matrices, in particular to optimize the matrix numbering in order to minimize its bandwidth. It is not a general **Graph** class.

Each node, represented by a `vector<Number>`, contains the connection between this node and others. Node n_i and n_j are "connected" if n_j belongs to `vector<Number>` defining n_i . The degree of a node is the number of nodes connected to it. It is implemented as a child class of `vector<vector<Number> >`:

```
class Graph : public std::vector<std::vector<Number> >
{
public:
    Graph(Number); //basic constructor
    Number nodeDegree(Number) const;
    Number maximumDegree() const;
    ...
};
```

It provides some tools to compute matrix bandwidth and length of skyline storage of matrix, and to optimize numbering to reduce the bandwidth:

```
pair<Number, Number> bandwidthAndSkyline() const;
Number bandwidth() const;
Number skylineSize() const;
vector<Number> renumber(Number);
Number renumEngine(vector<bool>&, vector<Number>&, Number&, Number,
                  Number&, Number&);
```

and some printing facilities:

```
void print(std::ostream&, const String&) const;
void print(const String& title = "") const;
void printNodes(std::ostream&, const vector<Number>&);
void printNodes(const vector<Number>&);
```

library :	utils
header :	Graph.hpp
implementation :	Graph.cpp
unitary tests :	test_Graph.cpp
header dependences :	

3.18 The Value class

The **Value** class is a utility class to encapsulates user's data involved in a differential expression. Such data may be a function (**Function** object) or any real or complex scalar, vector or matrix value. This class is designed as follows:

```
class Value
{
protected :
    ValueType type_; //type of returned value ( _real, _complex)
    StrucType struct_; //structure of returned value ( _scalar, _vector, _matrix,
                                                         _vectorofvector, _vectorofmatrix, _matrixofmatrix)
    void * value_p; //void pointeur to the true value
    mutable bool conjugate_; //temporary conjugate state flag
```

```
mutable bool transpose_; //temporary transposition state flag
```

The attributes *conjugate_* and *transpose_* are temporary flags used when chaining Value object in a differential expression. They have no meaning outside a particular context!

In order to recast safely the void pointer *value_p* pointing to the actual data, this class uses the Run Time Information, to store the names of data type supported:

```
typedef std::pair<ValueType, StructType> structPair;
typedef std::pair<String, structPair> mapPair;
static std::map<String, structPair> theValueTypeRTInames;
static void valueTypeRTInamesInit();
```

The only constructors allowed are the following:

```
Value(const real_t &);
Value(const Vector<real_t> &);
Value(const Matrix<real_t> &);
Value(const Vector<Vector<real_t> > &);
Value(const Vector<Matrix<real_t> > &);
Value(const Matrix<Matrix<real_t> > &);
Value(const complex_t &);
Value(const Vector<complex_t> &);
Value(const Matrix<complex_t> &);
Value(const Vector<Vector<complex_t> > &);
Value(const Vector<Matrix<complex_t> > &);
Value(const Matrix<Matrix<complex_t> > &);
```

They correspond to the data types that are supported by this encapsulation class. These constructors create in memory a copy of the given value.

There are two templated functions to check the data types:

```
template<typename T> static bool checkTypeInList(const T &);
template<typename T> static bool isTypeInList(const T&);
template<typename T> void checkType(const T&) const;
```

The two first ones checks if the type T is in the list of allowed types and the second one compares the type T to the type of the current Value. To get the actual value, you have to use the templated member function:

```
template<class T> T& value() const;
```

Note that you have to explicit the template argument T. This function checks the value type and recasts the void pointer.

Besides, there are accessors/modifiers member functions:

```
ValueType valueType() const{return type_;}
StructType structType() const{return struct_;}
bool conjugate()const {return conjugate_;}
bool& conjugate() {return conjugate_;}
void conjugate(bool v) const {conjugate_=v;}
bool transpose()const {return transpose_;}
bool& transpose() {return transpose_;}
void transpose(bool v) const {transpose_=v;}
```

In order to conjugate or transpose (or both) a value in an expression, this class provides the functions:


```
Value& conj(Value &);
Value& trans(Value &);
Value& adj(Value &);
```

Be care, these functions do not modify the value but just set the *conjugate_*/*transpose_* flag to its opposite.

Finally, there are some printing functions:

```
static void Value::printValueTypeRTINames(std::ostream&);
void Value::print(std::ostream &) const;
friend std::ostream& operator<<(std::ostream&,const Value &);
```

Here, we give a small example to illustrate how the `Value` class works:

```
Value::printValueTypeRTINames(cout);    //print on stdout the RTI names of value
types
real_t s=1.;
Value Vs(s);                          //create Value object
cout<<"real Value Vs(s) :"<<Vs;        //print Value
real_t r =Vs.value<real_t>();          //get as a real
Vector<complex_t> vc(3,complex_t(0,1)); //a vector of complex values
Value Vvc(vc);
cout<<"complex vector Value Vvc(vc) :"<<Vvc; //print Value
Vector<complex_t> wc=Vvc.value<Vector<complex_t>> >(); //get as a complex vector
```

```
library : operator
header : Value.hpp
implementation : Value.cpp
unitary tests : test_operator.cpp
header dependences : config.h, utils.h
```

3.19 The KdTree class

The `KdTree` class implements the well known kdtree which is a binary tree where objects are separated using component by component separation. Usually objects are points but the following implementation is templated to deal with any class `T` that have components that can be separated. More precisely, the `T` class has to provide the following:

```
T::SepValueType;
T::operator()(int);
T::dim();
int Tcompare(const T& t, int c, const T::SepValueType& s);
//return -1 (t_c<s), 1 (t_c>s) or 0 (t_c=s)
int maxSeparator(const T& t1, const T& t2, int &c, T::SepValueType& s);
// find the component index c where separation is maximal, update s as the middle
value and return the comparison of t1 and t2 (1,-1,0)
```

Such tree may find the nearest object of a given object in a fast way ($n \log n$ in the most cases and n in worst cases).

The `KdTree` class is based on the `KdNode` class which is in fact the main class, `KdTree` handling the root node. A `KdNode` is either a separation line (component that separates two points and value of separation) or a terminal leaf containing a separated object.

KdNode class

```
template <class T>
class KdNode
{
private :
    typedef typename T::SepValueType SvType;
    KdNode * parent_;      //!< pointer to parent node
    KdNode * left_;        //!< pointer to left node
    KdNode * right_;       //!< pointer to right node
    const T * obj_;        //!< object pointer linked to node
    int separator_;        //!< separation component
    SvType sv_;            //!< separation value
}
```

It has two basic constructors, a destructor and a node insertion function

```
KdNode();
KdNode(const T *O,int s, KdNode * p);
~KdNode();
void insert(const T *);
```

The `insert` function is the fundamental function used in building of kdtree : it travels in tree up to find a free box containing the object to be inserted; if no free box is found, the last non empty box containing object to be inserted and an other object is split in two boxes. The tree obtained by this method has not an optimal balancing !

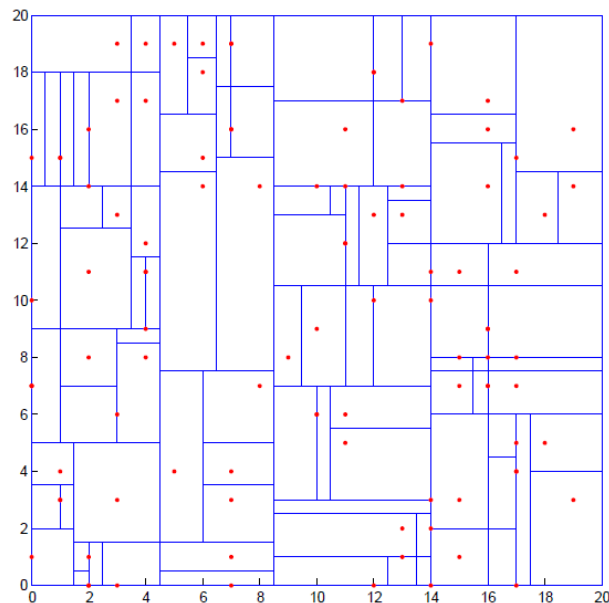


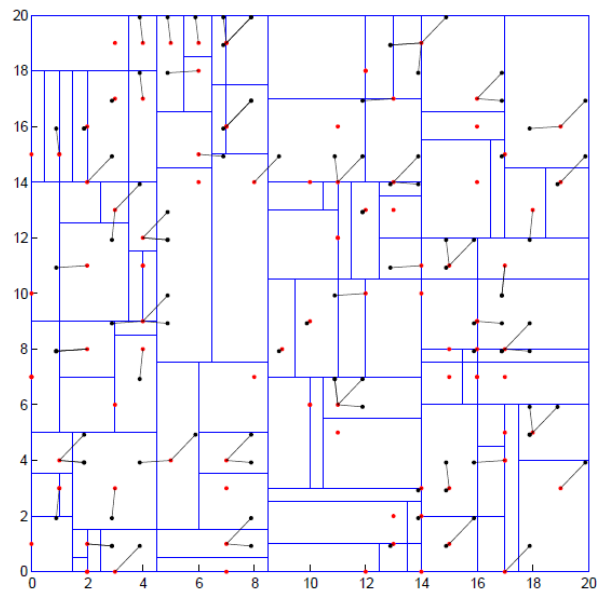
Figure 3.2: example of separation of 2d points

The class provides some utilities

```
Number& depth(Number &) const;      //!< return node depth
void print(std::ostream&, std::string &) const;
void printBoxes(std::ostream &os, Box<SvType> b) const;
template <class S>
friend std::ostream& operator<<(std::ostream& os, const KdNode<S> & node);
```

and the important function to search the nearest object of a given object:

```
void searchNearest(const T *, const T *&, SvType &);
```



KdTree class

As already mentioned **KdTree** handles the root node of tree and mainly encapsulated the **KdNode** functions

```
template <class T>
class KdTree
{
private :
    typedef typename T::SepValueType SvType;    //separator value type
    KdNode<T>* root;                            //rootnode of the tree
    Dimen dimT;                                //object dimension
}
```

```
KdTree() {root=new KdNode<T>();dimT=0;}
KdTree(const std::vector<T>&);
~KdTree() {delete root;};
bool isVoid() const;
Dimen dim() const;
void insert(const T&);
void insert(const std::vector<T>&);
Number depth() const;
const T* searchNearest(const T &) const;
void print(std::ostream& out) const;
template <typename R>
    void printBoxes(std::ostream &os, const Box<R>& rb);
template <typename R>
    void printBoxes(const std::string& file, const Box<R>& rb);
template <class S>
    friend std::ostream& operator<<(std::ostream& os, const KdTree<S>& node);
```

library :	utils
header :	KdTree.hpp
implementation :	KdTree.hpp
unitary tests :	test_KdTree.cpp
header dependences :	space.h, config.h, utils.h

3.20 The polynomials classes

In order to deal with general polynomial spaces (using formal polynomials), XLiFE++ provides the following template classes

- **MonomialT** to deal with monomials of the form $x^i y^j z^k$
- **PolynomialT** to deal with polynomials as combination of monomials, say

$$p(x, y, z) = \sum_{i,j,k \geq 0} \alpha_{ijk} x^i y^j z^k$$

- **PolynomialBasisT** to deal with set of polynomials, may be a poynomials basis:

$$P[x, y, z] = \text{span} \{P_\ell, \ell = 1, L\}$$

- **PolynomialsBasisT** to deal with set of vector polynomials, may be a polynomial basis:

$$\mathbf{P}[x, y, z] = \text{span} \left\{ \mathbf{P}_\ell = \begin{pmatrix} P_\ell^x \\ P_\ell^y \\ P_\ell^z \end{pmatrix}, \ell = 1, L \right\}$$

- **PolyNodeT** to represent a polynomial as tree, useful to evaluate polynomial in a best way (Horner algorithm)

These classes can manage one, two or three variables and are templated by the type of the variable (K) which is of `real_t` type by default.

Note that there is no specific class to deal with vector polynomial, they are described as `std::vector<PolynomialT>`.

For users, some useful aliases are available :

```
typedef MonomialT<real_t> Monomial;
typedef PolynomialT<real_t> Polynomial;
typedef PolynomialBasisT<real_t> PolynomialBasis;
typedef PolynomialsBasisT<real_t> PolynomialsBasis;
```

Some usual scalar polynomials spaces may be constructed in 1d, 2d or 3d:

$$\begin{aligned} Pn[x, y, z] &: \sum a_{ijk} x^i y^j z^k & i, j, k \geq 0, i + j + k \leq n \\ PHn[x, y, z] &: \sum a_{ijk} x^i y^j z^k & i, j, k \geq 0, i + j + k = n \\ Qn[x, y, z] &: \sum a_{ijk} x^i y^j z^k & i, j, k \geq 0, i, j, k \leq n \\ Qns[x, y, z] &: \sum a_{ijk} x^i y^j z^k & i, j, k \geq 0, i \leq nx, j \leq ny, k \leq nz \end{aligned}$$

Some particular vector polynomials spaces related to FE space are also provided (see **PolynomialsBasisT** class). They can be addressed using the enumeration:

```
enum PolynomialSpace {_Pk, _PHk, _Qk, _Qks, _Rk, _SHk, _Dk, _DQk, _DQ2k};
```

Main algebraic operations are supported and some formal derivative operations too. Polynomials may be evaluated at point (x, y, z) using operator $()$.

Besides, Polynomials may have a tree representation using `PolyNodeT` class. It may be useful when evaluating large polynomials.

3.20.1 The MonomialT class

This class deals with monomials of the form :

$$x_1^{a_1} x_2^{a_2} x_3^{a_3}$$

storing only the three power numbers a_1, a_2, a_3 :

```
template <typename K = real_t>
class MonomialT
{
public:
    dimen_t a1, a2, a3;
    ...
};
```

Thus, it allows only to deal with monomials of 3 variables of type K. Note that no coefficient is attached to a monomial.

`MonomialT` class provides only one fundamental constructor, overloaded operator $()$ to evaluate monomial at a point or a vector of points, product by a monomial, comparison operators and print facilities:

```
MonomialT(dimen_t p1=0, dimen_t p2=0, dimen_t p3=0)
K operator()(const K& x1, const K& x2 = K(1), const K& x3 = K(1)) const
K operator()(const std::vector<K>& x) const
MonomialT<K>& operator*=(const MonomialT<K>& m);
void swapVar(dimen_t v1, dimen_t v2, dimen_t v3);
string_t asString() const;
void print(std::ostream& out) const;
```

```
MonomialT<K> operator*(const MonomialT<K>& m1, const MonomialT<K>& m2);
friend std::ostream& operator<<(std::ostream& out, const MonomialT& m);
bool operator== (const MonomialT<K>& m1, const MonomialT<K>& m2);
bool operator!= (const MonomialT<K>& m1, const MonomialT<K>& m2);
bool operator < (const MonomialT<K>& m1, const MonomialT<K>& m2);
bool operator > (const MonomialT<K>& m1, const MonomialT<K>& m2);
```

3.20.2 The PolynomialT class

The `PolynomialT` class deals with polynomial up to 3 variables of value type K stored as a list of pairs of `MonomialT` object and a coefficient of type K:

```
template <typename K=real_t>
class PolynomialT
{
public:
    std::list<std::pair<MonomialT<K>, K>> monomials;
```

```

    real_t epsilon;
    mutable PolyNodeT<K> tree;
    ...
}

```

The member data `epsilon` is used to round to 0 the very small coefficients of the polynomial. By default its value is `100000*theEpsilon` (about 10^{-10} in double precision). The member data `tree` may store a tree representation (`PolyNode` object) of the polynomial, that is useful to evaluate it at a point in a faster and more stable way (generalized Horner algorithm).

The class provides simple constructors from monomials and a copy constructor:

```

PolynomialT();
PolynomialT(const MonomialT<K>&, const K& =K(1));
PolynomialT(const MonomialT<K>&, const K&, const MonomialT<K>&, const K&);
PolynomialT(const PolynomialT<K>&);
PolynomialT<K>& operator=(const PolynomialT<K>& p);

```

There are some useful tools:

```

void push_back(const K&, const MonomialT<K>&); //add a monomial part
(const_) iterator begin() (const) ;
(const_) iterator end() (const) ;
dimen_t degree() const;
void clean(real_t asZero); //remove "zero" part
void clean();
void swapVar(dimen_t v1, dimen_t v2, dimen_t v3);
static PolynomialT<K> zero();
bool isZero() const;

```

To evaluate the polynomial two methods are available. One evaluating the polynomial as a linear combination of monomials, the other one using a generalized Horner algorithm based on a tree representation of the polynomial:

```

void buildTree() const;
K eval(const K& x1, const K& x2 = K(1), const K& x3 = K(1)) const ;
K evalTree(const K& x1, const K& x2 = K(1), const K& x3 = K(1)) const ;
K operator() (const K& x1, const K& x2 = K(1), const K& x3 = K(1)) const ;
K operator() (const std::vector<K>& x) const ;

```

If tree representation is built, the `eval` functions used always tree representation.

Main algebraic operations are available, either as internal or external functions to the class:

```

PolynomialT<K>& operator *=(const MonomialT<K>&);
PolynomialT<K>& operator *=(const K&);
PolynomialT<K>& operator /=(const K& k);
PolynomialT<K>& operator +=(const MonomialT<K>&);
PolynomialT<K>& operator -=(const MonomialT<K>&);
PolynomialT<K>& operator +=(const PolynomialT<K>&);
PolynomialT<K>& operator -=(const PolynomialT<K>&);
PolynomialT<K>& operator *=(const PolynomialT<K>&);

//extern operation
template <typename K>
PolynomialT<K> operator *(const PolynomialT<K>&, const MonomialT<K>&);
PolynomialT<K> operator *(const MonomialT<K>&, const PolynomialT<K>&);
PolynomialT<K> operator *(const MonomialT<K>&, const K&);
PolynomialT<K> operator *(const K&, const MonomialT<K>&);
PolynomialT<K> operator /(const MonomialT<K>&, const K&);

```

```

PolynomialT<K> operator *(const PolynomialT<K>&, const K&);
PolynomialT<K> operator *(const K&, const PolynomialT<K>&);
PolynomialT<K> operator /(const PolynomialT<K>&, const K&);
PolynomialT<K> operator -(const PolynomialT<K>&);
PolynomialT<K> operator +(const PolynomialT<K>&, const PolynomialT<K>&);
PolynomialT<K> operator -(const PolynomialT<K>&, const PolynomialT<K>&);
PolynomialT<K> operator *(const PolynomialT<K>&, const PolynomialT<K>&);

```

Using the operator (.), polynomials may be chained (say $q(p1(x, y, z), p2(x, y, z), p3(x, y, z))$) :

```

template <typename K>
PolynomialT<K> operator () (const PolynomialT<K>& px,
                           const PolynomialT<K>& py=PolynomialT<K>(),
                           const PolynomialT<K>& pz=PolynomialT<K>()) const;
PolynomialT<K>& replace (VariableName, const PolynomialT<K>&);

```

It is also possible to get some derivatives and integrals of polynomials:

```

template <typename K>
PolynomialT<K> dx(const MonomialT<K>&);
PolynomialT<K> dy(const MonomialT<K>&);
PolynomialT<K> dz(const MonomialT<K>&);
vector<PolynomialT<K>> grad(const MonomialT<K>&, dimen_t =3);
vector<PolynomialT<K>> curl(const MonomialT<K>&, dimen_t =3);
PolynomialT<K> derivative(VariableName, const MonomialT<K>&);
PolynomialT<K> integral(VariableName, const MonomialT<K>&);

PolynomialT<K> dx(const PolynomialT<K>&);
PolynomialT<K> dy(const PolynomialT<K>&);
PolynomialT<K> dz(const PolynomialT<K>&);
vector<PolynomialT<K>> grad(const PolynomialT<K>&, dimen_t =3);
vector<PolynomialT<K>> curl(const PolynomialT<K>&, dimen_t =3);
PolynomialT<K> derivative(VariableName, const PolynomialT<K>&);
PolynomialT<K> integral(VariableName, const PolynomialT<K>&);

vector<PolynomialT<K>> dx(const vector<PolynomialT<K>>&);
vector<PolynomialT<K>> dy(const vector<PolynomialT<K>>&);
vector<PolynomialT<K>> dz(const vector<PolynomialT<K>>&);
vector<PolynomialT<K>> curl(const vector<PolynomialT<K>>&);
PolynomialT<K> div(const vector<PolynomialT<K>>&);
PolynomialT<K> dot(const vector<PolynomialT<K>>&, const vector<K>&);
vector<PolynomialT<K>> derivative(VariableName, const vector<PolynomialT<K>>&);
vector<PolynomialT<K>> integral(VariableName, const vector<PolynomialT<K>>&);

```

Finally, some print facilities are provided:

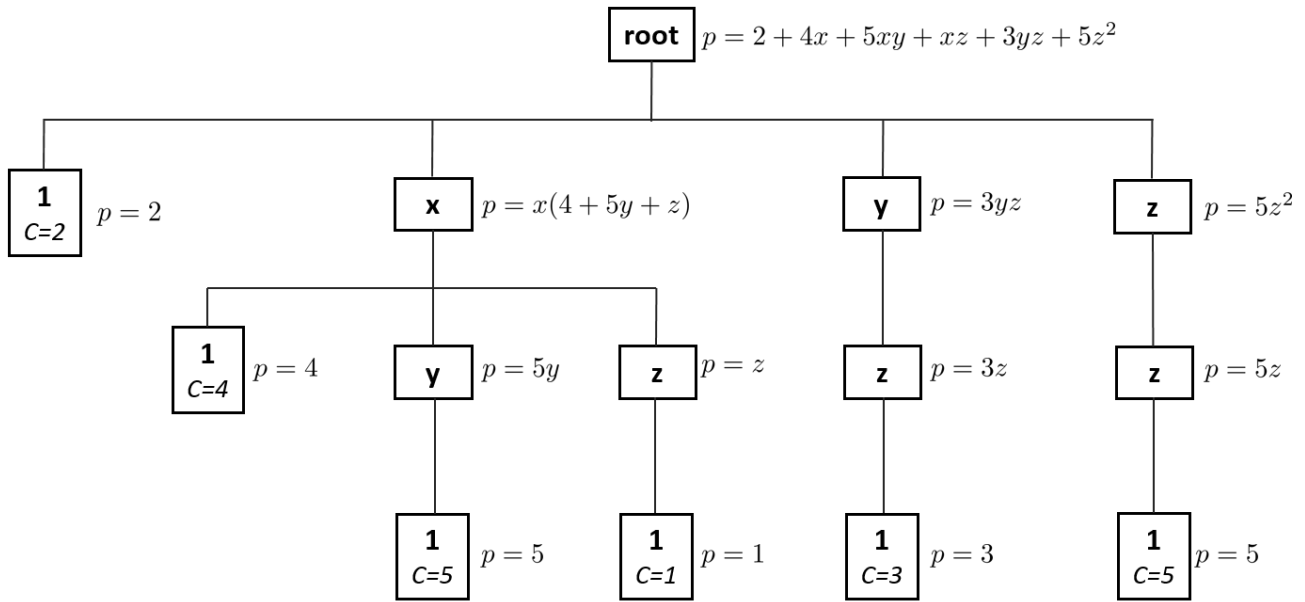
```

string_t asString() const;
void print(std::ostream&) const;
friend std::ostream& operator<<(std::ostream&, const PolynomialT&)

```

3.20.3 The PolyNodeT class

In order to be computed in a faster and a more stable way, a polynomial may be represented as a tree and evaluated by traveling this tree. In this tree representation, each node represents one of the factor 1, x_1 , x_2 or x_3 . When it is a leaf, the coefficient of the corresponding monomial is stored. For instance the polynomial $2 + 4x + 5xy + xz + 3yz + 5z^2$ has the following tree :



Horizontal lines represents addition while vertical lines represent product. Note that the tree representation is not unique.

The `PolyNodeT` class manages for each node a factor type and a coefficient, and some pointers to travel the tree:

```

template <typename K=real_t>
class PolyNodeT
{
public:
    dimen_t type;           // 0: 1, 1: x1, 2:x2, 3:x3
    K coef;
    PolyNodeT<K>* parent, * child, * right;
    ...
};
  
```

It offers only a few member functions to build the polynomial tree:

```

PolyNodeT(dimen_t =0, PolyNodeT<K>* =0, const K& =K(0));
~PolyNodeT();
void clear();
void insert(const MonomialT<K>&, const K&);
bool rootNode() const;
bool isZero() const {return (coef==K(0) && child==0);}
void print(std::ostream&) const;
void printTree(std::ostream&, std::string&) const;
string_t varString() const;
  
```

The most important function is the operator `()`:

```

K operator()(const K& x, const K& y=K(1), const K& z=K(1)) const;
  
```

that evaluates the polynomial at a point (x, y, z) .

3.20.4 The PolynomialBasisT class

A space of polynomials (of finite dimension p) may be described by one of its basis, say a set of p polynomials. The `PolynomialBasisT` class manages this basis as a list of `PolynomialT` objects. In fact it inherits from `std::list<PolynomialT<K>>`:

```

template <typename K=real_t>
class PolynomialBasisT : public std::list<PolynomialT<K>>
{public:
    dimen_t dim;           // number of variables in Polynomials
    string_t name;         // basis name
    ...
};

```

There exists some basic constructors from monomials and a general constructor for rather standard polynomial spaces identified by the enumerator `PolynomialSpace`:

```

enum PolynomialSpace {_Pk, _PHk, _Qk, _Qks, _Rk, _SHk, _Dk, _DQk, _DQ2k};

PolynomialBasisT(dimen_t =0, const string_t& = "");
PolynomialBasisT(dimen_t, const MonomialT<K>&, const string_t& na="");
PolynomialBasisT(dimen_t, const MonomialT<K>&, const MonomialT<K>&,
                  const string_t& = "");
PolynomialBasisT(PolynomialSpace, dimen_t, dimen_t, dimen_t =0, dimen_t =0);
void buildPk(dimen_t);
void buildPHk(dimen_t);
void buildQk(dimen_t);
void buildQks(dimen_t, dimen_t =0, dimen_t=0);
void buildTree();

```

The (scalar) polynomial spaces that can be constructed are (n number of variables, α multi-index):

polynomial space	dimension	XLiFE++
$P_k = \left\{ \sum_{ a \leq k} a_\alpha x^\alpha \right\}$	$C_k^{n+k} = \frac{(n+k)!}{k! n!}$	_Pk
$\tilde{P}_k = \left\{ \sum_{ \alpha =k} a_\alpha x^\alpha \right\}$	$n = 2 : k + 1$ $n = 3 : \frac{1}{2}(k+2)(k+1)$	_PHk
$Q_k = \left\{ \sum_{\alpha_1 \leq k, \alpha_2 \leq k, \alpha_3 \leq k} a_\alpha x^\alpha \right\}$	$(k+1)^n$	_Qk
$\tilde{Q}_{lmn} = \left\{ \sum_{\alpha_1 \leq l, \alpha_2 \leq m, \alpha_3 \leq n} a_\alpha x^\alpha \right\}$	$(l+1)(m+1)(n+1)$	_Qks

Several functions are devoted to the management of the list:

```

void push_back(const PolynomialT<K>&);
void add(const MonomialT<K>&);
void add(const PolynomialT<K>&);
void add(const PolynomialBasisT&);
iterator begin();
iterator end();
const_iterator begin() const;
const_iterator end() const;
size_t size() const;
void resize(size_t n);
void clean();
void clean(real_t asZero);
dimen_t degree() const;
void swapVar(dimen_t v1, dimen_t v2, dimen_t v3);

```




The `PolynomialsBasisT` class is intended to deal with polynomial basis but there is no tool to check independance of polynomials!

It is also possible to evaluate all the polynomials at a point using different ways:

```
std::vector<K>& eval(std::vector<K>& res, const K& x1, const K& x2 = K(1),
                  const K& x3 = K(1)) const;
std::vector<K> eval(const K& x1, const K& x2 = K(1), const K& x3 = K(1)) const;
std::vector<K>& evalTree(std::vector<K>& res, const K& x1, const K& x2 = K(1),
                      const K& x3 = K(1)) const;
std::vector<K> evalTree(const K& x1, const K& x2 = K(1),
                      const K& x3 = K(1)) const;
std::vector<K> operator()(const K& x1, const K& x2 = K(1),
                      const K& x3 = K(1)) const;
std::vector<K>& operator()(std::vector<K>& res, const K& x1, const K& x2 = K(1),
                      const K& x3 = K(1)) const;
```

As usual, there are some print facilities:

```
void print(std::ostream&) const;
friend std::ostream& operator<<(std::ostream&, const PolynomialBasisT<K>& );
```

3.20.5 The PolynomialBasisT class

In finite element approximation, some vector polynomial spaces are required. This is the role of the `PolynomialBasisT` class. There is no class to deal with vector polynomial, they are represented by some `vector<PolynomialT>`. As a consequence, the `PolynomialBasisT` class inherits from `list<std::vector<PolynomialT<K>>>`:

```
template <typename K=real_t>
class PolynomialBasisT : public std::list<std::vector<PolynomialT<K> > >
{
public:
    dimen_t dimVar;           // dimension of Polynomials (number of variables)
    dimen_t dimVec;           // dimension of vectors of Polynomials
    string_t name;            // basis name
    ...
};
```

Several constructors are available. Some construct product spaces from `PolynomialBasisT` objects. There is a general constructor allowing to get particular vector polynomial spaces enumerated in `PolynomialSpace`:

```
PolynomialBasisT(dimen_t =0, dimen_t =0, const string_t& = "");
PolynomialBasisT(const PolynomialBasisT<K>&, dimen_t, const string_t& na="");
PolynomialBasisT(const PolynomialBasisT<K>&, const PolynomialBasisT<K>&,
                const string_t& = "");
PolynomialBasisT(const PolynomialBasisT<K>&, const PolynomialBasisT<K>&,
                const PolynomialBasisT<K>&, const string_t& na="");
PolynomialBasisT(PolynomialSpace, dimen_t, dimen_t);
void buildSHk(dimen_t);
void buildRk(dimen_t);
void buildDk(dimen_t);
void buildDQk(dimen_t);
void buildDQ2k(dimen_t);
void buildTree();
```

The following FE spaces are available:

polynomial space	dimension	XLiFE++
$S_k = \{p \in (\tilde{P}_k)^n; x.p = 0\}$	$n \dim \tilde{P}_k - \dim \tilde{P}_{k+1}$	_Sk
$D_k = (P_{k-1})^n \oplus \tilde{P}_{k-1}x$	$n \dim P_{k-1} + \dim \tilde{P}_{k-1}$	_Dk
$R_k = (P_{k-1})^n \oplus S_k$	$n \dim P_{k-1} + \dim S_k$	_Rk
$DQk = Q_{k,k-1,k-1} \times Q_{k-1,k,k-1} \times Q_{k-1,k-1,k}$	$3k(k-1)^2$	_DQk
$DQ2k_{2d} = (P_k)^2 \oplus \text{span} \left\{ \text{curl } x_1^{k+1} x_2, \text{curl } x_2^{k+1} x_1 \right\}$	$2(\dim P_k + 1)$	_DQ2k

The operator $*$ allows to build tensor polynomials space, say if $P[x_1, x_2]$ and $Q[x_3]$ are two polynomials sets, $(P * Q)[x_1, x_2, x_3]$ is the set:

$$\{p_i(x_1, x_2)q_j(x_3), i = 1, m, j = 1, n\},$$

```
PolynomialBasisT<K> operator*(const PolynomialBasisT<K>& P,
                               const PolynomialBasisT<K>& Q)
```

Member functions are similar to member functions of the `PolynomialBasisT` class:

```
void push_back(const std::vector<PolynomialT<K> >&);
void add(const PolynomialT<K>&);
void add(const PolynomialT<K>&, const PolynomialT<K>&);
void add(const PolynomialT<K>&, const PolynomialT<K>&,
         const PolynomialT<K>&);
void add(const std::vector<PolynomialT<K> >&);
void add(const PolynomialsBasisT&);
iterator begin();
iterator end();
const_iterator begin() const;
const_iterator end() const;
size_t size() const;
void resize(size_t);
dimen_t degree() const;
void swapVar(dimen_t, dimen_t, dimen_t);
void clean();
void clean(real_t asZero);

//evaluation of basis
vector<vector<K> >& eval(vector<vector<K> >&, const K&, const K& = K(1),
                      const K& = K(1)) const;
vector<vector<K> > eval(const K&, const K& = K(1), const K& = K(1)) const;
vector<vector<K> >& evalTree(vector<vector<K> >&, const K&,
                          const K& = K(1), const K& = K(1)) const;
vector<vector<K> > evalTree(const K&, const K& = K(1), const K& = K(1)) const;
vector<vector<K> >& operator()(vector<vector<K> >&, const K&,
                           const K& = K(1), const K& = K(1)) const;

//print facilities
void print(std::ostream&) const;
friend std::ostream& operator<<(std::ostream&, const PolynomialsBasisT<K>&)
```

library :	utils
header :	polynomials.hpp
implementation :	polynomials.hpp
unitary tests :	test_polynomials.cpp
header dependences :	config.h, utils.h

4

The *geometry* library

The *geometry* library collects all the classes and functionalities related to meshes. The main class is the `Mesh` class which handles all stuff to describe a 1D, 2D or 3D mesh. This class provides also basic meshing tools (for simple geometries such as quadrangle, hexahedron, disk, sphere, ...), and import and export tools. The other important class is the `GeomDomain` class, which means geometrical domain, describing parts of mesh, in particular boundaries or parts of boundary. `GeomDomain` objects are involved in integral linear or bilinear forms as integration domain (see the *form* lib). The atomic object of any mesh or geometric domain is the geometric element (`GeomElement` class) which can handle either a full description of a geometric element (`MeshElement` class) or the description of a side of geometric element (say parent elements and side numbers). Note that a side element may also carry a `MeshElement` structure if required; by default it does not.

So the *geometry* library is organized as follows:

- `Mesh` class to deal with 1D, 2D or 3D mesh,
- `Geometry` class to handles global geometric data such as bounding box, parametrisations,
- `GeomDomain` class to deal with geometrical domains (whole computational domain, subdomains, ...),
- `GeomElement` class to deal with elements of a mesh or geometrical domains,
- `MeshElement` class handling full description of an element of mesh (nodes, numbering, ...),
- `GeomMapData` class storing geometric data (jacobian matrix of the map to reference element, differential element, normal vector, ...).

The scheme on figure 4.1 shows the organisation of the *geometry* library and its interactions with the *space* and *finiteElements* libraries.

4.1 Defining geometries

Whatever the geometry is, the common characteristics are a dimension, a shape, a name, a bounding box and a more convenient box for geometry inclusion than the previous one, and easier to define than the convex hull, the so-called "minimal box". This is the reason why we define 3 classes : `BoundingBox`, `MinimalBox` and `Geometry`.

4.1.1 BoundingBox and MinimalBox classes

The bounding box is the smallest box whose faces are parallel to axis containing the geometry. As a segment in 1D, a rectangle in 2D or a parallelepiped in 3D, it can be defined from 2, 3 or 4 points, as illustrated in the following figure :

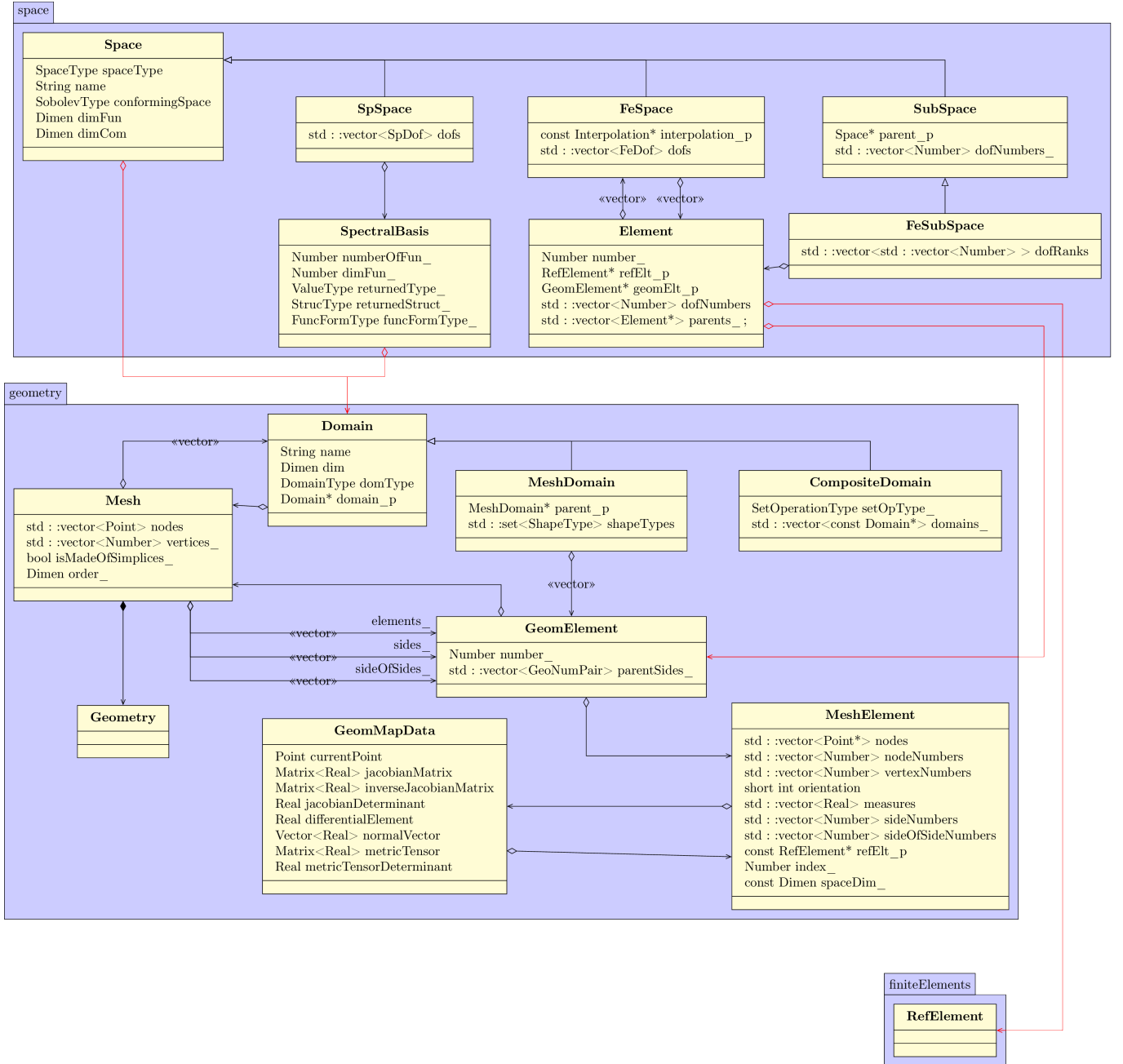


Figure 4.1: *geometry* library organisation

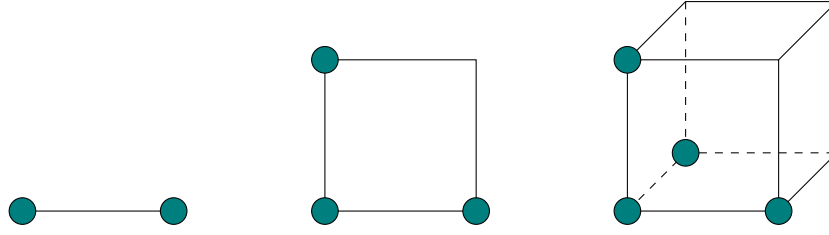


Figure 4.2: Definition of a box from points

For example, if we define a segment whose boundaries are the points $A(0,0,-1)$ and $B(1,2,3)$, the bounding box is $[0;1] \times [0;2] \times [-1;3]$. The `BoundingBox` class is then defined as a vector of pairs of real numbers :

```
typedef std::pair<Real, Real> RealPair;
class BoundingBox
{
private :
    std::vector<RealPair> bounds_; ///< Bounding values of the box
                                [xmin, xmax] x [ymin, ymax] x [zmin, zmax]
```

It offers a various list of constructors, from set of real pairs or set of `Point` :

```
BoundingBox() {}
BoundingBox(const std::vector<RealPair>&);
BoundingBox(Real, Real, Real, Real, Real, Real);
BoundingBox(Real, Real, Real, Real);
BoundingBox(Real, Real);
BoundingBox(const Point&, const Point&, const Point&, const Point&);
BoundingBox(const Point&, const Point&, const Point&);
BoundingBox(const Point&, const Point&);
BoundingBox(const std::vector<Point>& vp);
```

There is also :

- a various list of accessors :

```
std::vector<RealPair> bounds() const; // return bounds in all direction
RealPair bounds(Dimen i) const; // return bounds in direction i (1 to dim)
RealPair xbounds() const; // return bounds in first direction
RealPair ybounds() const; // return bounds in second direction
RealPair zbounds() const; // return bounds in third direction
Point minPoint() const; // return the min point (left, bottom, front)
Point maxPoint() const; // return the max point (right, top, back)
```

- print facilities :

```
void print(std::ostream&) const; // print BoundingBox
String asString() const; // format as string : [a,b]x[c,d]x[e,f]

std::ostream& operator<<(std::ostream&, const BoundingBox&); //output box
std::ostream& operator<<(std::ostream&, const RealPair&); //output pair of reals
```

- some utility functions, such as merging 2 bounding boxes or giving its dimension :

```
Dimen dim() const; // dimension of the bounding box
BoundingBox& operator +=(const BoundingBox&); // merge a bounding box
```

As the segment example shows, the dimension of the geometry and the dimension of the bounding box are independent. So if we want to check if a geometry is inside another geometry, this is not the right box to use, if we want to be independent of the true shape of the geometry. That is why the `MinimalBox` class is defined :

```
class MinimalBox
{
    protected :
        std::vector<Point> bounds_;
```

You can notice that it is `Point` that are stored, and only the 2, 3 or 4 points necessary to define the box, as shown in Figure 4.2

The `MinimalBox` class offers quite the same constructors as the `BoundingBox` class :

```
MinimalBox() {}
MinimalBox(const std::vector<Point>&);
MinimalBox(const std::vector<RealPair>&);
MinimalBox(Real, Real, Real, Real, Real, Real);
MinimalBox(Real, Real, Real, Real);
MinimalBox(Real, Real);
MinimalBox(const Point&, const Point&, const Point&, const Point&);
MinimalBox(const Point&, const Point&, const Point&);
MinimalBox(const Point&, const Point&);
```

It also offers :

- some accessors :

```
std::vector<Point> bounds() const;
Point boundPt(Dimen i) const; // return bounds in direction i (1 to dim)
Point minPoint() const;
Point maxPoint() const;
```

- print facilities :

```
void print(std::ostream&) const;
String asString() const;

std::ostream& operator<<(std::ostream&, const MinimalBox&); // output MinimalBox
```

- some utility functions, the same as in `BoundingBox`, plus a function to return all vertices of the box, not only the points stored in `bounds_`.

```
Dimen dim() const;
MinimalBox& operator +=(const MinimalBox&);
std::vector<Point> vertices() const;
```



A minimal box of a segment will always be a segment. A minimal box of a 2D plane geometry will always be a rectangle.

Now that both boxes classes are clarified, we can now look at the main class : `Geometry`.

4.1.2 The Geometry class

A geometry can be of 3 natures : canonical geometries, such as `Rectangle` or `Ball`, so-called "loop" geometries defined by its set of geometrical boundaries, and so-called "composite" geometries defined by union of elementary geometries and holes. The definition of the `Geometry` class takes into account these different natures :

```

enum ShapeType
{
    _noshape=0,
    _point ,
    _segment , segment=_segment ,
    _triangle , triangle=_triangle ,
    _quadrangle , quadrangle=_quadrangle ,
    _tetrahedron , tetrahedron=_tetrahedron ,
    _hexahedron , hexahedron=_hexahedron ,
    _prism , prism=_prism ,
    _pyramid , pyramid=_pyramid ,
    _ellArc , _circArc ,
    _polygon , _parallelogram , _rectangle , _square , _ellipse , _disk , _ellipsoidPart ,
        _spherePart ,
    _setofelems , _trunkPart , _cylinderPart , _conePart ,
    _polyhedron , _parallelepiped , _cuboid , _cube , _ellipsoid , _ball , _trunk ,
    _revTrunk , _cylinder , _revCylinder , _cone , _revCone ,
    _composite , _loop , _extrusion
};

class Geometry
{
public:
    BoundingBox boundingBox;
    MinimalBox minimalBox;
    mutable bool force; // tag to force inclusion when geometry engines fails
    mutable bool crackable; // tag to define if the geometry should be cracked or
        not
protected:
    string_t domName_; // geometry name
    Dimen dim_; // physical space dimension
    ShapeType shape_; // geometrical shape
    std::vector<string_t> sideNames_; //names of sides (boundaries)
    std::vector<string_t> theNamesOfVariables_; // variable names (x,y,z) ,...
    string_t teXFilename_; // name of the file containing the
        tex code drawing the geometry
private:
    mutable std::map<number_t, Geometry*> components_; // for composite/loop
        geometry
    std::map<number_t, std::vector<number_t> > geometries_; // for composite
        geometry
    std::map<number_t, std::vector<number_t> > loops_; // for loop geometry
    mutable Transformation* extrusion_; // for extrusion geometry
    number_t layers_; // for extrusion geometry

```

name and sideNames_ are devoted to store domain names or side domain names.

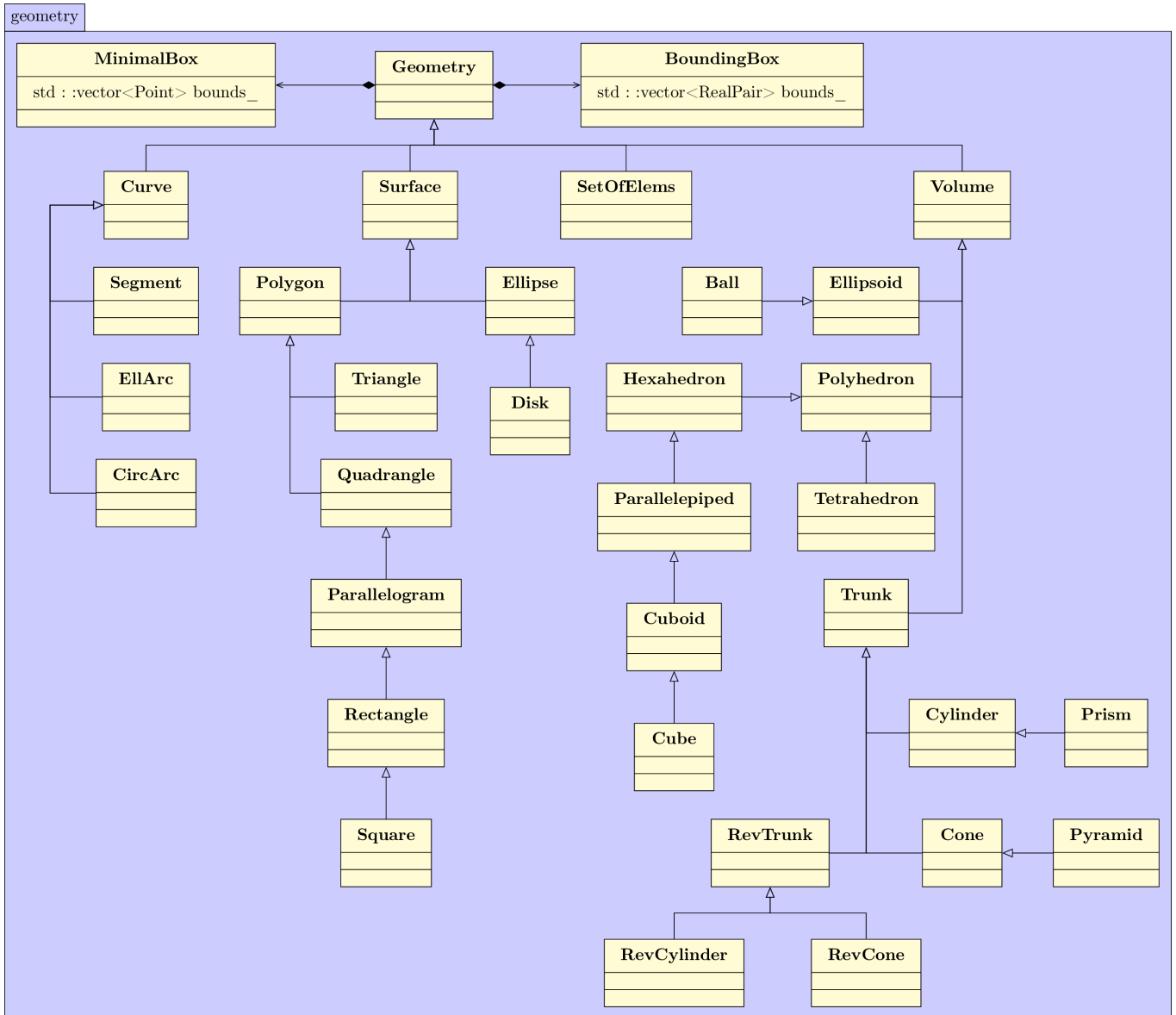


Figure 4.3: *Geometry* child classes

The *Geometry* is the base class for a large set of canonical geometries, so the management of "composite" and "loop" geometries implies 2 main vectors of *Geometry* pointers and a set of virtual functions to cast a *Geometry* into one of its child :

```

//! access to child Xxxxxxx object (const)
virtual const Xxxxxxx* xxxxxxxx() const { error("bad_geometry", name,
    words("shape",shape_), words("shape",-yyyyyyy)); return 0; }
//! access to child Xxxxxxx object
virtual Xxxxxxx* xxxxxxxx() { error("bad_geometry", name, words("shape",shape_),
    words("shape",-yyyyyyy)); return 0; }

```

xxxxxxx must be one of the following keyword : segment, ellArc, circArc, polygon, triangle, quadrangle, parallelogram, rectangle, square, ellipse, disk, polyhedron, tetrahedron, hexahedron, parallelepiped, cuboid, cube, ellipsoid, ball, revTrunk, trunk, cylinder, revCylinder, prism, cone, revCone, pyramid, setOfElems
 yyyyyyy must be the corresponding *ShapeType* item.

The *Geometry* class offers constructors from *BoundingBox*, dimension or both :

```

Geometry(): force(false), crackable(false), domName(""), dim_(0),
    shape_(noshape), extrusion_(0), layers_(0) { sideNames_.resize(0); } // void
    constructor
Geometry(const Geometry& g); // copy constructor
// basic constructor from bounding box
Geometry(const BoundingBox&, const string_t& na = "?", ShapeType sht=noshape,
    const string_t& nx = "x", const string_t& ny = "y", const string_t& nz = "z");
Geometry(const BoundingBox&, dimen_t, const string_t& na = "?",
    ShapeType sht=noshape, const string_t& nx = "x", const string_t& ny = "y",
    const string_t& nz = "z");
Geometry(dimen_t, const string_t& na = "?", ShapeType sht=noshape,
    const string_t& nx = "x", const string_t& ny = "y", const string_t& nz = "z");
virtual ~Geometry() {}

```

It also offers :

- some accessors :

```

dimen_t dim() const;
ShapeType shape() const;
void shape(ShapeType sh);
string_t domName() const;
void domName(const string_t& nm);
string_t texFilename() const;
void texFilename(const string_t& fn);
const std::vector<string_t> sideNames() const;
std::vector<string_t> sideNames();
std::map<number_t, Geometry*> components();
const std::map<number_t, Geometry*> components() const;
const std::map<number_t, std::vector<number_t>> geometries() const;
std::map<number_t, std::vector<number_t>> loops() const;
number_t layers() const;
Transformation* extrusion();
const Transformation* extrusion() const;

```

- some print facilities :

```

void print(std::ostream&) const; //!< print Geometry
friend std::ostream& operator<<(std::ostream&, const Geometry&); //!< output
    Geometry

```

- some functions to compute or update boxes :

```

virtual void computeMB(); //! compute the minimal box
void updateBB(const std::vector<RealPair>& bb) { boundingBox = BoundingBox(bb); }

```

As the minimal box depends on the true shape of the geometry, it is a virtual function reimplemented in child classes.

- some virtual functions to get data from canonical geometries, such as the vector of points defining the geometry, or the parameters for the subdivision mesh algorithm :

```

virtual std::vector<const Point*> boundNodes() const;
virtual std::vector<Point*> nodes();
virtual std::vector<const Point*> nodes() const;
virtual std::vector<std::pair<ShapeType, std::vector<const Point*>>> curves()
    const;
virtual std::vector<std::pair<ShapeType, std::vector<const Point*>>> surfs() const;

```

- member functions to determine if a geometry is coplanar to another geometry or inside another geometry, and external functions or operators to define "composite", "loop" or "extrusions" geometries.

```

bool isCoplanar(const Geometry& g) const;
bool isInside(const Geometry&) const; //check if geometry is inside an other one

friend Geometry operator+(const Geometry&, const Geometry&); //union
Geometry& operator+=(const Geometry&);
friend Geometry operator-(const Geometry&, const Geometry&); //difference (hole)
Geometry& operator-=(const Geometry&);

// definition of a geometry 2D/3D from its boundary 1D/2D
friend Geometry surfaceFrom(const Geometry&, String name = "");
friend Geometry volumeFrom(const Geometry&, String name = "");
friend Geometry extrude(const Geometry& g, const Transformation& t, number_t
    layers, string_t domName, std::vector<string_t> sidenames);
friend Geometry extrude(const Geometry& g, const Transformation& t, number_t
    layers, std::vector<string_t> sidenames);

```

Operators + and - work as follows :

- if both arguments are canonical geometries (neither "composite" nor "loop"), it puts pointers to them in the `components_` attribute and if one geometry is inside the other, it puts it in the `geometries` attribute.
- If at least one of the argument is "composite" or "loop", it inserts the canonical geometry correctly, or merge the 4 maps.

`surfaceFrom` and `volumeFrom` allow to define surfaces and respectively volumes from their boundaries, so-called "loop" geometries. The mechanism is the same : taking a "composite" geometry as argument, it puts every component in the `borders_` attribute and defines the loop in the `loops_` attribute, so that we can define "composite" geometries of "loop" geometries, but it is for the moment forbidden.

4.1.3 Curve, Surface and Volume classes

The `Curve` class is the base class for every canonical 1D geometry. It has no attributes and only general constructors :

```

class Curve : public Geometry
{
public:
    Curve();
protected:
    void buildParam(const Parameter& p);
    void buildDefaultParam(ParameterKey key);
    std::set<ParameterKey> getParamsKeys();
public:
    Curve(const Curve& c) : Geometry(c) {} ///< copy constructor
    virtual Geometry* clone() const { return new Curve(*this); } ///< virtual copy
    constructor for Geometry
    virtual ~Curve() {} ///< virtual destructor
};

```

The `Surface` class is the base class for every canonical 2D geometry. Actually, it has no attributes and only constructors :

```

class Surface : public Geometry
{
public:
    ///< default constructor for 2D geometries
    Surface();

protected:

```

```

void buildParam(const Parameter& p);
void buildDefaultParam(ParameterKey key);
std::set<ParameterKey> getParamsKeys();

public:
    Surface(const Surface& s) : Geometry(s) {} ///< copy constructor
    virtual Geometry* clone() const { return new Surface(*this); } ///< virtual
        copy constructor for Geometry
    virtual Surface* cloneS() const { return new Surface(*this); } ///< virtual
        copy constructor for Surface
    virtual ~Surface() {} ///< virtual destructor

    virtual std::vector<Point> p() const
    { error("not_yet_implemented", "std::vector<Point> Surface::p() const"); return
        std::vector<Point>(); }
    virtual std::vector<number_t> n() const
    { error("not_yet_implemented", "std::vector<Point> Surface::n() const"); return
        std::vector<number_t>(); }
    virtual Point p(number_t i) const
    { error("not_yet_implemented", "Point Surface::p(Number i) const"); return
        Point(); }
    virtual number_t n(number_t i) const
    { error("not_yet_implemented", "Number Surface::n(Number i) const"); return 0; }
    virtual number_t& n(number_t i)
    { error("not_yet_implemented", "Number& Surface::n(Number i)"); return *new
        number_t(0); }
    virtual std::vector<real_t> h() const
    { error("not_yet_implemented", "std::vector<Real> Surface::h() const"); return
        std::vector<real_t>(); }
    virtual real_t h(number_t i) const
    { error("not_yet_implemented", "Number Surface::h(Number i) const"); return 0.; }
    virtual real_t& h(number_t i)
    { error("not_yet_implemented", "Number& Surface::h(Number i)"); return *new
        real_t(0); }

    bool isPolygon()
    {
        if (shape_ == _polygon || shape_ == _triangle || shape_ == _quadrangle ||
            shape_ == _parallelogram || shape_ == _rectangle ||
            shape_ == _square) { return true; }
        return false;
    }

    ///< format as string
    virtual string_t asString() const { error("shape_not_handled",
        words("shape", shape_)); return string_t(); }
};

```

The **Volume** class is the base class for every canonical 3D geometry. Actually, it has no attributes and only constructors :

```

class Volume : public Geometry
{
public:
    ///< default constructor for 3D geometries
    Volume();

protected:
    void buildParam(const Parameter& p);
    void buildDefaultParam(ParameterKey key);
    std::set<ParameterKey> getParamsKeys();

public:
    Volume(const Volume& v) : Geometry(v) {} ///< copy constructor

```

```

virtual Geometry* clone() const { return new Volume(*this); } ///< virtual copy
constructor for Geometry
virtual ~Volume() {} ///< virtual destructor

///< format as string
virtual string_t asString() const { error("shape_not_handled",
    words("shape",shape_)); return string_t(); }
};

```

4.1.4 Definition of canonical geometries

In this subsection, we will explain how to define canonical geometries. All constructors for these classes are writtent in the same way : with **Parameter** arguments, so that the user can use a key-value system.

The key-value system

Some keys are easy to determine : the center for rectangles, squares, ellipses, disks, cuboids, cubes, ellipsoids and balls, the radius for disks, balls, and cylinders, To do so, global keys are defined (type **Parameter**). Here is the list of available keys with authorized data types and classes using them:

key	authorized types	concerned classes
_angle1	single integer or real value	class Disk
_angle2	single integer or real value	class Disk
_apex	single integer or real value, or Point	classes Cone , Pyramid , RevCone
_apogee	single integer or real value, or Point	class EllArc
_basis	classes Polygon , Triangle , Quadrangle , Parallelogram , Rectangle , Square , Ellipse , Disk	classes Trunk , Cylinder , Prism , Cone , Pyramid
_center	single integer or real value, or Point	classes EllArc , CircArc , Rectangle , Square , Ellipse , Disk , Cuboid , Cube , Ellipsoid , Ball , RevCone
_center1	single integer or real value, or Point	classes Trunk , Cylinder , Cone , RevTrunk , RevCylinder
_center2	single integer or real value, or Point	classes Trunk , Cylinder , RevTrunk , RevCylinder
_direction	std::vector of real values or Point	classes Cylinder , Prism
_domain_name	single string	every class
_end_shape	enum GeometricEndShape	class RevCone
_end1_shape	enum GeometricEndShape	classes RevTrunk , RecCylinder
_end2_shape	enum GeometricEndShape	classes RevTrunk , RecCylinder
_end_distance	single unsigned integer or real positive value	class RevCone
_end1_distance	single unsigned integer or real positive value	classes RevTrunk , RecCylinder
_end2_distance	single unsigned integer or real positive value	classes RevTrunk , RecCylinder
_faces	vector of Polygon	class Polyhedron
_hsteps	single real value, std::vector of real values or Reals	every class but Polyhedron
_length	single unsigned integer or real positive value	classes Square , Cube
_nbocants	single unsigned integer value	classes Cube , Ellipsoid , Ball
_nbsubdomains	single unsigned integer value	classes RevTrunk , RevCylinder , RevCone
_nnodes	single unsigned integer value, std::vector of integer values or Numbers	every class but Polyhedron
_origin	single integer or real value, or Point	classes Rectangle , Square , Cuboid , Cube , Trunk , Cylinder , Prism

key	authorized types	concerned classes
_radius	single unsigned integer or real positive value	classes Disk , Ball , RevCylinder , RevCone
_radius1	single unsigned integer or real positive value	class RevTrunk
_radius2	single unsigned integer or real positive value	class RevTrunk
_scale	single unsigned integer or real positive value	class Trunk
_side_names	single string or std::vector of strings	every class but Polyhedron
_type	single unsigned integer value	classes Ellipse , Disk , Ellipsoid , Ball , RevTrunk , RevCylinder , RevCone
_varnames	single string or std::vector of strings	every class
_vertices	vector of Point	class Polygon
_v1	single integer or real value, or Point	every class, but Polygon , Polyhedron , RevTrunk , RevCylinder , RevCone
_v2	single integer or real value, or Point	every class, but Polygon , Polyhedron , RevTrunk , RevCylinder , RevCone
_v3	single integer or real value, or Point	classes Triangle , Quadrangle , Tetrahedron , Hexahedron , Prism , Pyramid
_v4	single integer or real value, or Point	classes Quadrangle , Parallelogram , Rectangle , Square , Tetrahedron , Hexahedron , Parallelepiped , Cuboid , Cube , Pyramid
_v5	single integer or real value, or Point	classes Hexahedron , Parallelepiped , Cuboid , Cube
_v6	single integer or real value, or Point	class Hexahedron
_v7	single integer or real value, or Point	class Hexahedron
_v8	single integer or real value, or Point	class Hexahedron
_xlength	single unsigned integer or real positive value	classes Rectangle , Ellipse , Cuboid , Ellipsoid
_xmin	single integer or real value	classes Rectangle , Cuboid
_xmax	single integer or real value	classes Rectangle , Cuboid
_ylength	single unsigned integer or real positive value	classes Rectangle , Ellipse , Cuboid , Ellipsoid
_ymin	single integer or real value	classes Rectangle , Cuboid
_ymax	single integer or real value	classes Rectangle , Cuboid
_zlength	single unsigned integer or real positive value	class Cuboid
_zmin	single integer or real value	class Cuboid
_zmax	single integer or real value	class Cuboid

A constructor of a geometry will take several [Parameter](#) object and transfer the building work to the 3 main following routines:

```
// main function to build a geometry
void build(const std::vector<Parameter>& ps);
```

```

// build the list of available keys
std::set<ParameterKey> getParamsKeys();
// deal with one key and do the appropriate work
void buildParam(const Parameter& gp);
// deal with unused keys having default values and do the appropriate work
void buildDefaultParam(ParameterKey key);

```

For some geometries, there are some additional routines called by `build`.

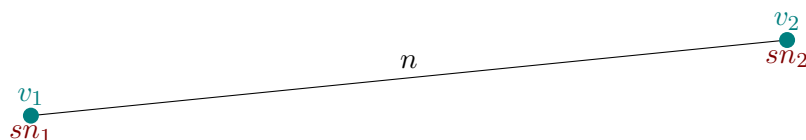
The `Segment` class

A segment is just a straight line between 2 points, with a number of nodes.

```

class Segment : public Curve
{
private:
    Point p1_, p2_;
    Number n_;
    std::vector<real_t> h_;

```



It offers constructors, taking from 2 to 5 `Parameter`. Authorized keys and what they do are:

attribute	key to use to define it	optional	default value
p1_, p2_	(_v1, _v2)	no	no
n_	_nnodes	yes	2
	or (_xmin, _xmax)		
n_	_nnodes	yes	2
h_	_hsteps	yes	no
domName_	_domain_name	yes	""
sideNames_	_side_names	yes	""

It also offers :

- some accessors
- implementation of inherited virtual functions concerning segments :

```

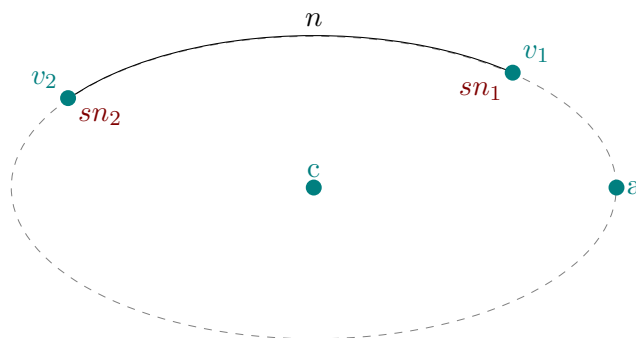
virtual void computeMB();
virtual std::vector<Point*> nodes();
virtual std::vector<const Point*> nodes() const;
virtual std::vector<std::pair<ShapeType, std::vector<const Point*> > > curves()
    const;

```

The `EllArc` class

To define an elliptic arc, you need 4 points : the center of the ellipse, a point on the main axis (apogee) and the bounds of the arc. When omitted, the apogee point is defined as the first bound of the arc. An elliptic arc must be smaller than a half-ellipse, to be defined correctly. This is a GMSH restriction.


```
class EllArc : public Curve
{
protected:
    Point c_, a_, p1_, p2_;
    Number n_;
    std::vector<real_t> h_;
```



It offers constructors, taking from 3 to 7 **Parameter**. Authorized keys and what they do are:

attribute	key to use to define it	optional	default value
c_, a_, p1_, p2_	(_center, _apogee, _v1, _v2) or (_center, _v1, _v2)	no	no
n_	_nnodes	yes	2
h_	_hsteps	yes	no
domName_	_domain_name	yes	""
sideNames_	_side_names	yes	""

It also offers :

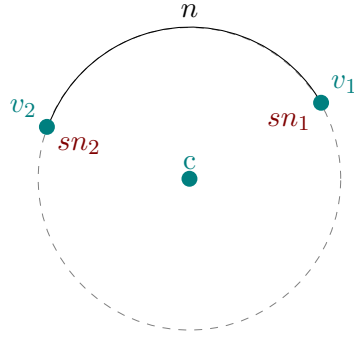
- some accessors
- implementation of inherited virtual functions concerning elliptic arcs :

```
virtual void computeMB();
virtual std::vector<Point*> nodes();
virtual std::vector<const Point*> nodes() const;
virtual std::vector<std::pair<ShapeType, std::vector<const Point*> > > curves()
    const;
```

The CircArc class

To define a circular arc, you need 3 points : the center of the circle and the bounds of the arc. A circular arc must be smaller than a half-circle, to be defined correctly. This is a GMSH restriction.

```
class CircArc : public Curve
{
protected:
    Point c_, p1_, p2_;
    number_t n_;
    std::vector<real_t> h_;
```



It offers constructors, taking from 3 to 6 [Parameter](#). Authorized keys and what they do are:

attribute	key to use to define it	optional	default value
c -, $p1$ -, $p2$ -	<code>_center</code> , <code>_v1</code> and <code>_v2</code>	no	no
n -	<code>_nnodes</code>	yes	2
h -	<code>_hsteps</code>	yes	no
$domName$ -	<code>_domain_name</code>	yes	""
$sideNames$ -	<code>_side_names</code>	yes	""

It also offers :

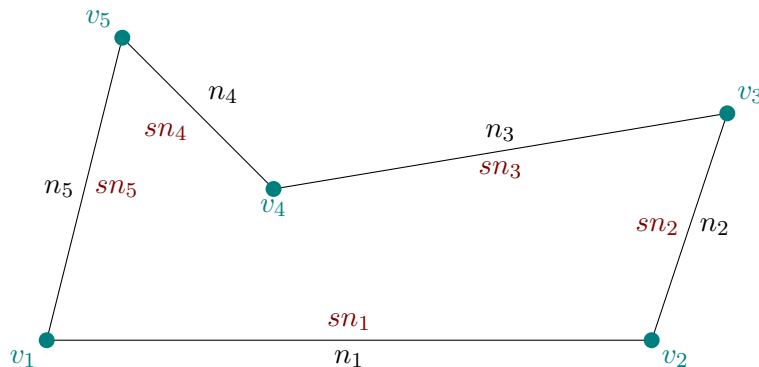
- some accessors
- implementation of inherited virtual functions concerning circular arcs :

```
virtual void computeMB();
virtual std::vector<Point*> nodes();
virtual std::vector<const Point*> nodes() const;
virtual std::vector<std::pair<ShapeType, std::vector<const Point*>>> curves()
    const;
```

The [Polygon](#) class

To define a polygon, you give the ordered list of vertices.

```
class Polygon : public Surface
{
protected:
    std::vector<Point> p_;
    std::vector<real_t> h_;
    std::vector<number_t> n_;
```



It offers constructors, taking from 1 to 4 [Parameter](#). Authorized keys and what they do are:

attribute	key to use to define it	optional	default value
p_	_vertices	no	no
n_	_nnodes	yes	2
h_	_hsteps	yes	no
domName_	_domain_name	yes	""
sideNames_	_side_names	yes	""

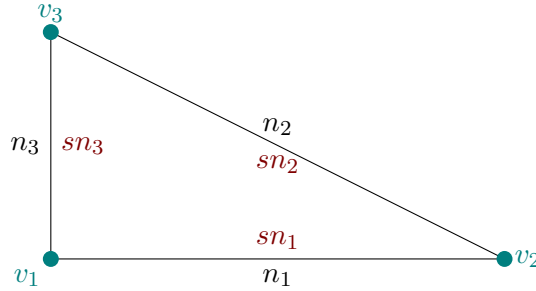
It also offers :

- some accessors
- implementation of inherited virtual functions concerning polygon :

```
virtual void computeMB();
virtual std::vector<const Point*> boundNodes() const;
virtual std::vector<Point*> nodes();
virtual std::vector<const Point*> nodes() const;
virtual std::vector<std::pair<ShapeType, std::vector<const Point*> > > curves()
    const;
virtual std::vector<std::pair<ShapeType, std::vector<const Point*> > > surfs() const;
```

The [Triangle](#) class

To define a triangle, you give the 3 vertices.



It offers constructors, taking from 3 to 6 [Parameter](#). Authorized keys and what they do are:

attribute	key to use to define it	optional	default value
p_	_v1, _v2 and _v3	no	no
n_	_nnodes	yes	2
h_	_hsteps	yes	no
domName_	_domain_name	yes	""
sideNames_	_side_names	yes	""

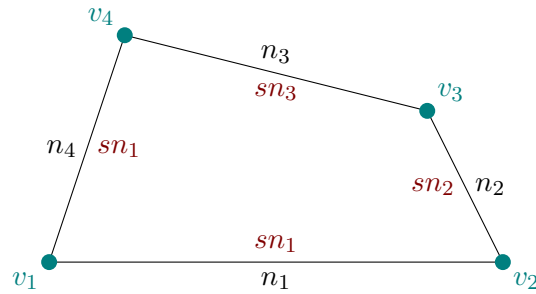
It also offers :

- some accessors
- implementation of inherited virtual functions concerning triangles :

```
virtual void computeMB();
virtual std::vector<const Point*> boundNodes() const;
virtual std::vector<Point*> nodes();
virtual std::vector<const Point*> nodes() const;
virtual std::vector<std::pair<ShapeType, std::vector<const Point*> > > curves()
    const;
virtual std::vector<std::pair<ShapeType, std::vector<const Point*> > > surfs() const;
```

The **Quadrangle** class

To define a quadrangle, you give the 4 vertices.



It offers constructors, taking from 4 to 7 **Parameter**. Authorized keys and what they do are:

attribute	key to use to define it	optional	default value
p_	_v1 , _v2 , _v3 and _v4	no	no
n_	_nnodes	yes	2
h_	_hsteps	yes	no
domName_	_domain_name	yes	""
sideNames_	_side_names	yes	""

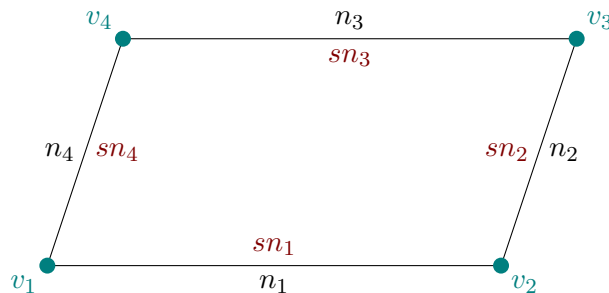
It also offers :

- some accessors
- implementation of inherited virtual functions concerning quadrangles :

```
virtual void computeMB();
virtual std::vector<const Point*> boundNodes() const;
virtual std::vector<Point*> nodes();
virtual std::vector<const Point*> nodes() const;
virtual std::vector<std::pair<ShapeType, std::vector<const Point*>>> curves()
    const;
virtual std::vector<std::pair<ShapeType, std::vector<const Point*>>> surfs() const;
```

The **Parallelogram** class

To define a parallelogram, you give 3 vertices. p_3 is indeed unnecessary. It can be calculated from the 3 other vertices.



It offers constructors, taking from 3 to 6 **Parameter**. Authorized keys and what they do are:

attribute	key to use to define it	optional	default value
p_	_v1, _v2 and _v4	no	no
n_	_nnodes	yes	2
h_	_hsteps	yes	no
domName_	_domain_name	yes	""
sideNames_	_side_names	yes	""

It also offers :

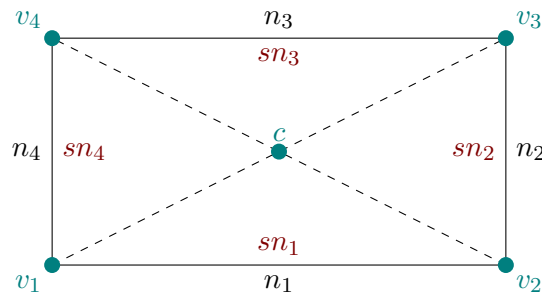
- some accessors
- implementation of inherited virtual functions concerning parallelograms :

```
virtual void computeMB();
virtual std::vector<const Point*> boundNodes() const;
virtual std::vector<Point*> nodes();
virtual std::vector<const Point*> nodes() const;
virtual std::vector<std::pair<ShapeType, std::vector<const Point*> > > curves()
    const;
virtual std::vector<std::pair<ShapeType, std::vector<const Point*> > > surfs() const;
```

The Rectangle class

To define a rectangle, you give 3 vertices, as for [Parallelogram](#). You can also give the center (or the origin) and the lengths, or you can gives bounds. To store temporarily these values, the [Rectangle](#) class has internal attributes.

```
class Rectangle : public Parallelogram
{
protected:
    Point center_, origin_;
    bool isCenter_, isOrigin_;
    real_t xlength_, ylength_;
private:
    real_t xmin_, xmax_, ymin_, ymax_;
    bool isBounds_;
```



It offers constructors, taking from 3 to 7 [Parameter](#). Authorized keys and what they do are:

attribute	key to use to define it	optional	default value
p_	(_v1, _v2, _v4) or (_center, _xlength, _ylength) or (_origin, _xlength, _ylength) or (_xmin, _xmax, _ymin, _ymax)	no	no
n_	_nnodes	yes	2
h_	_hsteps	yes	no
domName_	_domain_name	yes	""
sideNames_	_side_names	yes	""

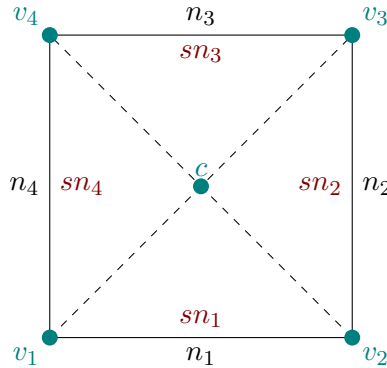
It also offers :

- some accessors
- implementation of inherited virtual functions concerning rectangles :

```
virtual void computeMB();
virtual std::vector<const Point*> boundNodes() const;
virtual std::vector<Point*> nodes();
virtual std::vector<const Point*> nodes() const;
virtual std::vector<std::pair<ShapeType, std::vector<const Point*>>> curves()
    const;
virtual std::vector<std::pair<ShapeType, std::vector<const Point*>>> surfs() const;
```

The Square class

To define a square, you give 3 vertices. You can also give the center (or the origin) and the length.



It offers constructors, taking from 2 to 6 **Parameter**. Authorized keys and what they do are:

attribute	key to use to define it	optional	default value
p_	(_v1, _v2, _v4) or (_center, _xlength, _ylength) or (_origin, _xlength, _ylength)	no	no
n_	_nnodes	yes	2
h_	_hsteps	yes	no
domName_	_domain_name	yes	""
sideNames_	_side_names	yes	""

It also offers :

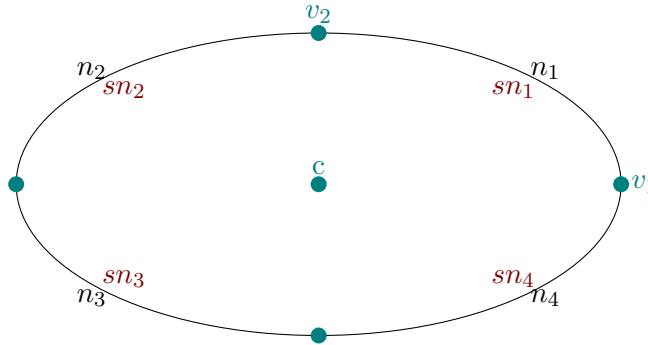
- some accessors

- implementation of inherited virtual functions concerning squares :

```
virtual void computeMB();
virtual std::vector<const Point*> boundNodes() const;
virtual std::vector<Point*> nodes() const;
virtual std::vector<const Point*> nodes() const;
virtual std::vector<std::pair<ShapeType, std::vector<const Point*>>> curves()
    const;
virtual std::vector<std::pair<ShapeType, std::vector<const Point*>>> surfs() const;
```

The Ellipse class

To define an elliptic surface, you have to define a center and 2 apogees (c , p_1 and p_2 in the following figure):



You can also define an ellipse from its center and the axis lengths, when axes are x-axis and y-axis. The main attributes are center_, p1_, p2_, p3_, p4_, those that will be used in meshing.

```
class Ellipse : public Surface
{
protected:
    Point center_, p1_, p2_, p3_, p4_;
    real_t xlength_, ylength_;
    bool isAxis_;
    number_t n1_, n2_, n3_, n4_;
    std::vector<real_t> h_;
    real_t thetamin_;
    real_t thetamax_;
    dimen_t type_;
```

It offers constructors, taking from 3 to 9 **Parameter**. Authorized keys and what they do are:

attribute	key to use to define it	optional	default value
center_, p1_, p2_, p3_, p4_	(_center, _v1 and _v2) or (_center, _xlength, _ylength)	no	no
n1_, n2_, n3_, n4_	_nnodes	yes	2
h_	_hsteps	yes	no
domName_	_domain_name	yes	""
sideNames_	_side_names	yes	""
thetamin_	_angle1	yes	0
thetamax_	_angle2	yes	360
type_	_type	yes	1

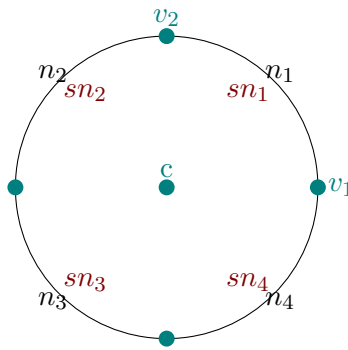
It also offers :

- some accessors
- implementation of inherited virtual functions concerning elliptic surfaces and disks :

```
virtual void computeMB();
virtual std::vector<const Point*> boundNodes() const;
virtual std::vector<Point*> nodes();
virtual std::vector<const Point*> nodes() const;
virtual std::vector<std::pair<ShapeType, std::vector<const Point*> > > curves()
    const;
virtual std::vector<std::pair<ShapeType, std::vector<const Point*> > > surfs() const;
```

The Disk class

To define a disk, you have to do the same way as [Ellipse](#).



It offers constructors, taking from 2 to 9 [Parameter](#). Authorized keys and what they do are:

attribute	key to use to define it	optional	default value
center_, p1_, p2_, p3_, p4_	(_center, _v1 and _v2) or (_center, _radius)	no	no
n1_, n2_, n3_, n4_	_nnodes	yes	2
h_	_hsteps	yes	no
domName_	_domain_name	yes	""
sideNames_	_side_names	yes	""
thetamin_	_angle1	yes	0
thetamax_	_angle2	yes	360
type_	_type	yes	1

It also offers :

- some accessors
- implementation of inherited virtual functions concerning elliptic surfaces and disks :

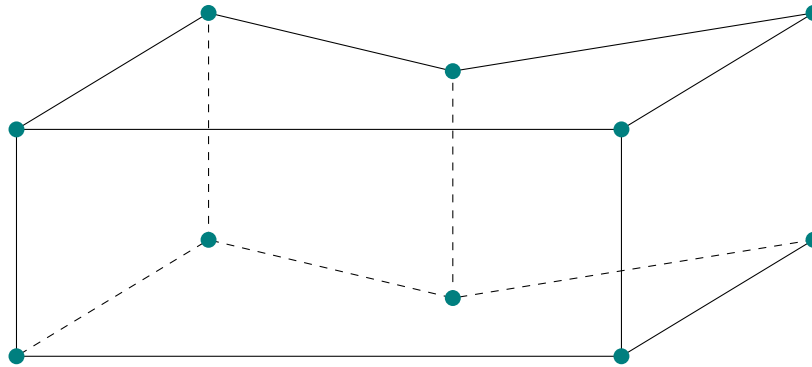
```
virtual void computeMB();
virtual std::vector<const Point*> boundNodes() const;
virtual std::vector<Point*> nodes();
virtual std::vector<const Point*> nodes() const;
virtual std::vector<std::pair<ShapeType, std::vector<const Point*> > > curves()
    const;
virtual std::vector<std::pair<ShapeType, std::vector<const Point*> > > surfs() const;
```


The Polyhedron class

A polyhedron is defined by its polygonal faces.

```
class Polyhedron : public Volume
{
private:
    std::vector<Polygon*> faces_;
protected:
    std::vector<Point> p_;
    std::vector<number_t> n_;
    std::vector<real_t> h_;
```

p_, n_ and h_ are defined here but used in child classes.



It offers constructors, taking from 1 to 2 **Parameter**. Authorized keys and what they do are:

attribute	key to use to define it	optional	default value
faces_	_faces	no	no
domName_	_domain_name	yes	""

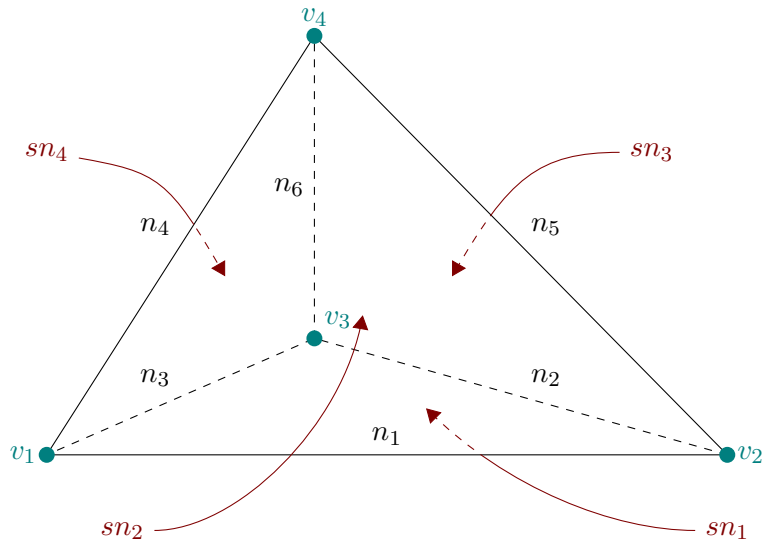
It also offers :

- some accessors
- implementation of inherited virtual functions concerning hexahedron, parallelepipeds and cubes :

```
virtual void computeMB();
virtual std::vector<const Point*> boundNodes() const;
virtual std::vector<Point*> nodes();
virtual std::vector<const Point*> nodes() const;
virtual std::vector<std::pair<ShapeType, std::vector<const Point*>>> curves()
    const;
virtual std::vector<std::pair<ShapeType, std::vector<const Point*>>> surfs() const;
```

The Tetrahedron class

To define a tetrahedron, you just have to give the 4 vertices.



It offers constructors, taking from 4 to 7 **Parameter**. Authorized keys and what they do are:

attribute	key to use to define it	optional	default value
p_	_v1, _v2, _v3 and _v4	no	no
n_	_nnodes	yes	2
h_	_hsteps	yes	no
domName_	_domain_name	yes	""
sideNames_	_side_names	yes	""

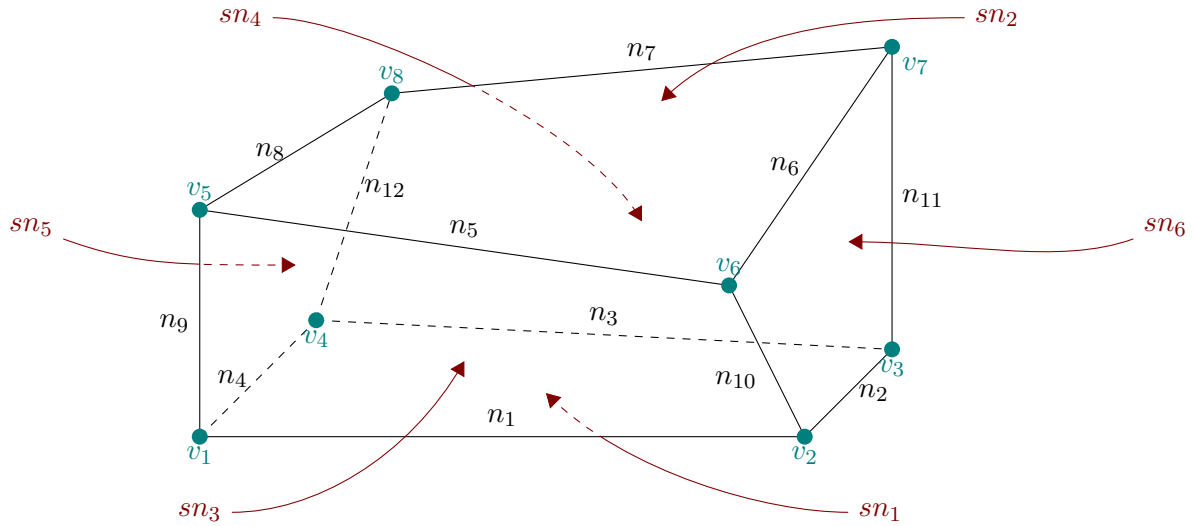
It also offers :

- some accessors
- implementation of inherited virtual functions concerning tetrahedron :

```
virtual void computeMB();
virtual std::vector<const Point*> boundNodes() const;
virtual std::vector<Point*> nodes();
virtual std::vector<const Point*> nodes() const;
virtual std::vector<std::pair<ShapeType, std::vector<const Point*> > > curves()
    const;
virtual std::vector<std::pair<ShapeType, std::vector<const Point*> > > surfs() const;
```

The Hexahedron class

To define a hexahedron, you just have to give the 8 vertices.



It offers constructors, taking from 8 to 11 [Parameter](#). Authorized keys and what they do are:

attribute	key to use to define it	optional	default value
p_	_v1, _v2, _v3, _v4, _v5, _v6, _v7 and _v8	no	no
n_	_nnodes	yes	2
h_	_hsteps	yes	no
domName_	_domain_name	yes	""
sideNames_	_side_names	yes	""

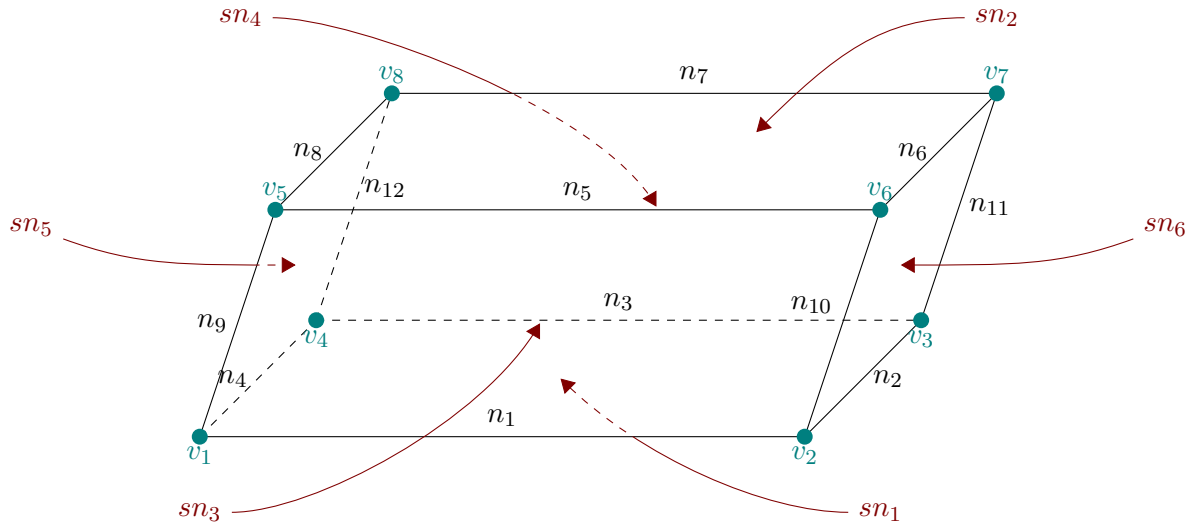
It also offers :

- some accessors
- implementation of inherited virtual functions concerning hexahedra :

```
virtual void computeMB();
virtual std::vector<const Point*> boundNodes() const;
virtual std::vector<Point*> nodes();
virtual std::vector<const Point*> nodes() const;
virtual std::vector<std::pair<ShapeType, std::vector<const Point*> > > curves()
    const;
virtual std::vector<std::pair<ShapeType, std::vector<const Point*> > > surfs() const;
```

The [Parallelepiped](#) class

To define a parallelepiped, you just have to give 4 vertices (p_1 , p_2 , p_4 and p_5 in the following figure):



It offers constructors, taking from 4 to 7 [Parameter](#). Authorized keys and what they do are:

attribute	key to use to define it	optional	default value
p_	_v1, _v2, _v4 and _v5	no	no
n_	_nnodes	yes	2
h_	_hsteps	yes	no
domName_	_domain_name	yes	""
sideNames_	_side_names	yes	""

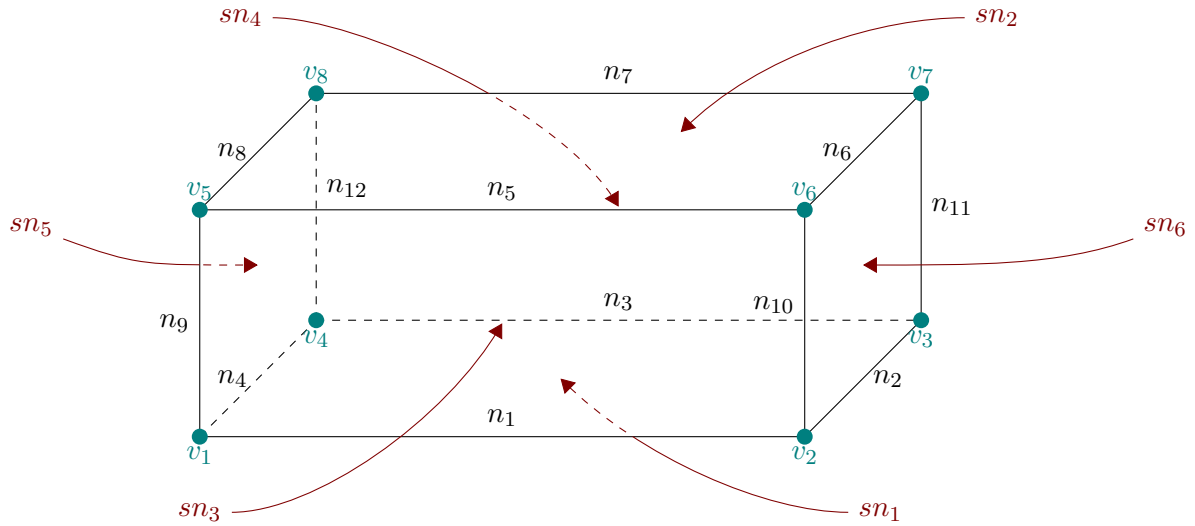
It also offers :

- some accessors
- implementation of inherited virtual functions concerning parallelepipeds :

```
virtual void computeMB();
virtual std::vector<const Point*> boundNodes() const;
virtual std::vector<Point*> nodes();
virtual std::vector<const Point*> nodes() const;
virtual std::vector<std::pair<ShapeType, std::vector<const Point*>>> curves()
    const;
virtual std::vector<std::pair<ShapeType, std::vector<const Point*>>> surfs() const;
```

The [Cuboid](#) class

To define a cuboid, you just have to give 4 vertices, as for [Parallelepiped](#). And as for [Rectangle](#), you can define it by its center or its origin and its lengths, or define it by its bounds



It offers constructors, taking from 4 to 9 [Parameter](#). Authorized keys and what they do are:

attribute	key to use to define it	optional	default value
p_	(_v1, _v2, _v4, _v5) or (_center, _xlength, _ylength, _zlength) or (_origin, _xlength, _ylength, _zlength) or (_xmin, _xmax, _ymin, _ymax, _zmin, _zmax)	no	no
n_	_nnodes	yes	2
h_	_hsteps	yes	no
domName_	_domain_name	yes	""
sideNames_	_side_names	yes	""

It also offers :

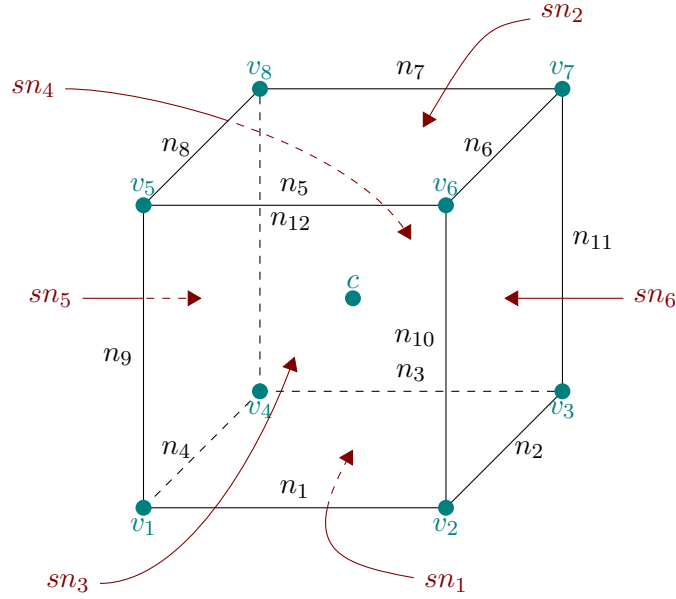
- some accessors
- implementation of inherited virtual functions concerning cuboids :

```
virtual void computeMB();
virtual std::vector<const Point*> boundNodes() const;
virtual std::vector<Point*> nodes();
virtual std::vector<const Point*> nodes() const;
virtual std::vector<std::pair<ShapeType, std::vector<const Point*>>> curves()
    const;
virtual std::vector<std::pair<ShapeType, std::vector<const Point*>>> surfs() const;
```

The [Cube](#) class

To define a cube, you just have to give 4 vertices. You can also give the center or the origin and the length.

```
class Cube : public Cuboid
{
private:
    dimen_t nbocants_;
```



It offers constructors, taking from 2 to 7 **Parameter**. Authorized keys and what they do are:

attribute	key to use to define it	optional	default value
p_	(_v1 , _v2 , _v4 , _v5) or (_center , _length) or (_origin , _length)	no	no
n_	_nnodes	yes	2
h_	_hsteps	yes	no
domName_	_domain_name	yes	""
sideNames_	_side_names	yes	""
nbocants_	_nbocants	yes	8

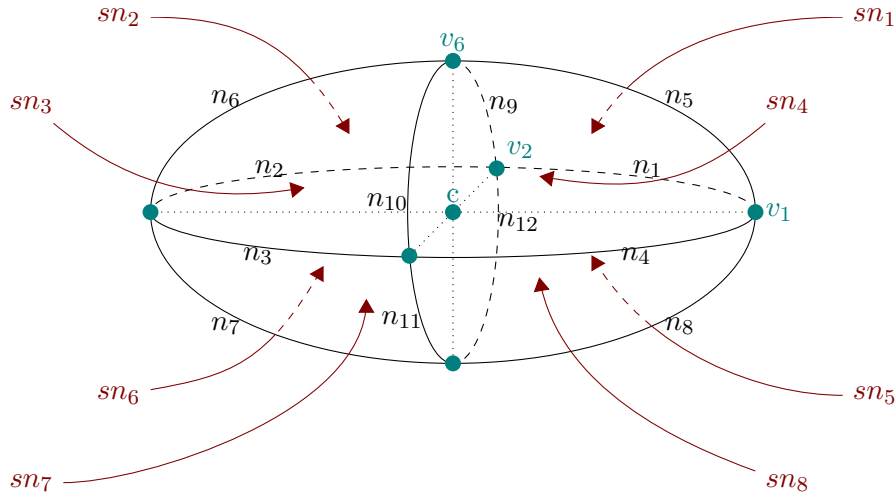
It also offers :

- some accessors
- implementation of inherited virtual functions concerning cubes :

```
virtual void computeMB();
virtual std::vector<const Point*> boundNodes() const;
virtual std::vector<Point*> nodes();
virtual std::vector<const Point*> nodes() const;
virtual std::vector<std::pair<ShapeType, std::vector<const Point*>>> curves()
    const;
virtual std::vector<std::pair<ShapeType, std::vector<const Point*>>> surfs() const;
```

The Ellipsoid class

To define an ellipsoid, you do the same way as for an ellipse, namely defining a center and 3 apogees (c , p_1 , p_2 and p_6 in the following figure):



```
class Ellipsoid : public Volume
{
protected:
    Point center_, p1_, p2_, p3_, p4_, p5_, p6_;
    real_t xlength_, ylength_, zlength_;
    bool isAxis_;

    number_t n1_, n2_, n3_, n4_, n5_, n6_, n7_, n8_, n9_, n10_, n11_, n12_;
    std::vector<real_t> h_;

    dimen_t nbocants_;
    dimen_t type_;
};
```

It offers constructors, taking from 4 to 9 **Parameter**. Authorized keys and what they do are:

attribute	key to use to define it	optional	default value
p-	(_center, _v1, _v2, _v6) or (_center, _xlength, _ylength, _zlength) or (_origin, _xlength, _ylength, _zlength)	no	no
n-	_nnodes	yes	2
h-	_hsteps	yes	no
domName-	_domain_name	yes	""
sideNames-	_side_names	yes	""
nbocants-	_nbocants	yes	8
type-	_type	yes	1

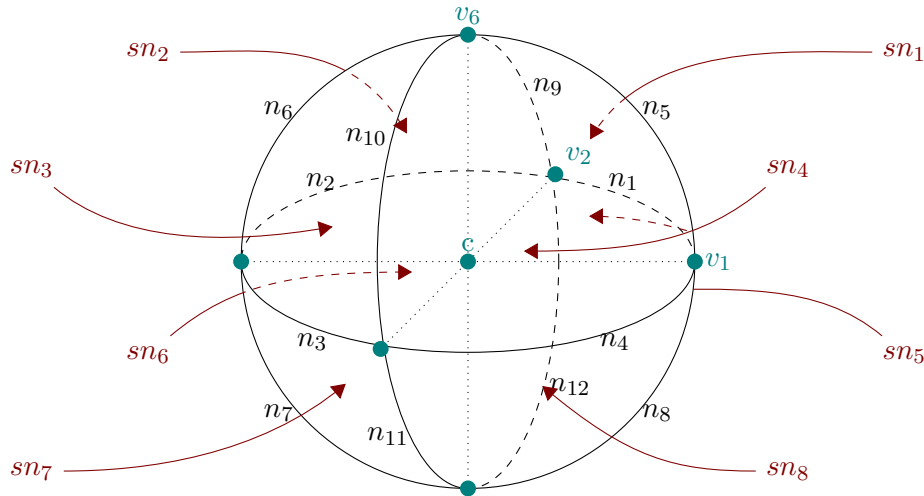
It also offers :

- some accessors
- implementation of inherited virtual functions concerning ellipsoids :

```
virtual void computeMB();
virtual std::vector<const Point*> boundNodes() const;
virtual std::vector<Point*> nodes();
virtual std::vector<const Point*> nodes() const;
virtual std::vector<std::pair<ShapeType, std::vector<const Point*> > > curves()
    const;
virtual std::vector<std::pair<ShapeType, std::vector<const Point*> > > surfs() const;
```

The `Ball` class

To define a ball, you do the same way as for an ellipsoid.



It offers constructors, taking from 2 to 9 `Parameter`. Authorized keys and what they do are:

attribute	key to use to define it	optional	default value
p_	(<code>_center</code> , <code>_v1</code> , <code>_v2</code> , <code>_v6</code>) or (<code>_center</code> , <code>_length</code>) or (<code>_origin</code> , <code>_length</code>)	no	no
n_	<code>_nnodes</code>	yes	2
h_	<code>_hsteps</code>	yes	no
domName_	<code>_domain_name</code>	yes	""
sideNames_	<code>_side_names</code>	yes	""
nbocants_	<code>_nbocants</code>	yes	8
type_	<code>_type</code>	yes	1

It also offers :

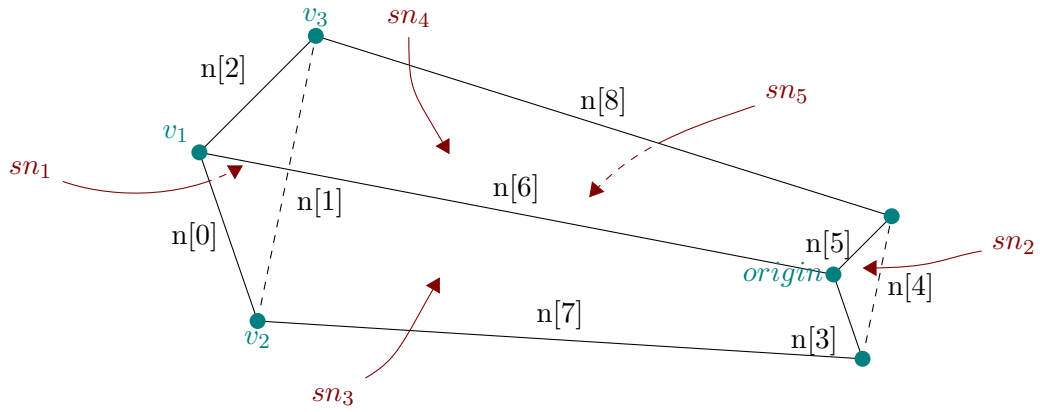
- some accessors
- implementation of inherited virtual functions concerning balls :

```
virtual void computeMB();
virtual std::vector<const Point*> boundNodes() const;
virtual std::vector<Point*> nodes();
virtual std::vector<const Point*> nodes() const;
virtual std::vector<std::pair<ShapeType, std::vector<const Point*>>> curves()
    const;
virtual std::vector<std::pair<ShapeType, std::vector<const Point*>>> surfs() const;
```

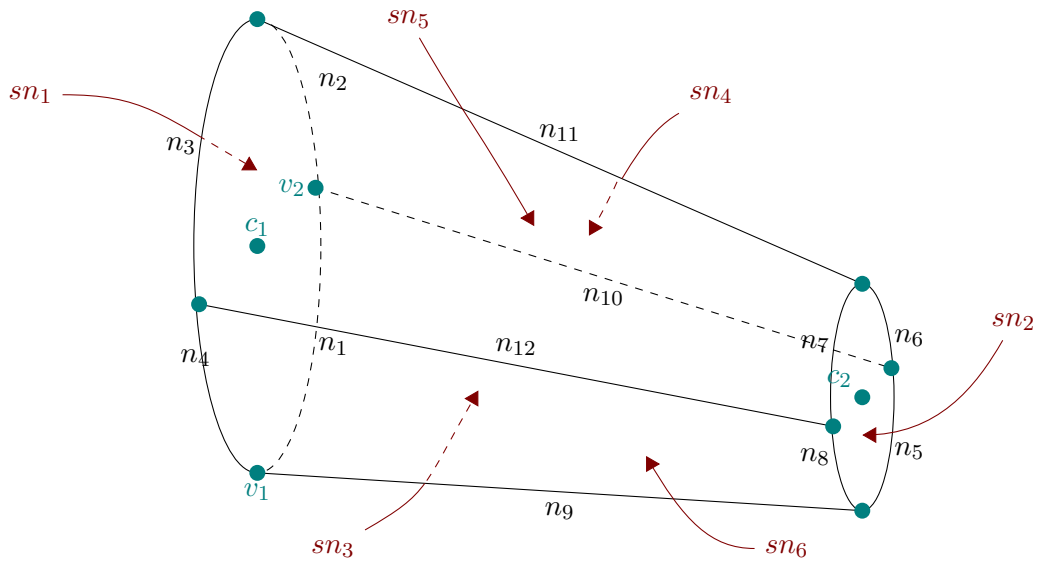
The `Trunk` class

A trunk is a generalized truncated cone. To define a trunk, you need to give a surface, namely a polygonal surface or a elliptical surface. To define the other surface, you just need to give a point of this surface (*origin*), and the scale factor according to the first surface.

For a trunk with polygonal basis, *origin* is the equivalent of the first vertex of the surface you give, as you can see on the following figure of a trunk with triangular basis. The triangle being defined by its vertices p_1 , p_2 and p_3 , *origin* is the equivalent of p_1 :



For a trunk with elliptical basis, *origin* is the center of the second basis, as you can see on the following figure of a trunk with elliptical basis.



```
class Trunk : public Volume
{
protected:
    Surface * basis_;
    real_t scale_;
    std::vector<Point> p_;
    std::vector<number_t> n_;
    std::vector<real_t> h_;
    Point origin_, center1_, p1_, p2_;
    bool isElliptical_, isN_;
};
```

It offers constructors, taking from 3 to 8 **Parameter**. Authorized keys and what they do are:

attribute	key to use to define it	optional	default value
p_	(_center1, _v1, _v2, _center2, _scale) or (_basis, _origin, _scale)	no	no
n_	_nnodes	yes	2
h_	_hsteps	yes	no
domName_	_domain_name	yes	""
sideNames_	_side_names	yes	""

It also offers :

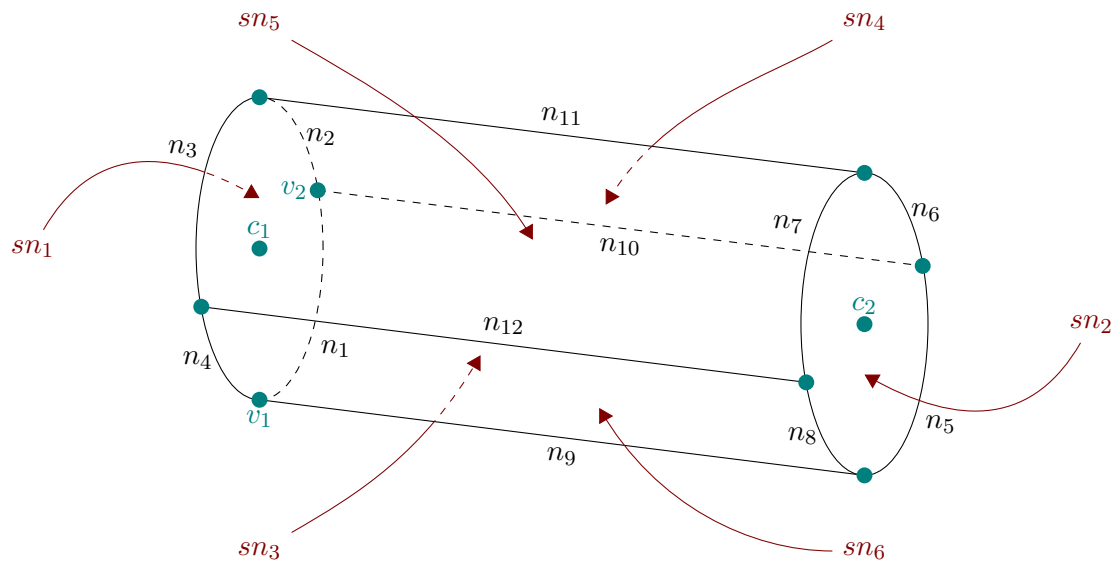
- some accessors
- implementation of inherited virtual functions concerning trunks :

```
virtual void computeMB();
virtual std::vector<const Point*> boundNodes() const;
virtual std::vector<Point*> nodes();
virtual std::vector<const Point*> nodes() const;
virtual std::vector<std::pair<ShapeType, std::vector<const Point*> > > curves()
    const;
virtual std::vector<std::pair<ShapeType, std::vector<const Point*> > > surfs() const;
```

The Cylinder class

A cylinder is a truncated cone whose apex is at infinite distance. So it is the geometry defined by the extrusion of a surface by translation.

```
class Cylinder : public Trunk
{
protected:
    //! direction vector
    Point dir_;
```



It offers constructors, taking from 2 to 7 **Parameter**. Authorized keys and what they do are:

attribute	key to use to define it	optional	default value
p_	(_center1, _v1, _v2, _center2) or (_basis, _direction)	no	no
n_	_nnodes	yes	2
h_	_hsteps	yes	no
domName_	_domain_name	yes	""
sideNames_	_side_names	yes	""

It also offers :

- some accessors

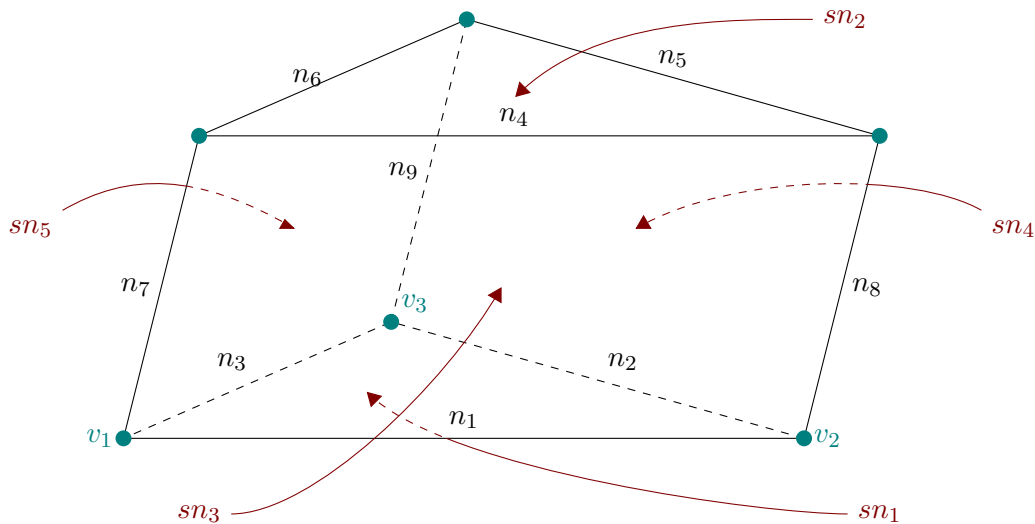
- implementation of inherited virtual functions concerning cylinders :

```
virtual void computeMB();
virtual std::vector<const Point*> boundNodes() const;
virtual std::vector<Point*> nodes();
virtual std::vector<const Point*> nodes() const;
virtual std::vector<std::pair<ShapeType, std::vector<const Point*>>> curves()
    const;
virtual std::vector<std::pair<ShapeType, std::vector<const Point*>>> surfs() const;
```

The Prism class

A prism is by definition a cylinder whose basis is a polygonal surface. Often a prism refers to a cylinder with triangular basis (as the finite element cell).

```
class Prism : public Cylinder
{
private:
    bool isTriangular_;
    Point p3_;
```



It offers constructors, taking from 2 to 7 **Parameter**. Authorized keys and what they do are:

attribute	key to use to define it	optional	default value
p-	(_v1 , _v2 , _v3 , _direction) or (_basis , _direction)	no	no
n-	_nnodes	yes	2
h-	_hsteps	yes	no
domName-	_domain_name	yes	""
sideNames-	_side_names	yes	""

It also offers :

- some accessors
- implementation of inherited virtual functions concerning prisms :

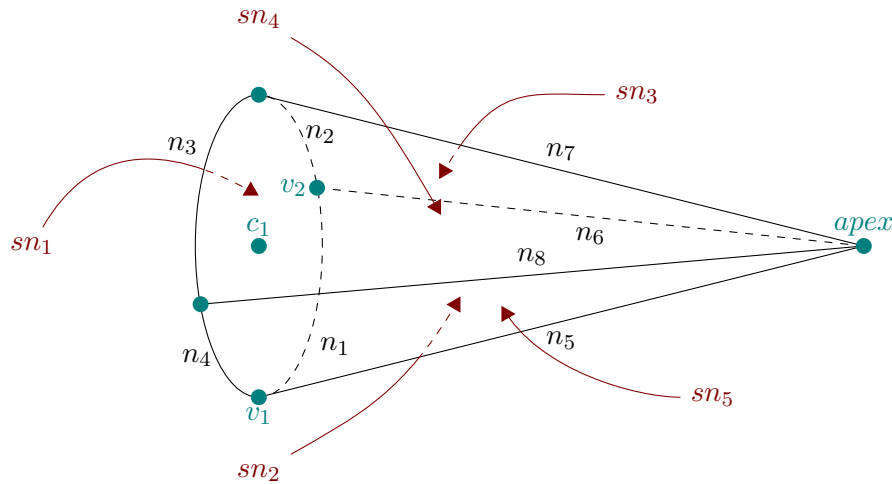
```

virtual void computeMB();
virtual std::vector<const Point*> boundNodes() const;
virtual std::vector<Point*> nodes();
virtual std::vector<const Point*> nodes() const;
virtual std::vector<std::pair<ShapeType, std::vector<const Point*>>> curves()
    const;
virtual std::vector<std::pair<ShapeType, std::vector<const Point*>>> surfs() const;

```

The Cone class

A cone is defined by a surface and an apex. As for trunks and cylinders, you can also define directly a cone with elliptical basis.



It offers constructors, taking from 3 to 7 **Parameter**. Authorized keys and what they do are:

attribute	key to use to define it	optional	default value
p-	(_center1 , _v1 , _v2 , _apex) or (_basis , _apex)	no	no
n-	_nnodes	yes	2
h-	_hsteps	yes	no
domName-	_domain_name	yes	""
sideNames-	_side_names	yes	""

It also offers :

- some accessors
- implementation of inherited virtual functions concerning cones :

```

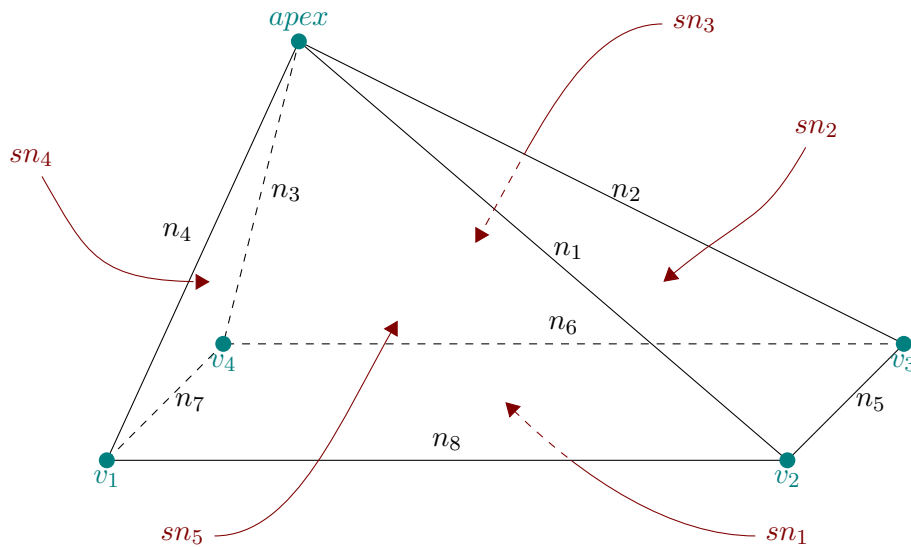
virtual void computeMB();
virtual std::vector<const Point*> boundNodes() const;
virtual std::vector<Point*> nodes();
virtual std::vector<const Point*> nodes() const;
virtual std::vector<std::pair<ShapeType, std::vector<const Point*>>> curves()
    const;
virtual std::vector<std::pair<ShapeType, std::vector<const Point*>>> surfs() const;

```

The `Pyramid` class

A pyramid is a cone with a polygonal basis. Often a pyramid refers to a cone with quadrangular basis (as the finite element cell).

```
class Pyramid : public Cone
{
    private:
        bool isQuadrangular_;
        Point p3_, p4_;
}
```



It offers constructors, taking from 2 to 8 `Parameter`. Authorized keys and what they do are:

attribute	key to use to define it	optional	default value
<code>p_</code>	<code>(_v1, _v2, _v3, _v4, _apex)</code> or <code>(_basis, _apex)</code>	no	no
<code>n_</code>	<code>_nnodes</code>	yes	2
<code>h_</code>	<code>_hsteps</code>	yes	no
<code>domName_</code>	<code>_domain_name</code>	yes	""
<code>sideNames_</code>	<code>_side_names</code>	yes	""

It also offers :

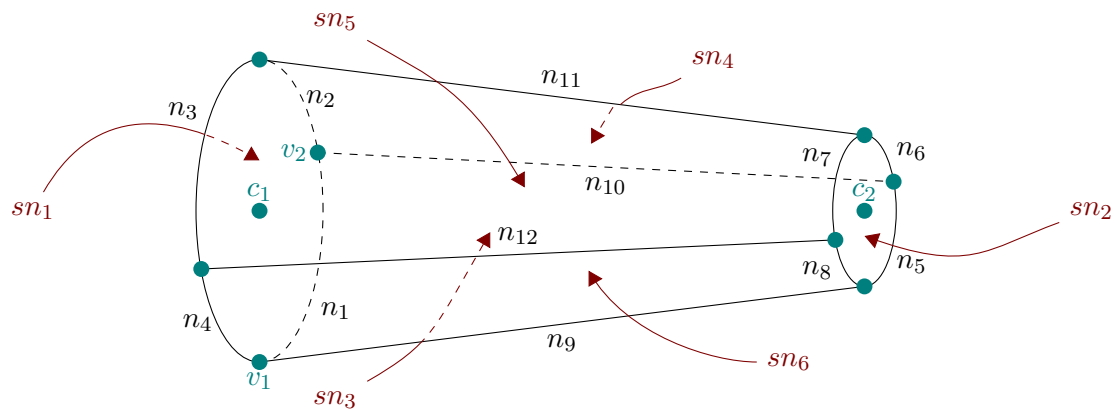
- some accessors
- implementation of inherited virtual functions concerning pyramids :

```
virtual void computeMB() ;
virtual std::vector<const Point*boundNodes() const;
virtual std::vector<Point*nodes() ;
virtual std::vector<const Point*nodes() const;
virtual std::vector<std::pair<ShapeType, std::vector<const Point*> > > curves()
    const;
virtual std::vector<std::pair<ShapeType, std::vector<const Point*> > > surfs() const;
```

The RevTrunk class

A revolution trunk is a right trunk with circular basis. **RevTrunk** offers you more geometry abilities. Indeed, you can decide to add extensions at ends of the revolution trunk. Extensions can be : none, flat, ellipsoid, or cone.

```
class RevTrunk : public Trunk
{
protected:
    real_t radius1_, radius2_;
    number_t nbSubDomains_;
    GeometricEndShape endShape1_;
    real_t distance1_;
    GeometricEndShape endShape2_;
    real_t distance2_;
    dimen_t type_;
```



It offers constructors, taking from 4 to 13 **Parameter**. Authorized keys and what they do are:

attribute	key to use to define it	optional	default value
p_	(_center1, _center2, _radius1, _radius2)	no	no
endShape1_	_end1_shape	yes	_gesflat
endShape2_	_end2_shape	yes	_gesFlat
distance1_	_end1_distance	yes	0.
distance2_	_end2_distance	yes	0.
n_	_nnodes	yes	2
h_	_hsteps	yes	no
domName_	_domain_name	yes	""
sideNames_	_side_names	yes	""
nbSubDomains_	_nbsubdomains	yes	1
type_	_type	yes	1

It also offers :

- some accessors
- implementation of inherited virtual functions concerning revolution trunks :

```
virtual void computeMB();
virtual std::vector<const Point*> boundNodes() const;
virtual std::vector<Point*> nodes();
virtual std::vector<const Point*> nodes() const;
```

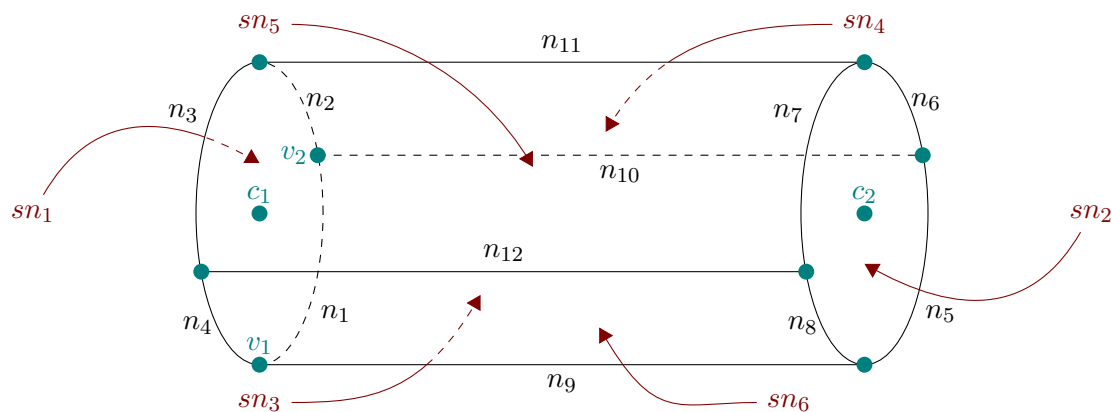
```

virtual std::vector<std::pair<ShapeType, std::vector<const Point*>>> curves()
    const;
virtual std::vector<std::pair<ShapeType, std::vector<const Point*>>> surfs() const;

```

The RevCylinder class

A revolution cylinder is a revolution trunk where both radii are equal. So, we need centers of both bases, and the radius. As [RevTrunk](#), [RevCylinder](#) offers you the ability to add extensions at ends of the revolution cylinder.



It offers constructors, taking from 3 to 12 [Parameter](#). Authorized keys and what they do are:

attribute	key to use to define it	optional	default value
p_	(_center1, _center2, _radius)	no	no
endShape1_	_end1_shape	yes	_gesflat
endShape2_	_end2_shape	yes	_gesFlat
distance1_	_end1_distance	yes	0.
distance2_	_end2_distance	yes	0.
n_	_nnodes	yes	2
h_	_hsteps	yes	no
domName_	_domain_name	yes	""
sideNames_	_side_names	yes	""
nbSubDomains_	_nbsubdomains	yes	1
type_	_type	yes	1

It also offers :

- some accessors
- implementation of inherited virtual functions concerning revolution cylinders :

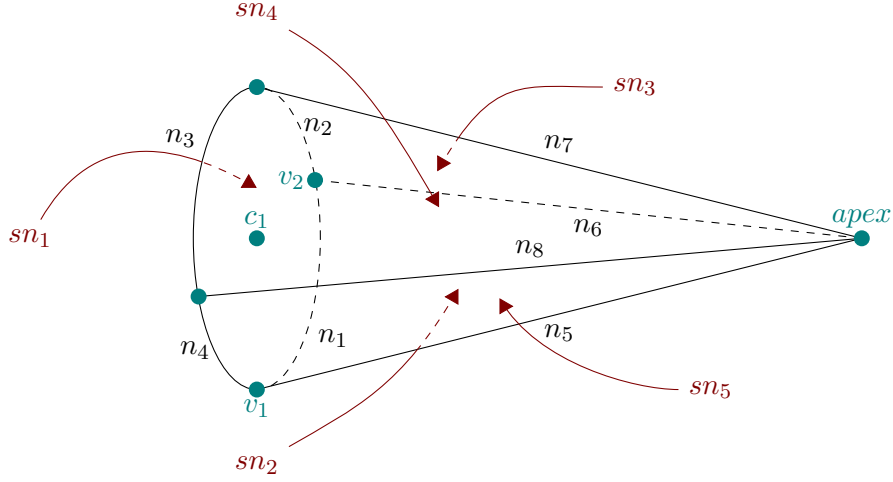
```

virtual void computeMB();
virtual std::vector<const Point*> boundNodes() const;
virtual std::vector<Point*> nodes();
virtual std::vector<const Point*> nodes() const;
virtual std::vector<std::pair<ShapeType, std::vector<const Point*>>> curves()
    const;
virtual std::vector<std::pair<ShapeType, std::vector<const Point*>>> surfs() const;

```

The `RevCone` class

A revolution cone is a revolution trunk where second radius is equal to 0. As `RevTrunk`, `RevCone` offers you more the ability to add an extension to the basis of a revolution cone.



It offers constructors, taking from 3 to 10 `Parameter`. Authorized keys and what they do are:

attribute	key to use to define it	optional	default value
p_	(<code>_center</code> , <code>_radius</code> , <code>_apex</code>)	no	no
endShape1_	<code>_end_shape</code>	yes	<code>_gesflat</code>
distance1_	<code>_end_distance</code>	yes	0.
n_	<code>_nnodes</code>	yes	2
h_	<code>_hsteps</code>	yes	no
domName_	<code>_domain_name</code>	yes	""
sideNames_	<code>_side_names</code>	yes	""
nbSubDomains_	<code>_nbsubdomains</code>	yes	1
type_	<code>_type</code>	yes	1

It also offers :

- some accessors
- implementation of inherited virtual functions concerning revolution cones :

```
virtual void computeMB();
virtual std::vector<const Point*> boundNodes() const;
virtual std::vector<Point*> nodes();
virtual std::vector<const Point*> nodes() const;
virtual std::vector<std::pair<ShapeType, std::vector<const Point*>>> curves()
    const;
virtual std::vector<std::pair<ShapeType, std::vector<const Point*>>> surfs() const;
```

The `SetOfElems` class

4.2 Mesh management

The `Mesh` class handles mesh description, that is a collection of mesh elements. `Mesh` objects are built either from internal meshing tools or from mesh files provided by external meshing tools as GMSH for instance. To avoid complexity to end users, we decide to have a unique mesh class with no inheritance.

It means that each meshing tool should be a member function of the `Mesh` class. These member functions are distributed in various files (see the Meshing tools section).

Basically, a mesh is a collection of points (vertices and intermediate points), of geometric elements (`GeomElement`) to carry the computation domain partition, of geometric domains (`Domain`): a list of elements or a list of sides of elements. Note that curved elements are supported using geometric interpolation of order greater than one. This `Mesh` class supports also mesh of surface in 3D space and mesh of line in 2D space. Besides the basic geometric data, it is also able to manage global list of sides (say faces of a 3D mesh, edges of a 2D mesh) or global list of sides of sides (edges of a 3D mesh). By default these lists are not constructed.

General informations on a mesh such as mesh name, space dimension, name of variables, bounding box are collected in the `Geometry` class. In future, this class should be developed to handle other informations, in particular boundary parametrisations useful to describe in an analytical way the domain to mesh, allowing to provide high order mesh and refinement meshing tools.

4.2.1 The Mesh class

```
class Mesh
{
public :
    Geometry* geometry_p;           // global geometric information
    std::vector<Point> nodes;        // list of all mesh nodes
    Number lastIndex_;              // last index of GeomElements

protected :
    String name_;                   // name for documentation purposes
    String comment_;                // comment documentation purposes
    std::vector<GeomElement*> elements_; // list of geometric elements
                                     (MeshElement)
    std::vector<GeomDomain*> domains_; // list of geometric domains
    std::vector<Number> vertices_;    // list of vertices indexed by their
                                     number in nodes vector
    bool isMadeOfSimplices_;          // only segments, triangles or tetrahedra
    Dimen order_;                    // order of the mesh = max of element
                                     orders
    std::vector<GeomElement*> sides_;  // list of sides (edges in 2D, face in
                                     3D), built if required
    std::vector<GeomElement*> sideOfSides_; // list of sides of sides (edges in 3D),
                                     built if required
    std::vector<std::vector<GeoNumPair> > vertexElements_; // for each vertex v, list
                                                             of elements having vertex v, built if required
    mutable Mesh* firstOrderMesh_p; // underlying first order mesh when mesh
                                     has elements of order greater than one
    ...
}
```

`GeomElement` class is described in detail in section 4.4. In few words, the `GeomElement` class stores in a `MeshElement` object the numbering vectors of a mesh element (node numbers, vertex numbers, side numbers, side of side numbers) and a pointer to the reference element (`RefElement`) supporting the geometric interpolation of the element (define the map to reference geometric element). In case of a side element, the `GeomElement` class stores the parent sides and the side numbers. Note that in that case, `MeshElement` object does not necessarily exist!

Regarding meshing tools, the `Mesh` class provides various constructor : one reading a mesh file, two general constructors from any geometry, a constructor meshing one reference element, a constructor extruding a 1D/2D mesh and a constructor converting mesh elements to an other type:

```

Mesh();
Mesh(const String& filename, const String&, IOFormat mft = _undefFormat);
Mesh(const Mesh&);
~Mesh();
Mesh(const Geometry& g, Number order = 1, MeshGenerator mg = _defaultGenerator,
      const String& name = ""); // for 1D geometry
Mesh(const Geometry& g, ShapeType sh, Number order = 1, MeshGenerator mg =
      _defaultGenerator, const String& name = ""); // for 2D and 3D geometries
Mesh(ShapeType shape); // for reference element
Mesh(const Mesh& ms, const Point& O, const Point& d, number_t nbl, number_t
      namingDomain=0, number_t namingSection=0, number_t namingSide=0, const
      string_t& meshName="") //for extruded mesh
Mesh(const Mesh& mesh, ShapeType sh, const string_t name=""); // for element
converter

```

See subsection 4.2.2 for an exhaustive list of meshing tools.

As most of members are protected, there are useful accessors:

```

const std::vector<GeomElement*>& elements() const;
const std::vector<Domain*>& domains() const;
const std::vector<GeomElement*>& sides() const;
const std::vector<GeomElement*>& sideOfSides() const;
const std::vector<Number>& vertices() const;
const String& name() const;
bool isMadeOfSimplices() const;
const GeomElement& element(Number i) const;
const Domain& domains(Number i) const;
const GeomElement& side(Number i) const;
const GeomElement& sideOfSide(Number i) const;
const Number vertex(Number i) const;
const Point& vertexPoint(Number i) const;
Dimen spaceDim() const;
Dimen meshDim() const;
Number nbOfElements() const;
Number nbOfDomains() const;
Number nbOfSides() const;
Number nbOfSideOfSides() const;
Number nbOfVertices() const;
Number nbOfNodes() const;

```

There are all constant and some are useful shortcuts.

The `Mesh` class provides four important member functions to complete on demand the construction of a mesh: functions to build the list of sides, the list of sides of sides and the list of vertex elements and a function to compute element orientation (sign of jacobian determinant) and the measure of element and its sides:

```

void buildSides();
void buildSideOfSides();
void buildVertexElements();
void buildGeomData();

```

Note that `buildSides` and `buildSideOfSides` functions update some data of `GeomElements`.

Finally, the class has usual printing facilities:

```

void print(std::ostream &) const;
friend std::ostream& operator<<(std::ostream&, const Mesh&);

```

Example

We give a basic example of using the `Mesh` class. It concerns the meshing of the rectangle $[0,1] \times [1,3]$ with regular P1 geometric elements (20 by 40) where the boundary $[0,1] \times \{1\}$ is named `Gamma_1` and the boundary $\{0\} \times [1,3]$ is named `Gamma_2`:

```
std::vector<String> sidenames(4,"");sidenames[0]="Gamma_1";sidenames[2]="Gamma_2";
Mesh mesh2d(Rectangle(0,1,1,3),_triangle,20,40,sidenames,"P1 mesh of [0,1]x[1,3]");
mesh2d.buildSides(); //build sides list
verboseLevel(3);std::cout<<mesh2d;
```

It produces the following output:

```
Mesh'P1 mesh of [0,1]x[1,3]'
space dimension : 2, element dimension : 2
Geometry rectangle [0,1]x[1,3] of dimension 2, BoundingBox [0,1]x[1,3], names of variable : x y
number of elements : 1600, number of vertices : 861, number of nodes : 861, number of domains : 3 ( Omega Gamma_1 Gamma_2 )
list of elements (1600) :
geometric element 1 : triangle_Lagrange_1, orientation +1 measure = 0.00125
nodes : 2 ->(0.05, 0) 22 ->(0, 0.05) 1 ->(0, 0)
vertices : 2 ->(0.05, 0) 22 ->(0, 0.05) 1 ->(0, 0)
measure of sides = 0.0707107 0.05 0.05
sides : 61 21 1 sideOfSides : unset adjacent elements : 2
geometric element 2 : triangle_Lagrange_1, orientation +1 measure = 0.00125
nodes : 22 ->(0, 0.05) 2 ->(0.05, 0) 23 ->(0.05, 0.05)
vertices : 22 ->(0, 0.05) 2 ->(0.05, 0) 23 ->(0.05, 0.05)
measure of sides = 0.0707107 0.05 0.05
sides : 61 62 63 sideOfSides : unset adjacent elements : 1 3 41
geometric element 3 : triangle_Lagrange_1, orientation +1 measure = 0.00125
nodes : 3 ->(0.1, 0) 23 ->(0.05, 0.05) 2 ->(0.05, 0)
vertices : 3 ->(0.1, 0) 23 ->(0.05, 0.05) 2 ->(0.05, 0)
measure of sides = 0.0707107 0.05 0.05
sides : 64 62 2 sideOfSides : unset adjacent elements : 2 4
...
geometric element 1600 : triangle_Lagrange_1, orientation +1 measure = 0.00125
nodes : 860 ->(0.95, 2) 840 ->(1, 1.95) 861 ->(1, 2)
vertices : 860 ->(0.95, 2) 840 ->(1, 1.95) 861 ->(1, 2)
measure of sides = 0.0707107 0.05 0.05
sides : 2458 2459 2460 sideOfSides : unset adjacent elements : 1599
list of vertices (861) :
1 -> (0, 0)
2 -> (0.05, 0)
3 -> (0.1, 0)
...
861 -> (1, 2)
list of nodes (861) :
1 -> (0, 0)
2 -> (0.05, 0)
3 -> (0.1, 0)
...
861 -> (1, 2)
list of sides (2460) :
side 1 -> geometric side element 1601 : side 3 of element 1
side 2 -> geometric side element 1602 : side 3 of element 3
side 3 -> geometric side element 1603 : side 3 of element 5
...
side 2460 -> geometric side element 4059 : side 3 of element 1600
list of sides of sides (0) : unset
list of domains (3) :
Domain 'Omega' of dimension 2 from mesh 'P1 mesh of [0,1]x[1,3]'
geometric element 1 : triangle_Lagrange_1, orientation +1 measure = 0.00125
geometric element 2 : triangle_Lagrange_1, orientation +1 measure = 0.00125
geometric element 3 : triangle_Lagrange_1, orientation +1 measure = 0.00125
...
geometric element 1600 : triangle_Lagrange_1, orientation +1 measure = 0.00125
Domain 'Gamma_1' of dimension 1 from mesh 'P1 mesh of [0,1]x[1,3]'
geometric side element 1601 : side 3 of element 1
geometric side element 1602 : side 3 of element 3
geometric side element 1603 : side 3 of element 5
...
geometric side element 1620 : side 3 of element 39
```

```

Domain 'Gamma_2' of dimension 1 from mesh 'P1 mesh of [0,1]x[1,3]'
geometric side element 1621 : side 2 of element 1
geometric side element 1622 : side 2 of element 41
geometric side element 1623 : side 2 of element 81
...
geometric side element 1660 : side 2 of element 1561

```

4.2.2 Meshing tools

In this section, we will study in details how work the `Mesh` constructors from `Geometry`.

```

Mesh(const Geometry& g, Number order = 1, MeshGenerator mg = _defaultGenerator,
     const String& name = ""); // constructor from 1D geometry
Mesh(const Geometry& g, ShapeType sh, Number order = 1, MeshGenerator mg =
     _defaultGenerator, const String& name = ""); // constructor from 2D and 3D
     geometries

```

The `MeshGenerator` argument can have the following values :

```

enum MeshGenerator
{
    _defaultGenerator=0,
    _structured, structured=_structured,
    _subdiv, subdiv=_subdiv,
    _gmsh, gmsh=_gmsh
};

```

We will now look at the behavior of the different generators :

Structured meshes

The "structured" generator works only on `Segment`, `Rectangle`, and `Parallelepiped` geometries, and for P1, Q1, prisme 1 or pyramid 1 finite elements. In this way, you call the following underlying methods :

```

void meshP1Segment(const Segment&, Number, const std::vector<String>&, const
String& na = "");
void meshP1Rectangle(const Rectangle&, Number, Number, const std::vector<String>&,
const String& na = "");
void meshQ1Rectangle(const Rectangle&, Number, Number, const std::vector<String>&,
const String& na = "");
void meshP1Parallelepiped(const Parallelepiped&, Number, Number, Number, const
std::vector<String>&, const String& na = "");
void meshQ1Parallelepiped(const Parallelepiped&, Number, Number, Number, const
std::vector<String>&, const String& na = "");
void meshPr1Parallelepiped(Parallelepiped&, number_t, number_t, number_t, const
std::vector<string_t>&, const string_t& na = "");
void meshPy1Parallelepiped(Parallelepiped&, number_t, number_t, number_t, const
std::vector<string_t>&, const string_t& na = "");

```

Mesher with *subdivision* algorithm

The subdivision algorithm enables to mesh a wide set of geometries with triangles, quadrangles, tetrahedrons or hexahedrons: `Ball`, `Cube`, `RevCylindricVolume` and `SetOfElems`. This algorithm takes into account 2 main parameters : `nbsubdiv` and `order`. The first one controls the number of subdivisions to be done (mesh refinement), and the second one is the polynomial order of approximation of the geometry, which can take any (positive integer) value.

To use subdivision algorithm, dedicated constructors for the geometries are available:

```

Cube(Real x, Real y, Real z, Real edgeLen = 1., int nbOctants=1, number_t
    nbsubdiv=0);
Cube(const Point& p1, const Point& p2, const Point& p3, const Point& p4, int
    nbOctants, number_t nbsubdiv);
Cube(Real x, Real y, Real z, Real edgeLen, Real theta1, Dimen axis1, int
    nbOctants=1, number_t nbsubdiv=0);
Cube(Real x, Real y, Real z, Real edgeLen, Real theta1, Dimen axis1, Real theta2,
    Dimen axis2, int nbOctants=1, number_t nbsubdiv=0);
Cube(Real x, Real y, Real z, Real edgeLen, Real theta1, Dimen axis1, Real theta2,
    Dimen axis2, Real theta3, Dimen axis3, int nbOctants=1, number_t nbsubdiv=0);
Ball(Real x, Real y, Real z, Real radius = 1., int nbOctants=8, number_t
    nbsubdiv=0, number_t type=1);
Ball(Real x, Real y, Real z, Real radius, Real theta1, Dimen axis1, int
    nbOctants=8, number_t nbsubdiv=0, number_t type=1);
Ball(Real x, Real y, Real z, Real radius, Real theta1, Dimen axis1, Real theta2,
    Dimen axis2, int nbOctants=8, number_t nbsubdiv=0, number_t type=1);
Ball(Real x, Real y, Real z, Real radius, Real theta1, Dimen axis1, Real theta2,
    Dimen axis2, Real theta3, Dimen axis3, int nbOctants=8, number_t nbsubdiv=0,
    number_t type=1);
RevCylindricVolume(Point p1, Point p2, Real radius, int nbslices=0, number_t
    nbsubdiv=0, number_t type=1, CylinderEndShape endShape1=_cesFlat,
    CylinderEndShape endShape2=_cesFlat, Real distance1=0., Real distance2=0.);
SetOfElems(const std::vector<Point>& pts, const std::vector<std::vector<number_t>
    >& tri, const std::vector<std::vector<number_t> >& bound, const number_t
    nbsubdiv=1);

```

The parameter `nbOctants` determines which part of the geometry is to be meshed ; it is clarified in the following figures :

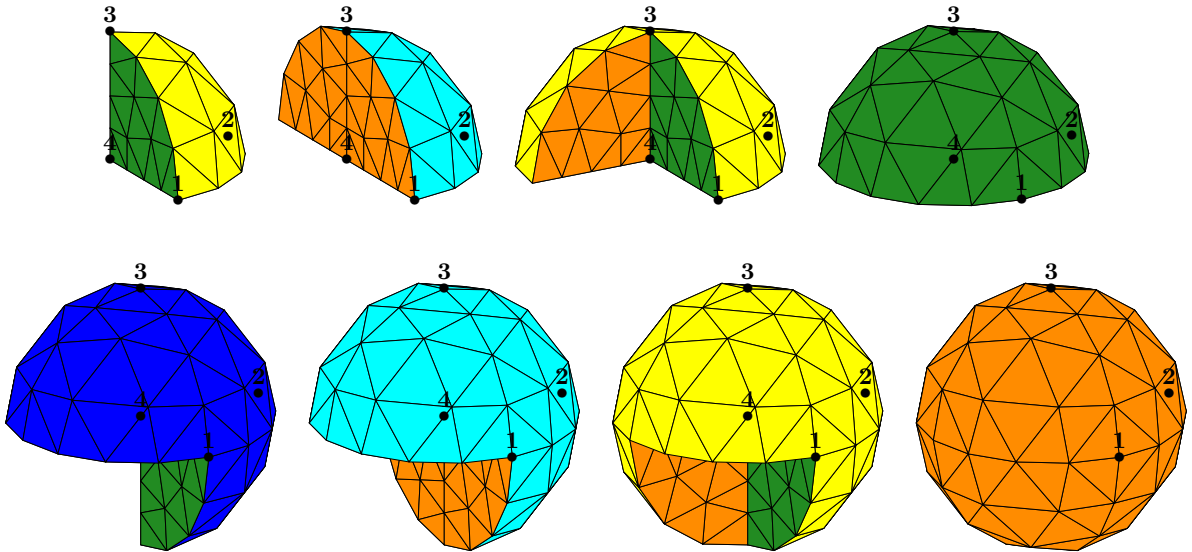


Figure 4.4: Meshes of the different portions of the sphere according to the number of octants.

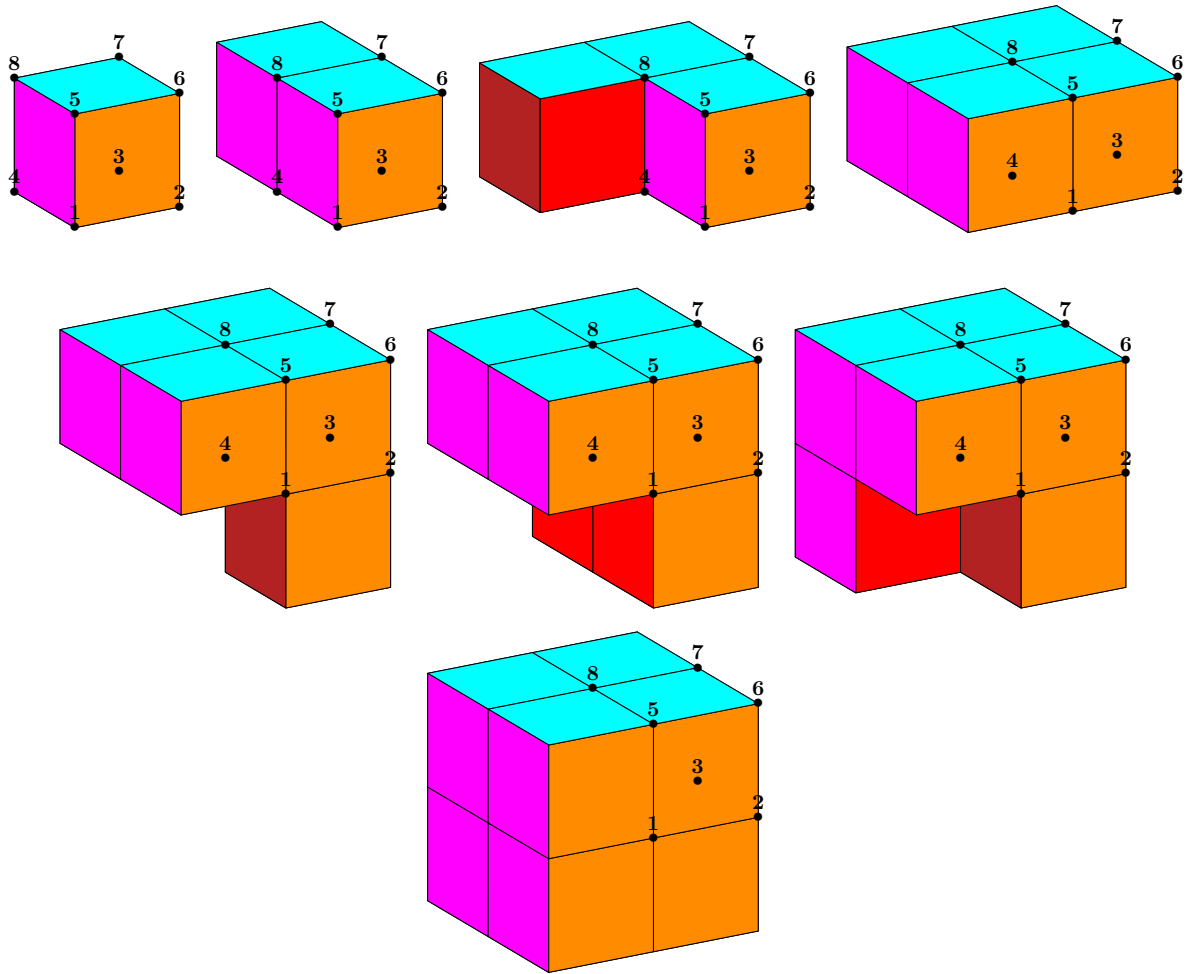


Figure 4.5: Meshes of the different portions of the cube according to the number of octants.

In this way, you call the following underlying methods :

```
void subdvMesh(Ball& sph, const ShapeType shape, const int nbOctants, const
number_t nbsubdiv=0, const number_t order=1, const number_t type=1, const
string_t& name = "", const string_t& TeXFilename = "");
void subdvMesh(Cube& cub, const ShapeType shape, const int nbOctants, const
number_t nbsubdiv=0, const number_t order=1, const string_t& name = "", const
string_t& TeXFilename = "");
void subdvMesh(RevTrunk& cyl, const ShapeType shape, const number_t nbSubDomains=1,
const number_t nbsubdiv=0, const number_t order=1, const number_t type=1, const
string_t& name = "", const string_t& TeXFilename = "");
void subdvMesh(Disk& carc, const ShapeType shape, const number_t nbsubdiv, const
number_t order, const number_t type, const string_t& name, const string_t&
TeXFilename);
void subdvMesh(const std::vector<Point>& pts, const
std::vector<std::vector<number_t>>& elems, const
std::vector<std::vector<number_t>>& bounds, const ShapeType shape, const
number_t nbsubdiv=0, const number_t order=1, const string_t& name = "", const
string_t& TeXFilename = "");
```

Meshes with nested calls to GMSH

This generator enables to mesh whatever geometry, canonical, "composite" or "loop". To use it, you need GMSH installed on your computer before XLife++ is installed, in order to be detected. You can define

meshes as if it was GMSH directly, so that finite elements up to order 5 are allowed here. In this way, you call the underlying external functions :

```
void Mesh::saveToGeo(Geometry& g, number_t order, const string_t& filename);
void Mesh::saveToGeo(Geometry& g, ShapeType sh, number_t order, const string_t&
    filename);
void saveSegmentToGeo(Segment& s, ShapeType sh, std::ofstream& fout,
    std::vector<PhysicalData>& pids, bool withLoopsStorage = true, bool
    withSideNames = true);
void saveEllArcToGeo(EllArc& a, ShapeType sh, std::ofstream& fout,
    std::vector<PhysicalData>& pids, bool withLoopsStorage = true, bool
    withSideNames = true);
void saveCircArcToGeo(CircArc& a, ShapeType sh, std::ofstream& fout,
    std::vector<PhysicalData>& pids, bool withLoopsStorage = true, bool
    withSideNames = true);
void savePolygonToGeo(Polygon& p, ShapeType sh, std::ofstream& fout,
    std::vector<PhysicalData>& pids, bool withLoopsStorage = true, bool
    withSideNames = true);
void saveTriangleToGeo(Triangle& t, ShapeType sh, std::ofstream& fout,
    std::vector<PhysicalData>& pids, bool withLoopsStorage = true, bool
    withSideNames = true);
void saveQuadrangleToGeo(Quadrangle& q, ShapeType sh, std::ofstream& fout,
    std::vector<PhysicalData>& pids, bool withLoopsStorage = true, bool
    withSideNames = true);
void saveEllipseToGeo(Ellipse& e, ShapeType sh, std::ofstream& fout,
    std::vector<PhysicalData>& pids, bool withLoopsStorage = true, bool
    withSideNames = true);
void savePolyhedronToGeo(Polyhedron& p, ShapeType sh, std::ofstream& fout,
    std::vector<PhysicalData>& pids, bool withLoopsStorage = true, bool
    withSideNames = true);
void saveTetrahedronToGeo(Tetrahedron& t, ShapeType sh, std::ofstream& fout,
    std::vector<PhysicalData>& pids, bool withLoopsStorage = true, bool
    withSideNames = true);
void saveHexahedronToGeo(Hexahedron& h, ShapeType sh, std::ofstream& fout,
    std::vector<PhysicalData>& pids, bool withLoopsStorage = true, bool
    withSideNames = true);
void saveEllipsoidToGeo(Ellipsoid& e, ShapeType sh, std::ofstream& fout,
    std::vector<PhysicalData>& pids, bool withLoopsStorage = true, bool
    withSideNames = true);
void saveTrunkToGeo(Trunk& t, ShapeType sh, std::ofstream& fout,
    std::vector<PhysicalData>& pids, bool withLoopsStorage = true, bool
    withSideNames = true);
void saveCylinderToGeo(Cylinder& c, ShapeType sh, std::ofstream& fout,
    std::vector<PhysicalData>& pids, bool withLoopsStorage = true, bool
    withSideNames = true);
void saveConeToGeo(Cone& c, ShapeType sh, std::ofstream& fout,
    std::vector<PhysicalData>& pids, bool withLoopsStorage = true, bool
    withSideNames = true);
void saveRevTrunkToGeo(RevTrunk& t, ShapeType sh, std::ofstream& fout,
    std::vector<PhysicalData>& pids, bool withLoopsStorage = true, bool
    withSideNames = true);
void saveRevCylinderToGeo(RevCylinder& c, ShapeType sh, std::ofstream& fout,
    std::vector<PhysicalData>& pids, bool withLoopsStorage = true, bool
    withSideNames = true);
void saveRevConeToGeo(RevCone& c, ShapeType sh, std::ofstream& fout,
    std::vector<PhysicalData>& pids, bool withLoopsStorage = true, bool
    withSideNames = true);
```


Mesher from extrusion

The extrusion constructor can extrude any 1D or 2D mesh in the \vec{OD} direction using nbl layers of same width (regular extrusion). More precisely, any point M of the section mesh is extruded in nbl points :

$$M_k = M + O + k * \vec{OD}.$$

When a 1D section, extruded mesh is made with quadrangles. When a 2D triangular mesh section, extruded mesh is made with prisms and when a 2D quadrangular mesh section, extruded mesh is made with hexahedra.

```
Mesh(const Mesh& ms, const Point& O, const Point& D , number_t nbl,
      number_t namingDomain=0, number_t namingSection=0, number_t namingSide=0,
      const string_t& meshName=""); //constructor
void buildExtrusion(const Mesh& ms, const Point& O, const Point& D,
                    number_t nbl, number_t namingDomain, number_t namingSection,
                    number_t namingSide, const string_t& meshName); //effective
```

The boundary domains created by the extrusion process come from the boundary domains of the original section. This process is controlled by the 3 parameters `namingDomain`, `namingSection`, `namingSide` taking one of the values 0, 1 or 2, with the following rules:

- 0 : domains are not created
- 1 : one extruded domain is created for any domain of the original section
- 2 : for each layer, one extruded domain is created for any domain of the original section

Be cautious, the side domains of extruded domain are created from side domains of the given section. Thus, if the given section has no side domains, the extruded domains will have no side domains! The naming convention is the following:

- Domains and side domains keep their name with the extension "_e" or "_e1", "_e2", ..., "_en"
- Section domains have the name of original domains with the extension "_0", ..., "_n"
- When `namingDomain=0`, the full domain is always created and named "Omega".

The figure 4.6 illustrates the naming rules of domains.

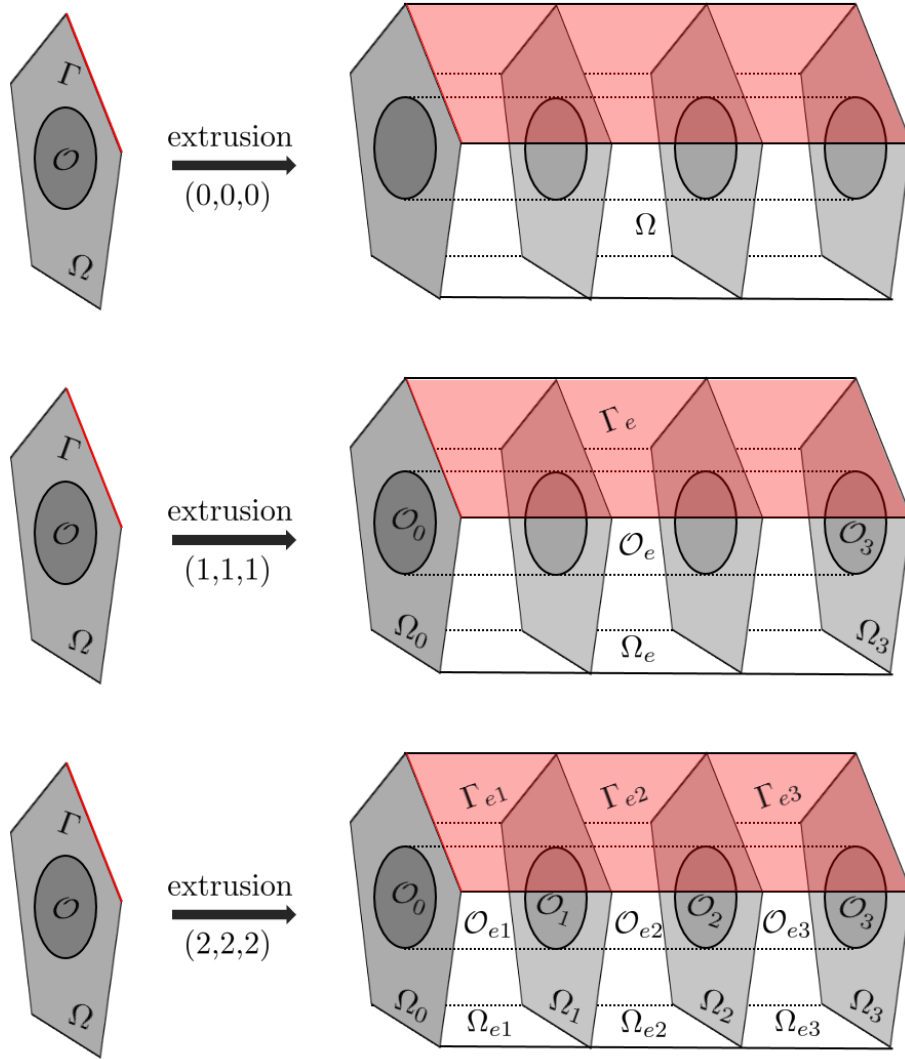


Figure 4.6: Mesh extrusion, domains naming

Note that the `meshPr1Parallelepiped` function uses this extrusion process.

Mesher from element splitting

Sometimes it may be useful to split elements into elements of another type, for instance to produce a mesh of pyramids that are not provided by standard meshing software. To do this, a general constructor is offered :

```
Mesh(const Mesh& mesh, ShapeType sh, const string_t name="");
```

that calls the effective functions that process the splitting.

Up to now, only one splitting function is available:

```
void buildPyramidFromHexadron(const Mesh& hexMesh, const string_t& meshName="");
```

that produces a pyramid mesh from an hexahedron mesh.

Note that the `meshPy1Parallelepiped` function uses this splitting function.

library :	geometry
header :	Mesh.hpp
implementation :	Mesh.cpp, subdivision/SubdvMesh.cpp
unitary tests :	test_Mesh.cpp
header dependences :	config.h, utils.h, GeomElement.hpp, Domain.hpp, Geometry.hpp

4.3 Domain management

Main objects in variational formulation are integrals over a geometrical domain. Geometrical domains can be any subset of any dimension of the computational geometrical domain, usually given by a mesh. But they can also be defined from analytical parametrisations. Besides, it may be useful to consider unions of domains and sometimes, intersections of domains. Such "composite" domains are involved when evaluating linear combination of integrals over different domains. To deal with all this kind of geometrical domains, the following classes are provided:

- **MeshDomain** class to deal with domain defined from a list of mesh elements,
- **CompositeDomain** class to deal with domain defined as an union or an intersection of MeshDomains,
- **AnalyticalDomain** class to deal with domain defined from parametrisations (not yet implemented).
- **PointsDomain** class to deal with domain managing only a list of Points (cloud)

These classes inherit from the **GeomDomain** base class.

In order for the end user to deal with only a single class (**GeomDomain**), we use the "abstract/non abstract" pattern: **GeomDomain** has a **GeomDomain*** member attribute that points either to a child (end user object) or to itself (child object). Constructors of **GeomDomain** class construct child objects and member functions (either virtual or not) are interfaces with children.

4.3.1 The GeomDomain base class

The **GeomDomain** base class has only two pointers as member attributes, one for general information and the other for child

```
class GeomDomain
{protected :
    DomainInfo* domainInfo_p;    // pointer to domain information class
    GeomDomain* domain_p;        // pointer to its child or itself if a child
    static std::vector<const GeomDomain*> theDomains; // list of all domains
    ...
}
```

where **DomainInfo** is the simple class:

```
class DomainInfo
{public :
    String name;                // name of domain
    Dimen dim;                  // dimension of domain (the max when compositeDomain)
    DomainType domType;         // type of domain
    const Mesh* mesh_p;         // pointer to mesh
    String description;         // additionnal information
    DomainInfo(const String&, Dimen, DomainType, const Mesh*, const String& = "");
};
```

The type of domain is defined by the enumeration:

```
enum DomainType {_undefDomain = 0, _analyticDomain, _meshDomain,
                 _compositeDomain, _pointsDomain}
```

In order to not have clones of domains, created domains are listed in the static vector `theDomains`.



There is no reason to instantiate child objects without using base class constructors like. Thus, only base class objects are collected in the static vector `theDomains` and not child objects!

The `GeomDomain` base class provides a basic constructor and child constructors like that construct a child object in memory and store its pointer in the `domain_p` attribute:

```
GeomDomain(const String & na="", dimen_t d=0, const Mesh* m=0);
GeomDomain(const Mesh&, const String&, Dimen, const String&=""); //build meshdomain
GeomDomain(SetOperationType sot,
           const GeomDomain&, const GeomDomain&,
           const String & na=""); //build composite domain with two domains
GeomDomain(SetOperationType sot,
           const std::vector<const GeomDomain*>&,
           const String & na=""); //build composite domain with n domains
~GeomDomain(); //destructor
```

The set operations available are listed in the `SetOperationType` enumeration:

```
enum SetOperationType {_union, _intersection};
```

The class has the following accessors, some to access to `DomainInfo` attributes, the others to access to child attributes through polymorphism behaviour:

```
const String& name() const;
Dimen dim() const;
const DomainType domType() const;
const Mesh* mesh() const;
const String& description() const;
virtual const MeshDomain* meshDomain() const; // non const also
virtual const CompositeDomain* compositeDomain() const; // non const also
virtual const AnalyticalDomain* analyticalDomain() const; // non const also
```

For instance, the virtual accessor `meshDomain()` has the following implementation in the base class:

```
const MeshDomain* GeomDomain::meshDomain() const
{ if(domain_p!=this) return domain_p->meshDomain();
  error("domain_notmesh");
  return 0;
}
```

and the `meshDomain()` function in the `MeshDomain()` child class is:

```
const MeshDomain* MeshDomain::meshDomain() const {return this;}
```

Besides, some useful member functions are provided

```
void rename(const String&) ; //rename domain
void addSuffix(const String&); //add a suffix to domain name
Dimen spaceDim() const; //return the space dimension
const String domTypeName() const; //return domain type name
virtual bool isUnion() const; //true if union
virtual bool isIntersection() const; //true if intersection
virtual bool include(const GeomDomain&) const; //true if includes a domain
virtual Number numberOfElements() const; // number of geometric elements
virtual GeomElement* element(Number); //access to k-th element (k>=1)
```

```
virtual void setMaterialId(Number); //set material id (>0) for all elements
virtual void setDomainId(Number); //set a domain id for all elements
virtual void clearGeomMapData(); //clear geometric data
virtual const GeomDomain* largestDomain(std::vector<const GeomDomain*>) const;
```

There are some static member functions to search domains in the domain list and to print the domain list:

```
static const GeomDomain* findDomain(const GeomDomain*);
static const GeomDomain* findDomain(SetOperationType,
                                     const std::vector<const Domain*>&);
static void printTheDomains(std::ostream&);
```

To manage the union of domains, the following functions are proposed:

```
const GeomDomain* geomUnionOf(std::vector<const GeomDomain*>&);
GeomDomain& merge(const std::vector<GeomDomain*>&, const String& name); //merge n
domains
GeomDomain& merge(GeomDomain&, GeomDomain&, const String& name); //merge 2 domains
...
```

The `geomUnionOf` function constructs (or identifies) the symbolic union of domains, say in the meaning of composite domains, whereas `merge` functions create a true union of domains by merging list of elements. Finally, the class provides print facilities:

```
virtual void GeomDomain::print(std::ostream&) const;
std::ostream& operator<<(std::ostream&, const GeomDomain&);
```

4.3.2 The MeshDomain child class

The `MeshDomain` child class handles a domain defined as a list of mesh elements, say a list of `GeomElement` objects that is a basic stone of a mesh.

```
class MeshDomain : public GeomDomain
{ public :
    vector<GeomElement*> geomElements; //list of geometric elements
    ...
}
```

It manages other important informations :

```
protected :
    MeshDomain* parent_p; //parent mesh domain if submesh domain
public :
    set<ShapeType> shapeTypes; //list of element shape types in mesh domain
    const MeshDomain* extensionof_p; //side domain pointer extension pointer
    mutable pair<OrientationType, const GeomDomain*> normalOrientationRule;
    //orientation rule
```

`shapeTypes` is the list of element shape types contains at least one item and more than one if the mesh domain is a mixture of different elements. For instance if there are some triangles and quadrangles, the `shapeTypes` list has two items (`_triangle`, `_quadrangle`).

The `MeshDomain` pointer `parent_p` gives a link to its parent domain when it is a side domain.

The `MeshDomain` pointer `extensionof_p` gives a link to its "child" when domain is an extension of a side domain. The extension of a side domain is the set of elements having a side (edge/face) located on the side domain. Such domains are required to compute boundary terms that involves non tangential derivative, for instance:

$$\int_{\Gamma} \partial_x u v.$$

`normalOrientationRule` stores informations about the way the normals of a manifold domain are oriented :

- an orientation type from the enumeration

```
enum OrientationType{ _undefOrientationType , _towardsInfinite ,
                     _outwardsInfinite , _towardsDomain , _outwardsDomain };
```

- a `GeomDomain` pointer providing the domain involved when `_towardsDomain` or `_outwardsDomain` is selected



When not specified, `_towardsInfinite` is chosen for an immersed manifold and `_outwardsDomain` for a boundary (may be hazardous when the boundary is an interface).

Two structures useful to locate element from point, may be also constructed if they are required:

```
mutable std::map<Point, std::list<GeomElement*> > vertexElements;
mutable KdTree<Point> kdtree;
```

The first one gives for any vertex of the domain, the list of elements having this vertex and the second is a kdtree structure (tree of points) allowing to get in a short time the vertex closest to a given point. These structures are built on the fly by interpolation tools.

The class manages also some flags to know what quantities are already computed :

```
mutable bool orientationComputed;
mutable bool jacobianComputed;
mutable bool diffEltComputed;
mutable bool normalComputed;
mutable bool inverseJacobianComputed;
```

This class provides a single basic constructor and the member functions related to virtual member functions of the `GeomDomain` base class:

```
MeshDomain(const Mesh&,const String&,Dimen);
MeshDomain* meshDomain();
const MeshDomain* meshDomain() const;
```

Some accessors and basic tools are provided:

```
MeshDomain* parent() const {return parent_p;};
void setShapeTypes();
virtual MeshDomain* meshDomain();
virtual const MeshDomain* meshDomain() const;
virtual bool isUnion() const;
virtual bool isIntersection() const;
bool isSideDomain() const;
virtual Number numberOfElements() const;
virtual GeomElement* element(Number);
virtual void setMaterialId(Number);
virtual void setDomainId(Number);
```

The `setMaterialId` and `setDomainId` end user functions propagates Ids to all geometric elements of the domain. It is the way to get some physical information in FE computation at a low level.

There are more specific tools related to mesh domains:

```

//node access
std::set<Number> nodeNumbers() const;
std::vector<Point> nodes() const;
//set functions
virtual bool include(const GeomDomain&) const;
bool include(const MeshDomain& d) const;
bool isUnionOf(const std::vector<const GeomDomain*>&)const;
const GeomDomain* largestDomain(std::vector<const GeomDomain*>)const;
//Geometric tool
void setNormalOrientation(OrientationType, const GeomDomain& ) const;
void setNormalOrientation(OrientationType = _undefOrientationType,
                          const GeomDomain* gp=0) const;
void setOrientationForManifold(OrientationType = _towardsInfinite) const;
void setOrientationForBoundary(OrientationType ort = _outwardsDomain,
                              const GeomDomain* =0) const;

void reverseOrientations() const;
void buildGeomData() const;
void clearGeomMapData();
//create domain extension
const GeomDomain* extendDomain() const;
//tools to locate element
void buildVertexElements() const;
void buildKdTree() const;
GeomElement* locate(const Point&) const;

```

The `buildGeomData` function constructs measure and orientation of the elements of the domain.

The orientation of an element is the sign s such that the normal computed times s is the desired normal. When domain is a side domain, the orientations are either constructed using a tracking algorithm (`setOrientationForManifold`) or using a "volumic" algorithm (`setOrientationForBoundary`) when the side domain is the boundary of a "volumic" domain. Contrary to most FE softwares, XLIFFE++ does not assume that the numbering of nodes provided by meshing tools gives the outward normals! So the normal are always oriented by XLIFFE++.



The computation of normal vector orientations is not done when mesh is generated or loaded but it is done during the construction of a space involving a manifold or during the computation of a Term involving normal vectors.

The `locate` function is a powerful tool that returns one of the elements including a given point (null pointer if no element). It is used in all interpolation operations. It is wellknown that this operation is time expansive. This is the reason why additional structures `vertexElements` and `kdtree` may be constructed if point location is required. This structures are constructed only once by `buildVertexElements` and `buildKdTree` functions at the first call of `locate`. The time to locate a point is generally (may be worth) in $\log(n)$ (n the number of vertices).

Some print/export stuff is available:

```

virtual void print(std::ostream&) const;
friend std::ostream& operator<<(std::ostream&, const MeshDomain&);
void saveNormalsToFile(const string_t&, IOFormat iof=_vtk) const;

```

4.3.3 The CompositeDomain child class

The `CompositeDomain` child class describes set operations (union and intersection) of geometrical domains (`Domain` objects). It is an abstract description that it means that elements lists of `MeshDomain` objects are not merged or intersected!

```

class CompositeDomain : public GeomDomain
{protected :
    SetOperationType setOpType_;           //type of the set operation
    std::vector<const Domain*> domains_;    //list of domains
    ...

```

`SetOperationType` enumerates `_union` and `_intersection` values.

This class provides a single basic constructor, an accessor to the set operation and the member functions related to virtual member functions of the `GeomDomain` base class:

```

CompositeDomain(SetOperationType, const std::vector<const GeomDomain*>&,const
    String&);
SetOperationType setOpType() const;
const std::vector<const GeomDomain*>& domains() const;
CompositeDomain* compositeDomain();
const CompositeDomain* compositeDomain();
bool isUnion() const;
bool isIntersection() const;
std::vector<const GeomDomain*> basicDomains() const;    //recursive
virtual Number numberOfElements() const;
void print(std::ostream&) const;

```

The `basicDomains()` function is recursive and provides the list of all basic domains, that are domains which are not composite domains.

4.3.4 The PointsDomain child class

This class deals with domain only defined by a list of points (cloud):

```

class PointsDomain : public GeomDomain
{
public :
    std::vector<Point> points;
    PointsDomain(const std::vector<Point>&, const String& = "");
    PointsDomain* pointsDomain();
    const PointsDomain* pointsDomain() const;
    const Point& point(Number n) const;
    Point& point(Number n);
    virtual bool include(const GeomDomain&) const;
    virtual void print(std::ostream&) const;
};

```

It may be useful but it is not used in core of the library!

4.3.5 The AnalyticalDomain child class

This class has not be yet implemented (here for future usage).

4.3.6 The Domains class

`Domains` is an alias of `PCollection<GeomDomain>` class that manages a collection of `GeomDomain`, in fact a `std::vector<GeomDomain*>` (see the definition of `PCollection`). It may be used as following:

```

Strings sn("Gamma1", "Gamma2", "Gamma3", "Gamma4");
Mesh mesh2d(Rectangle(_xmin=0, _xmax=0.5, _ymin=0, _ymax=1, _nnodes=Numbers(3,6),
    _domain_name="Omega", _side_names=sn), _triangle, 1, _structured);
Domain omega=mesh2d.domain("Omega"),
    gamma1=mesh2d.domain("Gamma1"), gamma2=mesh2d.domain("Gamma2"),
    gamma3=mesh2d.domain("Gamma3"), gamma4=mesh2d.domain("Gamma4");

```



```

Domains ds3 (gamma1 , gamma2 , gamma3 ) ;
Domains ds4 ;
for (Number i=0; i < 4; i++) ds4 << mesh2d . domain ( i + 1 ) ;
Domains ds5 ( 5 ) ;
for (Number i=1; i <= 5; i++) ds5 ( i ) = mesh2d . domain ( i - 1 ) ;

```

C¹¹ If C++11 is available (the library has to be compiled in C++11), the following syntax is also working:

```

Domains dsi={gamma1 , gamma2 , gamma3 } ;

```

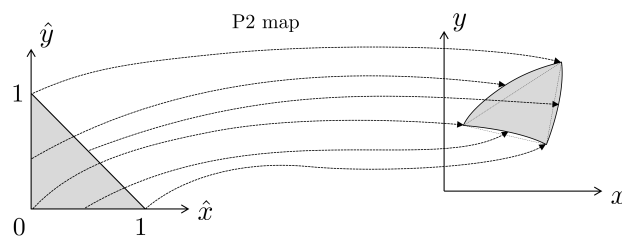
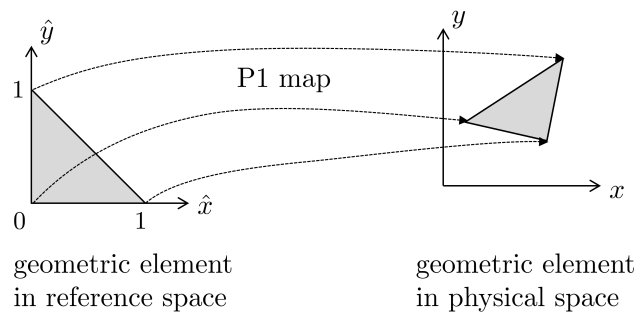
```

library : geometry
header : GeomDomain.hpp
implementation : GeomDomain.cpp
unitary tests : test_Domain.cpp
header dependences : config.h, utils.h

```

4.4 Geometric element

The atomic object of a mesh is a geometric element, that is a segment, a polygon or a polyhedron of physical space with flat or curved faces or edges. A general way to describe it, consists in giving a polynomial interpolation and a reference geometric element in reference space. Thus, a geometric element is the image by the map of a reference geometric element. This map is defined from the polynomial interpolation and some nodes of the geometric element: vertices and extra points regarding the order of the polynomial interpolation.



Geometric elements are the geometric supports of finite element. In other words, a finite element is defined from a geometric element and an interpolation (different from interpolation used in geometric element!), see [Element](#) class in [space](#) library.

As some finite elements may be required on a boundary of domain, geometric element can be a side of geometric element and even a side of side. It is the reason why a geometric element can possibly manage side informations (parent and side number).

4.4.1 The `GeomElement` class

The `GeomElement` class has only four protected members:

```
class GeomElement
{protected :
    const Mesh* mesh_p;           //pointer to the mesh
    Number number_;               //unique id number
    MeshElement* meshElement_p;   //pointer to a MeshElement
    std::vector<GeoNumPair> parentSides_; //if a side element
    ...
}
```

The most important member `meshElement_p` is a pointer to a `MeshElement` object collecting real geometric informations (nodes, numbering, reference element, map data, ...). This class is detailed in the next section.

Besides, it manage some additional informations :

```
Number materialId;    // material id (default= 0)
Number domainId;     // domain id (default=0)
Real theta, phi;      // angles to describe local curvature (default 0,0)
```

`GeoNumPair` is an alias for storing a pair of parent geometric element and a side number:

```
typedef std::pair<GeomElement*,Number> GeoNumPair;
```

When the geometric element is a side of a geometric element, `parentSides_` is not empty and by default, the `meshElement_p` is null. Note that it is possible to build it. Of course, when geometric element is not a side element (say plain element), `parentSides_` is empty.

This class provides few public constructors, an assignment operator and a destructor:

```
GeomElement(const Mesh* m=0, Number n = 0);
GeomElement(const Mesh*, const RefElement*, Dimen, Number); //for plain element
GeomElement(GeomElement*, Number, Number);                 //for side element
GeomElement(const GeomElement&);
GeomElement& operator =(const GeomElement&);
```

some simple accessors:

```
Number number() const;
Number& number();
bool hasMeshElement() const;
bool isSideElement() const;
```

and complex accessors working for plain or side elements, even if `meshElement_p` is null :

```
const MeshElement* meshElement() const;
MeshElement* meshElement();
const GeomElement* parent(Number i=0) const;
const GeoNumPair parentSide(Number i) const;
std::vector<GeoNumPair>& parentSides();
Dimen elementDim() const;
const RefElement* refElement(Number s = 0) const;
const GeomRefElement* geomRefElement(Number s = 0) const;
```

```

Number numberOfVertices() const;
Number numberOfSides() const;
Number numberOfSideOfSides() const;
ShapeType shapeType(Number s = 0) const;
Number vertexNumber(Number i) const;
Number nodeNumber(Number i) const;
std::vector<Number> vertexNumbers(Number s = 0) const;
std::vector<Number> nodeNumbers(Number s = 0) const;

```

`s` means a side number of element and when it is null, function applies to element itself. Most of these member functions are recursive. It allows to get data for plain, side and side of side elements (side of side of side is not managed). Be cautious, these member functions does not check the integrity of data. For instance, the function `meshElement()` may return a null pointer (case of a side element).

For a side element, it is possible to construct, if really necessary, its `MeshElement` structure using the member function:

```

void buildSideMeshElement();
void deleteMeshElement();           // destroy meshElement information

```

There is an encoding function which constructs a string key with global numbers of vertices of the geometric element (`s=0`) or its side (`s`). This function is used to build global lists of sides (edge in 2D or face in 3D), of side of sides (edge in 3D) and vertices (see the `Mesh` class).

```

String encodeElement(Number s=0) const;

```

Related to these global lists, the following functions return temporary lists of adjacent elements by side, by side of side or by vertex:

```

GeoNumPair elementsSharingSide(Number s) const;
std::vector<GeoNumPair> elementsSharingVertex(Number v, bool o=false) const;
std::vector<GeoNumPair> elementsSharingSideOfSide(Number s, bool o=false) const;

```

When the boolean argument `o` is true, it means that elements in lists are adjacent **only** by side of side or only by vertex. When it is false, all elements are listed. Obviously, these functions work only if the global lists of sides, of side of sides and vertices have been constructed before. By default, they are not (see the `Mesh` class).

The class provides a function to test if a point belongs to element:

```

bool contains(const std::vector<Real>& p);

```

It is possible to compare `GeomElement` objects by their indices (`number_`) using the operators:

```

bool operator== (const GeomElement&, const GeomElement&);
bool operator<  (const GeomElement&, const GeomElement&);

```

For output purpose, you can split a `GeomElement` in first order elements, either simplices or of same shape :

```

std::vector<GeomElement*> splitP1() const;
std::vector<GeomElement*> splitO1() const;

```

Finally, the `GeomElement` class provides usual printing facilities:

```

void print(std::ostream&) const;
friend std::ostream& operator<< (std::ostream&, const GeomElement&);

```

4.4.2 The MeshElement class

MeshElement is the real class storing geometric element data:

```
class MeshElement
{public:
    std::vector<Point*> nodes;           //nodes of element
    std::vector<Number> nodeNumbers;     //global numbers of node
    std::vector<Number> vertexNumbers;   //global numbers of vertices
    std::vector<Number> sideNumbers;     //global numbers of sides
    std::vector<Number> sideOfSideNumbers; //global numbers of side of sides
    short int orientation;              //element orientation
    std::vector<Real> measures;          //measure of element and its sides
    GeomMapData* geomMapData_p;         //useful data for geometric map
private:
    Number index_;                      //element index
    const Dimen spaceDim_;              //space dimension
    const RefElement* refElt_p;         //pointer to associated Reference Element
    ...
}
```

The vector **nodes** gives a direct access to the node coordinates of the element (it is redundant). The numbering vectors **nodeNumbers**, **vertexNumbers**, **sideNumbers** and **sideOfSideNumbers** are related to global vectors defined in **Mesh** ; **nodes**, **vertices_**, **sides_** and **sideOfSide_**. Note that the **sideNumbers** and **sideOfSideNumbers** vectors are not built by default (see the **buildSides** and **buildSideOfSides** member functions of **Mesh** class). The **nodeNumbers** and **vertexNumbers** vectors are the same for one order polynomial geometric interpolation.

The orientation of an element is defined by default as the sign of the determinant of the jacobian of the geometric map from reference element to geometric element; this map being well defined through the **RefElement** pointer **refElt_p** and the geometric element nodes. But, this orientation may be changed by the function **setOrientationForManifold** which set the orientation in order the normal computed from jacobian be always the outward normal.

Besides, the **MeshElement** class manages a pointer to a **GeomMapData** object storing computational data related to the geometric map: jacobian matrix, its inverse, its determinant, ... see the next section.



As it refers to some **Mesh** class members, **MeshElement** object should not be instantiated independantly of a mesh.

The class has only one basic constructor initializing the **RefElement** pointer, the space dimension and the element index:

```
MeshElement();
MeshElement(const RefElement*, Dimen, Number);
~MeshElement();
```

It means that numbering informations has to be given outside the constructor. Generally, this is done by meshing tools.

There are some useful accessors, most of them providing same information as those defined in the **GeomElement** interface class:

```
Number index() const;
Dimen spaceDim() const;
Number order() const;
Real measure(const Number s = 0) const;
const RefElement* refElement(Number s = 0) const;
const GeomRefElement* geomRefElement(Number i = 0) const;
```

```

Dimen elementDim() const;
ShapeType shapeType(Number s = 0) const;
Number vertexNumber(Number i) const;
Number nodeNumber(Number i) const;
std::vector<Number> verticesNumbers(Number s=0) const;
Number numberOfNodes() const;
Number numberOfVertices() const;
Number numberOfSides() const;
Number numberOfSideOfSides() const;

```

Besides, there are some computation functions related to geometric element:

```

void computeMeasures();
void computeMeasure();
void computeMeasureOfSides();
void computeOrientation();
bool contains(const std::vector<Real>&);
void clearGeomMapData();

```

The functions related to jacobian computations are defined in the `GeomMapData` class. The `geomMapData_p` pointer may be used to store map data and the function `clearGeomMapData()` deletes the `GeomMapData` object.

Finally, `MeshElement` informations are displayed with:

```

void print(std::ostream&) const;
friend std::ostream& operator<< (std::ostream&, const MeshElement&);

```

library :	geometry
header :	GeomElement.hpp
implementation :	GeomElement.cpp
unitary tests :	test_GeomElement.cpp
header dependences :	config.h, utils.h

4.5 Geometric map data

The `GeomMapData` class is helpful to compute and store jacobian matrices and various values related to the map from a reference element to a geometric element.

4.5.1 Differential calculus

Mapping

The map from Lagrange reference element (say \hat{E}) to any Lagrange geometric element (say E) is defined by

$$F(\hat{x}) = \sum_{k=1,q} M_k \hat{\tau}_k(\hat{x})$$

where $(M_k)_{k=1,q}$ are the nodes of geometric element E and $(\hat{\tau}_k)_{k=1,q}$ are the shape functions of reference element \hat{E} .

Note that \hat{E} and E may be not in the same dimension space, think to a triangle in 3D space or a segment in 2D or 3D space. Denote n the space dimension (the dimension of point M_i) and p the dimension of element (the dimension of point of reference element and variable \hat{x} too). So the map F acts from \mathbb{R}^p to \mathbb{R}^n . If the element E is not degenerated (its measure as element of dimension p

is not null), F is injective and C^1 , by construction $E = F(\widehat{E})$, thus F is a diffeomorphism from \widehat{E} onto E .

For a side S of element E , image of \widehat{S} by map F , the restriction of F on \widehat{S} is defined by:

$$F_S(\widehat{x}) = \sum_{k / M_k \in S} M_k \widehat{\tau}_k(\widehat{x}).$$

Differential element in integrals

The $n \times p$ jacobian matrix is defined by $(\widehat{x} = (\widehat{x}_1, \dots, \widehat{x}_p))$:

$$J_{ij}(\widehat{x}) = \sum_{k=1,q} (M_k)_i \partial_{\widehat{x}_j} \widehat{\tau}_k(\widehat{x})$$

- When $n = p$ it is assumed that the Jacobian matrix is invertible (no degenerate element) and J^{-1} denotes its inverse.
- When $n = p + 1$, it is assumed that the columns of jacobian $(J_{.j}(\widehat{x}))_{j=1,p}$ form a basis of the tangential plane of the element at point \widehat{x} . There exists a $p \times p$ sub matrix of $J(\widehat{x})$ invertible, say $\widetilde{J}(\widehat{x})$.
- When $n = p + 2$, only in 3D ($n = 3$ and $p = 1$), the unique column of jacobian $(J_{.1}(\widehat{x}))$ is a tangential vector of the element at point \widehat{x} and at least, one component of $J_{.1}(\widehat{x})$ is not null.

Most of finite element computation are integrals over E involving the mapping to the reference element (g an integrable function):

$$\int_E g(x) dx = \int_{\widehat{E}} g \circ F(\widehat{x}) \gamma(\widehat{x}) d\widehat{x}$$

where $\gamma(\widehat{x})$ denotes the differential element defined by:

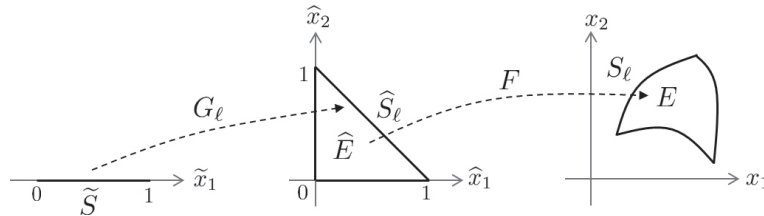
$$\gamma(\widehat{x}) = \begin{cases} |\det J(\widehat{x})| & \text{if } p = n \\ |\vec{n}(\widehat{x})| & \text{if } p = n - 1 \\ |J_{.1}(\widehat{x})| & \text{if } p = n - 2 = 1 \end{cases}$$

where $\vec{n}(\widehat{x})$ denotes the following normal to the element (oriented area):

$$\vec{n}(\widehat{x}) = \begin{cases} J_{.1}(\widehat{x}) \times J_{.2}(\widehat{x}) & \text{if } p = n - 1 = 2 \\ J_{.1}(\widehat{x}) \perp & \text{if } p = n - 1 = 1 \end{cases}$$

Integrals over a side S_ℓ of element E involve the mapping from a side reference element \widetilde{S} :

$$S_\ell = H_\ell(\widetilde{S}) = F \circ G_\ell(\widetilde{S})$$



where G_ℓ maps the side reference element \widetilde{S} to the side ℓ of the reference element \widehat{E} , say \widehat{S}_ℓ . As reference elements are flat elements, G_ℓ may be chosen as a first order polynomial map from \mathbb{R}^{n-1} to \mathbb{R}^n given by:

$$G_\ell(\widehat{x}) = \sum_{k=1,q} \widehat{M}_{\ell k} \widehat{\tau}_k(\widehat{x}) \quad (\widehat{M}_{\ell k} \text{ vertices of } \widehat{S}_\ell).$$

The mapping of the integral on side S is:

$$\int_S g(x) dx = \int_{\tilde{S}} g \circ F \circ G_\ell(\tilde{x}) \mu(\tilde{x}) d\tilde{x}$$

where the differential element is given by:

$$\mu(\tilde{x}) = \begin{cases} |(J_{H_\ell})_{.1} \times (J_{H_\ell})_{.2}| & \text{if } n = 3 \\ |(J_{H_\ell})_{.1}| & \text{if } n = 2 \end{cases}.$$

As $J_{H_\ell} = J_F J_{G_\ell}$, the differential element $\mu(\tilde{x})$ may be written :

$$\mu(\tilde{x}) = \begin{cases} |J_F(J_{G_\ell})_{.1} \times J_F(J_{G_\ell})_{.2}| & \text{if } n = 3 \\ |J_F(J_{G_\ell})_{.1}| & \text{if } n = 2 \end{cases}.$$

An other way to get the differential element, consist in introducing the side reference element with the same order polynomial interpolation as the reference element and the map:

$$\tilde{H}_\ell = \sum_{k=1,q} M_{\ell k} \tilde{\tau}_k \quad (M_{\ell k} \text{ nodes of } S_\ell)$$

with $(\tilde{\tau}_k)$ the shape functions of the side reference element, differents from $(\tilde{\tau}_k)$ if the interpolation is not of order 1. As

$$H_\ell(\tilde{x}) = \sum_{k / M_k \in S_\ell} M_k \hat{\tau}_k(G_\ell(\tilde{x})) = \sum_k M_{\ell k} \hat{\tau}_{\ell k}(G_\ell(\tilde{x}))$$

and $\hat{\tau}_{\ell k}(G_\ell(\tilde{x})) = \tilde{\tau}_k$, maps H_ℓ and \tilde{H}_ℓ are the same.

The jacobian is given by:

$$(J_{H_\ell})_{ij} = \sum_{k=1,q} (M_{\ell k})_i \partial_{\tilde{x}_j} \tilde{\tau}_k$$

and the differential element is given by:

$$\mu(\tilde{x}) = \begin{cases} |(J_{H_\ell})_{.1} \times (J_{H_\ell})_{.2}| & \text{if } n = 3 \\ |(J_{H_\ell})_{.1}| & \text{if } n = 2 \end{cases}.$$

Gradient

Sometimes, differential operators may appear in integrals. For any differentiable function h defined on E , we set $\hat{h} = h \circ F$. The following relations holds

- "volumic" computation, $p = n$:

$$\nabla h = J(\hat{x})^{-t} \hat{\nabla} \hat{h}$$

- surfacic gradient, $p = n - 1 = 2$:

$$\nabla_S h = J(\hat{x}) T(\hat{x})^{-1} \hat{\nabla} \hat{h}$$

where $T(\hat{x})$ is the $p \times p$ metric tensor defined by:

$$T(\hat{x}) = J(\hat{x})^t J(\hat{x}).$$

Note that $\nabla_S h$ is a n vector.

- lineic gradient, $p = n - 2 = 1$ or $p = n - 1 = 1$:

$$\nabla_L h = \frac{\hat{h}'}{|J_{.1}(\hat{x})|} \frac{J_{.1}(\hat{x})}{|J_{.1}(\hat{x})|} = \frac{\hat{h}'}{|J_{.1}(\hat{x})|} \tau$$

In this expression, $\nabla_L h$ is a n vector. The scalar tangential derivative is thus given by

$$\partial_\tau h = \frac{\hat{h}'}{|J_{.1}(\hat{x})|}.$$

4.5.2 The GeomMapData class

In view of main differential calculus formulas, the `GeomMapData` class has the following members:

```
class GeomMapData
{private :
    const MeshElement* geomElement_p;    //current geometric element
    Point currentPoint;                  //point used in computation
public :
    Matrix<Real> jacobianMatrix;          //jacobian matrix of map from reference
        element
    Matrix<Real> inverseJacobianMatrix;    //inverse jacobian matrix
    Real jacobianDeterminant;             //jacobian determinant
    Real differentialElement;             //differential element (abs(jac. det.))
    Vector<Real> normalVector;           //unit outward normal vector at a boundary
        point
    Vector<Real> metricTensor;            //symetric metric tensor (t_i | t_j)
    Real metricTensorDeterminant;         //metric_tensor determinant
    ...
};
```

This class provides three basic constructors and some computation functions:

```
GeomMapData(const MeshElement*, const Point&);
GeomMapData(const MeshElement*, std::vector<Real>::iterator);
GeomMapData(const MeshElement*);
void computeJacobianMatrix(Number side=0);
void computeJacobianMatrix(const std::vector<Real>& x, Number s=0);
Real computeJacobianDeterminant();
void invertJacobianMatrix();
void computeNormalVector();
void normalize();
void computeDifferentialElement();
Real diffElement();
Real diffElement(Number s);
void computeMetricTensor();
void computeSurfaceGradient(Real, Real, std::vector<Real>&);
```

Note that these computations requires the evaluation of the shape functions of the reference element at a given point. This evaluation is done by the three following functions :

```
Point geomMap(Number side = 0);
Point geomMap(const std::vector<Real>& x, Number side = 0);
Point piolaMap(Number side = 0);
```

library :	geometry
header :	GeomMapData.hpp
implementation :	GeomMapData.cpp
unitary tests :	test_GeomElement.cpp
header dependences :	config.h, utils.h, GeomElement.hpp

The *finiteElements* library concerns all the stuff related to the elementary objects of a finite elements discretization. When these objects are provided by a mesh tool, they are *GeomElement* (for geometric element) and when they also carry interpolation information, they are *Element*. In few words, a *Mesh* is a collection of *GeomElement* and a finite element *Space* is a collection of *Element*. In the framework of the finite elements, it is common (and useful) to link any element to a reference element, named in this library: *GeomRefElement* and *RefElement*. *GeomRefElement* is a class whom inherit fundamental geometric elements *GeomSegment*, *GeomTriangle*, ... and *RefElement* is a class whom inherit all the finite element types *LagrangeSegment*, *LagrangeTriangle*, *LagrangeTetrahedron*, The *Interpolation* class carries some general informations on the finite element such that the finite element type and its conforming space. Besides, this library provides integration method stuff (*IntegrationMethod*, *Quadrature*, *QuadratureRule*).

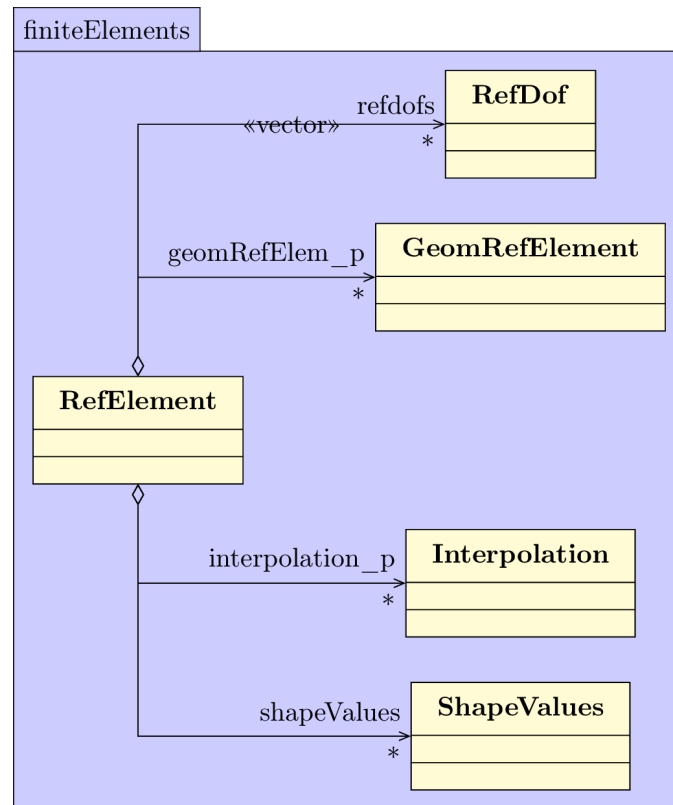


Figure 5.1: The *finiteElements* library main class diagram

5.1 The Interpolation class

The purpose of the *Interpolation* class is to store data related to finite element interpolation. It concerns only general description of an interpolation: type of the finite element interpolation (Lagrange, Hermite, ...), a subtype among standard, Gauss-Lobatto (only for Lagrange) and first or second family (only for Nedelec), the "order" of the interpolation for Lagrange family and the space conformity (L2, H1, Hdiv, ...). For instance, a Lagrange standard P1 with H1 conforming is the classic P1 Lagrange

interpolation but a Lagrange standard P1 with L2 conforming means a discontinuous interpolation (like Galerkin discontinuous approximation) where the vertices of an element are not shared. For the moment, this class is only intended to support the following choices:

type	subtype	order
Lagrange	standard, Gauss-Lobatto	any order
Hermite	standard	
Crouzet-Raviart	standard	order 1
Raviart-Thomas	standard	any order
Nedelec	first or second family	any order
Nedelec face	first or second family	any order
Nedelec edge	first or second family	any order



Nedelec face and Nedelec edge are specific to 3D. Nedelec face corresponds to the 2D Raviart-Thomas family and Nedelec edge corresponds to the 2D Nedelec family.



The **Interpolation** class is intended to support a lot of finite element family but all are not available. See child classes of **RefElement** class to know which families are actually available.

This class is mainly used with the **GeomElement** class (definition of the geometric transformation) and the **Space** class (definition of the finite element approximation related to the discretized space).

The class **Interpolation** has the following attributes:

```
class Interpolation
{private:
    const FEType type_;          //!interpolation type
    const FESubType subtype_;    //!interpolation subtype
    const number_t numtype_;     //!additionnal number type (degree for Lagrange)
    SobolevType conformSpace_;   //!conforming space
    bool isoparametric_;         //!isoparametric flag
    bool isoparametric_;         //!isoparametric flag
    String name_;                //!name of the interpolation type
    String subname_;             //!interpolation sub_name
    ...
};
```

FEType, **FESubType** and **SpaceType** are enumerations defined in the *config.hpp* file as follow:

```
enum SobolevType {L2=0,H1,Hdiv,Hcurl,Hrot=Hcurl,H2,Hinf};
enum FEType      {Lagrange,Hermite,CrouzeixRaviart,Nedelec,RaviartThomas,
                  NedelecFace,NedelecEdge};
enum FESubType   {standard=0,gaussLobatto,firstFamily,secondFamily};
```

In addition, there are a static attribute to manage a unique list of instantiated **Interpolation** objects.

```
static std::vector<Interpolation*> theInterpolations;
```

This class proposes some access member functions and functions returning some interpolation properties:

```
FEType type() const {return type_;}
FESubType subtype() const {return subtype_;}
number_t numtype() const {return numtype_;}
SobolevType conformSpace() const {return conformSpace_;}
String name() const {return name_;}
```

```
String subName() const {return subname_;}
String conformSpaceName() const;
bool isIsoparametric() const {return isoparametric_;}
void isIsoparametric(bool is) {isoparametric_ = is;}
bool isContinuous();
bool areDofsAllScalar();
number_t maximumDegree();
```

some error member functions:

```
void badType() const;
void badSubType() const;
void badSpace() const;
void badType(const number_t) const;
void badSubType(const number_t) const;
void badDegree(const number_t) const;
```

There is only one explicit constructor (no copy constructor):

```
Interpolation(FEType, FESubType, number_t, SobolevType);
```

that initializes the attributes and add the created object in the static list **theInterpolations**.

The process creating a **Interpolation** object is governed by other classes (in particular the **Space** class) using the function:

```
Interpolation* findInterpolation(FEType, FESubType, number_t, SpaceType);
Interpolation& interpolation(FEType, FESubType, number_t, SobolevType);
```

These functions try to find an existing **Interpolation** object corresponding to the given parameters, if it does not exist one, a new object is created by calling the constructor.

It is possible to work with a collection of **Interpolation** objects that are handles as a vector by the **Interpolations** class (see the definition of **PCollection**). It may be used as following:

```
Interpolations ints(3);
for (Number i=1; i<=3; i++) ints(i)=interpolation(Lagrange, _standard, i, H1);
```

library :	finiteElements
header :	Interpolation.hpp
implementation :	Interpolation.cpp
unitary tests :	test_Interpolation.cpp
header dependencies :	config.h, utils.h

5.2 The RefDof class

The **RefDof** class is a class representing a generalized degree of freedom in a reference element. It is defined by:

- A support that can be a point, a side, a side of side or the whole element. When it is a point, we also have its coordinates.
- A dimension, that means the number of components of shape functions of the DoF.

- Projections data (type and direction) when the DoF is a projection DoF. The projection type is 0 for a general projection, 1 for a u_n DoF, 2 for a $u \times n$ DoF.
- Derivative data (order and direction) when the DoF is a derivative DoF. The order is 0 when the DoF is defined by an 'integral' over its support, or n for a Hermite DoF derivative of order $n > 0$ or a moment DoF of order $n > 0$.
- A shareable property: A DoF is shareable when it is shared between adjacent elements according to space conformity. Such a DoF can be shared or not according to interpolation. For example, every DoF of a Lagrange H1-conforming element on a side is shared, while DoF in discontinuous interpolation are not.

```
class RefDof
{
private:
    bool sharable_;           //D.o.F is shared according to space conformity
    DofLocalization where_;   //hierarchic localization of dof
    number_t supportNum_;     //support localization
    number_t index_;          //rank of the dof in its localization
    dimen_t supportDim_;      //dimension of the D.o.F geometric support
    number_t nodeNum_;        //node number when a point dof
    dimen_t dim_;             //number of components of shape functions of D.o.F
    std::vector<real_t> coords_; //coordinates of D.o.F support ifn supportDim_=0
    number_t order_;          //order of a derivative D.o.F or a moment D.o.F
    std::vector<real_t> derivativeVector_; //direction vector(s) of a derivative
    ProjectionType projectionType_; //type of projection
    std::vector<real_t> projectionVector_; //direction vector(s) of a projection
    string_t name_;           //D.o.F-type name for documentation
}
```

This class proposes mainly a full constructor and accessors to member data.

In order to sort dofs in an efficient way, the following class is offered:

```
class DofKey
{
public:
    DofLocalization where;
    number_t v1, v2;           //used for vertex, edge, element dofs
    std::vector<number_t> vs;   //used for face dofs
    number_t locIndex;         //local dof index
}
```

associated to the comparison operator:

```
bool operator<(const DofKey& k1, const DofKey& k2);
```

library :	finiteElements
header :	RefDof.hpp
implementation :	RefDof.cpp
unitary tests :	test_segment.cpp, test_triangle.cpp, test_quadrangle.cpp,
	test_tetrahedron.cpp, test_hexahedron.cpp, test_prism.cpp
header dependences :	config.h, utils.h

5.3 Geometric data of a reference element

5.3.1 The `GeomRefElement` class

The `GeomRefElement` object carries geometric data of a reference element: type, dimension, number of vertices, sides and side of sides, measure, and coordinates of vertices. Furthermore, to help finding geometric data on sides, it also carries type, vertex numbers of each side on the first hand, and vertex numbers and relative number to parent side of each side of side on the second hand.

```
class GeomRefElement
{
protected:
    ShapeType shapeType_;           //element shape
    const Dimen dim_;               //element dimension
    const Number nbVertices_, nbSides_, nbSideOfSides_; //number of vertices, ...
    const Real measure_;            //length or area or volume
    vector<Real> centroid_;          //coordinates of centroid
    vector<Real> vertices_;          //coordinates of vertices
    vector<ShapeType> sideShapeTypes_; //shape type of each side
    vector<vector<Number>> sideVertexNumbers_;
    vector<vector<Number>> sideOfSideVertexNumbers_;
    vector<vector<Number>> sideOfSideNumbers_;

public:
    static vector<GeomRefElement*> theGeomRefElements; //vector carrying all run time
                                                         GeomReferenceElements
};
```

It offers:

- few constructors (the default one, a general one with dimension, measure, centroid, and number of vertices, edges and faces and one more specific for each dimension):

```
GeomRefElement(); //!< default constructor
GeomRefElement(ShapeType, const Dimen d, const Real m, const Real c,
               const Number v, const Number e, const Number f); //general
GeomRefElement(ShapeType, const Real m = 1.0, const Real c = 0.5); // 1D
GeomRefElement(ShapeType, const Real m, const Real c, const Number v); // 2D
GeomRefElement(ShapeType, const Real m, const Real c, const Number v,
               const Number e); // 3D
```

- a private copy constructor and a private assign operator,
- some public access functions to data:

```
Dimen dim() const { return dim_; }
Number nbSides() const { return nbSides_; }
Number nbSideOfSides() const { return nbSideOfSides_; }
Real measure() const { return measure_; }
std::vector<Real>::const_iterator centroid() const { return centroid_.begin(); }
std::vector<Real>::const_iterator vertices() const { return vertices_.begin(); }
std::vector<ShapeType> sideShapeTypes() { return sideShapeTypes_; }
const std::vector<std::vector<Number>>& sideVertexNumbers() const
{ return sideVertexNumbers_; }
const std::vector<std::vector<Number>>& sideOfSideVertexNumbers() const
{ return sideOfSideVertexNumbers_; }
```

- some public methods to get data on sides and side of sides:

```

String shape(const Number sideNum = 0) const; // returns element (side) shape as a
string
ShapeType shapeType(const Number sideNum = 0) const; //returns element (side) shape
as a ShapeType
bool isSimplex() const; //true if a simplex
Number nbVertices(const Number sideNum = 0) const; // returns number of element
(side) vertices
Number sideVertex(const Number id, const Number sideNum = 0) const; // returns
vertex number of element (side)
std::vector<Real>::const_iterator vertex(const Number vNum) const; // returns
coordinates of n-th vertex
Number sideVertexNumber(const Number vNum, const Number sideNum) const;
Number sideOfSideVertexNumber(const Number vNum, const Number sideOfSideNum) const;
Number sideOfSideNumber(const Number i, const Number sideNum) const; // returns
local number of i-th edge ( i = 1,2, ...)
void rotateVertices(const number_t newFirst, std::vector<number_t>&) const; //
circular permutation of vertices
std::vector<real_t> sideToElt(number_t, std::vector<real_t>::const_iterator) const;
virtual Real measure(const Dimen dim, const Number sideNum = 0) const = 0;
virtual std::vector<real_t> projection(const std::vector<real_t>&, real_t&) const;
// projection of a point onto ref element
virtual bool contains(std::vector<real_t>& p, real_t tol= theTolerance) const;
// test if a point belongs to current element

```

- some protected append methods for vertices:

```

//! build vertex 1d coordinates
void vertex(std::vector<Real>::iterator& it, const Real x1);
void vertex(std::vector<Real>::iterator& it, const Real x1, const Real x2);
void vertex(std::vector<Real>::iterator& it, const Real x1, const Real x2, const
Real x3);

```

- some public error handlers:

```

void noSuchSideOfSide(const Number) const;
void noSuchSide(const Number) const;
void noSuchSide(const Number, const Number, const Number = 0, const Number = 0)
const;
void noSuchFunction(const String& s) const;

```

- an external search function in the static run-time list of `GeomRefElement`:

```

GeomRefElement* findGeomRefElement(ShapeType);

```

library :	finiteElements
header :	GeomRefElement.hpp
implementation :	GeomRefElement.cpp
unitary tests :	test_segment.cpp, test_triangle.cpp, test_quadrangle.cpp,
	test_tetrahedron.cpp, test_hexahedron.cpp, test_prism.cpp
header dependencies :	config.h, GeomRefSegment.hpp, GeomRefTriangle.hpp,
	GeomRefQuadrangle.hpp, GeomRefTetrahedron.hpp,
	GeomRefHexahedron.hpp, GeomRefPrism.hpp

5.3.2 Child classes of `GeomRefElement`

The child classes of `GeomRefElement` are `GeomRefSegment` in 1D, `GeomRefTriangle` and `GeomRefQuadrangle` in 2D and `GeomRefHexahedron`, `GeomRefTetrahedron`, `GeomRefPrism` and `GeomRefPyramid` in 3D.

Except implementation of virtual functions, these child classes provide 2 new member functions used in class constructors:

```
void sideNumbering(); // local numbers of vertices on sides
void sideOfSideNumbering(); // local numbers of vertices on side of sides
```

The last one is not defined for `GeomRefSegment`.

library :	<code>finiteElements</code>
header :	<code>GeomRefSegment.hpp</code>
implementation :	<code>GeomRefSegment.cpp</code>
unitary tests :	<code>test_segment.cpp</code>
header dependences :	<code>config.h</code> , <code>GeomRefElement.hpp</code>
<hr/>	
library :	<code>finiteElements</code>
header :	<code>GeomRefTriangle.hpp</code>
implementation :	<code>GeomRefTriangle.cpp</code>
unitary tests :	<code>test_triangle.cpp</code>
header dependences :	<code>config.h</code> , <code>GeomRefElement.hpp</code>
<hr/>	
library :	<code>finiteElements</code>
header :	<code>GeomRefQuadrangle.hpp</code>
implementation :	<code>GeomRefQuadrangle.cpp</code>
unitary tests :	<code>test_quadrangle.cpp</code>
header dependences :	<code>config.h</code> , <code>GeomRefElement.hpp</code>
<hr/>	
library :	<code>finiteElements</code>
header :	<code>GeomRefTetrahedron.hpp</code>
implementation :	<code>GeomRefTetrahedron.cpp</code>
unitary tests :	<code>test_tetrahedron.cpp</code>
header dependences :	<code>config.h</code> , <code>GeomRefElement.hpp</code>
<hr/>	
library :	<code>finiteElements</code>
header :	<code>GeomRefHexahedron.hpp</code>
implementation :	<code>GeomRefHexahedron.cpp</code>
unitary tests :	<code>test_hexahedron.cpp</code>
header dependences :	<code>config.h</code> , <code>GeomRefElement.hpp</code>
<hr/>	
library :	<code>finiteElements</code>
header :	<code>GeomRefPrism.hpp</code>
implementation :	<code>GeomRefPrism.cpp</code>
unitary tests :	<code>test_prism.cpp</code>
header dependences :	<code>config.h</code> , <code>GeomRefElement.hpp</code>

library :	finiteElements
header :	GeomRefPyramid.hpp
implementation :	GeomRefPyramid.cpp
unitary tests :	test_pyramid.cpp
header dependences :	config.h, GeomRefElement.hpp

5.4 Reference elements

5.4.1 The ShapeValues class

A **ShapeValues** object stores the evaluation of a shape function at a given point. Thus, it carries 2 real vectors of:

```
class ShapeValues
{
public:
    std::vector<Real> w; //shape functions at a point
    std::vector< std::vector<Real> > dw; //derivatives of shape functions at a point
};
```

It offers:

- basic constructors:

```
ShapeValues(); //!< default constructor
ShapeValues(const ShapeValues&);
//copy constructor
ShapeValues& operator=(const ShapeValues&); // assignment operator=
ShapeValues(const RefElement&); //constructor with associated RefElement
ShapeValues(const RefElement&, const Dimen); //constructor with associated RefElement
```

- two public size accessors:

```
Dimen dim() const { return static_cast<Dimen>(dw.size()); }
Number nbDofs() const { return static_cast<Number>(w.size()); }
```

- a resize function:

```
void resize(const RefElement&, const Dimen); // resize
void set(const real_t); // set shape functions to const value
void assign(const ShapeValues&); // fast assign of shapevalues into current
```

- some mapping

```
void extendToVector(dimen_t d); //extend scalar shape function to vector
void map(const ShapeValues&, const GeomMapData&, number_t der=1); //standard mapping
void contravariantPiolaMap(const ShapeValues&, const GeomMapData&, number_t der=1);
//contravariant Piola mapping
void covariantPiolaMap(const ShapeValues&, const GeomMapData&, number_t der=1);
//covariant Piola mapping
void changeSign(const std::vector<real_t>&, dimen_t); //change sign of shape functions according to a sign vector
```

The contravariant and covariant Piola maps are respectively used for Hdiv and Hcurl finite element families. They preserve respectively the normal and tangential traces. If J denotes the jacobian of the mapping from the reference element to any element, the covariant map is $J^{-t}\hat{\mathbf{u}}$ and the contravariant map $J/|J|\hat{\mathbf{u}}$ where $\hat{\mathbf{u}}$ is a vector fields in the reference space.

library :	finiteElements
header :	ShapeValues.hpp
implementation :	ShapeValues.cpp
unitary tests :	
header dependences :	config.h, RefElement.hpp, utils.h

5.4.2 The RefElement class

The **RefElement** is the main class of the *finiteElement* library. First, it has an object of each type previously seen in this chapter, namely **Interpolation**, **GeomRefElement**, **ShapeValues** and **RefDof**. Next, It is the mother class of the wide range of true reference elements, identified by shape and interpolation type. That is why we need some information such as the number of DoFs, the same number on vertices, on sides, on side of sides, ...

To define completely a reference element, we also need data on sides and on side of sides, such as DoF numbers and reference elements. Side and side of side reference elements will be built only when they are needed, equivalent to say only they have meaning. It is the case for H1 finite elements, but not for Hcurl and Hdiv finite elements.

So, the **RefElement** attributes are:

```
class RefElement
{
public:
    GeomRefElement* geomRefElem_p; ///< geometric data is gathered from geometric
    reference element
    const Interpolation* interpolation_p; ///< interpolation parameters
    std::vector<RefDof*> refDofs; ///< vector of local reference Degrees of Freedom

protected:
    String name_; ///< reference element name (for documentation or debug purposes)
    Number nbDofs_; ///< number of Degrees Of Freedom
    Number nbPts_; ///< number of Points (for local coordinates of points)

    Number nbDofsOnVertices_; ///< sharable D.o.F on vertices are first D.o.F in
    element
    Number nbDofsInSideOfSides_; ///< sharable D.o.F on side of sides (edges),
    excluding the previous ones, are next
    Number nbDofsInSides_; ///< sharable D.o.F on sides (faces or edges), excluding
    the previous ones, are next
    Number nbInternalDofs_; ///< finally come non-sharable D.o.F's

    std::vector<RefElement*> sideRefElems_; ///< pointers to side reference elements
    (in 2d sideRefElems_ is set to sideOfSideRefElems_)
    std::vector<RefElement*> sideOfSideRefElems_; ///< pointers to side of side (1D)
    reference elements

public:
    std::vector<std::vector<Number>> > sideDofNumbers_; ///< list of dof numbers for
    each side
    std::vector<std::vector<std::vector<Number>> > > sideOfSideDofNumbers_; ///< list of dof numbers
    for each side of side (edge in 3D)
    mutable ShapeValues shapeValues;
```



```

static std::vector<RefElement*> theRefElements; ///< vector to store run-time
RefElement pointers

private:
static const bool isSharable_ = true;

```

It offers:

- some constructors, the default one and a constructor by shape type and interpolation:

```

RefElement(); ///< default constructor
RefElement(ShapeType, const Interpolation* ); ///< constructor by shape &
interpolation

```

Note that the copy constructor and the assignment operator are private.

- some public accessors:

```

///< return associated Geometric Reference Element
const Interpolation& interpolation() const { return *interpolation_p; }
///< returns (protected) reference element name
String name() const { return name_; }
///< returns (protected) number of points
Number nbPts() const { return nbPts_; }
///< returns (protected) number of D.o.F supported by vertices
Number nbDofsOnVertices() const { return nbDofsOnVertices_; }
///< returns (protected) number of D.o.F supported by edges excluding D.o.F support
by vertices
Number nbDofsInSideOfSides() const { return nbDofsInSideOfSides_; }
///< returns (protected) number of D.o.F supported by faces excluding D.o.F
supported by vertices or edges
Number nbDofsInSides() const { return nbDofsInSides_; }

///< returns this if sideNum = 0 or reference element of side sideNum (sideNum = 1,
...)
const RefElement* refElement(Number sideNum = 0) const;
///< return associated Geometric Reference Element (i=0) or GeomRefElement of its
side i
const GeomRefElement* geomRefElement(Number sideNum = 0) const;
///< returns number of element D.o.F's (sideNum = 0) number of D.o.F's on a given
sideDim side sideNum (sideNum = 1, ...)
Number nbDofs(const Number sideNum = 0, const Dimen sideDim = 0) const;
///< returns number of element internal D.o.F (sideNum = 0 ) or number of internal
D.o.F's on sideDim side sideNum (sideNum = 1, ...)
Number nbInternalDofs(const Number sideNum = 0, const Dimen sideDim = 0) const;
ShapeType shapeType() const; ///< returns shape of element

```

- some build functions for shape values of side and side of side reference elements:

```

virtual void computeShapeValues(std::vector<Real>::const_iterator it_pt, const bool
withDeriv = true) const = 0;
///< compute shape functions for all points of vector x
std::vector<ShapeValues> getShapeValues(const std::vector< std::vector<Real> >& x,
const bool withDeriv = true) const;
void sideOfSideRefElement(); ///< reference element constructors on element edges
void sideRefElement(); ///< reference element constructors on element faces

```

- some general build functions for DoFs:

```

void LagrangeRefDofs(const int, const int, const int, const Dimen);
void HermiteRefDofs(const int, const int, const Dimen, const Dimen);

```

```
void dotNRefDofs(const int, const int, const Dimen, const Dimen);
void crossNRefDofs(const int, const int, const Dimen, const Dimen);
```

As the inheritance diagram is based on shape, these functions specify the interpolation type

- For output purpose, you can split a `RefElement` in first order elements, either simplices or of same shape, at every order :

```
virtual std::vector<std::vector<number_t> > splitP1() const;
virtual std::vector<std::pair<ShapeType, std::vector<number_t> > > splitO1() const;
```

- some external functions to find a reference element in the static list of `RefElement` and create it if not found:

```
RefElement* findRefElement(ShapeType, const Interpolation*);
RefElement* selectRefSegment(const Interpolation*);
RefElement* selectRefTriangle(const Interpolation*);
RefElement* selectRefQuadrangle(const Interpolation*);
RefElement* selectRefTetrahedron(const Interpolation*);
RefElement* selectRefPrism(const Interpolation*);
RefElement* selectRefHexahedron(const Interpolation*);
RefElement* selectRefPyramid(const Interpolation*);
```

library :	<code>finiteElements</code>
header :	<code>RefElement.hpp</code>
implementation :	<code>RefElement.cpp</code>
unitary tests :	<code>test_segment.cpp</code> , <code>test_triangle.cpp</code> , <code>test_quadrangle.cpp</code> , <code>test_tetrahedron.cpp</code> , <code>test_hexahedron.cpp</code> , <code>test_prism.cpp</code> , <code>test_pyramid.cpp</code>
header dependences :	<code>config.h</code> , <code>utils.h</code> , <code>Interpolation.hpp</code> , <code>ShapeValues.hpp</code> , <code>GeomRefElement.hpp</code> , <code>RefDof.hpp</code>

5.4.3 Child classes for segment

The `RefSegment` class has three main child classes :

- `LagrangeStdSegment` which implements standard Lagrange element at any order
- `LagrangeGLSegment` which implements Lagrange element based on Gauss-Lobatto points at any order
- `HermiteStdSegment` which implements standard Hermite element, currently only P3 Hermite

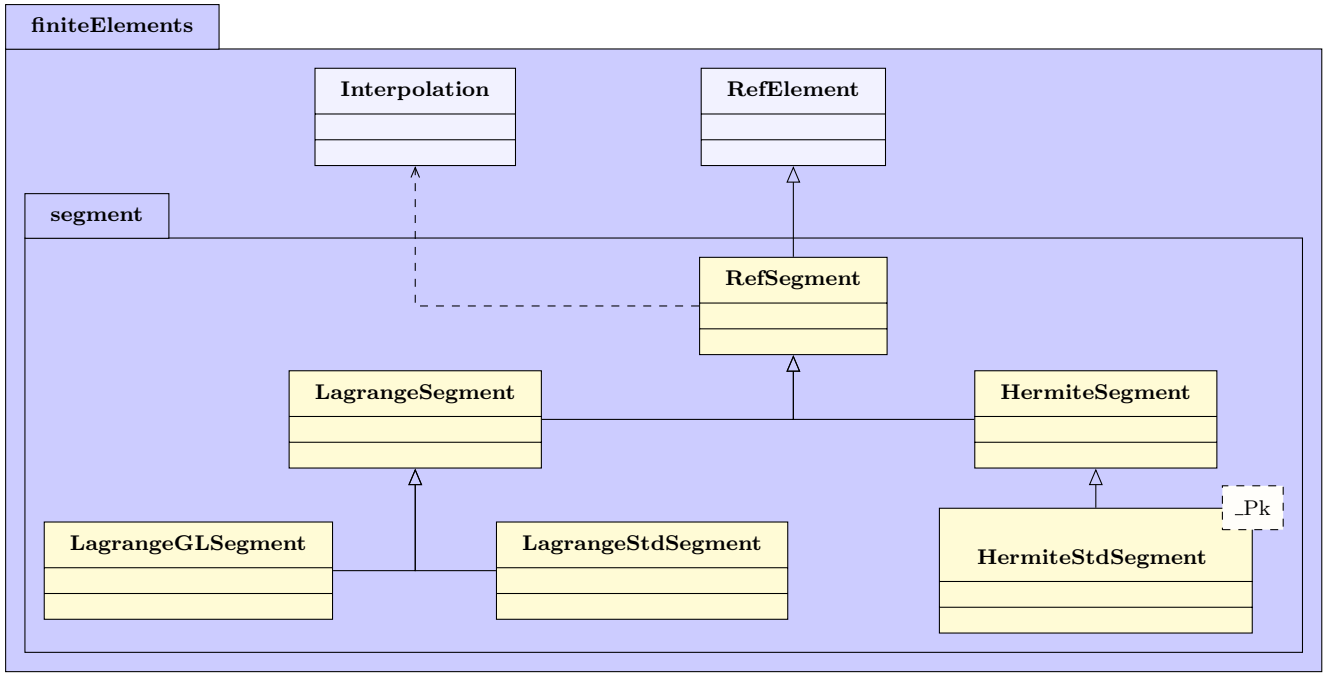


Figure 5.2: The `RefSegment` main class diagram.

5.4.4 Child classes for triangle

The `RefTriangle` class provides the following types of element:

- standard Lagrange element at any order `LagrangeStdTriangle <_Pk>` and `LagrangeStdTrianglePk`; the template `LagrangeStdTriangle<_Pk>` class is defined only for low order (up to 6) with best performance and `LagrangeStdTrianglePk` class is designed for any order. It uses a general representation of shape functions on xy-monomials basis get from the solving of a linear system. This method is not stable for very high order.
- `HermiteStdTriangle` which implements standard Hermite element, currently only P3 Hermite (not H2 conforming)
- `CrouzeixRaviartStdTriangle` which implements the Crouzeix-Raviart element, currently only P1 (not H1 conforming)
- `RaviartThomasTriangle` which implements the Raviart-Thomas elements (Hdiv conforming) : `RaviartThomasStdTriangleP1` for Raviart-Thomas standard P1 element and `RaviartThomasStdTrianglePk` for Raviart-Thomas at any order.
- `NedelecTriangle` which implements the Nedelec elements (Hcurl conforming) : `NedelecFirstTriangleP1` for Nedelec first family of order 1 element, `NedelecFirstTrianglePk` for Nedelec first family of any order and `NedelecSecondTrianglePk` for Nedelec second family of any order.

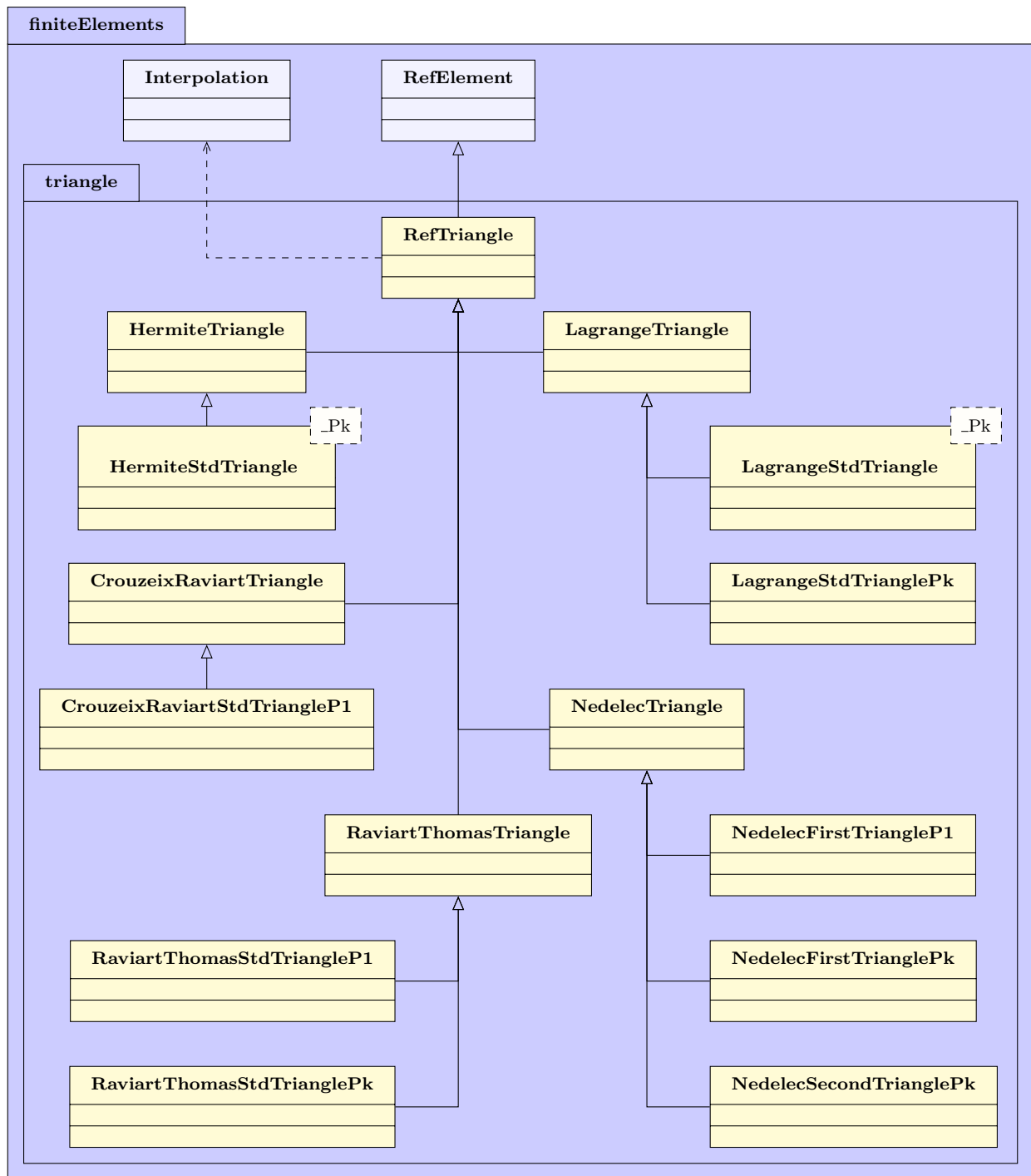


Figure 5.3: The `RefTriangle` main class diagram.

5.4.5 Child classes for quadrangle

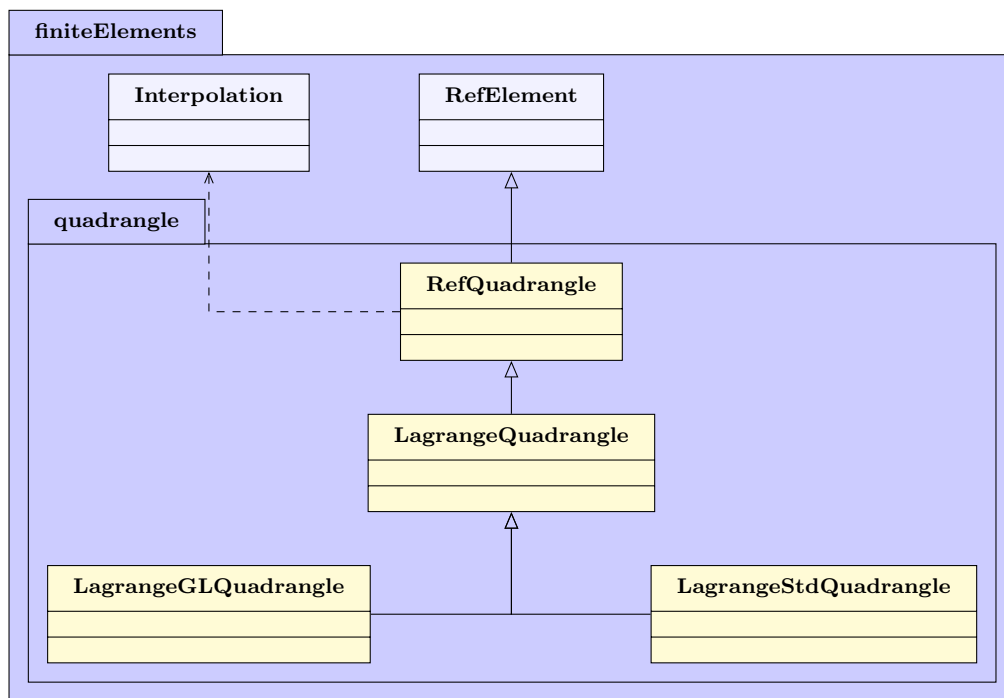


Figure 5.4: The `RefQuadrangle` main class diagram.

5.4.6 Child classes for tetrahedron

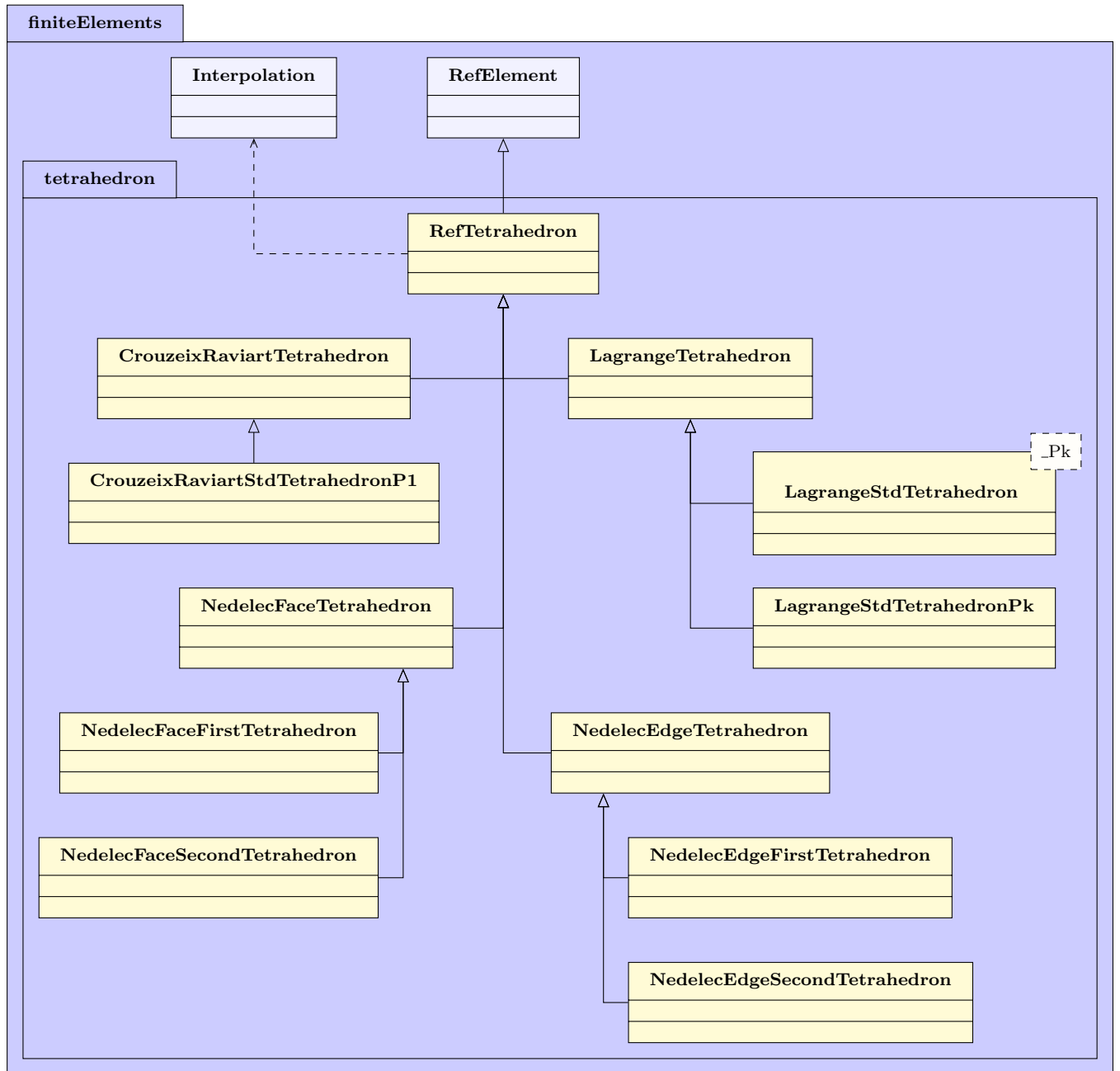


Figure 5.5: The `RefTetrahedron` main class diagram.

The `RefTetrahedron` class provides the following types of element:

- `LagrangeStdTetrahedron<_Pk>` class is defined only for low order (up to 6) with best performance and `LagrangeStdTetrahedronPk` class is designed for any order. It uses a general representation of shape functions on xyz-monoms basis get from the solving of a linear system. This method is not stable for very high order.
- `CrouzeixRaviartTetrahedron` which implements the Crouzet-Raviart element, currently only P1 (not H1 conforming)

- `NedelecFaceTetrahedron` which implements the Nedelec elements H_{div} conforming: `NedelecFaceFirstTetrahedronPk` for Nedelec first family of any order and `NedelecFaceSecondTetrahedronPk` for Nedelec second family of any order. The second family is not yet available.
- `NedelecEdgeTetrahedron` which implements the Nedelec elements H_{rot} conforming: `NedelecEdgeFirstTetrahedronPk` for Nedelec first family of any order and `NedelecEdgeSecondTetrahedronPk` for Nedelec second family of any order. The second family is not yet available.

Short identifiers of face/edge elements are defined in the following enumerations:

```
enum FeFaceType
{
    _RT_1 = 1, RT_1 = _RT_1, _NF1_1 = _RT_1, NF1_1 = _RT_1,
    _RT_2 ,   RT_2 = _RT_2, _NF1_2 = _RT_2, NF1_2 = _RT_2,
    _RT_3 ,   RT_3 = _RT_3, _NF1_3 = _RT_3, NF1_3 = _RT_3,
    _RT_4 ,   RT_4 = _RT_4, _NF1_4 = _RT_4, NF1_4 = _RT_4,
    _RT_5 ,   RT_5 = _RT_5, _NF1_5 = _RT_5, NF1_5 = _RT_5,
    _BDM_1 ,  BDM_1 = _BDM_1, _NF2_1 = _BDM_1, NF2_1 = _BDM_1,
    _BDM_2 ,  BDM_2 = _BDM_2, _NF2_2 = _BDM_2, NF2_2 = _BDM_2,
    _BDM_3 ,  BDM_3 = _BDM_3, _NF2_3 = _BDM_3, NF2_3 = _BDM_3,
    _BDM_4 ,  BDM_4 = _BDM_4, _NF2_4 = _BDM_4, NF2_4 = _BDM_4,
    _BDM_5 ,  BDM_5 = _BDM_5, _NF2_5 = _BDM_5, NF2_5 = _BDM_5
};

enum FeEdgeType
{
    _N1_1 = 1, N1_1 = _N1_1, _NE1_1 = _N1_1, NE1_1 = _N1_1,
    _N1_2 ,   N1_2 = _N1_2, _NE1_2 = _N1_2, NE1_2 = _N1_2,
    _N1_3 ,   N1_3 = _N1_3, _NE1_3 = _N1_3, NE1_3 = _N1_3,
    _N1_4 ,   N1_4 = _N1_4, _NE1_4 = _N1_4, NE1_4 = _N1_4,
    _N1_5 ,   N1_5 = _N1_5, _NE1_5 = _N1_5, NE1_5 = _N1_5,
    _N2_1 ,   N2_1 = _N2_1, _NE2_1 = _N2_1, NE2_1 = _N2_1,
    _N2_2 ,   N2_2 = _N2_2, _NE2_2 = _N2_2, NE2_2 = _N2_2,
    _N2_3 ,   N2_3 = _N2_3, _NE2_3 = _N2_3, NE2_3 = _N2_3,
    _N2_4 ,   N2_4 = _N2_4, _NE2_4 = _N2_4, NE2_4 = _N2_4,
    _N2_5 ,   N2_5 = _N2_5, _NE2_5 = _N2_5, NE2_5 = _N2_5
};
```

The naming convention is based on the FE periodic table. Note that there is an equivalence between $NF1_k$ and RT_k (Raviart Thomas), $NF2_k$ BDM_k (Brezzi Douglas Marini), $N1_k$ and $NE1_k$ and, $N2_k$ and $NE2_k$.



`NedelecEdgeTetrahedron` and `NedelecFaceTetrahedron` classes use a general process to build shape functions as polynomials related to moment dofs. To match dofs on shared edge or shared face, the ascending numbering of vertices is implicitly used even if it is not in fact. This method is particular touchy in case of face dofs of Nedelec Edge element. Indeed, such dofs depend on the choice of two tangent vectors, generally two edges of faces of the reference tetrahedron that are mapped to some edges of physical element in a non trivial way. To ensure a correct matching of such dofs on a shared face, a rotation has to be applied to shape values to guarantee that they corresponds to the same tangent vectors. This is the role of the member function `NedelecEdgeFirstTetrahedronPk::rotateDofs`. Note that if the element vertex numbering and face vertex numbering are ascending, `rotateDofs` does nothing. This trick is commonly used to overcome this difficulty but as `XLiFE++` makes no assumption on geometry, this dofs rotation is mandatory. More details on edge/face elements are provided in a specific paper.

5.4.7 Child classes for hexahedron

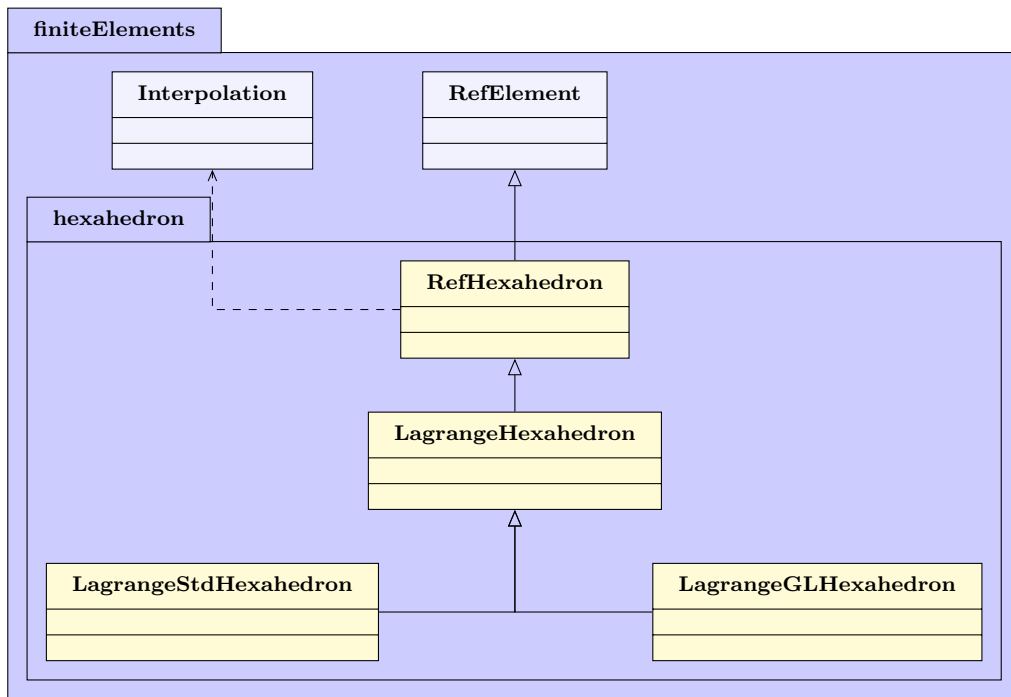


Figure 5.6: The **RefHexahedron** main class diagram.

5.4.8 Child classes for prism

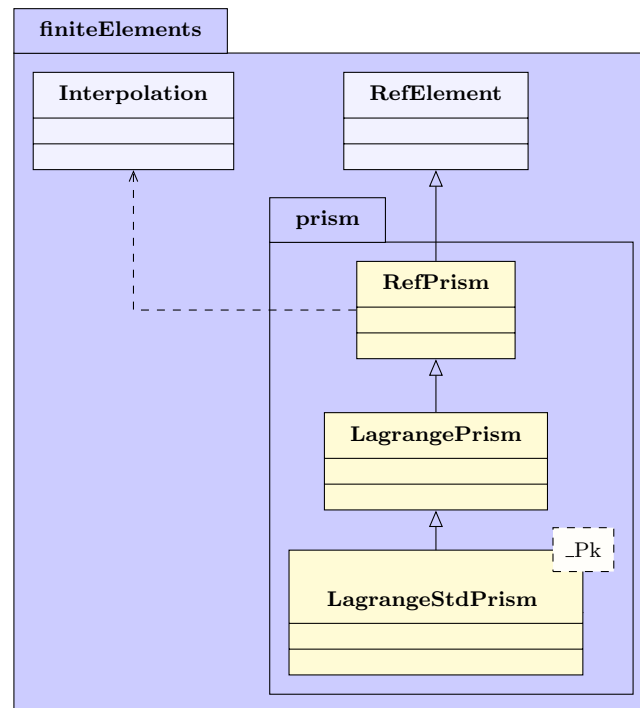


Figure 5.7: The **RefPrism** main class diagram.

5.4.9 Child classes for pyramid

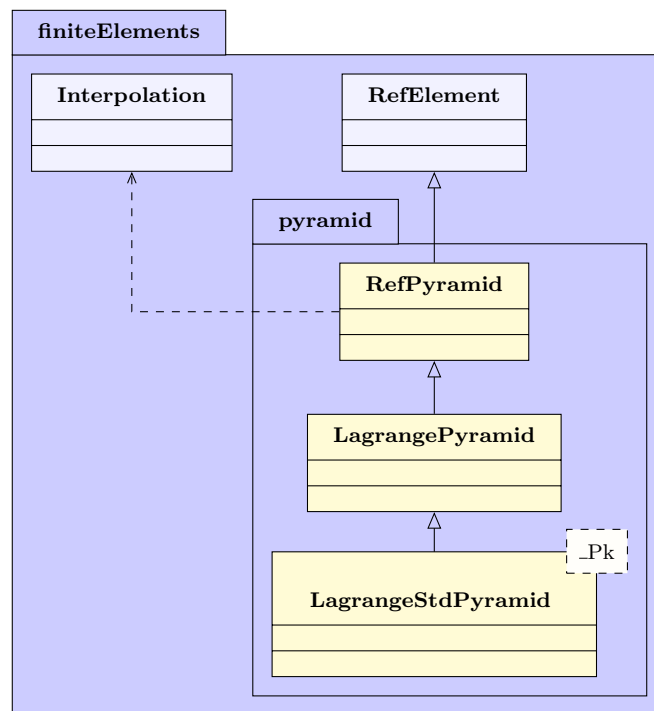
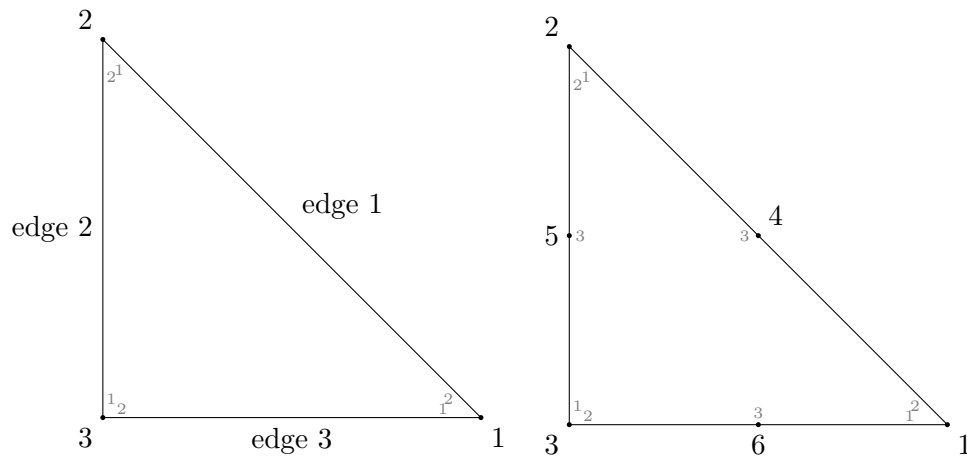


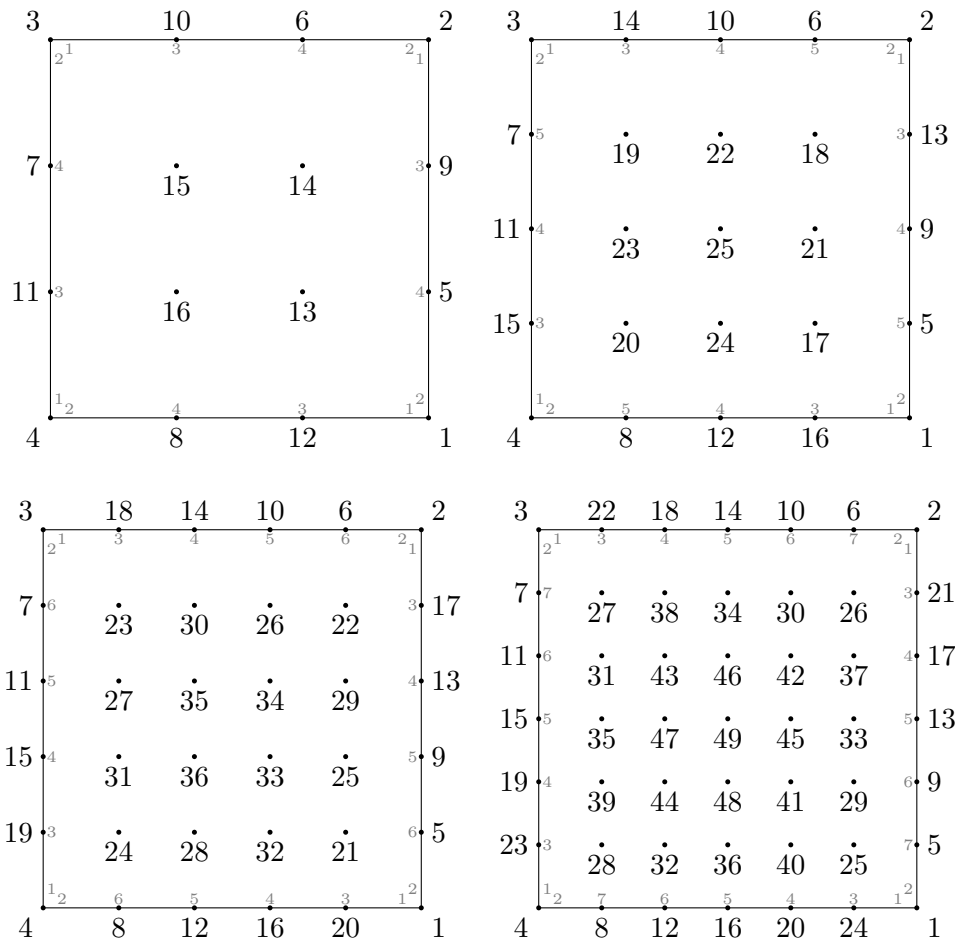
Figure 5.8: The `RefPyramid` main class diagram.

5.5 Examples of elements

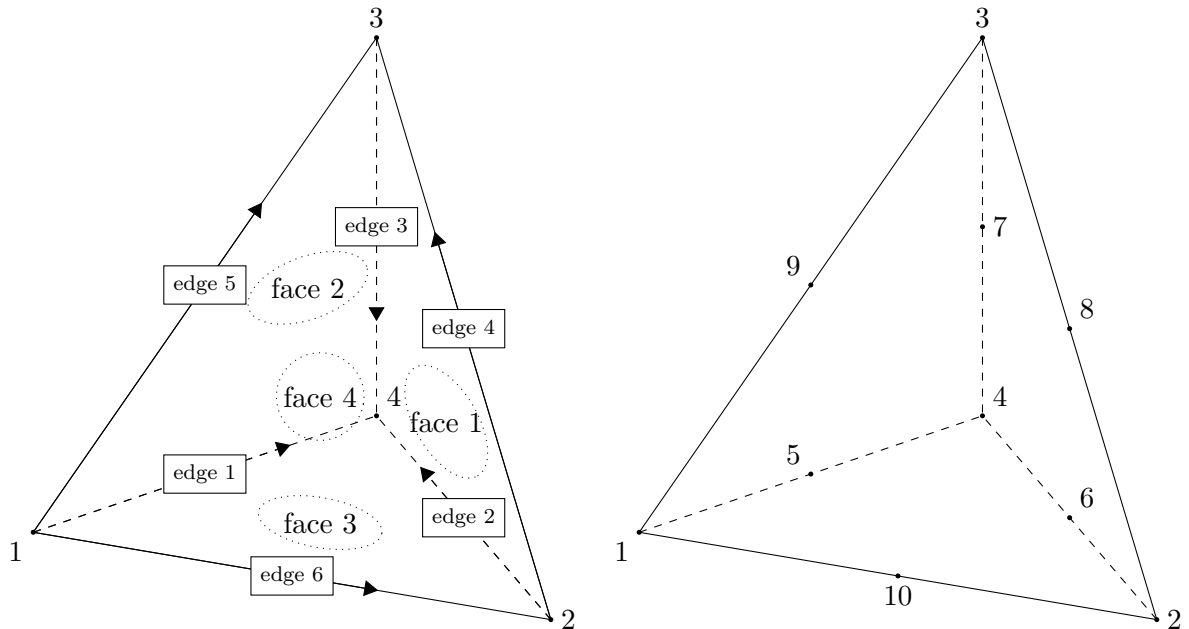
5.5.1 Examples of triangles

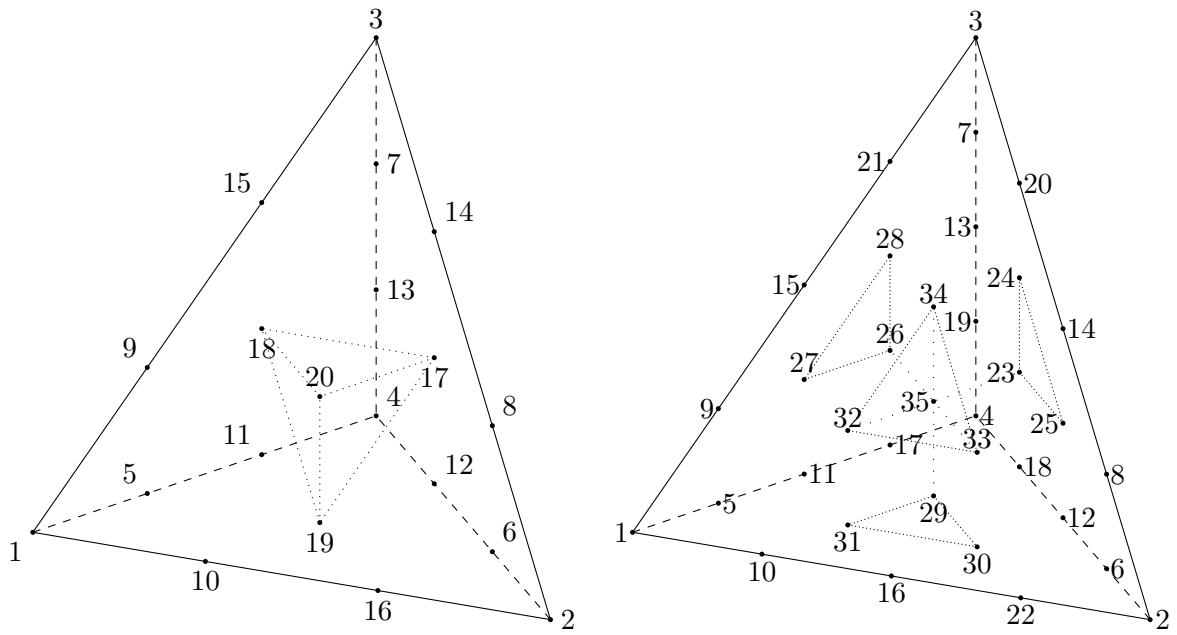




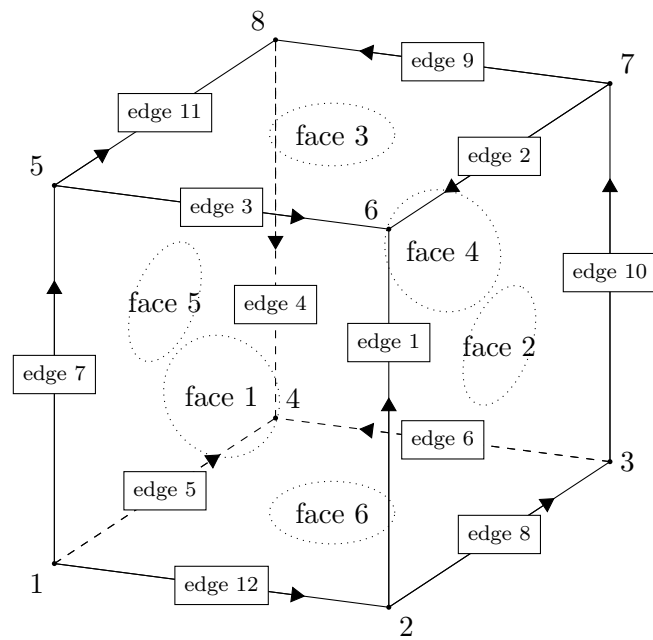


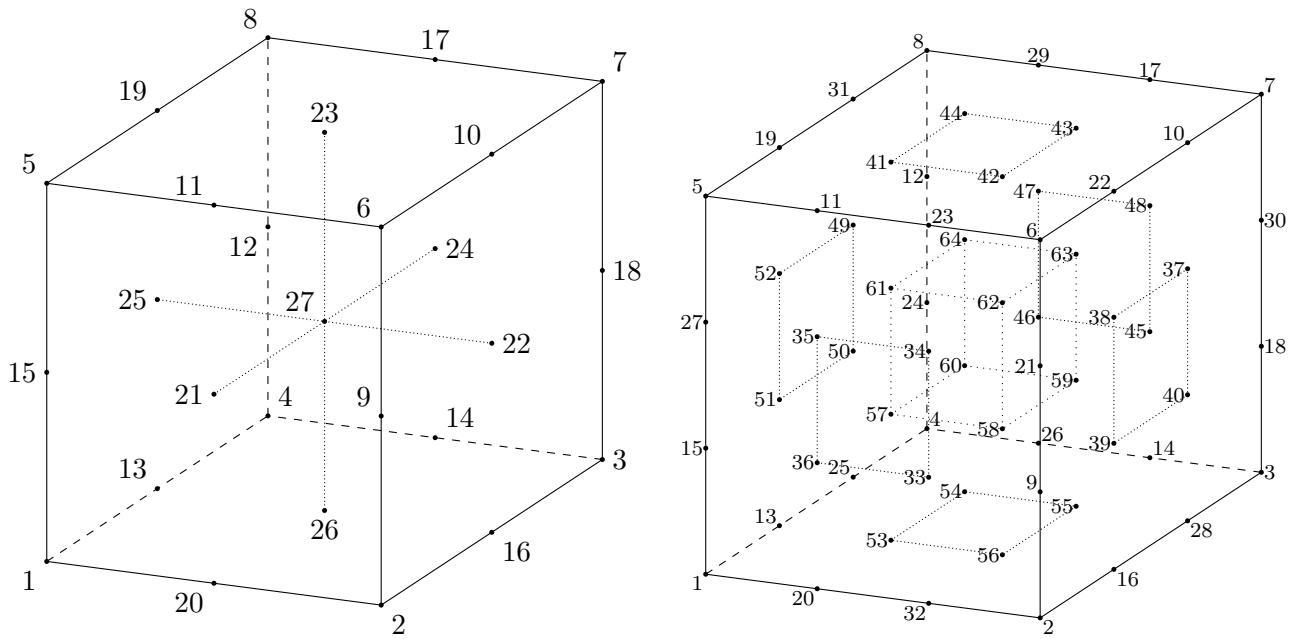
5.5.3 Examples of tetrahedra



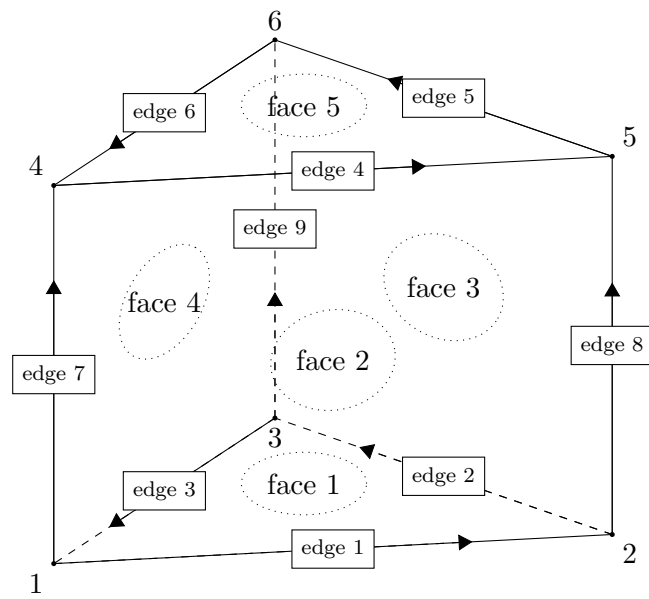


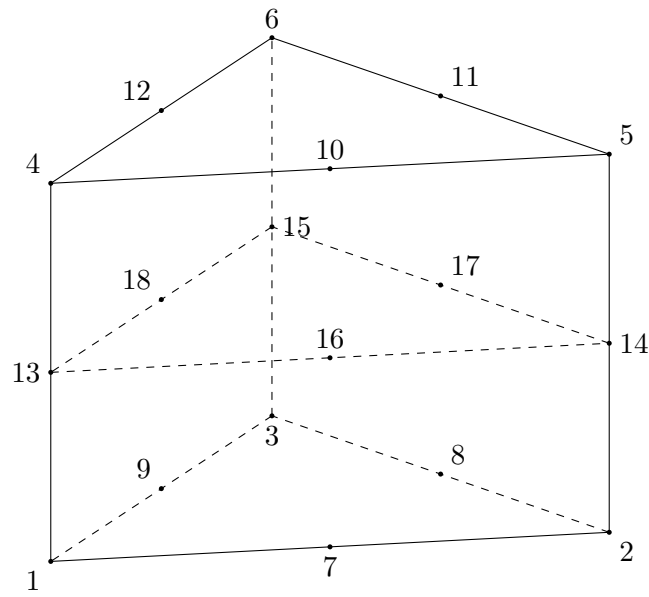
5.5.4 Examples of hexahedra



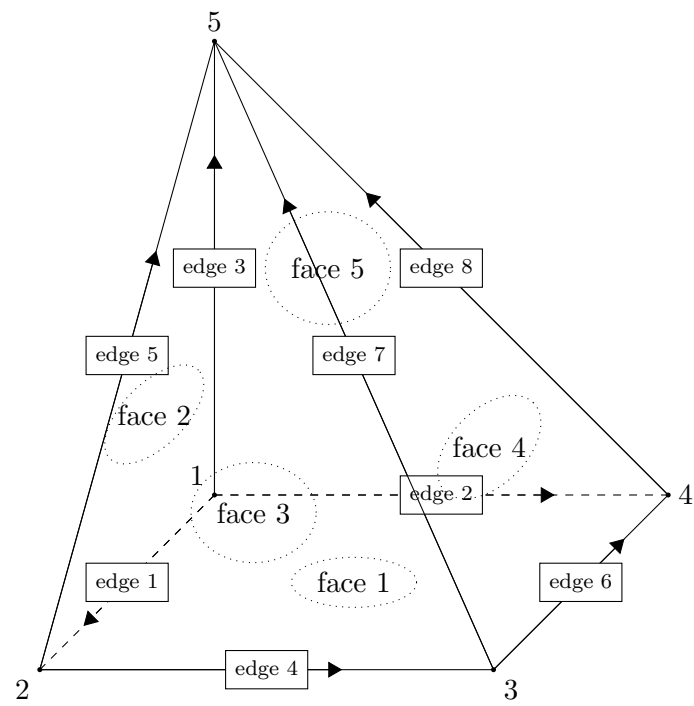


5.5.5 Examples of prisms





5.5.6 Examples of pyramids



5.6.1 The IntegrationMethod, SingleIM, DoubleIM classes

The IntegrationMethod class

The `IntegrationMethod` class, basis class of all integration methods, manages a name, a type of integral and two booleans:

```
class IntegrationMethod
{public :
    String name;
    IntegrationMethodType imType;
    bool requireRefElement;
    bool requirePhyElement;
};
```

Up to now, the types of integral defined in enumeration are:

```
enum IntegrationMethodType
{_undefIM, _quadratureIM, _polynomialIM, _productIM,
 _LenoirSalles2dIM, Lenoir_Salles_2d=_LenoirSalles2dIM,
 _LenoirSalles3dIM, Lenoir_Salles_3d=_LenoirSalles3dIM,
 _LenoirSalles2dIR, _LenoirSalles3dIR,
 _SauterSchwabIM, Sauter_Schwab=_SauterSchwabIM,
 _DuffyIM, Duffy=_DuffyIM,
 _HMatrixIM, H_Matrix=_HMatrixIM
};
```

The type `_HMatrixIM` is a very particular case because it is used to choose the HMatrix representation and compression methods which can be seen in some way as an integration method!

This class provides few general functions, most of them being virtual ones

```
IntegrationMethod(IntegrationMethodType imt=_undefIM, bool ref=false, bool
    phy=false, const String& na="")
virtual ~IntegrationMethod(){};
virtual bool isSingleIM() const =0;
virtual bool isDoubleIM() const =0;
virtual void print(std::ostream& os) const;
IntegrationMethodType type() const;
virtual bool useQuadraturePoints() const;

std::ostream& operator<<(std::ostream&, const IntegrationMethod&);
```

The SingleIM, DoubleIM classes

The `SingleIM`, `DoubleIM` derived classes provides only simple member functions :

```
class SingleIM : public IntegrationMethod
{public :
    bool isSingleIM() const {return true;}
    bool isDoubleIM() const {return false;}
    SingleIM(IntegrationMethodType imt=_undefIM);
    virtual void print(std::ostream& os) const;
}
```

```
class DoubleIM : public IntegrationMethod
{public :
    bool isSingleIM() const {return false;}
    bool isDoubleIM() const {return true;}
```



```

DoubleIM(IntegrationMethodType imt=_undefIM);
virtual void print(std::ostream& os) const;
}

```

5.6.2 The QuadratureIM class

The `QuadratureIM` class handles a list of `QuadratureIM` pointers indexed by `ShapeType` to deal with multiple quadrature rule in case of domain having more than one shape type. As the quadrature rule are often involved in context of FE computation, the `QuadratureIM` class may store shape values (pointer):

```

class QuadratureIM : public SingleIM
{protected :
    std::map<ShapeType, Quadrature *> quadratures_;
    std::map<ShapeType, std::vector<ShapeValues>*> shapevalues_;
}

```

It provides two basic constructors, accessors and print facilities (all are public):

```

QuadratureIM( ShapeType, QuadRule =_defaultRule , Number =0);
QuadratureIM(const std::set<ShapeType>&, QuadRule =_defaultRule , Number =0);
virtual bool useQuadraturePoints() const;
Quadrature* getQuadrature(ShapeType) const;
void setShapeValues(ShapeType, std::vector<ShapeValues>*);
std::vector<ShapeValues>* getShapeValues(ShapeType);
virtual void print(std::ostream& os) const;
std::ostream& operator<<(std::ostream&, const QuadratureIM&);

```

5.6.3 The ProductIM class

In order to represent a product of integration method over a product of geometric domain, the `ProductIM` class carries two `SingleIM` pointers:

```

class ProductIM : public DoubleIM
{protected :
    SingleIM* im_x;      ///< integration method along x
    SingleIM* im_y;      ///< integration method along y
}

```

and provides simple member functions:

```

ProductIM(SingleIM* imx=0, SingleIM* imy=0);
SingleIM* getxIM();
SingleIM* getyIM();
virtual bool useQuadraturePoints() const;
virtual void print(std::ostream& os) const; ///< print on stream
};

```

5.6.4 The Quadrature, QuadratureRule classes

The quadrature formulae have the following form :

$$\int_{\hat{E}} f(\hat{x}) d\hat{x} \approx \sum_{i=1,q} \omega_i f(\hat{x}_i)$$

where $(\hat{x}_i)_{i=1,q}$ are quadrature points belonging to reference element \hat{E} and $(w_i)_{i=1,q}$ are quadrature weights.

Up to now, there exist quadrature formulae for unit segment $]0, 1[$, for unit triangle, for unit quadrangle (square), for unit tetrahedron, for unit hexahedron (cube) and for unit prism. The following tables give the list of quadrature formulae :

General rules

	Gauss-Legendre	Gauss-Lobatto	Grundmann-Muller	symmetrical Gauss
segment	any odd degree	any odd degree		
quadrangle	any odd degree	any odd degree		odd degree up to 21
triangle	any odd degree		any odd degree	degree up to 10
hexahedron	any odd degree	any odd degree		odd degree up to 11
tetrahedron	any odd degree		any odd degree	degree up to 10
prism				degree up to 10
pyramid	any odd degree	any odd degree		degree up to 10

Particular rules

	nodal	miscellaneous
segment	P1 to P4	
quadrangle	Q1 to Q4	
triangle	P1 to P3	Hammer-Stroud 1 to 6
hexahedron	Q1 to Q4	
tetrahedron	P1, P3	Stroud 1 to 5
prism	P1	centroid 1, tensor product 1,3,5
pyramid	P1	centroid 1, Stroud 7

More precisely, the rules that are implemented :

- 1D rules
 - Gauss-Legendre rule, degree $2n + 1$, n points
 - Gauss-Lobatto rule, degree $2n - 1$, n points
 - trapezoidal rule, degree 1, 2 points
 - Simpson rule, degree 3, 3 points
 - Simpson 3 8 rule, degree 3, 4 points
 - Booleb rule, degree 5, 5 points
 - Wedge rule, degree 5, 6 points
- Rules over the unit simplex (less quadrature points BUT with negative weights)
 - TN Grundmann-Moller rule, degree $2n + 1$, C_{n+d+1}^n points
- Rules over the unit triangle $\{x > 0, y > 0, x + y < 1\}$
 - T2P2 MidEdge rule
 - T2P2 Hammer-Stroud rule
 - T2P3 Albrecht-Collatz rule, degree 3, 6 points, Stroud (p.314)
 - T2P3 Stroud rule, degree 3, 7 points, Stroud (p.308)
 - T2P5 Radon-Hammer-Marlowe-Stroud rule, degree 6, 7 points, Stroud (p.314)
 - T2P6 Hammer rule, degree 6, 12 points, G.Dhatt, G.Touzot (p.298)
 - symmetrical Gauss up to degree 10
- Rules over the unit tetrahedron $\{x > 0, y > 0, z > 0, x + y + z < 1\}$
 - T3P2 Hammer-Stroud rule, degree 2, 4 points, Stroud (p.307)
 - T3P3 Stroud rule, degree 3, 8 points, Stroud (p.308)
 - T3P5 Stroud rule, degree 5, 15 points, Stroud (p.315)
 - symmetrical Gauss up to degree 10

- Rules over the unit prism $\{0 < x, y < 1, x + y < 1, 0 < z < 1\}$
P1 nodal rule
tensor product of 2-points Gauss-Legendre rule on $[0,1]$ and Stroud 7 points formula on the unit triangle (degree 3)
tensor product of 3-points Gauss-Legendre rule on $[0,1]$ and Radon-Hammer-Marlowe-Stroud 7 points formula on the unit triangle (degree 5)
symmetrical Gauss up to degree 10
- Rules over the unit pyramid $\{0 < x, y < 1 - z, 0 < z < 1\}$
P1 nodal rule
symmetrical Gauss up to degree 10
- Rules built from lower dimension rules
tensor product of quadrature rules (quadrangle and hexahedron)
conical product of quadrature rules (triangle and pyramid)
rule built from 1D nodal rule which are quadrangle nodal rule
rule built from 1D nodal rule which are hexahedron nodal rule

References :

- A.H.Stroud, Approximate calculation of multiple integrals, Prentice Hall, 1971.
- G.Dhatt & G Touzot, The finite element method displayed, John Wiley & Sons, Chichester, 1984
- A. Grundmann & H.M. Moller, Invariant Integration Formulas for the N-Simplex by Combinatorial Methods, SIAM Journal on Numerical Analysis, Vol 15, No 2, Apr. 1978, pages 282-290.
- F.D. Witherden & P.E. Vincent, On the identification of symmetric quadrature rules for finite element methods, Computers & Mathematics with Applications, Volume 69, Issue 10, May 2015, Pages 1232-1241.

These quadrature formulae are provided by using two classes : the [Quadrature](#) class which handles the main quadrature objects and the [QuadratureRule](#) class carrying the quadrature points and weights.

The Quadrature class

The [Quadrature](#) class manages all informations related to quadrature :

```
class Quadrature
{public:
    GeomRefElement* geomRefElt_p; // geometric element bearing quadrature rule
    QuadratureRule quadratureRule; // embedded QuadratureRule object
protected:
    QuadRule rule_; // type of quadrature rule
    Number degree_; // degree of quadrature rule
    bool hasPointsOnBoundary_; // as it reads
    String name_; // name of quadrature formula
public:
    static std::vector<Quadrature*> theQuadratureRules;
    ...
}
```

QuadRule is a enumeration of all types of quadrature supported:

```
enum QuadRule
{_defaultRule = 0, _GaussLegendreRule, _symmetricalGaussRule, _GaussLobattoRule,
    Gauss_Lobatto = _GaussLobattoRule,
    _nodalRule, _miscRule, _GrundmannMollerRule, _doubleQuadrature };
}
```

`degree_` is either the degree of the quadrature rule or the degree of the polynomial interpolation in case of nodal quadrature rules. A `Quadrature` object has to be instantiated only once and the static vector `theQuadratureRules` collects all the pointers to `Quadrature` object. Note that the shape bearing the quadrature is given through the `GeomRefElement` pointer.

The class proposes only a default constructor, a full basic constructor and a destructor :

```
Quadrature();
Quadrature(ShapeType , QuadRule, Number, const String&, bool pob = false);
```

Note that the basic constructor initializes member attributes but does not load the quadrature points and quadrature weights (`QuadratureRule`)! The practical "construction" is done by the external function `findQuadrature`. The copy constructor and assignment operator are defined as private member function to avoid duplicated `Quadrature` object

The following accessors (read only) are provided:

```
String name() const;
QuadRule rule() const;
Number degree() const;
bool hasPointsOnBoundary() const;
Dimen dim() const;
Number numberOfPoints() const;
ShapeType shapeType() const;
std::vector<Real>::const_iterator point(Number i = 0) const;
std::vector<Real>::const_iterator weight(Number i = 0) const;
```



To construct a `Quadrature` object, the external function `findQuadrature` has to be used :

```
friend Quadrature* findQuadrature(ShapeType shape, QuadRule rule, Number degree,
    bool hpob= false);
```

It searches in the static vector `theQuadratureRules` the quadrature rule defined by a shape, a quadrature rule and a degree. If it is not found, a new `Quadrature` object is created on the heap. In any case a pointer to the `Quadrature` object is returned. There are few functions depending on the shape that are called during the creation process :

```
Quadrature* segmentQuadrature(QuadRule, Number);
Quadrature* triangleQuadrature(QuadRule, Number);
Quadrature* quadrangleQuadrature(QuadRule, Number);
Quadrature* tetrahedronQuadrature(QuadRule, Number);
Quadrature* hexahedronQuadrature(QuadRule, Number);
Quadrature* prismQuadrature(QuadRule, Number);
Quadrature* pyramidQuadrature(QuadRule, Number);
static void clear();
```

These functions loads quadrature points and quadrature weights in a `QuadratureRule` object using member functions of `QuadratureRule` class. The `clear` static function deletes quadrature objects stored in the `theQuadratureRules` static vector.

To choose quadrature rules, the class provides the static function `bestQuadRule` returning the 'best' quadrature rule for a given shape S and a given polynomial degree d .

```
static QuadRule bestQuadRule(ShapeType, Number);
```

Best rule has to be understood as the rule on shape S with the minimum of quadrature points integrating exactly polynomials of degree d . The following table gives the current best rules :

shape	degree	quadrature rule	number of points
segment	1	Gauss-Legendre 1	1
segment	2, 3	Gauss-Legendre 3	2
segment	4, 5	Gauss-Legendre 5	3
segment	$2n - 1$	Gauss-Legendre $2n - 1$	n

shape	degree	quadrature rule	number of points
quadrangle	1	Gauss-Legendre 1	1
quadrangle	2, 3	Gauss-Legendre 3	4
quadrangle	4, 5	symmetrical Gauss 5	8
quadrangle	6, 7	symmetrical Gauss 7	12
quadrangle	7, 9	symmetrical Gauss 9	20
quadrangle	10, 11	symmetrical Gauss 11	28
quadrangle	12, 13	symmetrical Gauss 13	37
quadrangle	14, 15	symmetrical Gauss 15	48
quadrangle	16, 17	symmetrical Gauss 17	60
quadrangle	18, 19	symmetrical Gauss 19	72
quadrangle	20, 21	symmetrical Gauss 21	85
quadrangle	$2n - 1 > 21$	Gauss-Legendre $2n - 1$	n^2

shape	degree	quadrature rule	number of points
triangle	1	centroid rule (misc,1)	1
triangle	2	P2 Hammer-Stroud (misc,2)	2
triangle	3	Grundmann-Moller 3	3
triangle	4	symmetrical Gauss 4	6
triangle	5	symmetrical Gauss 5	7
triangle	6	symmetrical Gauss 6	12
triangle	7	symmetrical Gauss 7	15
triangle	8	symmetrical Gauss 8	16
triangle	9	symmetrical Gauss 9	19
triangle	10	symmetrical Gauss 10	25
triangle	11	symmetrical Gauss 11	28
triangle	12	symmetrical Gauss 12	33
triangle	13	symmetrical Gauss 13	37
triangle	14	symmetrical Gauss 14	42
triangle	15	symmetrical Gauss 15	49
triangle	16	symmetrical Gauss 16	55
triangle	17	symmetrical Gauss 17	60
triangle	18	symmetrical Gauss 18	67
triangle	19	symmetrical Gauss 19	73
triangle	20	symmetrical Gauss 20	79
triangle	$2n - 1 > 20$	Gauss-Legendre $2n - 1$?

shape	degree	quadrature rule	number of points
hexahedron	1	Gauss-Legendre 1	1
hexahedron	2, 3	symmetrical Gauss 3	6
hexahedron	4, 5	symmetrical Gauss 5	14
hexahedron	6, 7	symmetrical Gauss 7	34
hexahedron	8, 9	symmetrical Gauss 9	58
hexahedron	10, 11	symmetrical Gauss 11	90
hexahedron	$2n - 1 \geq 12$	Gauss-Legendre $2n - 1$	n^3

shape	degree	quadrature rule	number of points
tetrahedron	1	centroid rule (misc,1)	1
tetrahedron	2	P2 Hammer-Stroud (misc,2)	4
tetrahedron	3	Grundmann-Moller 3	5
tetrahedron	4, 5	symmetrical Gauss 5	14
tetrahedron	6	symmetrical Gauss 6	24
tetrahedron	7	symmetrical Gauss 7	35
tetrahedron	8	symmetrical Gauss 8	46
tetrahedron	9	symmetrical Gauss 9	59
tetrahedron	10	symmetrical Gauss 10	81
tetrahedron	$2n - 1 \geq 11$	Grundmann-Moller $2n - 1$?

shape	degree	quadrature rule	number of points
prism	1	centroid rule (misc,1)	1
prism	2	symmetrical Gauss 2	5
prism	3	symmetrical Gauss 3	8
prism	4	symmetrical Gauss 4	11
prism	5	symmetrical Gauss 5	16
prism	6	symmetrical Gauss 6	28
prism	7	symmetrical Gauss 7	35
prism	8	symmetrical Gauss 8	46
prism	9	symmetrical Gauss 9	60
prism	10	symmetrical Gauss 10	85

shape	degree	quadrature rule	number of points
pyramid	1	centroid rule (misc,1)	1
pyramid	2	symmetrical Gauss 2	5
pyramid	3	symmetrical Gauss 3	6
pyramid	4	symmetrical Gauss 4	10
pyramid	5	symmetrical Gauss 5	15
pyramid	6	symmetrical Gauss 6	24
pyramid	7	symmetrical Gauss 7	31
pyramid	8	symmetrical Gauss 8	47
pyramid	9	symmetrical Gauss 9	62
pyramid	10	symmetrical Gauss 10	83

Finally, `Quadrature` class provides some print and error utilities:

```
void badNodeRule(int n_nodes);
void badDegreeRule();
void noEvenDegreeRule();
void upperDegreeRule();
friend std::ostream& operator<<(std::ostream&, const Quadrature&);
friend void alternateRule(QuadRule, ShapeType, const String&);
```

The QuadratureRule class

The `QuadratureRule` class stores quadrature points and quadrature weights and provides member functions to load them :

```
class QuadratureRule
{private:
    std::vector<Real> coords_; // point coordinates of quadrature rule
```

```
std::vector<Real> weights_; // weights of quadrature rule
Dimen dim_; // dimension of points
...
```

the following constructors are proposed:

```
QuadratureRule(const std::vector<Real>&, Real);
QuadratureRule(const std::vector<Real>&, const std::vector<Real>&);
QuadratureRule(Dimen , Number s);
```

and the following accessors are provided :

```
Dimen dim() const;
Number size() const;
std::vector<Real> coords() const;
std::vector<Real>::const_iterator point(Number i = 0) const;
std::vector<Real>::iterator point(Number i = 0);
std::vector<Real>::const_iterator weight(Number i = 0) const;
std::vector<Real>::iterator weight(const Number i = 0);
```

Some utilities are also helpful:

```
void point(std::vector<Real>::iterator&, Real, std::vector<Real>::iterator&, Real);
void point(std::vector<Real>::iterator&, Real, Real, std::vector<Real>::iterator&, Real);
void point(std::vector<Real>::iterator&, Real, Real, Real, std::vector<Real>::iterator&, Real);
void coords(std::vector<Real>::iterator&, Dimen, std::vector<Real>::const_iterator&);
void coords(std::vector<Real>::const_iterator);
void coords(const std::vector<Real>&);
void coords(Real);
void weights(const std::vector<Real>&);
void weights(Real);
friend std::ostream& operator<<(std::ostream&, const QuadratureRule&);
```

The main task of this class is to load quadrature points and weights. Thus, there are a lot of member functions to load these values. Some of them have a general purpose (for instance to build quadrature rules from tensor or conical products of 1D rules) and others are specific to one quadrature rule:

```
//tensor and conical product
QuadratureRule& tensorRule(const QuadratureRule&, const QuadratureRule&);
QuadratureRule& tensorRule(const QuadratureRule&, const QuadratureRule&, const QuadratureRule&);
QuadratureRule& conicalRule(const QuadratureRule&, const QuadratureRule&);
QuadratureRule& conicalRule(const QuadratureRule&, const QuadratureRule&, const QuadratureRule&);
//nodal tensor product rules for quadrangle and hexahedron
QuadratureRule& quadrangleNodalRule(const QuadratureRule&);
QuadratureRule& hexahedronNodalRule(const QuadratureRule&);
//1D Gauss rules
QuadratureRule& gaussLegendreRules(Number);
QuadratureRule& gaussLobattoRules(Number);
QuadratureRule& ruleOn01(Number);
//1D Newton-Cotes closed rules (P_k node rules on [0,1])
QuadratureRule& trapezoidalRule();
QuadratureRule& simpsonRule();
QuadratureRule& simpson38Rule();
QuadratureRule& booleRule();
//Grundmann-Moller rules for n dimensional simplex
QuadratureRule& TNGrundmannMollerRule(int , Dimen);
void TNGrundmannMollerSet(int); //utility for Grundmann-Moller
```

```

//special (Misc) rules for unit Triangle {  $x + y < 1$  ,  $0 < x, y < 1$  }
QuadratureRule& t2P2MidEdgeRule();
QuadratureRule& t2P2HammerStroudRule();
QuadratureRule& t2P3AlbrechtCollatzRule();
QuadratureRule& t2P3StroudRule();
QuadratureRule& t2P5RadonHammerMarloweStroudRule();
QuadratureRule& t2P6HammerRule();
//special (Misc) rules for unit Tetrahedron {  $x + y + z < 1$  ,  $0 < x, y, z < 1$  }
QuadratureRule& t3P2HammerStroudRule();
QuadratureRule& t3P3StroudRule();
QuadratureRule& t3P5StroudRule();
//special (Misc) rules for unit pyramid {  $0 < x, y < 1 - z$  ,  $0 < z < 1$  }
QuadratureRule& pyramidRule(const QuadratureRule&); //conical product
QuadratureRule& pyramidStroudRule(); //degree 7, 48 points
//symmetrical Gauss rules for any shape
QuadratureRule& symmetricalGaussTriangleRule(number_t); //degree up <= 20
QuadratureRule& symmetricalGaussQuadrangleRule(number_t); //odd degree <= 21
QuadratureRule& symmetricalGaussTetrahedronRule(number_t); //degree <= 10
QuadratureRule& symmetricalGaussHexahedronRule(number_t); //odd degree <= 11
QuadratureRule& symmetricalGaussPrismRule(number_t); //degree <= 10
QuadratureRule& symmetricalGaussPyramidRule(number_t); //degree <= 10

```

Here is a simple example of how to use quadrature objects :

```

Quadrature* q=findQuadrature(_triangle, _GaussLegendreRule, 3);
Real S=0, x,y;
Number k=0;
for (Number i =0; i<q->quadratureRule.size(); i++,k+=2)
{
    x=q->quadratureRule.coords()[k];
    y=q->quadratureRule.coords()[k+1];
    S+=q->quadratureRule.weights()[i]*std::sin(x*y);
}
std::cout<<"S="<<S;

```

5.6.5 How to add a new quadrature formula ?

When you want to add a new quadrature formula on a unit element, say *geomelt*,

- you have to modify the member function :
`Quadrature::geomeltQuadrature(QuadRule rule, Number deg)`
by adding in the right *case* statement the properties of your quadrature formula
- to add in `QuadratureRule` class the member function defining the quadrature points and weights.

Be care with the type and the number (degree) defining your quadrature formula, because they can be used by an other formula. To avoid this difficulty, it is possible to create new type of quadrature. In that case the enumeration `QuadRule` has to be modified, implying to update the dictionnaires and may be, the `Quadrature::bestQuadRule` static function.

5.6.6 Integration methods for singular kernels

To deal with single or double integral involving a kernel with a singularity, XLiFE++ provides some additional classes.

DuffyIM class

This class deals with double integral over segments involving 2d kernel K with $\log|x-y|$ singularity, in other words:

$$\int_{S_1} \int_{S_2} p_1(x) K(x, y) p_2(y) dy dx$$

```
class DuffyIM : public DoubleIM
{
public:
    Quadrature* quadSelf; //quadrature for self influence
    Quadrature* quadAdj;  //quadrature for adjacent elements
    number_t ordSelf;      //order of quadrature for self influence
    number_t ordAdj;       //order of quadrature for adjacent elements

    DuffyIM(IntegrationMethodType);
    DuffyIM(number_t =6);   //full constructor from quadrature order
    DuffyIM(const Quadrature&); //full constructor from a quadrature
```

SauterSchwabIM class

This class deals with double integral over triangles involving 3d kernel K with $|x-y|^{-1}$ singularity, say

$$\int_{T_1} \int_{T_2} p_1(x) K(x, y) p_2(y) dy dx$$

```
class SauterSchwabIM : public DoubleIM
{
public:
    Quadrature* quadSelf; //quadrature for self influence
    Quadrature* quadEdge;  //quadrature for elements adjacent by edge
    Quadrature* quadVertex; //quadrature for elements adjacent by vertex
    number_t ordSelf;      //order of quadrature for self influence
    number_t ordEdge;      //order of quadrature for edge adjacence
    number_t ordVertex;    //order of quadrature for vertex adjacence

    SauterSchwabIM(number_t =3); //basic constructor (order 3)
    SauterSchwabIM(Quadrature&); //full constructor from quadrature object
```

LenoirSalles2dIM and LenoirSalles3dIM classes

The **LenoirSalles2dIM** class can compute analytically

$$-\frac{1}{2\pi} \int_{S_1} \int_{S_2} p_1(x) \log|x-y| p_2(y) dy dx \quad (\text{single layer})$$

$$\frac{1}{2\pi} \int_{S_1} \int_{S_2} p_1(x) \frac{(x-y).n_y}{|x-y|^2} p_2(y) dy dx \quad (\text{double layer})$$

where p_1 and p_2 are either the P0 or P1 shape functions on segments S_1 and S_2 and the **LenoirSalles3dIM** class computes analytically

$$\frac{1}{4\pi} \int_{T_1} \int_{T_2} p_1(x) \frac{1}{|x-y|} p_2(y) dy dx \quad (\text{single layer})$$

$$-\frac{1}{4\pi} \int_{T_1} \int_{T_2} p_1(x) \frac{(x-y).n_y}{|x-y|^3} p_2(y) dy dx \quad (\text{double layer})$$

where p_1 and p_2 are the P0 shape functions (say 1) on triangles T_1 and T_2 .

These two classes have no member data and only one basic constructor.

LenoirSalles2dIR and LenoirSalles3dIR classes

The `LenoirSalles2dIR` class can compute analytically

$$-\frac{1}{2\pi} \int_S \log |x - y| p(y) dy dx \quad (\text{single layer})$$

$$\frac{1}{2\pi} \int_S \frac{(x - y) \cdot n_y}{|x - y|^2} p(y) dy dx \quad (\text{double layer})$$

where p is either the P0 or P1 shape functions on segment S and the `LenoirSalles3dIR` classe that computes analytically

$$\frac{1}{4\pi} \int_T \frac{1}{|x - y|} p(y) dy dx \quad (\text{single layer})$$

$$-\frac{1}{4\pi} \int_{TS} \frac{(x - y) \cdot n_y}{|x - y|^3} p(y) dy dx \quad (\text{double layer})$$

where p is the P0 or P1 shape functions on triangle T .

These two classes have no member data and only one basic constructor.

All the previous classes provides, a clone method, an access to the list of quadratures if there are ones and an interface to the computation functions:

```
virtual LenoirSalles2dIM* clone() const; //covariant
virtual std::list<Quadrature*> quadratures() const; //list of quadratures
template<typename K>
void computeIE(const Element* S, const Element* T, AdjacencyInfo& adj,
               const KernelOperatorOnUnknowns& kuv, Matrix<K>& res,
               IEcomputationParameters& iep,
               Vector<K>& vopu, Vector<K>& vopv, Vector<K>& vopk) const;
```

5.6.7 The IntegrationMethods class

The `IntegrationMethods` class collects some `IntgMeth` objects that handle an integration method pointer with some additional parameters:

```
class IntgMeth
{
public:
    const IntegrationMethod* intgMeth;
    FunctionPart functionPart; //_allFunction, _regularPart, _singularPart
    real_t bound; //bound value
}
```

The bound value may be used to select an integration method regarding a criteria. For instance, in BEM computation the criteria is the relative distance between the centroids of elements. Besides, using `functionPart` member, different integration method may be called on different part of the kernel, see `Kernel` class. This class has only some constructors and print stuff:

```
IntgMeth(const IntegrationMethod& im, FunctionPart fp=_allFunction, real_t b=0);
IntgMeth(const IntgMeth&);
~IntgMeth();
IntgMeth& operator=(const IntgMeth&);
void print(std::ostream&) const;
void print(PrintStream&) const;
friend ostream& operator<<(ostream&, const IntgMeth&)
```

For safe multithreading behaviour, copy constructor and assign operator do a full copy of `IntegrationMethod` object.

The `IntegrationMethods` class is simply a vector of `IntMeth` object:

```
class IntegrationMethods
{public:
    vector<IntMeth> intgMethods;
    typedef vector<IntMeth>::const_iterator const_iterator;
    typedef vector<IntMeth>::iterator iterator;
    ...}
```

This class provides many constructors depending to the number of integration methods given (up to 3 at this time), to some shortcuts :

```
IntegrationMethods() {};
IntegrationMethods(const IntegrationMethod&,FunctionPart=_allFunction , real_t=0);
...
void add(const IntegrationMethod&, FunctionPart=_allFunction , real_t=0);
void push_back(const IntMeth&);
```

To handle more than 3 integration methods, use the `add` member function.

Besides, the class offers some accessors and print stuff:

```
const IntMeth& operator [] (number_t) const ;
vector<IntMeth>::const_iterator begin() const;
vector<IntMeth>::const_iterator end() const;
vector<IntMeth>::iterator begin();
vector<IntMeth>::iterator end();
bool empty() const {return intgMethods.empty();}
void print(ostream&) const;
void print(PrintStream&) const;
friend ostream& operator<<(ostream&, const IntegrationMethods&);
```

In BEM computation, the following relative distance between two elements (E_i, E_j) is used:

$$dr(E_i, E_j) = \frac{\|C_i - C_j\|}{\max(\text{diam}(E_i), \text{diam}(E_j))} \text{ where } C_i \text{ is the centroid of } E_i.$$

So defining for instance

```
IntegrationMethods ims(SauterSchwabIM(4), 0., symmetrical_Gauss, 5, 1.,
    symmetrical_Gauss, 3);
```

the BEM computation algorithm will perform on function (all part):

Sauter-Schwab with quadrature of order 4 on segment	if $dr(E_i, E_j) = 0$
Symmetrical gauss quadrature of order 5 on triangle	if $0 < dr(E_i, E_j) \leq 1$
Symmetrical gauss quadrature of order 3 on triangle	if $dr(E_i, E_j) > 1$

library :	<code>finiteElements</code>
header :	<code>Quadrature.hpp</code> <code>QuadratureRule.hpp</code> <code>IntegrationMethod.hpp</code>
implementation :	<code>Quadrature.cpp</code> <code>QuadratureRule.cpp</code> <code>IntegrationMethod.cpp</code>
unitary tests :	<code>test_Quadrature.cpp</code>
header dependences :	<code>config.h</code> , <code>utils.h</code> , <code>LagrangeQuadrangle.hpp</code> , <code>LagrangeHexahedron.hpp</code> , <code>mathsResources.h</code>

The *space* library collects all the classes and functionalities related to functions spaces. In the framework of the code, a space is a finite dimension vector space of functions defined on a geometrical domain (see [GeomDomain](#)), in other words, the space spanned by a finite number of independent functions (global basis functions). This collection of functions may be given either explicitly (say spectral basis) or using local definition of functions on geometric elements (say finite element basis). Besides, a space may be also a subspace of a space (in the meaning of a function subspace or the trace of a space on a geometric sub domain) or a tensor product of spaces. The space we used is no more than the finite dimension space involved in discrete variational formulations except the essential boundary conditions. Then the [Space](#) class is an essential concept of the code, in particular for users who have to deal with. A finite element space is essentially a list of [Elements](#) which supports finite element. A spectral space is essentially a list of [GlobalFunction](#) defined by the user either in an analytical form or an interpolate form. Finally, there are two related classes: the [Unknown](#) class to deal with an abstract element of a space and [Dof](#) which represents an generalized index of an unknown.

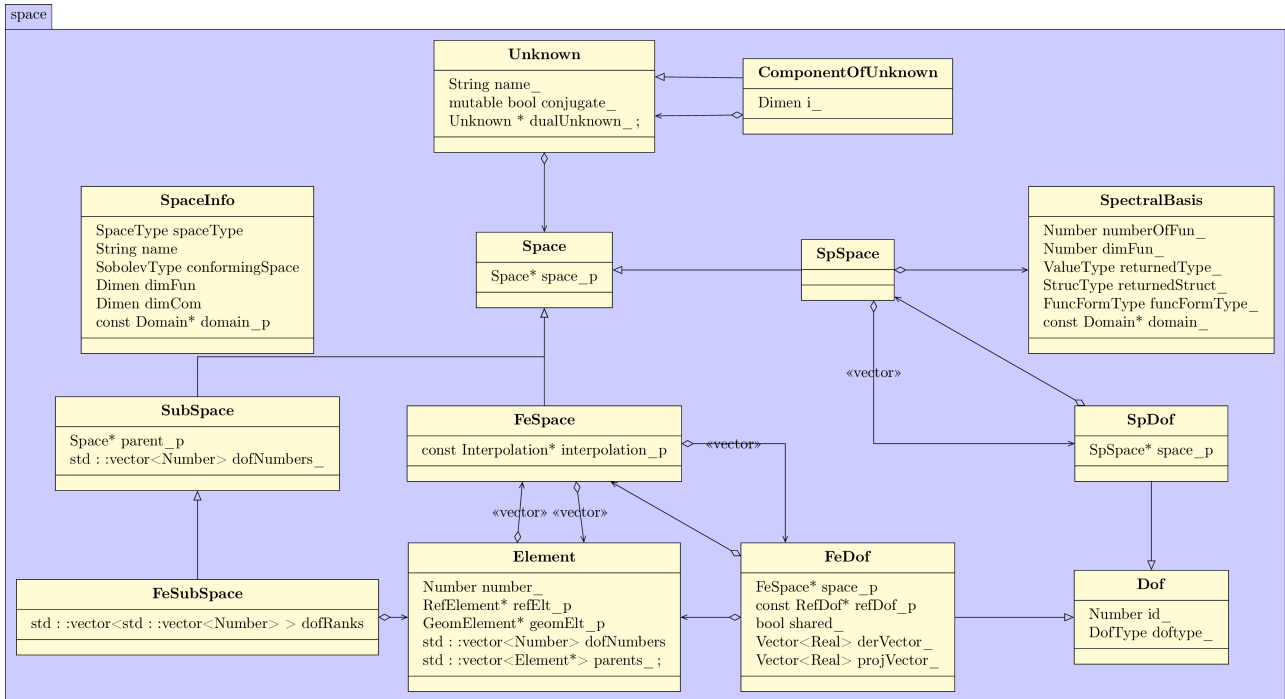


Figure 6.1: *space* library organisation

6.1 Space management

6.1.1 The Space and SpaceInfo classes

The [Space](#) class manages different kind of spaces using inherited classes: finite element space ([FeSpace](#)), spectral space ([SpSpace](#)), sub space ([SubSpace](#)) and product space ([ProdSpace](#)). To avoid the user to deal explicitly with this space hierarchy, we use the following programming paradigm: [Space](#) class has a pointer to a Space as attribute, which contains either its own address or the address of a child. In a

sense, the `Space` class is not an abstract class but has a the behaviour of an abstract one. When an end user declares a `Space` object, he induces the construction of a child `Space`. With this trick, an end user instantiates only `Space` object, never the child spaces. As a consequence, it requires a little more memory because member attributes are duplicated. In order to limit this effect, the common characteristics of a space are collected in the `SpaceInfo` class:

```
class SpaceInfo
{public :
    String name_; // name of space
    Domain * domain_p; // geometric domain supporting the space
    SobolevType spaceConforming_; // conforming space (L2,H1,Hdiv,Hcurl,H2,Hinf)
    dimen_t dimFun_; // number of components of the basis functions
    Dimen dimCom; // number of components of the unknowns
    SpaceType spaceType_; // space type
    (FESpace,SPSpace,SubSpace,ProductSpace)
    SpaceInfo() {};
    SpaceInfo(const String &,SobolevType,Domain *,dimen_t , SpaceType)
};
```

The general characteristics of a space are the geometric domain supporting the basis functions, the space conformity which "tells" the regularity of the basis functions and the dimension of the basis functions. The space conformity property is a way to distinguish discontinuous finite element approximation from continuous one. The space conformity is given by the `SobolevType` enumeration declared in the `config.hpp` header file:

```
enum SobolevType {L2=0, L2=L2,H1, H1=H1,_Hdiv, Hdiv=Hdiv,_Hcurl, Hcurl=Hcurl,
    _Hrot=Hcurl, Hrot=Hrot,_H2, H2=H2,_Hinf, Hinf=Hinf}
```

To be user friendly, several names are available for the same space.

The parent class `Space` has only two pointers and a static vector managing the list of all defined spaces:

```
class Space
{protected :
    Space * space_p; //pointer to the "true" space (child)
    SpaceInfo * spaceInfo_p; //pointer to the space information
    structure
public :
    static std::vector<Space*> theSpaces; //unique list of defined spaces
```

Regarding the type of spaces, the `Space` provides several constructors to construct child spaces:

```
//constructors of trivial space of dim n, say Rn
Space(number_t, const string_t&="Rn");
Space(const GeomDomain&, number_t, const string_t&="Rn");

//constructors addressing spectral space
Space(const SpectralBasis&, const string_t&); //from spectral basis
Space(const GeomDomain&, const Function&, number_t, dimen_t, const string_t&);
Space(const GeomDomain&, const Function&, number_t, dimen_t);
Space(const GeomDomain&, const Function&, number_t, const string_t&);
Space(const TermVectors&, const string_t&); //from interpolated functions

//constructors addressing FeSpace type from Interpolation or FE types
Space(const GeomDomain&, const Interpolation&, const string_t&, bool opt = true);
Space(const GeomDomain&, PolynomType, const string_t&, bool opt = true);
Space(const GeomDomain&, QolynomType, const string_t&, bool opt = true);
Space(const GeomDomain&, FeFaceType, const string_t&, bool opt = true);
Space(const GeomDomain&, FeEdgeType, const string_t&, bool opt = true);
```

```

//constructors addressing SubSpace type
Space(const GeomDomain&, Space&, const string_t& na = "");
Space(const GeomDomain&, const vector<number_t>&, Space&, const string_t& na = "");
Space(const std::vector<number_t>&, Space&, const string_t& na = "");

//tools related to constructors
void buildSpFun(const GeomDomain&, const Function&, number_t,
               dimen_t, const string_t&);
SubSpace* createSubSpace(const GeomDomain&, Space&, const string_t& na);
void createSubSpaces(const vector<const GeomDomain*>&, vector<Space*>&);
Space* buildSubspaces(const vector<const GeomDomain*>&, vector<Space*>&);

```

These constructors work as follows ((*child*)Space denotes any inherited space class):

```

(child)Space* csp_p=new (child)Space(...); //create the (child)Space
space_p=static_cast<Space*>(csp_p);        //cast (child)Space* to Space*
spaceInfo_p=csp_p->spaceInfo_p;            //copy the SpaceInfo pointer
theSpaces.push_back(this);                 //store space in list of spaces

```

Note that it is the child constructors that set all the attributes (general ones and specific ones).

The Space class provides some accessors, some being virtual:

```

string_t name() const;                //space name
SobolevType conformingSpace() const;  //space conformity
const GeomDomain* domain() const;     //related domain
dimen_t dimDomain() const;            //dimension of related domain
dimen_t dimPoint() const;            //dimension of points
dimen_t dimFun() const;              //dimension of basis functions
SpaceType typeOfSpace() const;        //type of space
SpaceType typeOfSubSpace() const;     //type of sub-space
virtual bool isSpectral() const;       //true if spectral space
virtual bool isFE() const;            //true if FE space or FeSubspace
const Space* space() const;           //pointer to true space object
virtual number_t dimSpace() const;     //space dimension
virtual number_t nbDofs() const;       //number of dofs
virtual const FeSpace* feSpace() const; //pointer to child fespace
virtual FeSpace* feSpace();            //pointer to child fespace
virtual const FeSubSpace* feSubSpace() const; //pointer to child fesubspace
virtual const SpSpace* spSpace() const; //pointer to child fesubspace
virtual const SubSpace* subSpace() const; //pointer to child subspace
virtual SubSpace* subSpace();          //pointer to child subspace
virtual const Space* rootSpace() const; //pointer to root space
virtual Space* rootSpace();            //pointer to root space
virtual ValueType valueType() const;   //value type of basis function
virtual StrucType strucType() const;   //structure type of basis function
virtual bool include(const Space*) const; //inclusion test

```

To understand how works the non abstract/abstract class paradigm, here is the implementation of the access function *feSpace*:

```

FeSpace* Space::feSpace() const
{ if(space_p!=this) return space_p->feSpace(); //call the true feSpace
  error("is_not_fespace",name());
  return 0;
}
FeSpace* FeSpace::feSpace() const {return *this;}

```

The test `space_p!=this` means that the current space is a user space (it has a child), the `fespace()` function is called by its child. By polymorphic behaviour, if `space_p` is a `FeSpace` the `FeSpace::fespace()` is called else `Space::fespace()` is again called, but with `space_p=this` and

an error is thrown because the current space is not a `FeSpace`.

As you will see in section 6.2, the dof number is relative to the parent space. To get the global number of a dof or the dof itself and to manage dofs, `Space` provides the following functions :

```
virtual number_t dofid(number_t) const;           //n-th DoF id of space
virtual std::vector<number_t> dofIds() const;      //DoF ids on space
virtual const DoF& dof(number_t) const;           //n-th dof (n=1,...)
virtual std::pair<number_t, number_t> renumberDofs(); //optimize dofs numbering
virtual void shiftDofs(number_t n);                //shift dofs numbering from n (SpSpace)
```

When space is a FE space or a FE subspace, tools to access to finite element informations are provided:

```
virtual const Interpolation* interpolation() const;
virtual number_t nbOfElements() const;
virtual const Element* element_p(number_t k) const;
virtual const Element* element_p(GeomElement* gelt) const;
virtual number_t numElement(GeomElement* gelt) const;
virtual vector<number_t> elementDofs(number_t k) const;
virtual vector<number_t> elementParentDofs(number_t k) const;
virtual const set<RefElement*>& refElements() const;
virtual void buildgelt2elt() const;
```

Finally, the `Space` provides some printing facilities:

```
void printSpaceInfo(std::ostream &) const;
virtual void print(std::ostream &) const;
friend std::ostream& operator<<(std::ostream&, const Space &);
```

Some functions managing subspaces (comparison, merging) are also available:

```
bool compareSpaceUsingSize(const Space*, const Space*);
vector<number_t> renumber(const Space*, const Space*);
Space& subSpace(Space&, const GeomDomain&);
inline Space& operator |(Space&, const GeomDomain&);
Space* mergeSubspaces(Space*&, Space*&, bool newSubspaces=false);
Space* mergeSubspaces(std::vector<Space*>&, bool newSubspaces=false);
Space* unionOf(std::vector<Space*>&);
vector<Point> dofCoords(Space&, const GeomDomain&);
```

Linked to the `Space` class, the `DoF` class stores characteristics of degree of freedom. Degree of freedom is like a key index of components of element of space. Generally, it handles a numeric index and carries some additionnal informations. From `DoF` class inherit the classes `FeDoF` and `SpDoF` (see section DoF).

library :	<code>space</code>
header :	<code>Space.hpp</code>
implementation :	<code>Space.cpp</code>
unitary tests :	<code>test_Space.cpp</code>
header dependences :	<code>config.h</code> , <code>geometry.h</code> , <code>Df.hpp</code>

Now, let's have a look at the child space classes.

6.1.2 The `FeSpace` class

A finite elements space is defined is by a finite set of functions given by interpolated form on a mesh. The inherited `FeSpace` class is defined as follows :


```

class FeSpace : public Space
{
protected :
    const Interpolation* interpolation_p;    //!< pointer to finite element
                                             interpolation description
    bool optimizedNumbering_;               //!< true if dof numbering is
                                             bandwidth optimized

public:
    std::vector<Element> elements;           //!< list of finite elements
    std::vector<FeDof> dofs;                //!< list of global degrees of freedom

    static Number lastEltIndex;             //!< to manage a unique index for
                                             elements
};

```

This class provides one constructor, using two member functions:

```

FeSpace(const Domain&, const Interpolation&, Dimen, const String&, bool opt=true);
void buildElements();    //!< construct the list of elements
void buildFeDofs();      //!< construct the list of FeDofs

```

The `buildElements()` member functions create the list of elements (`Element` objects) from the list of geometric elements belonging to the domain supporting the space. The `buildFeDofs()` function is the most important one. It constructs from reference dofs the list of global dofs using a general algorithm. It works as follows:

- case of no shared dof (L2 conforming space), concatenate all local dofs and exit
- case of shared dofs
 - construct a string index of dofs by traveling elements with the convention
 - "V i" for a dof on vertex i
 - "E i j l" for the l-th dof on edge [Vi, Vj], i<j, not a vertex dof
 - "F i j k l" for the l-th dof on face [Vi, Vj, Vk], i<j<k, not an edge dof
 - "N i l" for the l-th dof on element i (non shared dof)
 - construct the list of global dofs from this index and update elements

To insure the right matching between shared dofs on edge or face, this algorithm uses the `sideDofsMap` and `sideOfSideDofsMap` member functions of finite element classes. These functions relates dof numbering to one edge/face to an other edge/face.

The `FeSpace` class provides also some specific accessors:

```

virtual Number nbDofs() const;    //!< number of dofs (a dof may be a vector dof)
Dimen dimElements() const;    //!< return the dimension of FE elements
const Interpolation* interpolation() const;    //!< return pointer to interpolation
Number nbOfElements() const;    //!< number of elements

```

The `FeSpace` class provides also some specific renumbering functions:

```

Graph graphOfDofs();    //!< create dof connection graph
const Element* element_p(Number k) const;    //!< access to k-th (k>=0) element
                                             (pointer)
std::vector<Number> elementDofs(Number k) const;    //!< access to dofs ranks (local
                                             numbering) of k-th (>=0) element
std::vector<Number> elementParentDofs(Number k) const;    //!< access to dofs ranks
                                             (parent numbering, same as local) of k-th (>=0) element

```

library :	space
header :	FeSpace.hpp
implementation :	FeSpace.cpp
unitary tests :	test_Space.cpp
header dependencies :	config.h, Elements.hpp, Space.hpp, dof.hpp, finiteElement.h

6.1.3 The SpSpace class

A spectral space is defined by a finite set of functions given by analytic expression or by interpolated form. These global basis functions defined over a geometric domain are collected in the [SpectralBasis](#) class declared in the *SpectralBasis.hpp* header file (see the section [SpectralBasis](#)). The inherited [SpSpace](#) class is defined as follows:

```
class SpSpace : public Space
{
protected :
    SpectralBasis* spectralBasis_; //! object function encapsulating the spectral
    functions
    //!either defined in an analytical form or by interpolated functions (TermVector)
public :
    std::vector<SpDof> dofs; //! list of global degrees of freedom
};
```

This class provides one constructor, a destructor, several accessors and a printing function:

```
SpSpace(const String&, Domain&, number_t, dimen_t, SpectralBasis*);
~SpSpace();
SpectralBasis* spectralBasis() const;
virtual Number nbDofs() const; //!< number of dofs (a dof may be a vector dof)
```

library :	space
header :	SpSpace.hpp
implementation :	SpSpace.cpp
unitary tests :	test_Space.cpp
header dependencies :	config.h, Elements.hpp, Space.hpp, dof.hpp

6.1.4 The SubSpace and FeSubSpace classes

A subspace is part of a space. Its dofs are a subset of the parent dofs. The inherited [SubSpace](#) class is defined as follows:

```
class SubSpace : public Space
{
protected :
    Space* parent_p; //!< the parent space
    std::vector<Number> dofNumbers_; //!< global numbers of D.o.Fs in parent D.o.Fs
    numbering
};
```

This class provides two constructor, a destructor, and several specific accessors about the `dofNumbers_` attribute or about the type of subspace :

```
SubSpace() //!< default constructor
SubSpace(const Domain&, Space&, const String& na = ""); //!< constructor specifying
geom domain, parent space and name
```

```

~SubSpace(); //! destructor

std::vector<Number>& dofNumbers() //!< access to dofNumbers
    list
const std::vector<Number>& dofNumbers() const //!< access to dofNumbers
    list (const)
std::vector<Number> dofRootNumbers() const; //!< access to dofNumbers
    list in root space numbering
Number dofRootNumber(Number k) const; //!< access to dof number
    in root space numbering
virtual Number nbDofs() const //! number of dofs (a dof
    may be a vector dof)
virtual bool isFeSubspace();
virtual const FeSubSpace* fesubspace() const;

```

It also provides functions for numbering management:

```

void createNumbering(); //!< create numbering of subspace DoFs
void dofsOfFeSubspace(); //!< build dofs numbering of Fe SubSpace
void dofsOfSpSubspace(); //!< build dofs numbering of Sp SubSpace
void dofsOfSubSubspace(); //!< build dofs numbering of Subspace of SubSpace

```

library :	space
header :	SubSpace.hpp
implementation :	SubSpace.cpp
unitary tests :	test_Space.cpp
header dependencies :	config.h, Elements.hpp, Space.hpp, Dof.hpp

The **FeSubSpace** class concerns subspaces of finite element spaces. It is defined as follows:

```

class FeSubSpace : public SubSpace
{
public :
    std::vector<Element*> elements; //!< list of finite elements (or
        subelements)
    std::vector<std::vector<Number>> dofRanks; //!< numbering of D.o.Fs of
        domain elements in domain D.o.Fs
};

```

This class provides one constructor, a destructor, and several specific functions, also provided by the **FeSpace** class :

```

FeSubSpace(const Domain&, Space&, const String& na = ""); //!< constructor
    specifying geom domain, parent space and name
~FeSubSpace(); //!< destructor
Number nbOfElements() const; //!< number of elements
const Element* element_p(Number k) const; //!< access to k-th (k>=0)
    element (pointer)
std::vector<Number> elementDofs(Number k) const; //!< access to dofs ranks
    (local numbering) of k-th (>=0) element
std::vector<Number> elementParentDofs(Number k) const; //!< access to dofs ranks
    (parent numbering) of k-th (>=0) element

```

library :	space
header :	FeSubSpace.hpp
implementation :	FeSubSpace.cpp
unitary tests :	test_Space.cpp
header dependencies :	config.h, Elements.hpp, SubSpace.hpp, dof.hpp

6.1.5 The ProdSpace class

The `ProdSpace` class manages product of spaces using a vector of space pointers :

```
class ProdSpace : public Space
{
protected :
    std::vector<Space*> spaces_;
public :
    virtual Number dimSpace() const;
    virtual Number nbdofs() const;
    virtual const Space* rootSpace() const
    {return static_cast<const Space*>(this);}
    virtual Space* rootSpace()
    {return static_cast<Space*>(this);}
    virtual Number dofId(Number n) const
    {error("no_dof_numbering", name());}
    return 0;}
};
```

It is currently not used.

6.1.6 The Spaces class

`Spaces` is an alias of `PCollection<Space>` class that manages a collection of `Space` objects, in fact a `std::vector<Space*>`. Be cautious when using it because the `Space` pointers are shared; in particular when using temporary instance of `Space`! It is the reason why the `PCollectionItem` class is overloaded to protect pointer in assign syntax `sp(i)=Space(...)`:

```
template<> class PCollectionItem<Space>
{public:
    typename std::vector<Space*>::iterator itp;
    PCollectionItem(typename std::vector<Space*>::iterator it) : itp(it){}
    Space& operator = (const Space& sp); //protected assignment
    operator Space&(){return *itp;} //autocast PCollectionItem->Space&
};
```

Main usages of this class are the following:

```
Space V1(omega, _P1, "V1"), V2(omega, _P2, "V2"), V3(omega, _P3, "V3");
Spaces Vs2(V1, V2);
Spaces Vs3(V1, V2, V3);
Spaces Vs4; Vs4<<V1<<V2<<V3<<V1;
Spaces Vs5(5);
for (Number i=1; i<=5; i++)
    Vs5(i)=Space(omega, interpolation(Lagrange, _standard, i, H1), "V_"+tostring(i));
```

C¹¹₊₊ If C++11 is available (the library has to be compiled in C++11), the following syntax is also working:

```
Spaces vs={V1, V2, V3};
```

Collection of spaces can also be set by the following functions from one or several domains and one or several interpolations:

```

Spaces spaces(const Domains&,const Interpolations&,bool=true);
Spaces spaces(const Domains&,const Interpolation&,bool=true);
Spaces spaces(const GeomDomain&,const Interpolations&,bool=true);

```

```

library : space
header : ProdSpace.hpp
implementation : ProdSpace.cpp
unitary tests : test_space.cpp
header dependences : config.h, Space.hpp

```

6.2 DoF management

6.2.1 The Dof class

Linked to the **Space** class, the **Dof** class stores characteristics of degree of freedom. Degree of freedom is like a key index of components of element of space. Generally, it handles a numeric index and carries some additional informations. For instance, in case of Lagrange interpolation it carries the geometrical point associated to the basis function. From **Dof** class inherit the classes **FeDof** and **SpDof**.

```

class Dof
{protected :
    number_t id_;           //id of dof
    DofType doftype_;       //type of dof
public :
    Dof(DofType t=_otherDof, number_t i=0);
    virtual ~Dof() {};
    DofType doftype() const;
    number_t id() const;
    number_t& id();
    virtual void print(std::ostream&) const;
    friend std::ostream& operator<<(std::ostream&, const Dof &);
};

```

The type of Dof are enumerated by

```

enum DofType {_feDof, _spDof, _otherDof};

```

6.2.2 FeDof and SpDof classes

Inherited classes are very simple:

```

class FeDof: public Dof
{protected :
    FeSpace* space_p;       // pointer to FeSpace
    const RefDof* refDof_p; // pointer to a Reference D.o.F
    bool shared_;           // implies D.o.F is shared between elements
    Vector<Real> derVector_; // direction vector(s) of a derivative D.o.F
                           // (specific interpolation)
    Vector<Real> projVector_; // direction vector(s) of a projection D.o.F
                           // (specific interpolation)
    std::vector<std::pair<Element*, Number> > inElements; // list of elements
                           // related to Dof and local number of dof in elements
public :
    FeDof();
    FeDof(FeSpace& sp, number_t i=0)

```

```
virtual void print(std::ostream&) const;
};
```

```
class SpDof : public Dof
{protected :
    SpSpace * space_p;           //pointer to SpSpace
public :
    SpDof()
    SpDof(SpSpace& sp, number_t i)
    virtual void print(std::ostream&) const;
};
```

```
library : space
header : DoF.hpp
implementation : DoF.cpp
unitary tests : test_Space.cpp
header dependencies : config.h, utils.h
```

6.2.3 The DofComponent class

This class defined an extended representation of dof, useful to adress components of a vector dof:

```
class DofComponent
{
public :
    const Unknown* u_p;           //unknown
    Number dofnum;                //dof number
    Dimen numc;                   //component

    DofComponent(const Unknown* u=0, Number dn=0, Dimen nc=1);
    DofComponent dual() const;
    void print(std::ostream& os) const;
    friend std::ostream& operator<<(std::ostream&, const DofComponent&);
};
```

The dual() function returns the 'dual' dof component. Besides, this class provides some comparison operators (partial ordering) :

```
bool operator < (const DofComponent&, const DofComponent&);
bool operator ==(const DofComponent&, const DofComponent&);
bool operator !=(const DofComponent&, const DofComponent&);
```

and some functions manipulating list of component dofs :

```
vector<DofComponent> createCdofs(const Unknown* u, const vector<Number>& dofs);
vector<Number> renumber(const vector<DofComponent>&, const vector<DofComponent>&);
```

```
library : space
header : DoF.hpp
implementation : DoF.cpp
unitary tests : test_Space.cpp
header dependencies : config.h, utils.h
```

6.3 Unknown management

6.3.1 The Unknown class

The `Unknown` class manages element of a space (`Space`). It is mainly used as an abstract object in the description of a variational problem. It has the `ComponentOfUnknown` class as child which represents a component of a vector unknown. Do not confuse vector unknown on a scalar space and scalar unknown on a vector space. Vector unknown on a scalar space is a way to define an unknown on a product of scalar spaces (same space!), for instance the displacement field. Whereas on a vector space (say a space spanned by vector functions), unknown are generally a scalar one; for instance Hrot or Hdiv conform spaces. Up to now, it is not possible to deal with general product spaces.

```
class Unknown
{protected:
    String name_;           // name of the unknown
    Space * space_p;        // pointer to the space where the unknown lives
    mutable bool conjugate_; // temporary flag for conjugate operation
    Unknown * dualUnknown_; // pointer to dual Unknown (test function)
    Number rank_;           // rank to order unknowns
public:
    bool isUnknown;         // true if an unknown, false if a test function
    static std::vector<Unknown*> theUnknowns; // list of all Unknowns
```

The temporary `conjugate_` flag is used to mark in a `OperatorOnUnknown` construction that an unknown is conjugate. The static attribute `theUnknowns` traces all defined unknowns. Besides, each unknown has a rank, by default its creation number (first unknown created has rank 1 and so on). This rank is used to order blocks in vector or matrix in case of multiple unknowns terms.

There is no class to deal with test function. There exists only an alias `TestFunction` of `Unknown`:

```
typedef Unknown TestFunction;
```

So, to distinguish, unknown from test function, the flag `isUnknown` is used. It is set to true when using the constructor from a `Space` and set to false when using the constructor from unknown, implicitly the test function constructor:

```
Unknown();
Unknown(const string_t&, Space&,
        dimen_t d=1, number_t r=0); // unknown constructor from space
Unknown(Unknown&, const string_t& na="",
        number_t r=0);              // test function constructor from unknown
~Unknown();
```

and the following accessors:

```
String name() const;           // return name
Space * space() const;         // return pointer to space
UnknownType type() const;      // return type of unknown
StrucType strucType() const;  // return the structure type of an unknown
number_t index() const;        // return index of unknown
bool conjugate() const;        // return the conjugate state flag
bool& conjugate();             // return the conjugate state flag
void conjugate(bool v) const;  // set the conjugate state flag
friend Unknown& conj(Unknown &); // conjugate an unknown
Number rank() const;           // return the unknown rank
```

The member function `index()` returns the rank of the unknown in the `theUnknowns` list. It may be useful to order the unknowns. The member function `type()` returns the type of unknown as an element of the `UnknownType` enumeration (defined in `config.hpp` header file):

```
enum UnknownType {_feUnknown ,_spUnknown ,_mixedUnknown};
```

There are particular virtual member functions linked to the **ComponentOfUnknown** child class:

```
virtual dimen_t nbOfComponents() const;    // dimension of unknown
virtual bool isComponent() const;         // true if a ComponentOfUnknown
virtual const Unknown* parent() const;   // return parent (or itself)
virtual Dimen componentIndex() const;    // return component index if a
ComponentOfUnknown, 0 if not
virtual const ComponentOfUnknown* asComponent() const; // return Unknown as a
ComponentOfUnknown if it is
Unknown& operator [](dimen_t);           // create the ComponentOfUnknown u_i
```

Unknown class provides printing facility also:

```
friend std::ostream& operator<<(std::ostream&, const Unknown&);

void setRanks(std::vector<Unknown*>&, const std::vector<Number>&);
void setRanks(Unknown&, Number r);
void setRanks(Unknown&, Number, Unknown&, Number);
void setRanks(Unknown&, Number, Unknown&, Number, ...);
```

The **setRanks** functions are useful to define the internal order of unknowns in multiple unknowns objects such as **TermMatrix** or **TermVector**. The unknown ranks have to be unique but it is not mandatory that they follow. By default, first created unknown has rank 1, the second created has rank 2, and so on.

6.3.2 The ComponentOfUnknown class

To access to a component of a vector unknown as an unknown, the **ComponentOfUnknown** child class is provided:

```
class ComponentOfUnknown: public Unknown
{protected :
    const Unknown * u_p;    //Unknown which this unknown is a component
    Dimen i_;              //index of the component
public :
    ComponentOfUnknown(const Unknown &, dimen_t);
    virtual Dimen nbOfComponents() const;
    virtual Dimen dimFun() const;
    virtual bool isComponent() const;
    virtual const Unknown* parent() const;
    virtual Dimen componentIndex() const;
    virtual const ComponentOfUnknown* asComponent() const;
};
```

Note that the operator **Unknown::[]**(**Dimen**) creates a **ComponentOnUnknown** object returned as an **Unknown** object.

6.3.3 The Unknowns class

Unknowns is an alias of **PCollection<Unknown>** class that manages a collection of **Unknown** objects, in fact a **std::vector<Unknown*>**. Be cautious when using it because the **Unknown** pointers are shared; in particular when using temporary instance of **Unknown**! It is the reason why the **PCollectionItem** class is overloaded to protect pointer in assign syntax **sp(i)=Unknown(...)**:

```
template<> class PCollectionItem<Unknown>
{public:
    typename std::vector<Unknown*>::iterator itp;
```

```

PCollectionItem(typename std::vector<Unknown*>::iterator it) : itp(it){}
Unknown& operator = (const Unknown& sp); //protected assignment
operator Unknown&(){return **itp;}      //autocast PCollectionItem->Unknown&
};

```

Main usages of this class are the following:

```

Unknown u1(V1,"u1"), u2(V2,"u2"), u3(V3,"u3");
Unknowns us1(u1,u2,u3);
Unknowns us2; us2<<u1<<u2<<u3;
Unknowns us5(5);
for (Number i=1;i<=5;i++) us5(i)=Unknown(Vs5(i),"u_"+tostring(i));

```

C¹¹ If C++11 is available (the library has to be compiled in C++11), the following syntax is also working:

```

Unknowns us={u1,u2,u3};

```

In a same way, **TestFunctions** is an alias of **PCollection<TestFunction*>** class that manages a collection of **TestFunction** objects. It works as the **Unknowns** class. The following function allows to create a collection of testfunctions related to a collection of unknowns:

```

TestFunctions dualOf(const Unknowns& us);

```

library :	space
header :	Unknown.hpp
implementation :	Unknown.cpp
unitary tests :	test_Space.cpp
header dependences :	config.h, utils.h

6.4 The Element class

A finite element is mainly defined by its geometric support and its finite element interpolation. It carries also its dof numbering. The dedicated **Element** class is defined as follows:

```

class Element
{
protected:
    FeSpace* feSpace_p;           //pointer to Finite Element parent space
    Number number_;              //element number in FeSpace
public:
    FeSubSpace* feSubSpace_p;     //pointer to Finite Element parent sub space
    RefElement* refElt_p;         //pointer to reference element object
    GeomElement* geomElt_p;       //pointer to Geometric Element support
    std::vector<Number> dofNumbers; //global numbering of element dofs
    std::vector<Element*> parents_; //parent elements when a side of element
};

```

The **Element** class provides:

- One constructor

```

Element(FeSpace*, Number, RefElement*, GeomElement*, FeSubspace*=0);

```

- Some accessors


```

Dimen dim() const;           //dimension of the element
Number number() const;       //return number_ attribute
ShapeType shapeType() const; //shape of the element
FEType feType() const;       //FE type ( _Lagrange, _Hermite, ...)
Number feOrder() const;      //FE numtype (order for Lagrange)
Dimen dimFun() const;        //dimension of shape functions

```

- a dofs renumbering function and a tool to get index of dof

```

void dofsRenumbered(const std::vector<Number>&, std::vector<FeDof>&);
number_t refDofNumber(number_t) const;

```

- some utilities to compute normal vectors at a physical point and to access to:

```

ShapeValues computeShapeValues(const Point&, bool withderivative = false) const;
Vector<real_t> computeNormalVector(const Point&) const;
Vector<real_t> computeNormalVector() const;
const Vector<real_t>& normalVector() const;
Vector<real_t>& normalVector();

```

- some interpolation tools:

```

Point toReferenceSpace(const Point& p) const; // physical to reference space
template<typename T>
T& interpolate(const Vector<T>&, const Point&, const std::vector<number_t>&,
               T&, DiffOpType =_id) const;
template<typename T>
Vector<T>& interpolate(const Vector<T>&, const Point&,
                     const std::vector<number_t>&, Vector<T>&,
                     DiffOpType =_id) const;
template<typename T>
T& interpolate(const Vector<Vector<T> >&, const Point&,
               const std::vector<number_t>&, T&,
               DiffOpType = _id) const;
template<typename T>
T& interpolate(const VectorEntry&, const Point&,
               const std::vector<number_t>&, T&,
               DiffOpType =_id) const;

```



Interpolation tools are useful when evaluating an element of FE space at a given point, say P . It is an expansive process requiring to locate the element containing the point P , to get its location in reference space (inverse map) to apply local interpolation. The previous `interpolate` functions are not concerned by the localization of element containing P , they only compute local interpolation.

- a function dedicated to split a finite element in first order finite elements

```

std::vector<std::pair<ShapeType, std::vector<Number> > >
splitO1(std::map<Number,Number>* renumbering) const;

```

- some print functions

```

void print(std::ostream&) const; //!< print utility
friend std::ostream& operator<<(std::ostream&, const Element&);

```

library :	space
header :	Element.hpp
implementation :	Element.cpp
unitary tests :	test_Space.cpp
header dependencies :	config.h, utils.h, finiteElements.h, geometry.h, Dof.hpp, SpectralBasis.hpp

6.5 Spectral basis management

6.5.1 The SpectralBasis class

For spectral finite elements, the basis is defined by a finite set of functions globally defined over a geometric domain. The `SpectralBasis` class is devoted to this set of functions over a spectral space. This class is defined as follows:

```
class SpectralBasis
{
protected :
    Number numberOffun_;           //!< number of function in the basis
    Number dimFun_;                //!< dimension of the spectral functions
    const Domain* domain_;        //!< geometric domain support
    ValueType returnedType_;       //!< type of returned value (one among _real,
                                   _complex)
    StrucType returnedStruct_;     //!< structure of returned value (one among
                                   _scalar, _vector)
    FuncFormType funcFormType_;   //!< type of basis functions
                                   (_analytical, _interpolated)
};
```

The `SpectralBasis` class provides:

- One constructor

```
SpectralBasis(Number n, Number d, const Domain& g, ValueType r = _real, StrucType s
= _scalar);
```

- Some accessors

```
Number numberOffun() const;
Dimen dimFun() const;
Number dimSpace() const;
FuncFormType funcFormType() const;
```

- some functions about the basis functions :

```
virtual Function& functions(); //!< return basis function object (only for
    SpectralBasisFun)
template<typename T>
T& functions(Number, const Point&, T&); //!< compute n-th function
template<typename T>
Vector<T>& functions(const Point& P, Vector<T>& res); //!< compute all functions
```

- some print functions

```
virtual void print(std::ostream&) const = 0;
```

As you may have noticed, the `SpectralBasis` class is an abstract class. Indeed, a spectral basis can be defined with an analytical expression or by a set of interpolate functions. That is why the `SpectralBasis` class has 2 childs : `SpectralBasisFun` and `SpectralBasisInt`.

6.5.2 The SpectralBasisFun class

The `SpectralBasisFun` class is dedicated to a spectral basis defined with an analytical expression, namely a `Function` object.

```
class SpectralBasisFun : public SpectralBasis
{
    protected :
        Function functions_;
    public :
        SpectralBasisFun(Domain&, Function&, Number, Number); ///  
constructor
};
```

6.5.3 The SpectralBasisInt class

The `SpectralBasisInt` class is dedicated to a spectral basis defined by a set of interpolate functions, namely a set of `TermVector` objects:

```
class SpectralBasisInt : public SpectralBasis
{
    protected :
        std::vector<TermVector*> functions_;
    public :
        SpectralBasisInt(Domain&, Number, Number); ///  
constructor
};
```

library :	space
header :	SpectralBasis.hpp
implementation :	SpectralBasis.cpp
unitary tests :	test_Space.cpp
header dependencies :	config.h, utils.h, geometry.h

7

The *operator* library

The *operator* library collects the classes and functionalities related to the description of differential operators acting on unknown or test function in linear, bilinear form and essential condition expressions. As we want to propose to end users, a C++ description of such operators as close as possible to the mathematical description, few operators are overloaded and a lot of possibilities are offered, for instance:

mathematical expression	C++ translation	comment
$\nabla(u)$	<code>grad(u)</code>	u unknown
$\nabla(u) \cdot \nabla(v)$	<code>grad(u) grad(v)</code>	u unknown, v test function
$(A * \nabla(u)) \cdot \nabla(v)$	<code>(A*grad(u)) grad(v)</code>	u unknown, v test function, A a matrix
$(F(x) * \nabla(u)) \cdot \nabla(\bar{v})$	<code>(F*grad(u)) grad(conj(v))</code>	u unknown, v test function, F a function

As a consequence, this library is quite intricate. It is based on

- the class of differential operator: `DifferentialOperator` collecting all the differential operators that can be applied to an unknown or a test function,
- the class of operand: `Operand`, describing the operations performed at left and right of a differential operator acting on an unknown or a test function,
- the class of operator acting on unknown: `OperatorOnUnknown` which collects the differential operator acting on unknown and the left and right operands if exists.
- the `OperatorOnUnknowns` which relies two operators on unknown as they appear in a bilinear form.
- the class of linear combination of operators acting on unknown: `LcOperatorOnUnknown` which manages a list of pair of `OperatorOnUnknown` and complex coefficient. This class is also able to attach geometric domain to each pair.

A linear form will be defined from one `OperatorOnUnknown` object and a bilinear form from `OperatorOnUnknowns` objects; see the *form* library

Besides, this library provides additional classes to deal with expression involving `Kernel` functions:

- the `OperatorOnKernel` class which handles differential operator applied to a kernel.
- the `KernelOperatorOnUnknowns` class which relies two operators on unknown and a kernel.
- the `TensorKernel` class inherited from `Kernel` and implementing particular kernel.

7.1 The `DifferentialOperator` class

The `DifferentialOperator` class is intended to describe various operators that can be act on a scalar function or a vector function. They may be either differential operator (involving derivatives) or 0 order operators (not involving derivative). The attributes of this class are the following:

```

class DifferentialOperator
{
private:
    DiffOpType type_;           //type of differential operator
    number_t order_;           //derivation order
    bool requiresExtension_;    //does operator involve non-tangential derivatives
    bool requiresNormal_;      //is normal vector required for this operator
    String name_;              //operator name

```

The differential operators supported are list in the *DiffOpType* enumeration:

```

enum DiffOpType
{
    _id, _d0, _dt = _d0, _d1, _dx = _d1, _d2, _dy = _d2, _d3, _dz = _d3,
    _grad, _nabla = _grad, _div, _curl, _rot = _curl,
    _gradS, _nablaS = _gradS, _divS, _curlS, _rotS = _curlS,
    _nx, _ndot, _ncross, _ncrossncross, _ndotgrad, _ndiv, _ncrosscurl,
    _divG, _gradG, _nablaG = _gradG, _curlG, _rotG = _curlG, _epsilon,
    _jump, _mean}

```

with the following meaning (u is either a scalar or vector unknown, x, y and z are the cartesian coordinates and n is the normal):

enum value	mathematical	builtin functions	unknown
_id	identity	id(u) or u	scalar or vector
_d0=_dt	∂_t	d0(u) or dt(u)	scalar or vector
_d1=_dx	∂_x	d1(u) or dx(u)	scalar or vector
_d2=_dy	∂_y	d2(u) or dy(u)	scalar or vector
_d3=_dz	∂_z	d3(u) or dz(u)	scalar or vector
_grad=_nabla	∇	grad(u) or nabla(u)	scalar or vector
_div	div	div(u)	vector
_curl=_rot	curl	curl(u) or \verbrot(u)	vector
_gradS=_nablaS	∇_τ (surfacic)	gradS(u) or nablaS(u)	scalar or vector
_divS	div_τ (surfacic)	divS(u)	vector
_curlS=_rotS	curl_τ (surfacic)	curlS(u) or rotS(u)	vector
_gradG=_nablaG	$\nabla_{abc} = (a\partial_x, b\partial_y, c\partial_z)$	gradG(u,a,b,c) or nablaG(u,a,b,c)	scalar or vector
divG	$\nabla{abc} \cdot$	divG(u,a,b,c)	vector
_curlG=_rotG	$\nabla_{abc} \times$	curlG(u,a,b,c) or rotS(u,a,b,c)	vector
_epsilon	elastic tensor	epsilon(u)	vector
_nx	$n \cdot$	nx(u) or _n*u	scalar
_ndot	$n \cdot$	ndot(u) or _n.u	vector
_ncross	$n \times$	ncross(u) or _n^u	vector
_ncrossncross	$n \times n \times$	ncrossncross(u) or _n^_n^u	vector
_ndotgrad	$n \cdot \nabla$	ndotgrad(u) or _n.grad(u)	scalar
_ndiv	$n \text{div}$	ndiv(u) or _n*div(u)	vector
_ncrosscurl	$n \times \text{curl}$	ncrosscurl(u) or _n^curl(u)	vector
_jump	[] (jump across)	jump(u)	scalar or vector
_mean	{ } (mean across)	mean(u)	scalar or vector

There is only one definition of differential operators stored (using their pointers) in the static variable:

```

static std::vector<DifferentialOperator*> theDifferentialOperators;

```

There is an explicit constructor of DifferentialOperator objects but they are also created using the find function:

```
DifferentialOperator* findDifferentialOperator(DiffOpType);
```

which creates the differential operator of a given type if it does not exist and returns a pointer to the differential operator if it exists or has just been created.

This class provides some useful accessors functions (only read):

```
String name() const {return name_;}
DiffOpType type() const {return type_;}
number_t order() const {return order_;}
bool normalRequired() const {return requiresNormal_;}
bool extensionRequired() const {return requiresExtension_;}
```

and printing facilities:

```
std::ostream& operator<<(std::ostream&, const DifferentialOperator&);
void printListDiffOp(std::ostream&);
```

library :	operator
header :	DifferentialOperator.hpp
implementation :	DifferentialOperator.cpp
unitary tests :	test_operator.cpp
header dependences :	config.h, utils.h

7.2 The Operand class

The **Operand** class stores a **AlgebraicOperator**, a user data (either **Function** or **Value**) and conjugate, transpose flags. It is mainly used by the **OperatorOnUnknown** class as left or right-hand side of a generalized differential operator (see the next section). The only allowed algebraic operations defined in the **AlgebraicOperator** enumeration, are:

- the standard product, operator *****
- the inner product (hermitian product), operator **|**
- the cross product, operator **^**
- the contracted product, operator **%**

```
enum AlgebraicOperator {_product, _innerProduct, _crossProduct, _contractedProduct};
class Operand
{
protected :
    const Value* val_p;           // pointer to operand object (a Value)
    const Function* fun_p;        // pointer to Function object
    bool isFunction_;             // function flag
    bool conjugate_;              // true if the operand has to be conjugated
    bool transpose_;              // true if the operand has to be transposed
    AlgebraicOperator operation_; // operand operator
```

This class provides few public constructors:

```
Operand(const Function&, AlgebraicOperator); // from a Function
Operand(const Value&, AlgebraicOperator);    // from a Value (Value is copied)
Operand(const Value*, AlgebraicOperator);    // from a Value* (Value is not copied)
template <typename T>
Operand(const T &, AlgebraicOperator);      // from a scalar, a Vector or a Matrix
```

The templated constructor avoids the user to give explicitly a Value object, "ordinary" objects such scalar, vector or matrix may be passed to the constructor.

The other member functions are some accessors functions:

```
bool isFunction() const;
bool conjugate() const;
bool& conjugate();
bool transpose() const;
bool& transpose();
AlgebraicOperator operation() const;
const Value& value() const;
const Function& function() const;
template<class T> T& valueT() const;
```

There are two printing facilities:

```
void Operand::print(std::ostream&) const;
std::ostream& operator<<(std::ostream&,const Operand &);
```



The `Operand` class is an internal class linked to the `OperatorOnUnknown` class. Normally, an end user does not use this class.

library :	operator
header :	Operand.hpp
implementation :	Operand.cpp
unitary tests :	test_operator.cpp
header dependences :	config.h, utils.h

7.3 The OperatorOnUnknown class

The `OperatorOnUnknown` class is intended to store the description of an expression involving a differential operator acting on an unknown (or a test function) with possibly operations at the left and the right. More precisely, the general form of the operator on unknown is:

$$\text{tr}(\text{data}) \text{ op dif}(\text{tr}(\text{u})) \text{ op tr}(\text{data})$$

where

- **u** is an unknown,
- **dif** a differential operator,
- **data** is a user data (value or function),
- **op** an algebraic operation among * (product), | (inner product), ^ (cross product), : (contracted product),
- **tr** a transformation among conj (conjugate), tran (transpose) and adj (adjoint).

All is optional except the unknown **u**.

The simplest form is **u** and a more complex one is for instance, something like this (*f* a matrix and *g* a vector):

$$\text{adj}(\text{f})^* \text{ grad}(\text{conj}(\text{u}[2])) | \text{conj}(\text{g})$$

which mathematically reads:

$$f^* \nabla \bar{u}_2 \cdot \bar{g}$$

The class has the following attributes:

```
class OperatorOnUnknown
{protected :
    const Unknown * u_p;           //unknown involved in operator
    bool conjugateUnknown_;        //true if the unknown has to be conjugated
    DifferentialOperator * difOp_p; //differential operator involved in operator
    Operand * leftOperand_p;       //object before the diff operator
    Operand * rightOperand_p;      //object after the diff operator
    bool leftPriority_;             //priority order flag, true if left operand is
    prior
    Vector<complex_t> coefs_;       //coefficients for operator (gradG, divG, curlG)
    ValueType type_;               //returned value (_real, _complex)
    StrucType struct_;             //returned structure
    (_scalar, _vector, _matrix)
```

Operand is a class storing a user data (value or function) and an algebraic operation (* | ^ %).

The class provides some basic constructors (from unknown and differential operator type), some useful constructors (from unknown and function or value), a copy constructor, the assignment operator and a destructor:

```
OperatorOnUnknown(const Unknown* un=0, DiffOpType=_id);
OperatorOnUnknown(const Unknown& un, DiffOpType=_id);
OperatorOnUnknown(const Unknown&, const Function&, AlgebraicOperator, bool);
OperatorOnUnknown(const Unknown&, const Value&, AlgebraicOperator, bool);
OperatorOnUnknown(const OperatorOnUnknown &);
~OperatorOnUnknown();
OperatorOnUnknown& operator=(const OperatorOnUnknown &);
```

Note that the **DifferentialOperator** object are created on the fly when creating **OperatorOnUnknown** from **Unknown** and **DiffOpType**.

The class provides some accessors (read or write):

```
DiffOpType difOpType() const;
DifferentialOperator*& difOp();
ValueType& type();
Operand*& leftOperand();
Operand*& rightOperand();
Vector<complex_t>& coefs();
const Vector<complex_t>& coefs() const;
bool leftPriority() const;
bool& leftPriority();
```

and some utilities to update class attributes and print them:

```
void setReturnedType(const Unknown*, DiffOpType);
void updateReturnedType(AlgebraicOperator, ValueType, StrucType, bool);
void print(std::ostream&) const;
std::ostream& operator<<(std::ostream&, const OperatorOnUnknown &); //extern
```

The two first one manage the returned types and check consistency of operations and the two last functions output on stream the **OperatorOnUnknown** characteristics.

To deal with the general syntax of **OperatorOnUnknown**, a lot of external functions are required, in particular some overloaded operators. The first family of functions concerns the creation of simple

`OperatorOfUnknown` object from a differential operator and an unknown. Their names are the names of differential operator type (`DiffOpType` enumeration) without the leading underscore. We give only a few examples here (see `OperatorOnUnknown.hpp` header file for the complete list):

```
OperatorOnUnknown& id(const Unknown&);
OperatorOnUnknown& d0(const Unknown&);
...
OperatorOnUnknown& grad(const Unknown&);
OperatorOnUnknown& nabla(const Unknown&);
...
OperatorOnUnknown& nx(const Unknown&);
...
OperatorOnUnknown& gradG (const Unknown&,const complex_t&,const complex_t&,
                           const complex_t&,const complex_t&);
...
OperatorOnUnknown& mean(const Unknown&);
OperatorOnUnknown& jump(const Unknown&);
```

These functions constructs an `OperatorOnUnknown` in the heap memory and return a reference in order to be used in an chained expression. In this context, it may be occur a memory leak if nobody delete explicitly the `OperatorOnUnknown` object created on the fly.

To manage general syntax, the following c++ operators have been overloaded:

- * : usual consistent product between scalars, vectors and matrices
- | : inner product of vectors, for complex objects it is an hermitian product
- ^ : cross product of vectors
- % : contracted product (the second term is transposed)

These operators are enumerated in the `AlgebraicOperator` enumeration.

They are overloaded for `Unknown` or `OperatorOnUnknown` and unitaryVector (`UnitaryVector` enumerates the normal vector `_n` and the tangential vector `_t`). When differential operators involve operation with normal, you can use the previous function (for instance `ndot`) or use their mathematical syntax (for instance n^*u):

```
OperatorOnUnknown& operator*(UnitaryVector, const Unknown &); // n*u
OperatorOnUnknown& operator*(const Unknown &, UnitaryVector); // u*n
OperatorOnUnknown& operator|(UnitaryVector, const Unknown &); // n|u
OperatorOnUnknown& operator^(UnitaryVector, const Unknown &); // n^u
OperatorOnUnknown& operator|(const Unknown &, UnitaryVector); // u|n
OperatorOnUnknown& operator|(UnitaryVector, OperatorOnUnknown&); // n|grad(u)
OperatorOnUnknown& operator^(UnitaryVector, OperatorOnUnknown&); // n^(n^u) or
                           n^curl(u)
OperatorOnUnknown& operator*(UnitaryVector, OperatorOnUnknown&); // n*div
OperatorOnUnknown& operator|(OperatorOnUnknown&, UnitaryVector); // grad(u)|n
OperatorOnUnknown& operator*(OperatorOnUnknown&, UnitaryVector); // div*n
```

Note that all cases are not considered. It should be !?

To deal with left and right-hand side operations, these algebraic operators have been also overloaded for different types of objects (`Unknown`, `OperatorOnUnknown`, `Function`, `Value`, user c++ function or user value). The operator overloading with c++ function or user value avoids the user to explicitly encapsulate its own data (c++ function or scalar, vector, matrix) in `Function` object or `Value` object. There are shortcuts. In the following, only prototypes of `*` operator are presented.

7.3.1 Operations between Unknown and Function

```

OperatorOnUnknown& operator*(const Unknown &,const Function&); //  $u * F$ 
OperatorOnUnknown& operator*(const Function&,const Unknown &); //  $F * u$ 
template <typename T>
OperatorOnUnknown& operator*(const Unknown &,T( )(const Point&,Parameters&));
OperatorOnUnknown& operator*(T( )(const Point&,Parameters&),const Unknown &);
OperatorOnUnknown& operator*(const Unknown &,
                             T( )(const Vector<Point>&,Parameters&));
OperatorOnUnknown& operator*(T( )(const Vector<Point>&,Parameters&),
                             const Unknown &);
...

```

The two first one declarations concern Function object encapsulating c++ user functions whereas the templated forms concerns c++ user functions. As mentioned before, it avoids the user to encapsulate his c++ functions. Note that only c++ functions that can be encapsulated in Function object are working in templated forms. Other functions may be accepted in compilation process, but the result during execution may be hazardous!



Kernel type functions can not be use straightforward in operator. Contrary to the ordinary functions, they have to be encapsulated in **Function** objects!

7.3.2 Operations between Unknown and Value

```

OperatorOnUnknown& operator*(const Unknown&,const Value&); //  $u * V$ 
OperatorOnUnknown& operator*(const Value&,const Unknown&); //  $V * u$ 
template<typename T>
OperatorOnUnknown& operator*(const Unknown& ,const T& ) //  $u * T$ 
OperatorOnUnknown& operator*(const T& ,const Unknown&) //  $T * u$ 
...

```

The two first one declarations concern Value object encapsulating c++ user data and the templated forms concerns c++ user data. Only c++ user data that can be encapsulated in Value object are working in templated forms.

7.3.3 Operations between OperatorOnUnknown and Function

The principle is the same as in previous cases.

```

OperatorOnUnknown& operator*(OperatorOnUnknown&,const Function &); //  $op(u) * F$ 
OperatorOnUnknown& operator*(const Function &,OperatorOnUnknown&); //  $F * op(u)$ 
template<typename T>
OperatorOnUnknown& operator*(OperatorOnUnknown&, T( )(const Point&,Parameters&))
OperatorOnUnknown& operator*(T( )(const Point&,Parameters&), OperatorOnUnknown&)
OperatorOnUnknown& operator*(OperatorOnUnknown&,
                             T( )(const Vector<Point>&,Parameters&))
OperatorOnUnknown& operator*(T( )(const Vector<Point>&,Parameters&),
                             OperatorOnUnknown&)
...

```

7.3.4 Operations between OperatorOnUnknown and Value

```

OperatorOnUnknown& operator*(OperatorOnUnknown&,const Value &); //  $op(u) * V$ 
OperatorOnUnknown& operator*(const Value &,OperatorOnUnknown&); //  $V * op(u)$ 
template<typename T>
OperatorOnUnknown& operator*(OperatorOnUnknown&,const T&); //  $op(u) * T$ 
OperatorOnUnknown& operator*(const T&,OperatorOnUnknown&); //  $T * op(u)$ 
...

```

Most of the previous overloaded operators use the following update functions:

```
OperatorOnUnknown& updateRight (OperatorOnUnknown&, const
    Function&, AlgebraicOperator);
OperatorOnUnknown& updateLeft (OperatorOnUnknown&, const
    Function&, AlgebraicOperator);
OperatorOnUnknown& updateRight (OperatorOnUnknown&, const Value&, AlgebraicOperator);
OperatorOnUnknown& updateLeft (OperatorOnUnknown&, const Value&, AlgebraicOperator);
template<typename T>
    OperatorOnUnknown& updateRight (OperatorOnUnknown&, const T&, AlgebraicOperator);
    OperatorOnUnknown& updateLeft (OperatorOnUnknown&, const T&, AlgebraicOperator);
```

Note that it is possible to use the transformations *conj*, *tran* and *adj* on Unknown, Value or Function. These functions are defined in the files related to these classes. For simplicity, it is not possible to apply such transformations to an `OperatorOnUnknown` object or a part of the expression. For instance, the syntax *conj(grad(u))* is not supported.

Finally, we give some examples to show how this machinery works:

```
//define some C++ functions
real_t F(const Point &P, Parameters& pa = defaultParameters) {...}
Vector<real_t> vecF(const Point &P, Parameters& pa = defaultParameters) {...}
Matrix<real_t> matF(const Point &P, Parameters& pa = defaultParameters) {...}
//define some constant values
real_t pi=3.14159; complex_t i(0,1); vector<real_t> v(3,1); matrix<complex_t>
    A(3,3,i);
//assume u is an Unknown (a vector of dimension 3)
OperatorOnUnknown opu=u; //the simplest one
opu=grad(u); //involve only differential operator
opu=_n^curl(u); //equivalent to ncrosscurl(u)
opu=v^u; //cross product with vector
opu=vecF^u; //cross product with vector function
opu=A*grad(u); //left product with a matrix
opu=grad(u)*matF; //right product with a matrix function
opu=(matF*grad(u))%A; //product and contracted product
opu=curl(conj(u))|conj(v); //use conj transformation
```

Be careful with operator priority. The C++ priority rules are applied and not the mathematical ones. In doubt, use parenthesis. The code performs some constancy checking on operand and operator but only on structure (scalar, vector, matrix) not on the dimension (see *updateReturnedType* and *setReturnedType* functions).

library :	operator
header :	OperatorOnUnknown.hpp
implementation :	OperatorOnUnknown.cpp
unitary tests :	test_operator.cpp
header dependences :	config.h, utils.h

7.4 The OperatorOnUnknowns class

The `OperatorOnUnknowns` class deals with a pair of `OperatorOnUnknown` object related by an `AlgebraicOperator` (one of product, inner product, cross product or contracted product): It is useful to store syntax like *opu aop opv* occurring in bilinear forms.

```
class OperatorOnUnknowns
{
protected :
    OperatorOnUnknown opu_; //left operator on unknown
```

```

OperatorOnUnknown opv_; //right operator on unknown (test function)
AlgebraicOperator aop_; //algebraic operation (*,|,%,^)
}

```

The class provides basic constructor and accessors:

```

OperatorOnUnknowns() {};
OperatorOnUnknowns(const OperatorOnUnknown& operu, const OperatorOnUnknown& operv,
    AlgebraicOperator aop);
const OperatorOnUnknown& opu() const;
OperatorOnUnknown& opu();
OperatorOnUnknown& opv();
AlgebraicOperator algop();
ValueType valueType() const;

```

a function checking the consistency of expression:

```

bool checkConsistency();

```

The **checkConsistency** function checks that the given **OperatorOnUnknown** objects are consistent (in structure not in dimension) with the given algebraic operation. If not an error is handled.

Some print facilities are provided:

```

void print(std::ostream&) const;
friend std::ostream& operator<<(std::ostream&, const OperatorOnUnknowns&);

```

Besides, some extern functions acts like constructors:

```

OperatorOnUnknowns operator*(OperatorOnUnknown&,OperatorOnUnknown&); //opu * opv
OperatorOnUnknowns operator|(OperatorOnUnknown&,OperatorOnUnknown&); //opu | opv
OperatorOnUnknowns operator^(OperatorOnUnknown&,OperatorOnUnknown&); //opu ^ opv
OperatorOnUnknowns operator%(OperatorOnUnknown&,OperatorOnUnknown&); //opu % opv
OperatorOnUnknowns operator*(OperatorOnUnknown&,Unknown&); //opu * v
OperatorOnUnknowns operator|(OperatorOnUnknown&,Unknown&); //opu | v
OperatorOnUnknowns operator^(OperatorOnUnknown&,Unknown&); //opu ^ v
OperatorOnUnknowns operator%(OperatorOnUnknown&,Unknown&); //opu % v
OperatorOnUnknowns operator*(Unknown&,OperatorOnUnknown&); //u * opv
OperatorOnUnknowns operator|(Unknown&,OperatorOnUnknown&); //u | opv
OperatorOnUnknowns operator^(Unknown&,OperatorOnUnknown&); //u ^ opv
OperatorOnUnknowns operator%(Unknown&,OperatorOnUnknown&); //u % opv
OperatorOnUnknowns operator*(Unknown&,Unknown&); //u * v
OperatorOnUnknowns operator|(Unknown&,Unknown&); //u | v
OperatorOnUnknowns operator^(Unknown&,Unknown&); //u ^ v
OperatorOnUnknowns operator%(Unknown&,Unknown&); //u % v

```

library :	operator
header :	OperatorOnUnknowns.hpp
implementation :	OperatorOnUnknowns.cpp
unitary tests :	test_operator.cpp
header dependences :	OperatorOnUnknown.hpp, config.h, utils.h

7.5 The **LcOperatorOnUnknown** class

The **LcOperatorOnUnknown** class deals with linear combination of **OperatorOnUnknown** objects :

$$\sum_{i=1,n} c_i \text{ op}(u_i) | \text{ dom}_i$$

where c_i is a complex coefficient, $op(u_i)$ an operator on unknown u_i (`OperatorOnUnknown` object) and possibly a geometrical domain dom_i (`Domain` object). `LcOperatorOnUnknown` objects are useful to describe essential condition (see *essentialConditions* library):

$$\sum_{i=1,n} c_i op(u_i)|_{dom_i} = g.$$

The `LcOperatorOnUnknown` class inherits from `std::vector` class :

```
typedef std::pair<OperatorOnUnknown*, Complex> OpuValPair;

class LcOperatorOnUnknown : public std::vector<OpuValPair>
{
public :
    std::vector<Domain*> domains_;
}
```

The class provides some basic constructors and related stuff (copy, clear)

```
LcOperatorOnUnknown() {} ;
LcOperatorOnUnknown(const OperatorOnUnknown&, const Real& = 1.);
LcOperatorOnUnknown(const OperatorOnUnknown&, const Complex&);
LcOperatorOnUnknown(const OperatorOnUnknown&, Domain&, const Real& = 1.);
LcOperatorOnUnknown(const OperatorOnUnknown&, Domain&, const Complex&);
LcOperatorOnUnknown(const Unknown&, const Real& = 1.);
LcOperatorOnUnknown(const Unknown&, const Complex&);
LcOperatorOnUnknown(const Unknown&, Domain&, const Real& = 1.);
LcOperatorOnUnknown(const Unknown&, Domain&, const Complex&);
LcOperatorOnUnknown(const LcOperatorOnUnknown&);
~LcOperatorOnUnknown();
LcOperatorOnUnknown& operator =(const LcOperatorOnUnknown&);
void clear();
void copy(const LcOperatorOnUnknown& lc);
```

To insert new item in the list, there are insertion member functions and overloaded operators:

```
void insert(const OperatorOnUnknown&, GeomDomain* =0); // insert
    op(u)
void insert(const Real&, const OperatorOnUnknown&, GeomDomain* =0); // insert
    a*op(u)
void insert(const Complex&, const OperatorOnUnknown&, GeomDomain* =0); // insert
    a*op(u)
void insert(const Unknown&, GeomDomain* =0); // insert u
    on D
void insert(const Real&, const Unknown&, GeomDomain* =0); // insert
    a*u on D
void insert(const Complex&, const Unknown&, GeomDomain* =0); // insert
    a*u on D
LcOperatorOnUnknown& operator+=(const OperatorOnUnknown&); // lcop += opu
LcOperatorOnUnknown& operator+=(const Unknown&); // lcop += u
LcOperatorOnUnknown& operator+=(const LcOperatorOnUnknown&); // lcop += lcop
LcOperatorOnUnknown& operator-=(const OperatorOnUnknown&); // lcop -= opu
LcOperatorOnUnknown& operator-=(const Unknown&); // lcop -= u
LcOperatorOnUnknown& operator-=(const LcOperatorOnUnknown&); // lcop -= lcop
LcOperatorOnUnknown& operator*=(const Real&); // lcop *= r
LcOperatorOnUnknown& operator*=(const Complex&); // lcop *= c
LcOperatorOnUnknown& operator/=(const Real&); // lcop /= r
LcOperatorOnUnknown& operator/=(const Complex&); // lcop /= c
```

Extern operators may also be used :

```

LcOperatorOnUnknown operator+(const LcOperatorOnUnknown&);
LcOperatorOnUnknown operator-(const LcOperatorOnUnknown&);
LcOperatorOnUnknown operator+(const LcOperatorOnUnknown&, const OperatorOnUnknown&);
LcOperatorOnUnknown operator-(const LcOperatorOnUnknown&, const OperatorOnUnknown&);
LcOperatorOnUnknown operator+(const OperatorOnUnknown&, const LcOperatorOnUnknown&);
LcOperatorOnUnknown operator-(const OperatorOnUnknown&, const LcOperatorOnUnknown&);
LcOperatorOnUnknown operator+(const LcOperatorOnUnknown&, const Unknown&);
LcOperatorOnUnknown operator-(const LcOperatorOnUnknown&, const Unknown&);
LcOperatorOnUnknown operator+(const Unknown&, const LcOperatorOnUnknown&);
LcOperatorOnUnknown operator-(const Unknown&, const LcOperatorOnUnknown&);
LcOperatorOnUnknown operator+(const LcOperatorOnUnknown&, const
    LcOperatorOnUnknown&);
LcOperatorOnUnknown operator-(const LcOperatorOnUnknown&, const
    LcOperatorOnUnknown&);
LcOperatorOnUnknown operator*(const LcOperatorOnUnknown&, const Real&);
LcOperatorOnUnknown operator*(const LcOperatorOnUnknown&, const Complex&);
LcOperatorOnUnknown operator*(const Real&, const LcOperatorOnUnknown&);
LcOperatorOnUnknown operator*(const Complex&, const LcOperatorOnUnknown&);
LcOperatorOnUnknown operator/(const LcOperatorOnUnknown&, const Real&);
LcOperatorOnUnknown operator/(const LcOperatorOnUnknown&, const Complex&);
LcOperatorOnUnknown operator+(const OperatorOnUnknown&, const OperatorOnUnknown&);
LcOperatorOnUnknown operator-(const OperatorOnUnknown&, const OperatorOnUnknown&);
LcOperatorOnUnknown operator+(const OperatorOnUnknown&, const Unknown&);
LcOperatorOnUnknown operator-(const OperatorOnUnknown&, const Unknown&);
LcOperatorOnUnknown operator+(const Unknown&, const OperatorOnUnknown&);
LcOperatorOnUnknown operator-(const Unknown&, const OperatorOnUnknown&);
LcOperatorOnUnknown operator+(const Unknown&, const Unknown&);
LcOperatorOnUnknown operator-(const Unknown&, const Unknown&);

```

`GeomDomain` may be affected using `SetDomain` or `|` operator :

```

void setDomain(GeomDomain&);
LcOperatorOnUnknown& operator|(GeomDomain&);
LcOperatorOnUnknown operator|(const Unknown&, GeomDomain&);
LcOperatorOnUnknown operator|(const OperatorOnUnknown&, GeomDomain&);

```

Note that when setting the domain, all previous domain definitions are overwritten.

It provides the following accessors and property accessors:

```

bool isSingleUnknown() const;
const Unknown* unknown() const;
std::set<const Unknown*> unknowns() const;
bool withDomains() const;
bool isSingleDomain() const;
Domain* domain() const;
std::set<Domain*> domains() const;
Complex coefficient() const;
std::vector<Complex> coefficients() const;
std::set<DiffOpType> diffOperators() const;
void print(std::ostream&, bool withdomain=true) const;
friend std::ostream& operator<<(std::ostream&, const LcOperatorOnUnknown&);

```

To create essential condition from `LcOperatorOnUnknown`, `operator =` is overloaded as following:

```

EssentialCondition operator=(const Real &);
EssentialCondition operator=(const Complex &);
EssentialCondition operator=(const Function &);

```

These functions are implemented in *EssentialCondition.cpp*!

library :	operator
header :	LcOperatorOnUnknowns.hpp
implementation :	LcOperatorOnUnknowns.cpp
unitary tests :	test_EssentialCondition.cpp
header dependences :	OperatorOnUnknown.hpp, config.h, utils.h

7.6 The OperatorOnKernel class

Kernel objects describes function of two points, say $K(x, y)$. The `OperatorOnKernel` class is intended to store the differential operators acting on Kernel, one for x derivatives and one for y derivatives :

```
class OperatorOnKernel
{
protected :
    const Kernel* ker_p;           //Kernel involved in operator
    DifferentialOperator* xdifOp_p; //x differential operator involved in operator
    DifferentialOperator* ydifOp_p; //y differential operator involved in operator
    ValueType type_;               //type of returned value ( _real, _complex)
    StructType struct_;            //structure of returned value (_scalar,
    _vector, _matrix)
}
```

A null pointer `ker_p` means the kernel function $K(x, y) = 1$ and by default `xdifOp_p` and `ydifOp_p` point to identity differential operator.

`OperatorOnKernel` object are constructed either by constructor members or by external functions:

```
OperatorOnKernel();
OperatorOnKernel(const Kernel*, DiffOpType = _id, DiffOpType = _id, ValueType
    vt=_real, StructType st=_scalar);
OperatorOnKernel(const Kernel&, DiffOpType = _id, DiffOpType = _id);
OperatorOnKernel(const OperatorOnKernel&);
OperatorOnKernel& operator=(const OperatorOnKernel&);

OperatorOnKernel& id(const Kernel&);           //id(k)
OperatorOnKernel& grad_x(const Kernel&);       //grad_x(k)
OperatorOnKernel& grad_y(const Kernel&);       //grad_y(k)
...
OperatorOnKernel& id(OperatorOnKernel&);       //id(opk)
OperatorOnKernel& grad_x(OperatorOnKernel&);   //grad_x(opk)
OperatorOnKernel& grad_y(OperatorOnKernel&);   //grad_y(opk)
...
OperatorOnKernel& operator*(UnitaryVector, const Kernel&); //n*ker
OperatorOnKernel& operator*(const Kernel&, UnitaryVector); //ker*n
OperatorOnKernel& operator|(UnitaryVector, const Kernel&); //n|ker
OperatorOnKernel& operator|(const Kernel&, UnitaryVector); //ker|n
OperatorOnKernel& operator^(UnitaryVector, const Kernel&); //n^ker
OperatorOnKernel& operator*(UnitaryVector, OperatorOnKernel&); //n*opker
OperatorOnKernel& operator*(OperatorOnKernel&, UnitaryVector); //opker*n
OperatorOnKernel& operator|(UnitaryVector, OperatorOnKernel&); //n|opker
OperatorOnKernel& operator|(OperatorOnKernel&, UnitaryVector); //opker|n
OperatorOnKernel& operator^(UnitaryVector, OperatorOnKernel&); //n^opker
```

Note that operators return reference to object dynamically allocated.

The class provides some member accessors and property accessors

```
const Kernel* kernel() const;
DiffOpType xdifOpType() const;
DiffOpType ydifOpType() const;
```

```

DifferentialOperator*& xdifOp_();
DifferentialOperator& xdifOp() const;
DifferentialOperator*& ydifOp_();
DifferentialOperator& ydifOp() const;
ValueType& valueType();
ValueType valueType() const;
StrucType& strucType();
StrucType strucType() const;
Dimen xdiffOrder() const;
Dimen ydiffOrder() const;
bool voidKernel() const;

```

and print facilities

```

void print(std::ostream&) const;
friend std::ostream& operator<<(std::ostream& os, const OperatorOnKernel& opk);

```

Finally the class interfaces the computation of `OperatorOnKernel` at a couple of points or a list of couples of points using template functions:

```

template <typename T>
T& eval(const Point&, const Point&, T&) const;
template <typename T>
std::vector<T>& eval(const std::vector<Point>&, const std::vector<Point>&,
std::vector<T>&) const;

```

library :	operator
header :	OperatorOnKernel.hpp
implementation :	OperatorOnKernel.cpp
unitary tests :	test_operator.cpp
header dependencies :	DifferentialOperator.hpp, Operand.hpp, config.h, utils.h

7.7 The KernelOperatorOnUnknowns class

The `KernelOperatorOnUnknowns` class deals with expressions involving two operators on unknown linked with an `OperatorOnKernel`, that is

$$op_u \ aop_l \ op_k \ aop_r \ op_v$$

Such expression appears in integral equation or integral representation, for instance to deal with:

$$\int_{\Gamma} \int_{\Gamma} u(x) * G(x, y) * v(y) \, dx \, dy.$$

```

class KernelOperatorOnUnknowns
{
protected :
    OperatorOnUnknown opu_;
    OperatorOnUnknown opv_;
    AlgebraicOperator aopu_;
    AlgebraicOperator aopv_;
    OperatorOnKernel opker_;
}

```

The class provides one basic constructor and a lot of extern functions building `KernelOperatorOnUnknowns` object:


```

KernelOperatorOnUnknowns(const OperatorOnUnknown&, AlgebraicOperator, const
    OperatorOnKernel&, AlgebraicOperator, const OperatorOnUnknown&);

KernelOperatorOnUnknowns operator*(const OperatorOnUnknown&, const
    OperatorOnKernel&); //opu * opker
KernelOperatorOnUnknowns operator*(const OperatorOnKernel&, const
    OperatorOnUnknown&); //opker * opv
KernelOperatorOnUnknowns operator*(const OperatorOnUnknown&, const Kernel&);
    //opu * ker
KernelOperatorOnUnknowns operator*(const Kernel&, const OperatorOnUnknown&);
    //ker * opv
KernelOperatorOnUnknowns operator*(const Unknown&, const Kernel&);
    //u * ker
KernelOperatorOnUnknowns operator*(const Kernel&, const Unknown&);
    //ker * v
KernelOperatorOnUnknowns operator*(const OperatorOnUnknown&, const
    KernelOperatorOnUnknowns&); //opu * opker
KernelOperatorOnUnknowns operator*(const KernelOperatorOnUnknowns&, const
    OperatorOnUnknown&); //opker * opv
KernelOperatorOnUnknowns operator*(const Unknown&, const
    KernelOperatorOnUnknowns&); //u * opker
KernelOperatorOnUnknowns operator*(const KernelOperatorOnUnknowns&, const
    Unknown&); //opker * v
//same for operation inner product |, cross product ^ and contracted product %

template <typename T>
KernelOperatorOnUnknowns operator*(const Unknown& un, T(fun)(const Point&, const
    Point&, Parameters&)); //u * fun
template <typename T>
KernelOperatorOnUnknowns operator*(T(fun)(const Point&, const Point&, Parameters&),
    const Unknown& un); //fun * u
template <typename T>
KernelOperatorOnUnknowns operator*(T(fun)(const Vector<Point>&, const
    Vector<Point>&, Parameters&), const Unknown& un); //fun * u
template <typename T>
KernelOperatorOnUnknowns operator*(const Unknown& un, T(fun)(const
    Vector<Point>&, const Vector<Point>&, Parameters&)) //! u * func
//same for operation inner product |, cross product ^ and contracted product %

```

It provides also accessors and print facilities

```

const OperatorOnUnknown& opu() const;
const OperatorOnUnknown& opv() const;
OperatorOnUnknown& opu();
OperatorOnUnknown& opv();
AlgebraicOperator algopu() const;
AlgebraicOperator algopv() const;
AlgebraicOperator& algopu();
AlgebraicOperator& algopv();
const OperatorOnKernel& opker() const;
bool isKernelType() const;
ValueType valueType() const;

void print(std::ostream&) const;
friend std::ostream& operator<<(std::ostream&, const KernelOperatorOnUnknowns&);

```

library :	operator
header :	KernelOperatorOnUnknowns.hpp
implementation :	KernelOperatorOnUnknowns.cpp
unitary tests :	test_operator.cpp
header dependencies :	OperatorOnUnknown.hpp, OperatorOnKernel.hpp, config.h, utils.h

7.8 The TensorKernel class

The **TensorKernel** class inherited from **Kernel** handles some particular kernels, say tensor kernels:

$$K(x, y) = \sum_{m,n} \phi_n(y) \mathbb{A}_{mn} \psi_m(x).$$

This kernel is involved in DirichletToNeumann map.

 In case of complex basis functions, the **TensorKernel** is often involved in integral of the following form:

$$\sum_{mn} \mathbb{A}_{mn} \int_{\Sigma} w_j(y) * \bar{\phi}_n(y) \int_{\Gamma} \tau_i(x) * \psi_m(x)$$

where appears a conjugate operation on ϕ_n functions.

By default, computation functions do not apply the conjugate operation. So if you have to applied to, there is a flag to enforce the conjugate operation.

Because, **Kernel** class uses **SpectralBasis** class to handles functions ϕ_n and ψ_n , it can not be located in *utils* library. It is the reason why it is located in *term* library.

```
class TensorKernel : public Kernel
{
protected :
    const SpectralBasis* phi_p , *psi_p; //pointers to SpectralBasis object
    VectorEntry* matrix_p;               //matrix stored as a vector
public :
    bool isDiag;                          //!< true if matrix is diagonal matrix
    Function xmap, ymap;                  //!< maps applied to point x and y
    bool isConjugate;                     //!< conjugate phi_p in IESP computation (default=false)
```

SpectralBasis class manages either a family of explicit functions given by a **Function** object or a family of interpolated functions given as a vector of **TermVector** (see **SpectralBasisFun** class and **SpectralBasisInt** defined in *space* library). Stored as a **MatrixEntry** object, the matrix \mathbb{A} may be of real or complex type.

The **TensorKernel** class provides some explicit constructors from **SpectralBasis** and `vector<T>` (matrix \mathbb{A}) and some implicit constructors from **Unknown** associated to a spectral space equipped with a spectral basis, and `vector<T>`.

```
TensorKernel();
template<typename T>
TensorKernel(const SpectralBasis&, const std::vector<T>&, bool =false);
template<typename T>
TensorKernel(const SpectralBasis&, const std::vector<T>&, const SpectralBasis&);
template<typename T>
TensorKernel(const Unknown&, const std::vector<T>&, bool =false);
template<typename T>
TensorKernel(const Unknown&, const std::vector<T>&, const Unknown&);
template<typename T>
TensorKernel(const std::vector<TermVector>&, const std::vector<T>&, bool =false);
template<typename T>
```

```

TensorKernel(const std::vector<TermVector>&, const std::vector<T>&, const
std::vector<TermVector>&);
~TensorKernel();

```

Besides it provides some accessors and properties

```

const VectorEntry& vectorEntry() const {return *matrix_p;}
const SpectralBasis* phip() const {return phi_p;}
const SpectralBasis* psip() const {return psi_p;}
const SpectralBasis& phi() const {return *phi_p;}
const SpectralBasis& psi() const {return *psi_p;}
const TensorKernel* tensorKernel() const;
TensorKernel* tensorKernel();

Dimen dimOfPhi() const;
Dimen dimOfPsi() const;
Number nbOfPhi() const;
Number nbOfPsi() const;
bool phiIsAnalytic() const;
bool psiIsAnalytic() const;
virtual KernelType type() const;
virtual bool isSymbolic() const;
virtual ValueType valueType() const;
virtual StrucType structType() const;
bool sameBasis() const;

```

It proposes two template functions that compute the product of \mathbb{A}_m or \mathbb{A}_{mn} by something of type T, expecting a result of same type!

```

template<typename T>
T kernelProduct(Number, const T&) const;
template<typename T>
T kernelProduct(Number, Number, const T&) const;

```

library :	operator
header :	TensorKernel.hpp
implementation :	TensorKernel.cpp
unitary tests :	test_EssentialCondition.cpp
header dependences :	space.h, config.h, utils.h

This library provides the main `LinearForm` and `BilinearForm` classes which are the fundamental user classes to describe variational formulation. Linear and bilinear forms involved in variational formulation may have multiple expressions. For the moment, the library deals with:

- Single integral form involved in standard variational formulation, say

$$\int_D \mathcal{L}(u) \quad \text{or} \quad \int_D \mathcal{L}_1(u) O \mathcal{L}_2(v)$$

where D is a geometrical domain, u, v are unknowns or test functions, \mathcal{L} are linear operators (see `OperatorOnUnknown` class) and O is an algebraic operation. Canonical examples are:

$$\int_{\Omega} f u, \quad \text{or} \quad \int_{\Omega} A \nabla(u) \cdot \nabla(v)$$

and for Galerkin Discontinuous approach (S_{Ω} the internal sides of elements of Ω) :

$$\int_{S_{\Omega}} [u] \{v\}.$$

- Double integral form involved in integral representation or integral equation variational formulation, say

$$\int_{D_1} \int_{D_2} K(x_1, x_2) O \mathcal{L}(u)(x_1) \quad \text{or} \quad \int_{D_1} \int_{D_2} K(x_1, x_2) O \mathcal{L}(u)(x_1) O \mathcal{L}(v)(x_2)$$

where D_1 and D_2 are geometrical domains, u, v are unknowns or test functions, \mathcal{L} are linear operators and $K(x_1, x_2)$ is a kernel function.

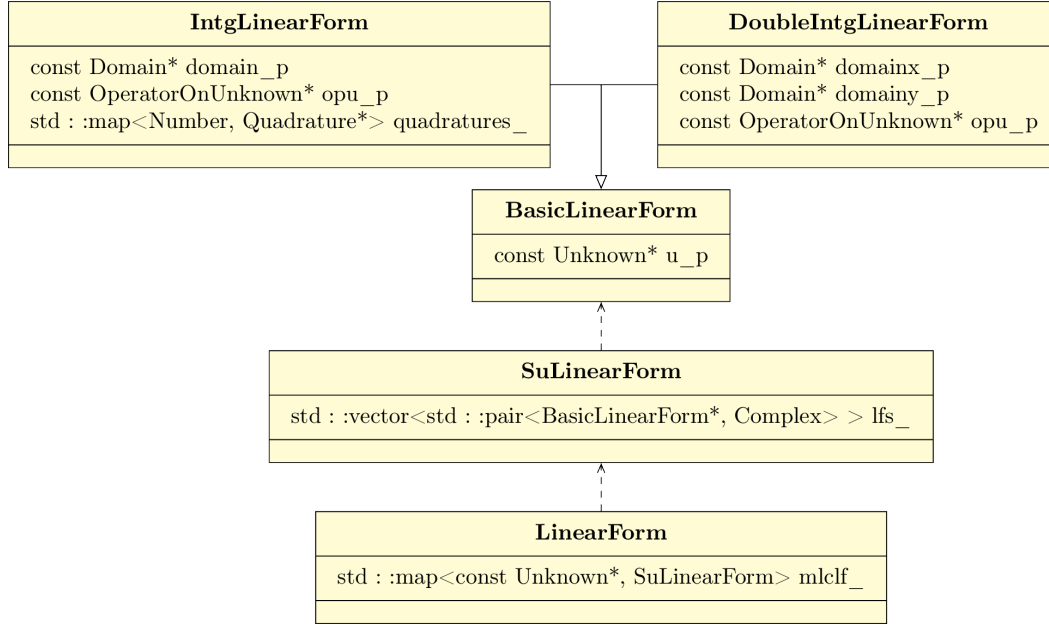
Besides, it should be friendly to deal with multiple unknowns formulation. For instance the formulation of Stokes problem involves the pressure and the velocity as unknowns. Thus, linear (resp. bilinear) forms taken into account are lists of linear combination of basic linear (resp. bilinear) forms. A basic linear (resp. bilinear) form is a simple or a double integral of an operator acting on unknown (resp. a "product" of operators acting on unknown), see the `OperatorOnUnknown` class. Only basic linear (resp. bilinear) forms having the same unknown (resp. couple of unknowns) may be combined in a linear combination. On the other hand, lists of linear combination of basic linear (resp. bilinear) forms may be algebraically combined, combination being done on each unknown (resp. couple of unknowns) separately. A multiple unknowns linear form may be seen as a vector of linear form indexed by unknown and a multiple unknowns bilinear form may be seen as a matrix of linear form indexed by a couple of unknowns. In this way, a single unknown linear form is a vector of size one and a "single" unknown bilinear form is a matrix one by one.

To intend to this organisation, the following classes are provided:

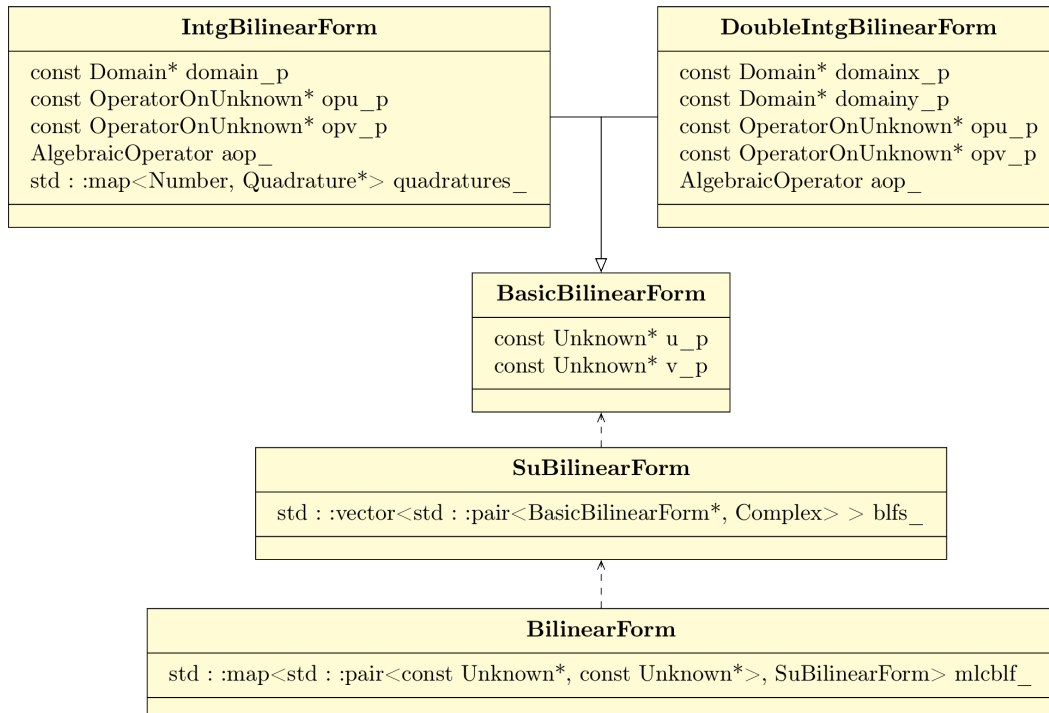
- `BasicLinearForm` (resp. `BasicBilinearForm`) class is the abstract base class of basic linear (resp. bilinear) form,
- `IntgLinearForm` (resp. `IntgBilinearForm`) class inherits from `BasicLinearForm` (resp. `BasicBilinearForm`) and deals with forms defined with an integral on a geometrical domain,

- **DoubleIntgLinearForm** (resp. **DoubleIntgBilinearForm**) class inherits from **BasicLinearForm** (resp. **BasicBilinearForm**) and deals with forms defined with an integral on a product of geometrical domains (double integral). Such integrals are involved in integral equation formulation,
- **LcLinearForm** (resp. **LcBilinearForm**) class stores a linear combination of **BasicLinearForm** (resp. **BasicBilinearForm**) objects having same unknown (resp. couple of unknowns),
- **LinearForm** (resp. **BilinearForm**) class stores a list of **LcLinearForm** (resp. **LcBilinearForm**) objects using map indexed by unknown (resp. couple of unknowns).

The organisation of linear forms looks like:



and the organisation of bilinear forms is similar:



8.1 Linear forms

Linear forms are managed using

- `BasicLinearForm` abstract class of basic linear form,
- `IntgLinearForm` class inheriting from `BasicLinearForm` and dealing with integral form,
- `DoubleIntgLinearForm` class inheriting from `BasicLinearForm` and dealing with double integral forms,
- `SuLinearForm` class storing linear combination of `BasicLinearForm` objects having the same unknown,
- `LinearForm` class storing a list of `SuLinearForm` objects using a map indexed by unknown.

8.1.1 The BasicLinearForm class

The `BasicLinearForm` class is an abstract class having only an unknown as attribute and its associated accessor:

```
class BasicLinearForm
{protected:
    const Unknown* u_p;           //pointer to unknown
public:
    const Unknown& up() const     //return the unknown
```

The child classes inheriting from `BasicLinearForm` have to provide the following member functions:

```
virtual ~BasicLinearForm() {};           //virtual destructor
virtual BasicLinearForm* clone() const =0; //clone of the linear form
virtual LinearFormType type() const =0;   //return the type of the linear form
virtual ValueType valueType() const=0;    //return the value type of the linear
form
virtual void print(std::ostream&) const =0; //print utility
```

The virtual `clone` function is used to construct a copy of the child objects from parent.

8.1.2 The IntgLinearForm class

The `IntgLinearForm` class handles linear forms defined by a (single) integral over a geometric domain:

$$\int_D \mathcal{L}(u)$$

where D is a geometric domain (`Domain` class) and \mathcal{L} a linear operator (`OperatorOnUnknown` class) acting on unknown u (`Unknownn` class).

This class is defined as follows:

```
class IntgLinearForm : public BasicLinearForm
{protected:
    const GeomDomain* domain_p; //geometric domain of the integral (pointer)
    const OperatorOnUnknown* opu_p; //operator on unknown (pointer)
```

It has some public constructors and some public accessors:

```

IntgLinearForm(const GeomDomain&,const OperatorOnUnknown&);
const OperatorOnUnknown* opu() const;
const GeomDomain* domain() const;
virtual BasicLinearForm* clone() const;
virtual LinearFormType type() const;
virtual ValueType valueType() const;
virtual void print(std::ostream&) const;

```

8.1.3 The DoubleIntgLinearForm class

The `DoubleIntgLinearForm` class handles linear forms defined by a double integral over a product of geometric domains:

$$\int_{D_x} \int_{D_y} \mathcal{L}(u)$$

where D_x , D_y are geometric domain (`GeomDomain` class) and \mathcal{L} a linear operator (`OperatorOnUnknown` class) acting on unknown u (`Unknown` class).

Except, there are two geometric domains, this class is very similar to the `IntgLinearForm` class:

```

class DoubleIntgLinearForm : public BasicLinearForm
{protected :
    const OperatorOnUnknown* opu_p; // operator on unknown (pointer)
    const GeomDomain* domainx_p; // first geometric domain (say x variable)
    const GeomDomain* domainy_p; // second geometric domain (say y variable)

```

It provides some public constructors and some public accessors:

```

DoubleIntgLinearForm(const GeomDomain&, const GeomDomain&,const OperatorOnUnknown&);
const OperatorOnUnknown* opu() const;
const GeomDomain* domainx() const;
const GeomDomain* domainy() const;
virtual BasicLinearForm* clone() const;
virtual LinearFormType type() const;
virtual ValueType valueType() const;
virtual void print(std::ostream&)const;

```

8.1.4 The SuLinearForm class

`BasicLinearForm` objects may be linearly combined to produce a linear combination of `BasicLinearForm` objects stored as a list of pair of `BasicLinearForm` object and a complex scalar in the `SuLinearForm` class.

```

typedef std::pair<BasicLinearForm*,complex_t> lfPair;
class SuLinearForm
{protected:
    std::vector<lfPair> lfs_; //list of pairs of basic linear form and coefficient
    ...

```

All the `BasicLinearForm` objects must have the same unknown ! It is the reason why this class has no pointer to an unknown; it refers to the first basic linear form to get its unknown.

Because `BasicLinearForm` objects are copied for safety reason, this class provides default and basic constructors but also a copy constructor, a destructor and the overload assignment operator:

```

SuLinearForm(){};
SuLinearForm(const SuLinearForm&);
~SuLinearForm();

```

```
SuLinearForm& operator=(const SuLinearForm&);
```

It provides few accessors (some being const and no const):

```
number_t size() const;
std::vector<lfPair>& lfs();
const std::vector<lfPair>& lfs() const;
lfPair& operator()(number_t n);
const lfPair& operator()(number_t n) const;
const Unknown* unknown() const;
const Space* space() const;
LinearFormType type() const;
ValueType valueType() const;
```

It is possible to perform linear combination of linear combinations using the following overloaded operators:

```
SuLinearForm& SuLinearForm::operator +=(const SuLinearForm&);
SuLinearForm& SuLinearForm::operator -=(const SuLinearForm&);
SuLinearForm& SuLinearForm::operator *=(const complex_t&);
SuLinearForm& SuLinearForm::operator /=(const complex_t&);
SuLinearForm operator-(const SuLinearForm&);
SuLinearForm operator+(const SuLinearForm&,const SuLinearForm&);
SuLinearForm operator-(const SuLinearForm&,const SuLinearForm&);
SuLinearForm operator*(const complex_t&,const SuLinearForm&);
SuLinearForm operator*(const SuLinearForm&,const complex_t&);
SuLinearForm operator/(const SuLinearForm&,const complex_t&);
bool SuLinearForm::checkConsistency(const SuLinearForm&) const;
```

The member function `checkConsistency` performs a test to insure that unknown is always the same.

Finally, there are some print facilities:

```
void SuLinearForm::print(std::ostream&)const;
std::ostream& operator<<(std::ostream&,const SuLinearForm&);
```

8.1.5 The LinearForm class

The `LinearForm` class is the end user class dealing with general linear form, either a single unknown linear form (a `SuLinearForm` object) or a multiple unknown linear form (a list of `SuLinearForm` objects). In this class, a single unknown linear form is a multiple unknown linear form with one unknown! The list of `SuLinearForm` objects is stored in a map of `SuLinearForm`, indexed by the pointer to `SuLinearForm` unknown:

```
class LinearForm
{protected :
    std::map<const Unknown *,SuLinearForm> mlclf_; //list of linear combinations of
        basic forms
    ...
```

To manage the map, the following aliases are defined:

```
typedef std::map<const Unknown *,SuLinearForm>::iterator it_mulc;
typedef std::map<const Unknown *,SuLinearForm>::const_iterator cit_mulc;
```



When the `SuLinearForm` unknown is an unknown component, the `SuLinearForm` object is attached to its parent item! In other words, a component unknown is not indexed in the map.

This class provides only one constructor from a linear combination of forms:


```
LinearForm(const SuLinearForm&);
```

and proposes some accessors and facilities:

```
bool isEmpty() const;
bool singleUnknown() const;
SuLinearForm& operator [] (const Unknown*); //protected
const SuLinearForm& operator [] (const Unknown*) const; //protected
const SuLinearForm& first() const;
LinearForm operator () (const Unknown&) const;
BasicLinearForm& operator () (const Unknown&, number_t);
const BasicLinearForm& operator () (const Unknown&, number_t) const;
```

Besides, there are two fundamental external end user's function (`intg`) which constructs `LinearForm` object:

```
LinearForm intg(const GeomDomain&, const OperatorOnUnknown&);
LinearForm intg(const GeomDomain&, const Unknown&);
LinearForm intg(const GeomDomain&, const GeomDomain&, const OperatorOnUnknown&);
LinearForm intg(const GeomDomain&, const GeomDomain&, const Unknown&);
```

In order to construct any linear forms, the algebraic operators (`+=` `-=` `*=` `/=` `+` `-` `*` `/`) are overloaded for different objects:

```
LinearForm& LinearForm::operator +=(const LinearForm&);
LinearForm& LinearForm::operator -=(const LinearForm&);
LinearForm& LinearForm::operator *=(const complex_t&);
LinearForm& LinearForm::operator /=(const complex_t&);
LinearForm operator+(const LinearForm&, const LinearForm&);
LinearForm operator-(const LinearForm&, const LinearForm&);
LinearForm operator*(const complex_t&, const LinearForm&);
LinearForm operator*(const LinearForm&, const complex_t&);
LinearForm operator/(const LinearForm&, const complex_t&);
```

Finally, the class provides usual print facilities:

```
void LinearForm::print(std::ostream&) const;
std::ostream& operator<<(std::ostream&, const LinearForm&);
```

Example

To end we show some characteristic examples. In these examples, `u` is a scalar unknown, `w` is a vector unknown, `f` either a scalar or a scalar function, `h` either a vector or a vector function, `g` a scalar or a scalar kernel and `Omega`, `Gamma` `Sigma` some geometric domains.

```
LinearForm l1=intg(Omega, f*u);
l1+=intg(Omega, h|grad(u));
l1=intg(Omega, f*u)+intg(Omega, h|grad(u)); //equivalent
l1=l1-2*intg(Gamma, Sigma, g*u);
LinearForm l2=intg(Omega, f*div(w))+intg(Gamma, h|(w^n));
LinearForm l3=l1+l2; //multiple unknowns
l3=intg(Omega, f*u)+intg(Omega, f*div(w))+intg(Omega, h|grad(u))
+2*intg(Gamma, Sigma, g*u)+intg(Gamma, h|(w^n)); //same result
```

library :	form
header :	LinearForm.hpp
implementation :	LinearForm.cpp
unitary tests :	test_form.cpp
header dependencies :	config.h, utils.h

8.2 Bilinear forms

Bilinear forms are managed using

- **BasicBilinearForm** abstract class of basic linear form,
- **IntgBilinearForm** class inheriting from **BasicBilinearForm** and dealing with integral form,
- **DoubleIntgBilinearForm** class inheriting from **BasicBilinearForm** and dealing with double integral forms,
- **SuBilinearForm** class storing linear combination of **BasicBilinearForm** objects having the same unknown,
- **BilinearForm** class storing a list of **SuBilinearForm** objects using a map indexed by unknowns.

8.2.1 The BasicBilinearForm class

The **BasicBilinearForm** class is an abstract class having only the unknowns as attribute and their associated accessors:

```
class BasicBilinearForm
{protected:
    const Unknown* u_p;           //pointer to unknown u
    const Unknown* v_p;           //pointer to unknown v (test function)
public:
    const Unknown& up() const;
    const Unknown& vp() const;
    ...
}
```

The 'u' letter refers always to the left unknown of the bilinear form and the 'v' letter to the right unknown (test function) of the bilinear form. In matrix representation 'u' stands for columns and 'v' for rows.

The child classes inheriting from **BasicBilinearForm** have to provide the following member functions:

```
virtual ~BasicBilinearForm() {};
virtual BasicBilinearForm* clone() const =0;
virtual LinearFormType type() const =0;
virtual ValueType valueType() const=0;
virtual void print(std::ostream&) const =0;
```

The virtual `clone()` function is used to construct a copy of the child objects from parent.

8.2.2 The IntgBilinearForm class

The **IntgBilinearForm** class handles linear forms defined by a (single) integral over a geometric domain:

$$\int_D \mathcal{L}_1(u) op \mathcal{L}_2(v)$$

where D is a geometric domain (`GeomDomain` class) and $\mathcal{L}_1, \mathcal{L}_2$ are linear operators (`OperatorOnUnknown` class) acting on unknowns u and v (`Unknown` class) and op is an algebraic product operator. In case of mesh domain, it handles also the quadrature rules, one rule by shape of elements.

This class is defined as follows:

```
class IntgBilinearForm : public BasicBilinearForm
{protected :
    const GeomDomain* domain_p; //geometric domain of the integral (pointer)
    const OperatorOnUnknown* opu_p; //operator on unknown u (pointer)
    const OperatorOnUnknown* opv_p; //operator on unknown v (pointer)
    AlgebraicOperator aop_; //algebraic operation between operators
    map<Number, Quadrature*> quadratures_; //!< pointers to quadrature rules
    ...
}
```

Note that the `quadratures_` map may be empty. This, is the case when the domain is not of type mesh domain. Generally, the elements of a mesh domain have the same shape but it may not. It is the reason why a map is used to store the quadratures .

It has one basic constructor, some public accessors and utilities:

```
IntgBilinearForm(const GeomDomain&, const OperatorOnUnknown&, AlgebraicOperator,
                const OperatorOnUnknown&, QuadRule = _defaultRule, Number =0)
const OperatorOnUnknown* opu() const;
const OperatorOnUnknown* opv() const;
const GeomDomain* domain() const;
virtual BasicBilinearForm* clone() const;
virtual LinearFormType type() const
virtual ValueType valueType() const
void setQuadrature(QuadRule, Number);
virtual void print(std::ostream&) const;
```

The `setQuadrature` function set the quadrature pointers from a `QuadRule` (enumeration of type of quadrature rule) and a quadrature number (degree or order of quadrature rule). For a full description of quadrature rules see section 5.6.2. When `QuadRule` is set to `_defaultRule` in constructor, the 'best' rule is chosen regarding the shape element and the degree of integrand (see `Quadrature::bestQuadRule` function).

8.2.3 The DoubleIntgBilinearForm class

The `DoubleIntgBilinearForm` class handles linear forms defined by a double integral over a product of geometric domains:

$$\int_{D_x} \int_{D_y} \mathcal{L}(u)$$

where D_x, D_y are geometric domain (`GeomDomain` class) and \mathcal{L} a linear operator (`OperatorOnUnknown` class) acting on unknown u (`Unknown` class).

Except, there are two geometric domains, this class is very similar to the `IntgBilinearForm` class:

```
class DoubleIntgBilinearForm : public BasicBilinearForm
{protected :
    const GeomDomain* domainx_p; //first geometric domain (say x variable)
    const GeomDomain* domainy_p; //second geometric domain (say y variable)
    const OperatorOnUnknown* opu_p; //operator on unknown u (pointer)
    const OperatorOnUnknown* opv_p; //operator on unknown v (pointer)
    ...
}
```

It provides a basic constructor and some public accessors:

```

DoubleIntgBilinearForm(const GeomDomain&, const GeomDomain&, const OperatorOnUnknown&,
                      AlgebraicOperator, const OperatorOnUnknown&);
const OperatorOnUnknown* opu() const;
const OperatorOnUnknown* opv() const;
const GeomDomain* domainx() const;
const GeomDomain* domainy() const;
virtual BasicBilinearForm* clone() const;
virtual LinearFormType type() const;
virtual ValueType valueType() const;
virtual void print(std::ostream&)const;

```



Be cautious in the definition of double integral. The syntax

```
BilinearForm a=intg(Sigma, Gamma, u*G*v);
```

has to be interpreted as

$$\int_{\Sigma_x} \int_{\Gamma_y} u(y) G(x, y) v(x) dy dx$$

where u (the right unknown) stands for the unknown, v (the left unknown) being the test function. In matrix representation, it means that u stands for columns and v for rows.

8.2.4 The SuBilinearForm class

BasicBilinearForm objects may be linearly combined to produce a linear combination of **BasicBilinearForm** objects stored as a list of pair of **BasicBilinearForm** object and a complex scalar in the **SuBilinearForm** class.

```

typedef std::pair<BasicBilinearForm*, complex_t> blfPair;
class SuBilinearForm
{protected:
    std::vector<blfPair> blfs_; //list of pairs of basic bilinear form and
    coefficient
    ...

```

All the **BasicBilinearForm** objects must have the same pair of unknowns ! It is the reason why this class has no pointer to unknowns; it refers to the first basic bilinear form to get its unknowns.

Because **BasicBilinearForm** objects are copied for safety reason, this class provides default and basic constructors but also a copy constructor, a destructor and the overload assignment operator:

```

SuBilinearForm(){};
SuBilinearForm(const SuBilinearForm&);
~SuBilinearForm();
SuBilinearForm& operator=(const SuBilinearForm&);

```

It provides few accessors (some being const and no const):

```

number_t size() const;
std::vector<blfPair>& blfs();
const std::vector<blfPair>& blfs() const;
blfPair& operator()(number_t n);
const blfPair& operator()(number_t n) const;
const Unknown* up() const;
const Unknown* vp() const;
const Space* uSpace() const;
const Space* vSpace() const;
LinearFormType type() const;
ValueType valueType() const;

```

It is possible to perform linear combination of linear combinations using the following overloaded operators:

```
SuBilinearForm& SuBilinearForm::operator +=(const SuBilinearForm&);
SuBilinearForm& SuBilinearForm::operator -=(const SuBilinearForm&);
SuBilinearForm& SuBilinearForm::operator *=(const complex_t&);
SuBilinearForm& SuBilinearForm::operator /=(const complex_t&);
SuBilinearForm operator-(const SuBilinearForm&);
SuBilinearForm operator+(const SuBilinearForm&,const SuBilinearForm&);
SuBilinearForm operator-(const SuBilinearForm&,const SuBilinearForm&);
SuBilinearForm operator*(const complex_t&,const SuBilinearForm&);
SuBilinearForm operator*(const SuBilinearForm&,const complex_t&);
SuBilinearForm operator/(const SuBilinearForm&,const complex_t&);
bool SuBilinearForm::checkConsistency(const SuBilinearForm&) const;
```

The member function `checkConsistency` performs a test to insure that pair of unknowns is always the same.

Finally, there are some print facilities:

```
void SuBilinearForm::print(std::ostream&)const;
std::ostream& operator<<(std::ostream&,const SuBilinearForm&);
```

8.2.5 The BilinearForm class

The `BilinearForm` class is the end user class dealing with general bilinear form, either a single unknown bilinear form (a `SuBilinearForm` object) or a multiple unknown bilinear form (a list of `SuBilinearForm` objects). In this class, a single unknown bilinear form is a multiple unknown bilinear form with one pair of unknowns! The list of `SuBilinearForm` objects is stored in a map of `SuBilinearForm`, indexed by a pair of pointers to unknown :

```
typedef std::pair<const Unknown*,const Unknown*> uvPair;

class BilinearForm
{protected :
    std::map<uvPair,SuBilinearForm> mlcblf_;; //list of linear combinations of basic
    forms
    ...
}
```

To manage the map, the following aliases are defined:

```
typedef std::map<uvPair,SuBilinearForm>::iterator it_mblc;
typedef std::map<uvPair,SuBilinearForm>::const_iterator cit_mblc;
```



When the `SuBilinearForm` unknowns owns an unknown component, the `SuBilinearForm` object is attached to its parent item! In other words, component unknowns are not indexed in the map.

This class provides only one constructor from a linear combination of forms:

```
BilinearForm(const SuBilinearForm&);
```

and proposes some accessors and facilities:

```
bool singleUnknown() const;
bool isEmpty() const;
SuBilinearForm& operator [] (const uvPair&); //protected
const SuBilinearForm& operator [] (const uvPair&) const; //protected
const SuBilinearForm& first() const;
BilinearForm operator () (const Unknown&,const Unknown&) const;
```

```
BasicBilinearForm& operator()(const Unknown&,const Unknown&,number_t);
const BasicBilinearForm& operator()(const Unknown&,const Unknown&,number_t) const;
```

Besides, there are two fundamental external enduser's function (`intg` and `intg_intg`) which constructs `BilinearForm` object:

```
BilinearForm intg(const GeomDomain&,const OperatorOnUnknown&,
                 AlgebraicOperator, const OperatorOnUnknown&);
BilinearForm intg(const GeomDomain&,const OperatorOnUnknowns&);
BilinearForm intg(const GeomDomain&,const GeomDomain&,
                 const OperatorOnUnknown&,AlgebraicOperator,
                 const OperatorOnUnknown&);
BilinearForm intg(const GeomDomain&,const GeomDomain&,const OperatorOnUnknowns&);
```

`OperatorUnknowns` is a simple class used to store the two operators and the algebraic operation.

In order to construct any linear forms, the algebraic operators (`+=` `-=` `*=` `/=` `+` `-` `*` `/`) are overloaded for different objects:

```
BilinearForm& BilinearForm::operator +=(const BilinearForm&);
BilinearForm& BilinearForm::operator -=(const BilinearForm&);
BilinearForm& BilinearForm::operator *=(const complex_t&);
BilinearForm& BilinearForm::operator /=(const complex_t&);
BilinearForm operator+(const BilinearForm&,const BilinearForm&);
BilinearForm operator-(const BilinearForm&,const BilinearForm&);
BilinearForm operator*(const complex_t&,const BilinearForm&);
BilinearForm operator*(const BilinearForm&,const complex_t&);
BilinearForm operator/(const BilinearForm&,const complex_t&);
```

Finally, the class provides usual print facilities:

```
void BilinearForm::print(std::ostream&)const;
std::ostream& operator<<(std::ostream&,const BilinearForm&);
```

Example

To end we show some characteristic examples. In these examples, `u,v` are scalar unknowns, `p, q` are vector unknowns, `f` either a scalar or a scalar function, `h` either a vector or a vector function, `g` a scalar or a scalar kernel and `Omega,Gamma Sigma` some geometric domains.

```
BilinearForm a1=intg(Omega,grad(u)|grad(v));
a1-=k2*intg(Omega,f*u*v);
a1=intg(Omega,grad(u)|grad(v))-intg(Omega,f*u*v); //same result
BilinearForm a2=intg(Omega,div(p)*q)+eps*intg(Omega,p*q);
BilinearForm a3=a1+a2;
a3=intg(Omega,grad(u)|grad(v))+intg(Omega,div(p)*q) //same result
   -intg(Omega,f*u*v)+eps*intg(Gamma,p*q);
```

```
library : form
header : BilinearForm.hpp
implementation : BilinearForm.cpp
unitary tests : test_form.cpp
header dependences : config.h, utils.h
```

The *largeMatrix* library deals with large matrices stored in different ways : dense storages (all the matrix values are stored), skyline storages (values on a row or a column are stored from the first non zero values) and compressed sparse storage (only non zero values are stored). For each of these storages, different "access" methods, driving how the matrix values are stored, are proposed : by rows, by columns, by rows and columns (say dual access) and symmetric access which is like dual access but with same row and column access. The symmetric access is well suited for matrices having symmetry property (symmetric, skew-symmetric, self-adjoint or skew-adjoint), or matrix with symmetric storage property, currently met in finite element approximation. Note that the row dense storage corresponds to the storage used by the *Matrix* class defined in the *utils*. As other access methods may be useful in finite element methods, it is the reason why the *largeMatrix* library provides dense storages.

Large matrices under consideration may be real or complex value matrices, but also matrices of real/complex matrices. This last case allows to deal with vector variational problems where the unknown and test function are vector functions, using the same storage structure as in the scalar case.

More precisely, the *largeMatrix* library is based on the template class *LargeMatrix<T>* which mainly handles a vector to store the matrix values and a pointer to a storage describing how the matrix values are stored. Different storage structures are organized in a hierarchy of classes, terminal classes carrying specific data and methods of storages, in particular the product of a matrix and a vector.

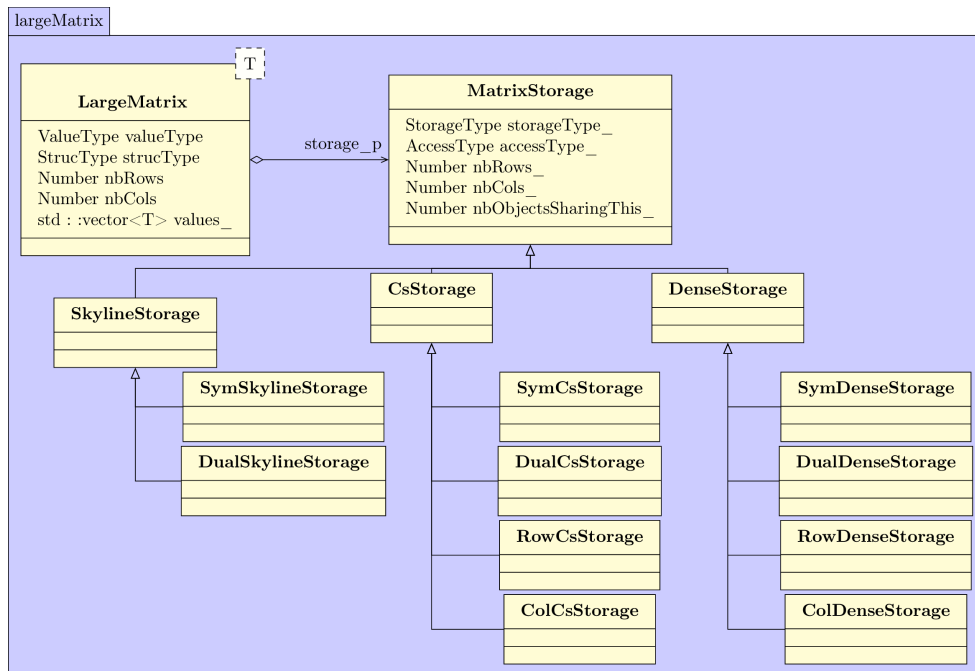


Figure 9.1: The *largeMatrix* library main class diagram

9.1 The LargeMatrix class

The `LargeMatrix` class deals with large matrices with different type of storages. It is a template class `LargeMatrix<T>` where `T` is either the type `Real`, `Complex`, `Matrix<Real>` or `Matrix<Complex>`. Although it is possible to instantiate `LargeMatrix<T>` with other types, most of functionalities do not work because some member functions are overloaded only for the previous types.

The class mainly manages a vector storing the matrix values and a pointer to a storage structure (`MatrixStorage` base class) :

```
template <typename T>
class LargeMatrix{
public :
    ValueType valueType;           // type of values (real, complex)
    StructType structType;         // struc of values (scalar, matrix)
    Number nbRows;                 // number of rows counted in T
    Number nbCols;                 // number of columns counted in T
    SymType sym;                   // type of symmetry
    Dimen nbRowsSub;               // number of rows of submatrices (1 if scalar values)
    Dimen nbColsSub;               // number of columns of submatrices (1 if scalar values)
    String name;                   // optionnal name, useful for documentation
protected :
    vector<T> values_;             // list of values ordered along storage method
    MatrixStorage* storage_p;      // pointer to the matrix storage
    ...
}
```

When `T=Matrix<Real>` or `T=Matrix<Complex>`, the number of rows and columns are counted in `Matrix` not in `scalar`!



The first value (index 0) of the vector `values_` is not used for storing a matrix value. It contains the default value (in general `T()`) returned when trying to access to a matrix value outside the storage. Be care with default `Matrix<T>()`, its a 0×0 matrix!

There are various constructors : three from a existing storage, some from file where the large matrix is loaded from the file and some from matrix dimensions and a default value. A large matrix which is created by the copy constructor (shallow copy) will share a same storage with the input one. Be care with these last constructors, they allocate the vector of values only for dense storage.

```
LargeMatrix();
LargeMatrix(const LargeMatrix<T>&);
LargeMatrix(MatrixStorage*, const T&= T(), SymType= _noSymmetry);
LargeMatrix(MatrixStorage* ms, dimPair dims, SymType sy = _noSymmetry);
LargeMatrix(MatrixStorage*, SymType);
LargeMatrix(Number, Number, StorageType = _dense, AccessType = _row, const T& =
    T());
LargeMatrix(Number, Number, StorageType = _dense, SymType = _symmetric, const T& =
    T());
LargeMatrix(const String&, StorageType, Number, Number, StorageType = _dense,
    AccessType = _row, Number nsr=1, Number nsc=1);
LargeMatrix(const String&, StorageType, Number, StorageType = _dense,
    SymType = _symmetric, Number nsr=1);
~LargeMatrix();

void init(MatrixStorage*, const T&, SymType);
void allocateStorage(StorageType, AccessType, const T&= T());
void setType(const T&);
```

The `StorageType`, `AccessType` and `SymType` are enumerations :


```
enum StorageType { _noStorage=0, _dense, _cs, _skyline, _coo };
enum AccessType { _noAccess=0, _sym, _row, _col, _dual };
enum SymType { _noSymmetry=0, _symmetric, _skewSymmetric, _selfAdjoint, _skewAdjoint,
               _diagonal };
```

The storage type `_cs` means compressed sparse. It refers to the well known CSR/CSC storage (see the next section for details on storage). The storage type `_coo` means coordinates storage (i, j, a_{ij}). So far, it is not proposed as a storage method for large matrix but large matrix may be saved to file in this format. Constructors use the utilities: `init`, `allocateStorage` and `setType`. The `setType` function uses some utilities (external functions) to get the dimensions of sub-matrices:

```
pair<Dimen, Dimen> dimsOf(const Real&);
pair<Dimen, Dimen> dimsOf(const Complex&);
pair<Dimen, Dimen> dimsOf(const Matrix<Real>&);
pair<Dimen, Dimen> dimsOf(const Matrix<Complex>&);
bool isDiagonal() const;
bool isId(const double & tol=0.) const;
```



Note that for the moment, it is not possible to load from a file a matrix as a matrix of matrices (use only `nsr=1` and `nsc=1` arguments in constructors from file).

There are few facilities to access to matrix values :

```
Number pos(Number i, Number j) const;
void positions(const vector<number_t>&, const vector<number_t>&,
              vector<number_t>&, bool = false) const;
vector<pair<number_t, number_t>> getCol(number_t, number_t = 1, number_t = 0) const;
vector<pair<number_t, number_t>> getRow(number_t, number_t = 1, number_t = 0) const;
T operator()(number_t adr) const;
T& operator()(number_t adr);
typename vector<T>::iterator at(number_t, number_t);
typename vector<T>::const_iterator at(number_t i, number_t j) const;
T operator()(Number i, Number j) const;
T& operator()(Number i, Number j, bool = true);
Number nbNonZero() const;
void resetZero();
```

The member function `pos(i,j)` returns the adress in vector `values_` of the matrix value M_{ij} for $i, j \geq 1$. The returned adress begins at 1 except when the value is outside the storage where the function returns 0. The access operator `(i,j)` returns the matrix value or a reference to it with the *non const* version. In that case, it is possible to activate (`errorOn=true`) an error handler to prevent access to values outside the storage. It is the default behaviour. When `errorOn=false`, the access operator returns a reference to `values_[0]`, thus you can modify its value. This trick avoids to perform test like `if(pos(i,j)!=0)...` before writing matrix values. To be safe at end, reset the `values_[0]` to 0 with `resetZero` function. Note that the skyline or compressed storage structure can not be dynamically changed when "inserting" a new value.



`pos(i,j)`, `getRow` and `getCol` function should not be used in heavy computations because it is time consuming. In heavy computation, write codes based on storage structure!

As `LargeMatrix` class provides a lot of functions to modify its contents, some being time expansive regarding the storage :

```
void deleteRows(number_t, number_t);
void deleteCols(number_t, number_t);
void setColToZero(number_t c1=0, number_t c2=0);
```

```

void setRowToZero(number_t r1=0, number_t r2=0);
void roundToZero(real_t aszero=10*theEpsilon);

LargeMatrix<T>& operator*=(const K&);
LargeMatrix<T>& operator/=(const K&);
LargeMatrix<T>& operator+=(LargeMatrix<T>&);
LargeMatrix<T>& operator-=(LargeMatrix<T>&);

LargeMatrix<T>& add(iterator&,iterator&);
LargeMatrix<T>& add(const vector<T>&, const vector<number_t>&,const
    vector<number_t>&);
LargeMatrix<T>& add(const T& c, const vector<number_t>&, const vector<number_t>&);
LargeMatrix<T>& add(const LargeMatrix<K>&, const vector<number_t>&,const
    vector<number_t>&, C);
void addColToCol(number_t, number_t, complex_t a, bool =false);
void addRowToRow(number_t, number_t, complex_t a, bool =false);
void copyVal(const LargeMatrix<T>&, const vector<number_t>&, const
    vector<number_t>&);
LargeMatrix<T>& assign(const LargeMatrix<K>&, const vector<number_t>&, const
    vector<number_t>&);

```

Some functions are provided to change the scalar/matrix representation of the matrix or the storage:

```

void toSkyline();
void toStorage(MatrixStorage*);
LargeMatrix<K>* toScalar(K);
void toScalarEntries(const vector<Matrix<K>> &, vector<K>&, const MatrixStorage&);
void toUnsymmetric(AccessType at=sym);
void extract2UmfPack(vector<int_t>& colPointer, vector<int_t>& rowIndex, vector<T>&
    matA) const;
bool getCsrColUmfPack(vector<int_t>& colPointer, vector<int_t>& rowIndex, const T&
    matA) const;

```

Matrices that are stored as Dense or Skyline can be factorized as LU, LDLt, LDL* and matrices that are stored as Skyline or Sparse can be factorized using UMFPack if it is available. All the stuff related to solve linear system is also available.

```

void ldltFactorize();
void ldltstarFactorize();
void luFactorize(bool withPermutation=true);
void umfpackFactorize();

void ldltSolve(vector<S1>& vec, vector<S2>& res) const;
void ldltstarSolve(vector<S>& vec, vector<T>& res) const;
void luSolve(vector<S1>& vec, vector<S2>& res) const;
void umfluSolve(vector<S1>& vec, vector<S2>& res) const;

void sorLowerMatrixVector(const vector<S1>& vec, vector<S2>& res, real_t) const;
void sorDiagonalMatrixVector(const vector<S1>& vec, vector<S2>& res, real_t) const;
void sorUpperMatrixVector(const vector<S1>& vec, vector<S2>& res, real_t) const;
void sorLowerSolve(const vector<S1>& vec, vector<S2>& res, real_t) const;
void sorDiagonalSolve(const vector<S1>& vec, vector<S2>& res, real_t) const;
void sorUpperSolve(const vector<S1>& vec, vector<S2>& res, real_t) const;

real_t umfpackSolve(const vector<S>& vec,
    vector<typename Conditional<NumTraits<ScalarType>>::IsComplex, ScalarType,
    S>::type>& res,
    bool soa=true);
real_t umfpackSolve(const vector<vector<S>*>&,
    vector<vector<typename Conditional<NumTraits<ScalarType>>::IsComplex,
    ScalarType, S>::type>*>&,
    bool soa=true);

```

```

real_t umfpackSolve(const vector<int_t>& colPointer, const vector<int_t>& rowIndex,
const vector<T>& values, const vector<S>& vec,
vector<typename Conditional<NumTraits<ScalarType>::IsComplex, ScalarType,
S>::type>& res,
bool soa=true);

```

In a same way, some functions interfaces eigen solvers:

```

friend number_t eigenDavidsonSolve(const LargeMatrix<K>* pA, const LargeMatrix<K>*
pB,
vector<pair<complex_t, Vector<complex_t>>>& res, number_t nev, real_t tol,
string_t which, bool isInverted, FactorizationType fac, bool isShift);
friend number_t eigenKrylovSchurSolve(const LargeMatrix<K>* pA, const
LargeMatrix<K>* pB,
vector<pair<complex_t, Vector<complex_t>>>& res, number_t nev, real_t tol,
string_t which, bool isInverted, FactorizationType fac, bool isShift);

```

The Frobenius norm $\sqrt{\sum |a_{ij}|^2}$ and infinity norm $\max_{|a_{ij}|}$ are provided:

```

real_t norm2() const;
real_t norminf() const;
real_t partialNormOfCol(number_t, number_t, number_t) const;

```

Some operations are available as external functions to the class.

The `LargeMatrix` supplies methods to add two large matrices, which share a same storage. The matrix result will use the same storage as the two added matrices. If the result doesn't point to the storage of addend, after the addition, it will be forced to use this storage, and the number of object sharing this storage increases by one. Similar to multiplication of matrix-vector, these functions are also external templates, with specializations to mixed real and complex types.

```

friend void addMatrixMatrix(const LargeMatrix<S>& matA, const LargeMatrix<S>& matB,
LargeMatrix<S>& matC);
friend void addMatrixMatrix(const LargeMatrix<Complex>& matA, const
LargeMatrix<Real>& matB, LargeMatrix<Complex>& matC);
friend void addMatrixMatrix(const LargeMatrix<Real>& matA, const
LargeMatrix<Complex>& matB, LargeMatrix<Complex>& matC);

```

The operator `+` is overloaded for `LargeMatrix` in the same manner.

```

LargeMatrix<T> operator+(const LargeMatrix<T>& matA, const LargeMatrix<T>& matB);
LargeMatrix<Complex> operator+(const LargeMatrix<Real>& matA, const
LargeMatrix<Complex>& matB);
LargeMatrix<Complex> operator+(const LargeMatrix<Complex>& matA, const
LargeMatrix<Real>& matB);

```



The symmetry of the result matrix depends on the ones of both addends. If one of these has `_noSymmetry` as `SymType`, the result will have the same `SymType` (i.e: `_noSymmetry`).

A large matrix can multiply by a scalar with the external templated functions

```

friend LargeMatrix<S> multMatrixScalar(const LargeMatrix<S>&, const S);
friend LargeMatrix<Complex> multMatrixScalar(const LargeMatrix<Complex>&, const
Real);
friend LargeMatrix<Complex> multMatrixScalar(const LargeMatrix<Real>&, const
Complex);

```

Like `addMatrixMatrix`, the result of `multMatrixScalar` shares the same storage of the input large matrix.

The operator `*` can be also overloaded in the same manner.

```

LargeMatrix<T> operator*(const LargeMatrix<T>& mat, const T v);
LargeMatrix<T> operator*(const T v, const LargeMatrix<T>& mat);
LargeMatrix<Complex> operator*(const LargeMatrix<Complex>& mat, const Real v);
LargeMatrix<Complex> operator*(const Real v, const LargeMatrix<Complex>& mat);
LargeMatrix<Complex> operator*(const LargeMatrix<Real>& mat, const Complex v);
LargeMatrix<Complex> operator*(const Complex v, const LargeMatrix<Real>& mat);

```

The product of matrix and vector is one of the most important operation required. The `LargeMatrix` class interfaces matrix-vector product and the vector-matrix matrix; the product being really done by the storage classes. They are external templated functions to the class (declared friend of class), with specializations to mixed real and complex types (only casting from real to complex is allowed) :

```

void multMatrixVector(const LargeMatrix<S>&, const vector<S>&, vector<S>&);
void multVectorMatrix(const LargeMatrix<S>&, const vector<S>&, vector<S>&);
void multMatrixVector(const LargeMatrix<Matrix<S> >&,
                      const vector<Vector<S> >&, vector<Vector<S> >&);
void multVectorMatrix(const LargeMatrix<Matrix<S> >&,
                      const vector<Vector<S> >&, vector<Vector<S> >&);

```

The operator `*` is overloaded for `LargeMatrix` and vector or vector and `LargeMatrix` in a same way :

```

friend vector<S> operator*(const LargeMatrix<S>&, const vector<S>&);
friend vector<S> operator*(const vector<S>&, const LargeMatrix<S>&);
friend vector<Vector<S> > operator*(const LargeMatrix<Matrix<S> >&,
                                   const vector<Vector<S> >&);
friend vector<Vector<S> > operator*(const vector<Vector<S> >&,
                                   const LargeMatrix<Matrix<S> >&);
...

```

The product of two matrices is provided, but be cautious, up to now the result matrix is a dense matrix (except if one matrix is a diagonal one):

```

void multMatrixMatrix(const LargeMatrix<SA>&, const LargeMatrix<SB>&,
                     LargeMatrix<SR>&);

```

The `LargeMatrix` also provides some functions to factorize a matrix and solve factorized system. Like `multMatrixVector`, these functions are external (friend of class), template with hybrid complex-real specialization.

```

void ldltFactorize(LargeMatrix<S>& mat);
void ldlstarFactorize(LargeMatrix<S>& mat);
void luFactorize(LargeMatrix<S>& mat);

```



These functions are available for `SkylineStorage` and `DenseStorage`.

If a matrix is factorized, some other operation are available

```

void multFactMatrixVector(const LargeMatrix<S1>&, const vector<S2>&, vector<S3>&);
void multVectorFactMatrix(const LargeMatrix<S1>&, const vector<S2>&, vector<S3>&);
void multInverMatrixVector(const LargeMatrix<S1>& mat, const vector<S2>& vec,
                           vector<typename Conditional<NumTraits<S1>::IsComplex, S1, S2>::type>& res,
                           FactorizationType);

```

In order to generate a diagonal matrix which uses an existent matrix as prototype, `LargeMatrix` class provides function

```
LargeMatrix<S> diagonalMatrix(const LargeMatrix<S>&, const S);
```

A special version of this function is a function to generate an identity matrix from an existing one.

```
LargeMatrix<S> identityMatrix(const LargeMatrix<S>&);
```



These functions work with `LargeMatrix` class having all kinds of matrix storage except `RowCsStorage` class and `ColCsStorage` class.

Finally, the class has input/output member functions :

```
void print(ostream&) const;
void viewStorage(ostream&) const;
ostream& operator<<(ostream&, const LargeMatrix<S>&);
void saveToFile(const String &, StorageType, bool encode=false) const;
void loadFromFile(const String&, StorageType);
String encodeFileName(const String&, StorageType) const;
```

Only two formats are allowed for saving matrix to file or loading matrix from file :

- the dense format (`_dense`), where all the matrix values are written row by row (one row by line), space character separates each value :

A11 A12 ... A1n		re(A11) im(A11) ... re(A1n) im(A1n)
A21 A22 ... A2n	if complex values	re(A21) im(A21) ... re(A2n) im(A2n)
...		...
Am1 Am2 ... Amn		re(Am1) im(Am1) ... re(Amn) im(Amn)

- the coordinates format (`_coo`)

i1 j1 Ai1j1		i1 j1 re(Ai1j1) im(Ai1j1)
i2 j2 Ai2j2	if complex values	i2 j2 re(Ai2j2) im(Ai2j2)
...		



For matrix of matrices (`T=Matrix<Real>` or `T=Matrix<Complex>`), submatrix structure is not preserved when writing matrix values!

To keep some general informations about matrix, with the argument `encode=true` in the `saveFile` function, the name of file may be modified by inserting the string :

```
(m_n_storageType_valueType_strucType_p_q)
```

where `m` and `n` are the "dimensions" of the matrix, `storageType` = `dense` or `coo`, `valueType` = `real` or `complex`, `strucType` = `scalar` or `matrix` and `p` and `q` are the dimensions of sub-matrices. In case of scalar value, this parameters are omitted. This trick avoids to include informations in file in order that it is easily loadable by other softwares, in particular Matlab.

Some examples

Manipulate a 3×2 matrix with row dense storage:

```
Vector<Real> x(2,2);
LargeMatrix<Real> A(3,2,_dense,_row,1.);
out<<"matrix A : "<<A;
out<<"access A(1,1)= "<<A(1,1);
out<<"product A*x : "<<A*x;
A.saveToFile("A_dense.mat",_dense);
LargeMatrix<Real> Areload("A_dense.mat",_dense,3,2,_dense,_row);
```

Manipulate a 3×2 matrix of 2×2 real matrices with dual dense storage:

```
Matrix<Real> I1(2, _idMatrix);
Vector<Real> x(6,3);
LargeMatrix<Matrix<Real> > B(6,3, _dense, _dual, 2*I1);
out<<"product X*B : "<<X*B;
```

Manipulate a 9×9 symmetric matrix with symmetric compressed storage:

```
//construct storage for a Q1 mesh
Number n=3;
vector<vector<Number> > elts((n-1)*(n-1), vector<Number>(4));
for(Number j=1; j<=n-1; j++)
  for(Number i=1; i<=n-1; i++) {
    Number e=(j-1)*(n-1)+i-1, p=(j-1)*n+i;
    elts[e][0]=p; elts[e][1]=p+1; elts[e][2]=p+n; elts[e][3]=p+n+1;
  }
SymCsStorage* cssym= new SymCsStorage(n*n, elts, elts);
//construct large matrix with symmetric compressed storage
LargeMatrix<Real> C(cssym, 1., _symmetric);
C.viewStorage(out);
Vector<Real> x(n*n, 0.); for(Number i=0; i<n*n; i++) x[i]=i;
out<<"product C*x : "<<C*x;
C.saveToFile("C-coo.mat", _coo);
//construct large matrix with symmetric compressed storage
LargeMatrix<Real> C1(cssym, 1., _symmetric);
out<<"addition C+C1 "<< C+C1;
// multiply a matrix with a scalar
out<<"product C*10.0 "<< C*10.0;
// generate a diagonal matrix
out<<"diagonal matrix from matrix C " << diagonalMatrix(C, 10.0);
// generate an identity matrix
out<<"identity matrix from matrix C " << identityMatrix(C);
```

Note that compressed storage has to be built before to instantiate a large matrix with compressed storage. The function `viewStorage` produced the output:

symmetric_compressed sparse (csr,csc), size = 29, 9 rows, 9 columns
(shared by 1 objects).

```
123456789
1 d.....
2 xd.....
3 .xd.....
4 xx.d.....
5 xxxxd....
6 .xx.xd...
7 ...xx.d..
8 ...xxxxd.
9 ....xx.xd
123456789
```

LDLt-Factorize a positive definite 3x3 matrix with symmetric skyline storage:

```
LargeMatrix<Real> rResSymSkylineLdlt(inputsPathTo("matrix3x3SymPosDef.data"),
    _dense, rowNum, _skyline, _symmetric);
ldltFactorize(rResSymSkylineLdlt);
out << "The result of LDLt factorizing sym skyline matrix is " <<
    rResSymSkylineLdlt << endl;
```

library :	largeMatrix
header :	LargeMatrix.hpp
implementation :	LargeMatrix.cpp
unitary tests :	test_LargeMatrixDenseStorage.cpp, test_LargeMatrixCsStorage.cpp
header dependences :	MatrixStorage.hpp, DenseStorage.hpp, CsStorage.hpp, SkylineStorage.hpp, config.h, utils.h

9.2 The MatrixStorage class

The **MatrixStorage** is the abstract base class of all supported matrix storages. Vectors carrying storage pointers are defined in child classes and the vector storing the matrix entries is defined outside the storage (see **LargeMatrix** class for instance). Various matrix storages are defined by two characteristics :

- a **StorageType** : **_dense** for dense storage, **_cs** for compressed sparse storage (CSR/CSC) and **_skyline** for skyline storage
- an **AccessType** : **_row** for a row wise access, **_col** for a column wise access, **_dual** for a row wise access for lower triangular part of the matrix and a column wise access for the upper triangular part of the matrix, and **_sym** if the storage is symmetric (only the lower triangular part is described with a row wise access).

```
class MatrixStorage
{protected:
    StorageType storageType_;           // storage type
    AccessType accessType_;            // access type
    Number nbRows_;                    // number of rows
    Number nbCols_;                    // number of columns
    Number nbObjectsSharingThis_;      // number of matrices sharing "this" storage
    ...
}
```

The number of rows and columns are counted in value type of the matrix.

The class provides some constructors which only initialize members of the class.

```
MatrixStorage();
MatrixStorage(StorageType, AccessType at = _noAccess);
MatrixStorage(StorageType, AccessType, Number, Number);
virtual ~MatrixStorage();
```

Protected members may be read by accessors :

```
String name() const;
AccessType accessType() const;
Number nbOfRows() const;
Number nbOfColumns() const;
Number numberOfObjects() const
void objectPlus();
void objectMinus();
```

The class provides virtual member functions returning the sizes of storage (number of matrix entries stored) :

```
virtual Number size() const = 0;
Number diagonalSize() const;
virtual Number lowerPartSize() const=0;
virtual Number upperPartSize() const=0;
```


Most of member functions depend on type of storage, so they are virtual member functions. Besides, as storages do not depend on value type of matrix, the storage classes are not templated by the matrix value type. Because virtual template functions are not allowed in C++, all the value type specializations are explicit in base class and in child classes. Up to now, the supported specialisations are `Real`, `Complex`, `Matrix<Real>` and `Matrix<Complex>`. We describe here only the `Real` specialisation.

There are two functions related to matrix entry position in storage, the first one gives the position of entry (i, j) with $i, j \geq 1$ and the second one compute the positions of the entries of the submatrix given by its row indices and column indices. These functions give **positions from index 1** if the entry is inside the storage else the returned position is 0. In the second function, it is possible to activate an error when entry index is outside the storage. In both functions, the symmetry has to be specified if the matrix has a symmetry property, in that case the position is always in lower triangular part. Be care with the fact that a matrix with no symmetry may have a symmetric storage!

```
virtual Number pos(Number i, Number j, SymType sym=_noSymmetry) const;
virtual void positions(const std::vector<Number>&, const std::vector<Number>&,
                      std::vector<Number>&, bool errorOn=true,
                      SymType=_noSymmetry) const;
```

To print, to save to file or load from file large matrices, the following functions are proposed :

```
virtual void printEntries(std::ostream&, const std::vector<Real>&,
                          Number vb = 0, SymType sym = _noSymmetry) const;
template<typename T>
void printDenseMatrix(std::ostream&, const std::vector<T>&, SymType
                      s=_noSymmetry) const;
template<typename T>
void printCooMatrix(std::ostream&, const std::vector<T>&, SymType s=_noSymmetry)
const;
virtual void loadFromFileDense(std::istream&, std::vector<Real>&, SymType, bool);
virtual void loadFromFileCoo(std::istream&, std::vector<Real>&, SymType, bool);
virtual void visual(std::ostream&) const;
```

The `printEntries` function print on output stream the stored matrix entries in a well suited presentation, the `printDenseMatrix` function print the matrix entries in a dense format as it was dense and the `printCooMatrix` function print the matrix entries in a coordinates format (i, j, A_{ij}) . A $m \times n$ matrix saved as dense produced a file like :

A11 A12 ... A1n		re(A11) im(A11) ... re(A1n) im(A1n)
A21 A22 ... A2n	if complex values	re(A21) im(A21) ... re(A2n) im(A2n)
...		...
Am1 Am2 ... Amn		re(Am1) im(Am1) ... re(Amn) im(Amn)

and when saved as coordinates format :

i1 j1 Ai1j1		i1 j1 re(Ai1j1) im(Ai1j1)
i2 j2 Ai2j2	if complex values	i2 j2 re(Ai2j2) im(Ai2j2)
...		...

Note that `printDenseMatrix` and `printCooMatrix` functions are templated functions. Their coding is not optimal because they use the virtual `pos` function and other extern utilities which "interpret" the type of the template :

```
void printDense(std::ostream&, const Real&, Number);
void printDense(std::ostream&, const Complex&, Number);
template<typename T> void printDense(std::ostream&, const Matrix<T>&, Number);
void printCoo(std::ostream&, const Real&, Number);
void printCoo(std::ostream&, const Complex&, Number);
```



```

template<typename T> void printCoo(std::ostream&, const Matrix<T>&, Number);
Number numberOfRows(const Real&);
Number numberOfRows(const Complex&);
Number numberOfCols(const Real&);
Number numberOfCols(const Complex&);
template<typename T> Number numberOfRows(const Matrix<T>&);
template<typename T> Number numberOfCols(const Matrix<T>& m);

```

The `visual` function prints only the matrix entries inside the storage as 'x' characters (d for diagonal entries):

```

matrix storage :row_compressed sparse (csr,csc), size = 49, 9 rows, 9 columns
(shared by 0 objects).
123456789
1  dx.xx...
2  xdx...
3  .xd.xx...
4  xx.dx.xx.
5  xxxdxxxx
6  .xx.xd.xx
7  ...xx.dx.
8  ...xxxdx
9  ....xx.xd
123456789

```

The `MatrixStorage` class provides the product matrix \times vector and the product vector \times matrix:

```

virtual void multMatrixVector(const std::vector<Real>&, const std::vector<Real>&,
std::vector<Real>&, SymType) const;
virtual void multMatrixVector(const std::vector<Matrix<Real>>&, const
std::vector<Vector<Real>>&, std::vector<Vector<Real>>&, SymType) const;
virtual void multVectorMatrix(const std::vector<Real>&, const std::vector<Real>&,
std::vector<Real>&, SymType) const;
virtual void multVectorMatrix(const std::vector<Matrix<Real>>&, const
std::vector<Vector<Real>>&, std::vector<Vector<Real>>&, SymType) const;

```

Note that mixing real vector/matrix and complex vector/matrix is also provided. Each child class implements the best algorithm for the product.

The addition of two matrices are supplied with:

```

virtual void addMatrixMatrix(const std::vector<Real>&, const std::vector<Real>&,
std::vector<Real>&) const;

```

Similar to multiplication of matrix-vector, this function is implemented in each child class. The `MatrixStorage` class also provides the matrix factorizations

```

virtual void ldlt(std::vector<Real>&, std::vector<Real>&,
const SymType sym = _symmetric) const;
virtual void ldlt(std::vector<Complex>&, std::vector<Complex>&,
const SymType sym = _symmetric) const;
virtual void lu(std::vector<Real>&, std::vector<Real>&,
const SymType sym = _noSymmetry) const;
virtual void lu(std::vector<Complex>&, std::vector<Complex>&,
const SymType sym = _noSymmetry) const;
virtual void ldlstar(std::vector<Real>&, std::vector<Real>&) const;
virtual void ldlstar(std::vector<Complex>&, std::vector<Complex>&) const;

```

as well as some special solvers

```

virtual void lowerD1Solver(const std::vector<Real>&, std::vector<Real>&,
std::vector<Real>&) const;
virtual void diagonalSolver(const std::vector<Real>&, std::vector<Real>&,

```

```

        std::vector<Real>&) const
virtual void upperD1Solver(const std::vector<Real>&, std::vector<Real>&,
        std::vector<Real>&, const SymType sym = _noSymmetry)
        const;
virtual void upperSolver(const std::vector<Real>&, std::vector<Real>&,
        std::vector<Real>&, const SymType sym = _noSymmetry) const;
virtual void sorDiagonalMatrixVector(const std::vector<Real>&,
        const std::vector<Real>&, std::vector<Real>&, const Real) const;
virtual void sorLowerMatrixVector(const std::vector<Real>&,
        const std::vector<Real>&, std::vector<Real>&,
        const Real, const SymType = _noSymmetry) const;
virtual void sorUpperMatrixVector(const std::vector<Real>&,
        const std::vector<Real>&, std::vector<Real>&,
        const Real, const SymType = _noSymmetry) const;
virtual void sorDiagonalSolve(const std::vector<Real>&,
        const std::vector<Real>&, std::vector<Real>&, const Real) const;
virtual void sorLowerSolve(const std::vector<Real>&,
        const std::vector<Real>&, std::vector<Real>&, const Real) const;
virtual void sorUpperSolve(const std::vector<Real>&,
        const std::vector<Real>&, std::vector<Real>&,
        const Real, const SymType sym = _noSymmetry) const;

```

Each descendant will implements the specific algorithm for these virtual functions. Values on the diagonal of a matrix is set with the function

```

virtual void setDiagValue(std::vector<Real>&, const Real);

```

library :	largeMatrix
header :	MatrixStorage.hpp
implementation :	MatrixStorage.cpp
unitary tests :	test_LargeMatrix...Storage.cpp
header dependences :	config.h, utils.h

9.3 Dense storages

Dense storages are proposed as childs of `MatrixStorage` class in order to be consistent with other storages. Contrary to the `Matrix` class (see `utils` lib) which is a row dense storage, several dense storages are offered for a $m \times n$ matrix (say A) :

- the row dense storage (`RowDenseStorage` class)

$$(A_{11}, A_{12}, \dots, A_{1n}, A_{21}, A_{22}, \dots, A_{2n}, \dots, A_{m1}, A_{m2}, \dots, A_{mn})$$

- the column dense storage (`ColDenseStorage` class)

$$(A_{11}, A_{21}, \dots, A_{m1}, A_{12}, A_{22}, \dots, A_{m2}, \dots, A_{1n}, A_{2n}, \dots, A_{mn})$$

- the dual dense storage (`DualDenseStorage` class), diagonal part is first stored, then strict lower triangular part and strict upper triangular part :

$$(A_{11}, A_{22}, \dots, A_{mm}, A_{21}, \dots, A_{i1}, \dots, A_{i,i-1}, \dots, A_{12}, \dots, A_{1j}, \dots, A_{j-1,j}, \dots) \text{ case } m = n$$

- the symmetric dense storage (`SymDenseStorage` class), diagonal part is first stored, then strict lower triangular part :

$$(A_{11}, A_{22}, \dots, A_{mm}, A_{21}, \dots, A_{i1}, \dots, A_{i,i-1}, \dots) \text{ case } m = n$$

Dual storage works also for non square matrices. Obviously, symmetric storage works only for square matrices!

All dense storages inherit from the `DenseStorage` abstract class which inherits from `MatrixStorage` abstract class.

9.3.1 The DenseStorage class

The `DenseStorage` class has no member attribute. There are some basic constructors just calling `MatrixStorage` constructors:

```
class DenseStorage : public MatrixStorage
{public :
    DenseStorage();
    DenseStorage(AccessType);
    DenseStorage(AccessType, Number);
    DenseStorage(AccessType, Number, Number);
    virtual ~DenseStorage() {}
    ...
}
```

`DenseStorage` objects do not have to be instantiated. This class acts as an interface to particular dense storages and gathers all common functionalities of dense storages, some being virtuals:

- public size functions:

```
Number size() const;
virtual Number lowerPartSize() const;
virtual Number upperPartSize() const;
```

- protected input/output functions (template line is written once):

```
template<typename Iterator>
void printScalarEntries(Iterator&, Number, Number, Number, Number, Number,
                        const String&, Number, std::ostream&) const;
void printScalarEntriesTriangularPart(Iterator&, Iterator&, Number, Number,
                                      Number, Number, Number, const String&,
                                      Number, std::ostream&) const;
void printMatrixEntries(Iterator&, Number, Number, const String&, Number,
                        std::ostream&) const;
void printMatrixEntriesTriangularPart(Iterator&, Iterator&, Number, Number,
                                      const String&, Number, std::ostream&) const;
void loadFromFileDense(std::istream&, std::vector<Real>&, SymType, bool);
void loadFromFileDense(std::istream&, std::vector<Complex>&, SymType, bool);
void loadFromFileCoo(std::istream&, std::vector<Real>&, SymType, bool);
void loadFromFileCoo(std::istream&, std::vector<Complex>&, SymType, bool);
```

- product of matrix and vector (template line is written once):

```
template<typename MatIterator, typename VecIterator, typename ResIterator>
void diagonalMatrixVector(MatIterator&, VecIterator&, ResIterator&,
                          ResIterator&) const;
void diagonalVectorMatrix(MatIterator&, VecIterator&, ResIterator&,
                          ResIterator&) const;
void lowerMatrixVector(MatIterator&, VecIterator&, VecIterator&,
                       ResIterator&, ResIterator&, SymType) const;
void lowerVectorMatrix(MatIterator&, VecIterator&, VecIterator&,
                       ResIterator&, ResIterator&, SymType) const;
void upperMatrixVector(MatIterator&, VecIterator&, VecIterator&,
                       ResIterator&, ResIterator&, SymType) const;
void upperVectorMatrix(MatIterator&, VecIterator&, VecIterator&,
                       ResIterator&, ResIterator&, SymType) const;
```

```

ResIterator&, ResIterator&, SymType) const;
void rowMatrixVector(MatIterator&, VecIterator&, VecIterator&,
ResIterator&, ResIterator&) const;
void rowVectorMatrix(MatIterator&, VecIterator&, VecIterator&,
ResIterator&, ResIterator&) const;
void columnMatrixVector(MatIterator&, VecIterator&, VecIterator&,
ResIterator&, ResIterator&) const;
void columnVectorMatrix(MatIterator&, VecIterator&, VecIterator&,
ResIterator&, ResIterator&) const;

```

Note that the product of matrix and vector are performed either by row or column access, for diagonal part, lower part or upper part. They use iterators to by-pass the templated matrix values vector! With this trick, product algorithms are all located in [DenseStorage](#) class.

- addition of two matrices:

```

template<typename Mat1Iterator, typename Mat2Iterator, typename ResIterator>
void sumMatrixMatrix(Mat1Iterator&, Mat2Iterator&, ResIterator&, ResIterator&)
const;

```

library :	largeMatrix
header :	DenseStorage.hpp
implementation :	DenseStorage.cpp
unitary tests :	test_LargeMatrixDenseStorage.cpp
header dependencies :	MatrixStorage.hpp, config.h, utils.h

9.3.2 RowDenseStorage, ColDenseStorage, DualDenseStorage and SymDenseStorage classes

All child classes ([RowDenseStorage](#), [ColDenseStorage](#), [DualDenseStorage](#) and [SymDenseStorage](#)) of [DenseStorage](#) class have the same structure. We give here only the description of [RowDenseStorage](#) class.

[RowDenseStorage](#) class manages dense storage where matrix values are stored row by row in values vector. It has no member attribute and has constructors just calling [DenseStorage](#) constructors:

```

class RowDenseStorage : public DenseStorage
{public :
    RowDenseStorage();
    RowDenseStorage(Number);
    RowDenseStorage(Number, Number);
    virtual ~RowDenseStorage() {}
    ...

```

The class provides all the virtual methods declared in [MatrixStorage](#) class :

```

Number pos(Number i, Number j, SymType s=_noSymmetry) const;
void positions(const std::vector<Number>&, const std::vector<Number>&,
std::vector<Number>&, bool errorOn=true, SymType=_noSymmetry ) const;
void printEntries(std::ostream&, const std::vector<Real>&,
Number vb = 0, SymType sym = _noSymmetry) const;
template<typename M, typename V, typename R>
void multMatrixVector(const std::vector<M>&, const std::vector<V>&,
std::vector<R>&) const;
void multVectorMatrix(const std::vector<M>&, const std::vector<V>&,
std::vector<R>&) const;
template<typename T>

```

```

void setDiagValueColDense(std::vector<T>& m, const T k);
template<typename M1, typename M2, typename R>
void addMatrixMatrix(const std::vector<M1>&, const std::vector<M2>&,
std::vector<R>&) const;
...

```

Only version of `multMatrixVector` and `multVectorMatrix` are written here for `Real` type but versions for `Complex`, `Matrix<Real>`, `Matrix<Complex>` and mixed types are also provided.

`DualDenseStorage` and `SymDenseStorage` classes provides also the size of strict lower and strict upper triangular part:

```

Number lowerPartSize() const;
Number upperPartSize() const;

```

XXX=Row, Col, Dual or Sym

library :	largeMatrix
header :	XXXDenseStorage.hpp
implementation :	XXXDenseStorage.cpp
unitary tests :	test_LargeMatrixDenseStorage.cpp
header dependences :	DenseStorage.hpp, MatrixStorage.hpp, config.h, utils.h

9.4 Compressed sparse storages

Compressed sparse storages are designed to store only non zero values of a matrix. It exists several methods to do it. The well known CSR/CSC storage being the most common one, we chose it. It consists in storing non zero values of a row (resp. column) in a compressed vector and their column indices (resp. row indices) in a compressed vector. A vector of addresses where begin rows (resp. columns) is also required. More precisely, for a $m \times n$ matrix

- row compressed storage is defined by the three vectors :

values = $(0, A_{1C_1}, \dots, A_{iC_i}, \dots, A_{mC_m})$

colIndex = $(C_1, \dots, C_i, \dots, C_m)$

rowPointer = $(p_1, \dots, p_i, \dots, p_m, p_{m+1})$

where C_i is the set of index j where $A_{ij} \neq 0$ and p_i is the position in vector colIndex of the first entry of row $i \leq m$; p_{m+1} gives the number of non zero values. By convention, all the indices begins at 0 and first entry of values vector is always 0!

For instance, the following 5×6 non symmetric matrix

$$A = \begin{bmatrix} 11 & 12 & 0 & 14 & 0 & 16 \\ 0 & 22 & 23 & 0 & 0 & 26 \\ 31 & 32 & 33 & 0 & 0 & 0 \\ 0 & 0 & 43 & 44 & 0 & 0 \\ 51 & 0 & 53 & 54 & 55 & 0 \end{bmatrix}$$

has the row compressed storage vectors:

values = (0 11 12 14 16 22 23 26 31 32 33 43 44 51 53 54 55)

colIndex = (0 1 3 5 1 2 5 0 1 2 2 3 0 2 3 4)

rowPointer = (0 4 7 10 12 16)

- row compressed storage is defined by the three vectors :

$$\begin{aligned} \text{values} &= (0, A_{\mathcal{R}_1 1}, \dots, A_{\mathcal{R}_j j}, \dots, A_{\mathcal{R}_n n}) \\ \text{rowIndex} &= (\mathcal{R}_1, \dots, \mathcal{R}_j, \dots, \mathcal{R}_n) \\ \text{colPointer} &= (p_1, \dots, p_j, \dots, p_n, p_{n+1}) \end{aligned}$$

where \mathcal{R}_j is the set of index i where $A_{ij} \neq 0$ and p_j is the position in rowIndex vector of the first entry of column $j \leq n$; p_{n+1} gives the number of non zero values. By convention, all the indices begins at 0 and first entry of values vector is always 0!

For instance, the previous 5×6 non symmetric matrix A has the column compress storage vectors:

$$\begin{aligned} \text{values} &= (0 \ 11 \ 31 \ 51 \ 12 \ 22 \ 32 \ 23 \ 33 \ 43 \ 53 \ 55 \ 16 \ 26) \\ \text{rowIndex} &= (0 \ 2 \ 4 \ 0 \ 1 \ 2 \ 1 \ 2 \ 3 \ 4 \ 0 \ 3 \ 4 \ 4 \ 0 \ 1) \\ \text{colPointer} &= (0 \ 4 \ 7 \ 10 \ 12 \ 16) \end{aligned}$$

- dual compressed storage is defined by the five vectors :

$$\begin{aligned} \text{values} &= (0, \text{diag}(A), A_{\mathcal{C}_1 1}, \dots, A_{\mathcal{C}_i i}, \dots, A_{\mathcal{C}_m m}, A_{\mathcal{R}_1 1}, \dots, A_{\mathcal{R}_j j}, \dots, A_{\mathcal{R}_n n}) \\ \text{colIndex} &= (\mathcal{C}_1, \dots, \mathcal{C}_i, \dots, \mathcal{C}_m) \\ \text{rowPointer} &= (p_1, \dots, p_i, \dots, p_m, p_{m+1}) \\ \text{rowIndex} &= (\mathcal{R}_1, \dots, \mathcal{R}_j, \dots, \mathcal{R}_n) \\ \text{colPointer} &= (q_1, \dots, q_j, \dots, q_n, q_{n+1}) \end{aligned}$$

where \mathcal{C}_i is the set of index j where $A_{ij} \neq 0$ and $j \leq \min(i-1, n)$, \mathcal{R}_j is the set of index i where $A_{ij} \neq 0$ and $i \leq \min(j-1, m)$, p_i (resp. q_j) is the position in colIndex (resp. rowIndex) vector of the first entry of row $i \leq m$ (resp. column $j \leq n$). The colIndex and rowPointer vectors describes the row compressed storage of the strict lower part of matrix while the rowIndex and colPointer vectors describes the column compressed storage of the strict upper part of matrix. In this storage, all the diagonal entries of matrix are always first stored even the null entries!

For instance, the previous 5×6 non symmetric matrix A has the column compress storage vectors:

$$\begin{aligned} \text{values} &= (0 \ 11 \ 22 \ 33 \ 44 \ 55 \ 31 \ 32 \ 43 \ 51 \ 53 \ 54 \ 23 \ 14 \ 44 \ 16 \ 26) \\ \text{colIndex} &= (0 \ 1 \ 2 \ 0 \ 2 \ 3) \\ \text{rowPointer} &= (0 \ 0 \ 0 \ 2 \ 3 \ 6) \\ \text{rowIndex} &= (0 \ 1 \ 0 \ 0 \ 1) \\ \text{colPointer} &= (0 \ 0 \ 1 \ 2 \ 2 \ 3 \ 5) \end{aligned}$$

Note that for an empty row i (resp. column j), $\text{rowPointer}(i) = \text{rowPointer}(i-1)$ (resp. $\text{colPointer}(j) = \text{colPointer}(j-1)$). By the way, $\text{rowPointer}(i) - \text{rowPointer}(i-1)$ gives the number of stored entries on row $i-1$.

- symmetric compressed storage is devoted to square matrix with symmetric storage, the matrix may be not symmetric.. In that case, only the storage of lower triangular part of matrix is described by the three vectors (those of row part of dual storage):

$$\begin{aligned} \text{values} &= (0, \text{diag}(A), A_{\mathcal{C}_1 1}, \dots, A_{\mathcal{C}_i i}, \dots, A_{\mathcal{C}_m m}) \text{ if } A \text{ has a symmetry property} \\ \text{values} &= (0, \text{diag}(A), A_{\mathcal{C}_1 1}, \dots, A_{\mathcal{C}_i i}, \dots, A_{\mathcal{C}_m m}, A_{\mathcal{C}_1 1}, \dots, A_{\mathcal{C}_j j}, \dots, A_{\mathcal{C}_m m}) \text{ if } A \text{ has no symmetry property} \\ \text{colIndex} &= (\mathcal{C}_1, \dots, \mathcal{C}_i, \dots, \mathcal{C}_m) \\ \text{rowPointer} &= (p_1, \dots, p_i, \dots, p_m, p_{m+1}) \end{aligned}$$

where the meaning of \mathcal{C}_i and p_i is the same as dual storage.

For instance, the 5×5 non symmetric matrix with symmetric storage

$$B = \begin{bmatrix} 11 & 0 & 13 & 0 & 15 \\ 0 & 22 & 23 & 0 & 0 \\ 31 & 32 & 33 & 34 & 35 \\ 0 & 0 & 43 & 44 & 45 \\ 51 & 0 & 53 & 54 & 55 \end{bmatrix}$$

has the symmetric compressed storage vectors:

$$\text{values} = (0 \ 11 \ 22 \ 33 \ 44 \ 55 \ 31 \ 32 \ 43 \ 51 \ 53 \ 54 \ 13 \ 23 \ 34 \ 15 \ 35 \ 45)$$

```
colIndex = (0 1 2 0 2 3)
rowPointer = (0 0 0 2 3 6)
```



Be care with the fact that actual matrix values begins at adress 1 in values vector defined outside the storage stuff.

9.4.1 The CsStorage class

The `CsStorage` class has no member attribute. There are some basic constructors just calling `MatrixStorage` constructors:

```
class CsStorage : public MatrixStorage
{public :
    CsStorage(AccessType = _dual);
    CsStorage(Number, AccessType = _dual);
    CsStorage(Number, Number, AccessType = _dual);
    virtual ~CsStorage() {}
protected :
    void buildCsStorage(const std::vector<std::vector<Number> >&,
                       std::vector<Number>&, std::vector<Number>&);
```

The `buildCsStorage` is a general member function building storage vectors from lists of column indices by row. It is used by any child class of `CsStorage` class. `CsStorage` objects do not have to be instantiated. This class acts as an interface to particular compressed storages and gathers all common functionalities of dense storages, some being virtuals (some template line declaration are omitted):

```
virtual Number size() const = 0;
virtual Number lowerPartSize() const;
virtual Number upperPartSize() const;
template<typename Iterator>
    void printEntriesAll(StrucType, Iterator&, const std::vector<Number>&,
                       const std::vector<Number>&, Number, Number, Number,
                       const String&, Number, std::ostream&) const;
    void printEntriesTriangularPart(StrucType, Iterator&, Iterator&,
                                   const std::vector<Number>&, const std::vector<Number>&,
                                   Number, Number, Number, const String&, Number,
                                   std::ostream&) const;
    void printCooTriangularPart(std::ostream&, Iterator&, const std::vector<Number>&,
                               const std::vector<Number>&, bool,
                               SymType sym = _noSymmetry) const;
    void printCooDiagonalPart(std::ostream&, Iterator&, Number) const;

template <typename T>
    void loadCsFromFileDense(std::istream&, std::vector<T>&, std::vector<Number>&,
                           std::vector<Number>&, SymType, bool);
    void loadCsFromFileCoo(std::istream&, std::vector<T>&, std::vector<Number>&,
                           std::vector<Number>&, SymType, bool);
    void loadCsFromFileDense(std::istream&, std::vector<T>&, std::vector<Number>&,
                           std::vector<Number>&, std::vector<Number>&,
                           std::vector<Number>&, SymType, bool);
    void loadCsFromFileCoo(std::istream&, std::vector<T>&, std::vector<Number>&,
                           std::vector<Number>&, std::vector<Number>&,
                           std::vector<Number>&, SymType, bool);
```

The `CsStorage` class provides the real code of product of matrix and vector, using iterators as arguments to by-pass the template type of matrix values. For dual and symmetric storage, the computation is performed by splitting the matrix in its diagonal, lower and upper part (template line are omitted):


```

template<typename MatIterator, typename VecIterator, typename ResIterator>
void diagonalMatrixVector(MatIterator&, VecIterator&, ResIterator&,
    ResIterator&) const;
void lowerMatrixVector(const std::vector<Number>&, const std::vector<Number>&,
    MatIterator&, VecIterator&, ResIterator&, SymType sym)
    const;
void upperMatrixVector(const std::vector<Number>&, const std::vector<Number>&,
    MatIterator&, VecIterator&, ResIterator&, SymType) const;
void rowMatrixVector(const std::vector<Number>&, const std::vector<Number>&,
    MatIterator&, VecIterator&, ResIterator&) const;
void columnMatrixVector(const std::vector<Number>&, const std::vector<Number>&,
    MatIterator&, VecIterator&, ResIterator&) const;
void diagonalVectorMatrix(MatIterator&, VecIterator&, ResIterator&,
    ResIterator&) const;
void lowerVectorMatrix(const std::vector<Number>&, const std::vector<Number>&,
    MatIterator&, VecIterator&, ResIterator&, SymType sym)
    const;
void upperVectorMatrix(const std::vector<Number>&, const std::vector<Number>&,
    MatIterator&, VecIterator&, ResIterator&, SymType) const;
void rowVectorMatrix(const std::vector<Number>&, const std::vector<Number>&,
    MatIterator&, VecIterator&, ResIterator&) const;
void columnVectorMatrix(const std::vector<Number>&, const std::vector<Number>&,
    MatIterator&, VecIterator&, ResIterator&) const;

```

Besides the product of matrix and vector, the **CsStorage** provides solvers for matrix in its diagonal, lower and upper part

```

template<typename MatIterator, typename VecIterator, typename ResIterator>
void bzSorDiagonalMatrixVector(MatIterator& itd, const VecIterator& itvb,
    ResIterator& itrb, const Real w) const;

template<typename MatIterator, typename VecIterator, typename XIterator>
void bzSorLowerSolver(const MatIterator&, const MatIterator&,
    VecIterator&, XIterator&, XIterator&,
    const std::vector<Number>&, const std::vector<Number>&,
    const Real) const;

template<typename MatIterator, typename VecIterator, typename XIterator>
void bzSorDiagonalSolver(const MatIterator& itdb, VecIterator& itbb,
    XIterator& itxb, XIterator& itxe, const Real w) const;

template<typename MatRevIterator, typename VecRevIterator, typename
    XRevIterator>
void bzSorUpperSolver(const MatRevIterator&, const MatRevIterator&,
    VecRevIterator&, XRevIterator&, XRevIterator&,
    const std::vector<Number>&, const std::vector<Number>&,
    const Real) const;

```

Addition of two matrices are implemented by

```

template<typename Mat1Iterator, typename Mat2Iterator, typename ResIterator>
void sumMatrixMatrix(Mat1Iterator&, Mat2Iterator&, ResIterator&, ResIterator&)
    const;

```

library :	largeMatrix
header :	CsStorage.hpp
implementation :	CsStorage.cpp
unitary tests :	test_LargeMatrixCstorage.cpp
header dependences :	MatrixStorage.hpp, config.h, utils.h

9.4.2 RowCsStorage and ColCsStorage classes

As mentioned before, row compressed storage are based on the column indices vector and the row pointers vector. So the `RowCsStorage` class carries this two vectors:

```
class RowCsStorage : public CsStorage
{protected :
    std::vector<Number> colIndex_; //vector of column index of non zero value
    std::vector<Number> rowPointer_; //vector of positions of begining of rows
    ...
```

The class provides a default constructor with no construction of storage vectors and constructors from a list of (i, j) index (coordinates) or a lists of column indices by row, constructing the compressed storage vectors:

```
RowCsStorage(Number nr=0, Number nc=0);
RowCsStorage(Number, Number, const std::vector< std::vector<Number> >&,
              const std::vector< std::vector<Number> >&);
RowCsStorage(Number, Number, const std::vector< std::vector<Number> >&);
~RowCsStorage() {};
```

The class provides most of the virtual methods declared in `MatrixStorage` class. For sake of simplicity, we report here only function with `Real` type but versions with `Complex`, `Matrix<Real>`, `Matrix<Complex>` and mixed types are also provided. Some template line declarations are also omitted.

```
Number size() const;
Number pos(Number i, Number j, SymType s=_noSymmetry) const;
void positions(const std::vector<Number>&, const std::vector<Number>&,
              std::vector<Number>&, bool errorOn=true,
              SymType s=_noSymmetry) const;
void printEntries(std::ostream&, const std::vector<Real>&, Number, SymType) const;
void printCooMatrix(std::ostream&, const std::vector<Real>&,
                   SymType s=_noSymmetry) const;
void loadFromFileDense(std::istream&, std::vector<Real>&, SymType, bool );
void loadFromFileCoo(std::istream&, std::vector<Real>&, SymType, bool);
template<typename M, typename V, typename R>
void multMatrixVector(const std::vector<M>&, const std::vector<V>&,
                     std::vector<R>&) const;
void multVectorMatrix(const std::vector<M>&, const std::vector<V>&,
                     std::vector<R>&) const;
void multMatrixVector(const std::vector<Real>&, const std::vector<Real>&,
                     std::vector<Real>&, SymType) const;
void multVectorMatrix(const std::vector<Real>&, const std::vector<Real>&,
                     std::vector<Real>&, SymType) const;
template<typename M1, typename M2, typename R>
void addMatrixMatrix(const std::vector<M1>&, const std::vector<M2>&,
                    std::vector<R>&) const; //!< templated row dense Matrix + Matrix
void addMatrixMatrix(const std::vector<Real>& m, const std::vector<Real>& v,
                    std::vector<Real>& rv) const
{ addMatrixMatrix<>(m, v, rv);}
```

There is no `printDenseMatrix` function because the general one defined in `MatrixStorage` class is sufficiently efficient.

The `ColCsStorage` class is very similar to the `RowCsStorage` class, except that it manages a row indices vector and a column pointers vector:

```
class ColCsStorage : public CsStorage
{protected :
    std::vector<Number> rowIndex_; //!< vector of row index of non zero value
```

```
std::vector<Number> colPointer_; ///  
...  
vector of positions of begining of cols
```

Its constructors have a similar structure (same arguments) and all prototypes of member functions are the same as `ColCsStorage` member functions. We do not repeat them. XXX=Row or Col

```
library : largeMatrix
header : XXXCsStorage.hpp
implementation : XXXCsStorage.cpp
unitary tests : test_LargeMatrixCsStorage.cpp
header dependences : CsStorage.hpp, MatrixStorage.hpp, config.h, utils.h
```

9.4.3 DualCsStorage and SymCsStorage classes

Dual compressed storage travels the matrix as diagonal, strict lower and upper parts. Column indices and row pointers vectors are attached to strict lower part and row indices and column pointers vectors are attached to strict upper part:

```
class DualCsStorage : public CsStorage
{protected :
    std::vector<Number> colIndex_; //vector of column index of non zero value
    std::vector<Number> rowPointer_; //vector of positions of begining of rows
    std::vector<Number> rowIndex_; //vector of row index of non zero value
    std::vector<Number> colPointer_; //vector of positions of begining of cols
    ...
```

`DualCsStorage` class provides basic constructor (no construction of storage vectors), constructor from a pair of global numbering vectors or list of column indices by rows. Both of them create the compressed storage vector using the auxiliary function `buildStorage`:

```
DualCsStorage(Number nr=0, Number nc=0);
DualCsStorage(Number, Number, const std::vector< std::vector<Number> >&,
               const std::vector< std::vector<Number> >&);
DualCsStorage(Number, Number, const std::vector<std::vector<Number> >&);
void buildStorage(const std::vector<std::vector<Number> >&);
~DualCsStorage() {};
```

The class provides most of the virtual methods declared in `MatrixStorage` class. For sake of simplicity, we report here only function with `Real` type but versions with `Complex`, `Matrix<Real>`, `Matrix<Complex>` and mixed types are also provided. Some template line declarations are also omitted.

```
Number size() const;
Number lowerPartSize() const ;
    Number upperPartSize() const ;
Number pos(Number i, Number j, SymType s=_noSymmetry) const;
void positions(const std::vector<Number>&, const std::vector<Number>&,
               std::vector<Number>&, bool errorOn=true,
               SymType s=_noSymmetry) const;
void printEntries(std::ostream&, const std::vector<Real>&, Number, SymType) const;
void printCooMatrix(std::ostream&, const std::vector<Real>&,
                    SymType s=_noSymmetry) const;
void loadFromFileDense(std::istream&, std::vector<Real>&, SymType, bool );
void loadFromFileCoo(std::istream&, std::vector<Real>&, SymType, bool);
template<typename M, typename V, typename R>
    void multMatrixVector(const std::vector<M>&, const std::vector<V>&,
                          std::vector<R>&) const;
```

```

void multVectorMatrix(const std::vector<M>&, const std::vector<V>&,
                     std::vector<R>&) const;
void multMatrixVector(const std::vector<Real>&, const std::vector<Real>&,
                     std::vector<Real>&, SymType) const;
void multVectorMatrix(const std::vector<Real>&, const std::vector<Real>&,
                     std::vector<Real>&, SymType) const;
template<typename M1, typename M2, typename R>
void addMatrixMatrix(const std::vector<M1>&, const std::vector<M2>&,
                    std::vector<R>&) const;
void addMatrixMatrix(const std::vector<Real>& m, const std::vector<Real>& v,
                    std::vector<Real>& rv) const
{ addMatrixMatrix<>(m, v, rv); }

```

The `SymCsStorage` class is very similar to the `DualCsStorage` class, except that it addresses only square matrix with symmetric storage and, thus, does not manage row indices vector and column pointers vectors because the upper part has same storage as lower part (transposed). Note that matrix may be have no symmetry property but a symmetric storage. In that case, the upper part is stored.

```

class SymCsStorage : public CsStorage
{protected :
    std::vector<Number> colIndex_;    ///  
vector of row index of non zero value
    std::vector<Number> rowPointer_; ///  
vector of positions of begining of cols
    ...

```

Contrary to the row compressed storage, in the `SymCsStorage`, the diagonal is first stored then the lower part of the matrix.

Its constructors have a similar structure (same arguments) to the `DualCsStorage` constructors and all prototypes of member functions are the same as `DualCsStorage` member functions. We do not repeat them.

Different from other `CsStorage` classes, the `SymCsStorage` class provides some functions to deal with the `SorSolver` class and `SsorSolver` class. Only specialized functions with `Real` are listed in the following

```

void sorDiagonalMatrixVector(const std::vector<Real>& m, const std::vector<Real>&
                             v, std::vector<Real>& rv, const Real w) const
{ sorDiagonalMatrixVector<>(m, v, rv, w); }
void sorLowerMatrixVector(const std::vector<Real>& m, const std::vector<Real>& v,
                          std::vector<Real>& rv, const Real w, const SymType sym) const
{ sorLowerMatrixVector<>(m, v, rv, w, sym); }
void sorUpperMatrixVector(const std::vector<Real>& m, const std::vector<Real>& v,
                          std::vector<Real>& rv, const Real w, const SymType sym) const
{ sorUpperMatrixVector<>(m, v, rv, w, sym); }
void sorDiagonalSolve(const std::vector<Real>& m, const std::vector<Real>& b,
                     std::vector<Real>& x, const Real w) const
{ sorDiagonalSolve<>(m, b, x, w); }
void sorLowerSolve(const std::vector<Real>& m, const std::vector<Real>& b,
                  std::vector<Real>& x, const Real w) const
{ sorLowerSolve<>(m, b, x, w); }
void sorUpperSolve(const std::vector<Real>& m, const std::vector<Real>& b,
                  std::vector<Real>& x, const Real w, const SymType sym) const
{ sorUpperSolve<>(m, b, x, w, sym); }

```

XXX=Dual or Sym

library :	<code>largeMatrix</code>
header :	<code>XXXCsStorage.hpp</code>
implementation :	<code>XXXCsStorage.cpp</code>
unitary tests :	<code>test_LargeMatrixCsStorage.cpp</code>
header dependences :	<code>CsStorage.hpp</code> , <code>MatrixStorage.hpp</code> , <code>config.h</code> , <code>utils.h</code>

9.5 Skyline storages

With dense storages, skyline storages are the only storages that are compatibles with factorisation methods like LU. Skyline storages consist in storage rows or columns from its first non zero value up to its last non zero value. When lower triangular part is addressed, the last non zero value on a row is the diagonal matrix entry. When upper triangular part is addressed, the last non zero value on a column is the diagonal matrix entry. Only dual and symmetric skyline storages are proposed here (resp. `DualSkylineStorage` and `SymSkylineStorage` classes).

- Dual skyline stores first the diagonal of the matrix, then the strict lower "triangular" part and the strict upper "triangular" part of the matrix using the following storage vectors (A is a $m \times n$ matrix):

values=(0, $A_{11}, \dots, A_{ii}, \dots, A_{il}, \dots, A_{i,p_i}, \dots, A_{i,l_i}, \dots, A_{q_j,j}, \dots, A_{q_j,c_j}$)

rowPointer=($s_1, \dots, s_i, \dots, s_m, s_{m+1}$)

colPointer=($t_1, \dots, t_j, \dots, t_n, t_{n+1}$)

where

$$l_i = \min(i-1, n-1), \quad c_j = \min(j-1, m-1)$$

$$p_i = \min 0 < j < l_i, \quad A_{ij} \neq 0 \text{ and } q_j = \min 0 < i < c_j, \quad A_{ij} \neq 0$$

$$s_1 = 0; \quad s_i = s_{i-1} + i - s_{i-1} \quad \forall i = 2, m, \text{ and } t_1 = 0; \quad t_j = s_{j-1} + j - t_{j-1} \quad \forall j = 2, n$$

The rowPointer (resp. colPointer) vector gives the position in lower (resp. upper) part of the first non zero entry of a row (resp. a column); s_{m+1} (resp. t_{n+1}) gives the number of entries stored in lower part (resp. upper part).

For instance, the following 5×6 non symmetric matrix

$$A = \begin{bmatrix} 11 & 12 & 0 & 14 & 0 & 16 \\ 0 & 22 & 23 & 0 & 0 & 26 \\ 31 & 32 & 33 & 0 & 0 & 0 \\ 0 & 0 & 43 & 44 & 0 & 0 \\ 51 & 0 & 53 & 54 & 55 & 0 \end{bmatrix}$$

has the following storage vectors:

values = (0 11 22 33 44 45 22 31 32 43 51 0 53 54 12 23 14 0 0 16 26 0 0)

rowPointer = (0 0 0 2 3 7)

colPointer = (0 0 1 2 5 8)

The length of row i (resp. column j) is given by $s_{i+1} - s_i$ (resp. $t_{j+1} - t_j$). The position in values vector of entry (i, j) is given by the following relations ($1 \leq i \leq m, 1 \leq j \leq n$):

$$\text{if } i = j \quad \text{adr}(i, j) = i$$

$$\text{if } p_i \leq j \leq l_i \quad \text{adr}(i, j) = \min(m, n) + s_{i+1} + j - i$$

$$\text{if } q_j \leq i \leq c_j \quad \text{adr}(i, j) = \min(m, n) + s_{m+1} + t_{j+1} + i - j$$

9.5.1 The SkylineStorage class

The `SkylineStorage` class has no member attribute. It has some basic constructors just calling `MatrixStorage` constructors:

```
class SkylineStorage : public MatrixStorage
{
public :
    SkylineStorage(AccessType = _dual);
}
```

```

SkylineStorage(Number, AccessType = _dual);
SkylineStorage(Number, Number, AccessType = _dual);
virtual ~SkylineStorage() {}

```

`SkylineStorage` objects do not have to be instantiated. This abstract class acts as an interface to particular skyline storages and gathers all common functionalities of skyline storages, some being virtuals (some template line declarations are omitted):

```

virtual Number size() const = 0;
virtual Number lowerPartSize() const = 0;
virtual Number upperPartSize() const {return 0;}
template<typename Iterator>
void printEntriesTriangularPart (StrucType, Iterator&, Iterator&, const
    std::vector<Number>&, Number, Number, Number, const String&, Number,
    std::ostream&) const;
void printCooTriangularPart(std::ostream&, Iterator&, const std::vector<Number>&,
    Number, Number, bool, SymType sym = _noSymmetry) const;
void printCooDiagonalPart(std::ostream&, Iterator&, Number) const;

template <typename T>
void loadSkylineFromFileDense(std::istream&, std::vector<T>&,
    std::vector<Number>&, std::vector<Number>&, SymType, bool);
void loadSkylineFromFileCoo(std::istream&, std::vector<T>&, std::vector<Number>&,
    std::vector<Number>&, SymType, bool);

template<typename MatIterator, typename VecIterator, typename ResIterator>
void diagonalMatrixVector(MatIterator&, VecIterator&, ResIterator&, ResIterator&)
    const;
void lowerMatrixVector(const std::vector<Number>&, MatIterator&, VecIterator&,
    ResIterator&, SymType sym) const;
void upperMatrixVector(const std::vector<Number>&, MatIterator&, VecIterator&,
    ResIterator&, SymType) const;
void diagonalVectorMatrix(MatIterator&, VecIterator&, ResIterator&, ResIterator&)
    const;
void lowerVectorMatrix(const std::vector<Number>&, MatIterator&, VecIterator&,
    ResIterator&, SymType sym) const;
void upperVectorMatrix(const std::vector<Number>&, MatIterator&, VecIterator&,
    ResIterator&, SymType) const;
void sumMatrixMatrix(Mat1Iterator&, Mat2Iterator&, ResIterator&, ResIterator&)
    const;

```

The `SkylineStorage` class provides the real code of product of matrix and vector, using iterators as arguments to by-pass the template type of matrix values.

In order to deal with some specific solvers, this class also comes with several following functions. For simplicity, we only declare template line once.

```

template<typename MatIterator, typename VecIterator, typename XIterator>
void bzLowerSolver(const MatIterator&, const MatIterator&, const VecIterator&,
    const XIterator&, const XIterator&, const std::vector<size_t>::const_iterator)
    const;
void bzLowerD1Solver(const MatIterator&, const VecIterator&, const XIterator&,
    const XIterator&, const std::vector<size_t>::const_iterator) const;
void bzLowerConjD1Solver(const MatIterator&, const VecIterator&, const XIterator&,
    const XIterator&, const std::vector<size_t>::const_iterator) const;
void bzDiagonalSolver(const MatIterator&, const VecIterator&, const XIterator&,
    const XIterator&) const;
void bzUpperSolver(const MatRevIterator&, const MatRevIterator&, const
    VecRevIterator&, const XRevIterator&, const XRevIterator&, const
    std::vector<size_t>::const_reverse_iterator) const;
void bzUpperD1Solver(const MatRevIterator&, const VecRevIterator&, const
    XRevIterator&, const XRevIterator&, const
    std::vector<size_t>::const_reverse_iterator) const;

```

```
void bzUpperConjD1Solver(const MatRevIterator&, const VecRevIterator&, const
XRevIterator&, const XRevIterator&, const
std::vector<size_t>::const_reverse_iterator) const;
```

Like the product of matrix and vector, these functions take advantage of iterators as arguments to by-pass the template type of matrix values.

library :	largeMatrix
header :	SkylineStorage.hpp
implementation :	SkylineStorage.cpp
unitary tests :	test_LargeMatrixSkylinestorage.cpp
header dependences :	MatrixStorage.hpp, config.h, utils.h

9.5.2 DualSkylineStorage and SymSkylineStorage classes

Dual skyline storage travels the matrix as diagonal, strict lower and upper parts. Row pointers vectors are attached to strict lower part and column pointers vectors are attached to strict upper part:

```
class DualSkylineStorage : public SkylineStorage
{protected :
    std::vector<Number> rowPointer_; // vector of positions of begining of rows
    std::vector<Number> colPointer_; // vector of positions of begining of cols
    ...
```

DualSkylineStorage class provides basic constructor (no construction of storage vectors), constructor from a pair of global numbering vectors or list of column indices by rows. Both of them create the compressed storage vector using the auxiliary function **buildStorage**:

```
DualSkylineStorage(Number nr = 0, Number nc = 0);
DualSkylineStorage(Number, Number, const std::vector< std::vector<Number> >&,
                    const std::vector< std::vector<Number> >&);
DualSkylineStorage(Number, Number, const std::vector< std::vector<Number> >&);
~DualSkylineStorage() {};
```

The class provides most of the virtual methods declared in **MatrixStorage** class. For sake of simplicity, we report here only function with **Real** type but versions with **Complex**, **Matrix<Real>**, **Matrix<Complex>** and mixed types are also provided. Some template line declarations are also omitted.

```
Number size() const;
Number lowerPartSize() const;
Number upperPartSize() const;
Number pos(Number i, Number j, SymType s=_noSymmetry) const;
void positions(const std::vector<Number>&, const std::vector<Number>&,
               std::vector<Number>&, bool errorOn=true,
               SymType s=_noSymmetry) const;
void printEntries(std::ostream&, const std::vector<Real>&, Number, SymType) const;
void printCooMatrix(std::ostream&, const std::vector<Real>&,
                   SymType s=_noSymmetry) const;
void loadFromFileDense(std::istream&, std::vector<Real>&, SymType, bool);
void loadFromFileCoo(std::istream&, std::vector<Real>&, SymType, bool);
template<typename M, typename V, typename R>
    void multMatrixVector(const std::vector<M>&, const std::vector<V>&,
                          std::vector<R>&) const;
    void multVectorMatrix(const std::vector<M>&, const std::vector<V>&,
                          std::vector<R>&) const;
    void multMatrixVector(const std::vector<Real>&, const std::vector<Real>&,
                          std::vector<Real>&, SymType) const;
```



```

void multVectorMatrix(const std::vector<Real>&, const std::vector<Real>&,
                     std::vector<Real>&, SymType) const;
template<typename M1, typename M2, typename R>
void addMatrixMatrix(const std::vector<M1>&, const std::vector<M2>&,
                    std::vector<R>&) const;
template<typename T>
void setDiagValueDualSkyline(std::vector<T>&, const T);
template<typename M>
void lu(std::vector<M>& m, std::vector<M>& fa) const;
template<typename M, typename V, typename X>
void lowerD1Solver(const std::vector<M>&, std::vector<V>&, std::vector<X>&) const;
void diagonalSolver(const std::vector<M>&, std::vector<V>&, std::vector<X>&)
    const;
void upperD1Solver(const std::vector<M>&, std::vector<V>&, std::vector<X>&, const
SymType) const;
void upperSolver(const std::vector<M>&, std::vector<V>&, std::vector<X>&) const;

```

The `SymSkylineStorage` class is very similar to the `DualSkylineStorage` class, except that it addresses only square matrix with symmetric storage and, thus, does not manage column pointers vectors because the upper part has same storage as lower part (with transposition) :

```

class SymSkylineStorage : public SkylineStorage
{protected :
    std::vector<Number> rowPointer_; ///  
...


```

Note that the matrix may be not symmetric. In that case its upper part is stored. Besides some similar functions to `DualSkylineStorage` class', the `SymSkylineStorage` provides more several functions to factorize matrix: LDLt and LDL* factorization

```

template<typename M>
void ldlt(std::vector<M>& m, std::vector<M>& fa, const SymType sym =
    _symmetric) const;
void ldltstar(std::vector<M>& m, std::vector<M>& fa) const;

```

XXX=Dual or Sym

library :	largeMatrix
header :	XXSkylineStorage.hpp
implementation :	XXSkylineStorage.cpp
unitary tests :	test_LargeMatrixSkylineStorage.cpp
header dependences :	SkylineStorage.hpp, MatrixStorage.hpp, config.h, utils.h

The *hierarchicalMatrix* library deals with large matrices that are represented by a hierarchical tree, each tree node being either a real submatrix (leaf) or a virtual submatrix addressing up to four nodes :

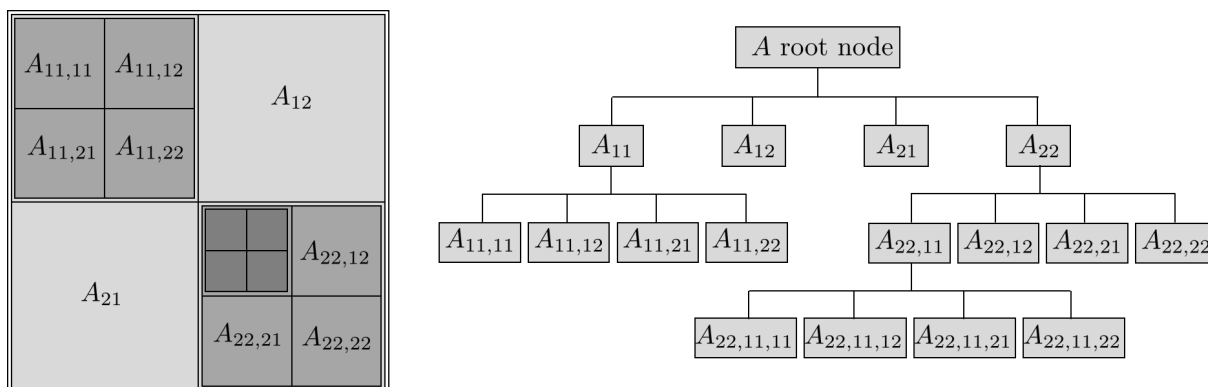


Figure 10.1: Hierarchical matrix

The matrix may be not squared and therefore the submatrices too.

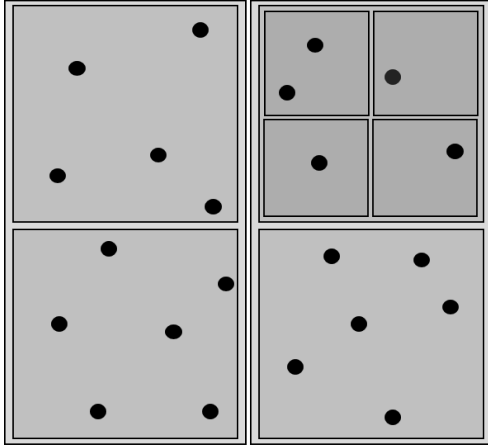
Such representation comes from mesh partition (clustering) where far interactions in BEM may be approximated by low rank matrices, so reducing the storage of matrix and making the matrix vector product faster. The *hierarchicalMatrix* library proposes

- some templated classes to deal with clusters of Points, FeDofs, Elements, ... : `ClusterTree<I>` and `ClusterNode<I>`
- some templated classes to deal with approximated matrices : `ApproximateMatrix<T>` and `LowRankMatrix<T>`
- some templated classes addressing hierarchical matrices : `HMatrix<T,I>`, `HMatrixNode<T,I>` and `HMatrixEntry<I>`

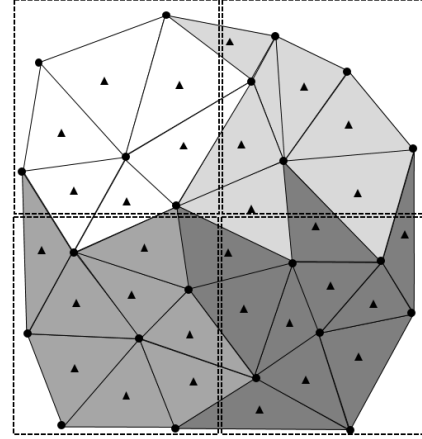
10.1 Cluster tree

The clustering of a set of objects consists in a partition of this set with a hierarchical structure. Each node of the cluster being a subset of the parent node subset. The partitioning of a subset is done according to some rules depending of the purpose of the cluster.

In the case of a mesh, we are interested in clusters of points (mesh nodes) or in clusters of elements where the partitioning rule aims to separate geometrically the points or the elements.



Cluster of points



Clustering of triangles using centroids

XLiFE++ provides two classes templated by the type T of objects to be clustered :

- `ClusterNode<T>` managing one node of the tree representation
- `ClusterTree<T>` managing the tree representation from the root node

10.1.1 The ClusterNode class

The `ClusterNode<T>` class describes one node of the tree representing the cluster where T is the type of objects to be clustered. It is assumed that T is a geometric object and that the three following functions are available:

```
dimen_t dim(const T&) const;           //dimension of T object
real_t coords(const T&, number_t i) const //i-th coordinates of T object
BoundingBox boundingBox(const T &)      //xy(z) bounding box of T object
```

For instance, these functions are provided for the XLiFE++ classes : `Point`, `FeDof`, `Element` allowing to cluster mesh using either some points of mesh (vertices), some finite element dofs or some finite elements.

The class manages standard informations:

```
vector<T>* objects_;           //pointer to the list of all objects
vector<number_t> numbers_;     //object numbers related to objects_ vector
ClusterNode* parent_;         //pointer to its parent (0 for root node)
ClusterNode* child_;          //pointer to its first child (0 = no child)
ClusterNode* next_;           //pointer to its brother, (0 = no brother)
number_t depth_;              //depth of node/leaf in tree (0 for root node)
BoundingBox boundingBox_;      //bounding box from characteristic points
BoundingBox realBoundingBox_;  //bounding box from bounding boxes of objects
```

The list of objects belonging to the cluster node is given by the `numbers_` vector that stores the indices of objects in the vector `objects_`. The clusters we deal with are supposed to be geometrical clusters and the clustering methods are mainly based on splitting of boxes, it is the reason why the class deals explicitly with bounding boxes. In the context of finite element, some additional data (mutable) can be handled :

```
vector<number_t> dofNumbers_;   //list of dof numbers related to the node
vector<Element*> elements_;     //list of element numbers related to the node
```

The class provides only one constructor assigning fundamental data (`objects_`, `parent_`, `child_`, `next_`, `depth_`) and a `clear` method that deletes recursively the subtree from the current node:

```
ClusterNode( vector<T>*, ClusterNode*, number_t, ClusterNode*=0, ClusterNode*=0);
void clear();
```



The copy constructor and assignment operator are the default ones, so the tree structure is not duplicated but only shared. Be cautious, when you use copy or assignment.

The class provides some functions to get some properties or data:

```
number_t dimSpace() const;    //geometric dimension
number_t size() const;       //number of objects
bool isVoid() const;         //true if void node
vector<ClusterNode*> getLeaves() const; //list of leave pointers
list<number_t> getNumbers(bool) const; //list of Numbers (recursive)
const BoundingBox& box() const; //return the real bounding box if allocated
real_t boxDiameter() const;  //diameter of the (real) bounding box
```

The main member function that builds the tree structure from a division process is:

```
void divideNode( ClusteringMethod cm, number_t mininleaf,
                number_t maxdepth=0, bool store = true);
```

where

- **ClusteringMethod** is the enumeration of clustering methods:

```
enum ClusteringMethod { _regularBisection, _boundingBoxBisection,
                        _cardinalityBisection, _uniformKdtree, _nonuniformKdtree };
```

Except the **_uniformKdtree** and **_nonuniformKdtree** methods, each others are bisection method in the direction where the boxes is larger, so the tree is a binary one. **_regularBisection** split boxes in two boxes of the same size, **_boundingBoxBisection** split bounding boxes of objects in two boxes of the same size, **_cardinalityBisection** split boxes in two boxes such that each new boxes have the same number of objects. The trees produced by this last method are better balanced. The **_uniformKdtree** and **_nonuniformKdtree** methods split any boxes in 2, 4 or 8 sub-boxes regarding the space dimension (1, 2 or 3); the sub-boxes are of the same size in any direction. Thus it produces binary tree in 1D, quadtree in 2D and octree in 3D. With **_uniformKdtree** method, all the boxes at the same level have the same size whereas with **_nonuniformKdtree** method, all the boxes are the minimal bounding boxes defined from the objects that they embed.

- **maxinleaf** is the maximum of objects in a leaf node, more precisely, a node is divided when its number of objects is greater than **maxinleaf**
- **maxdepth**, if not null, allows to stop the division process when **maxdepth** is reached. This stopping criteria has priority over the previous criteria.
- if **store** parameter is true, data (**numbers_**, bounding boxes, ...) are stored for each node else they are only stored for each leaves. By default its value is true, but if cluster is too large, not storing some data may save memory but some algorithms working on clusters could be slower.



Up to now, only three clustering methods are offered, but it is quite easy to add a new one by handling a new case in **divideNode** function. Bisection methods build binary tree but there is not limitation of the number of children in the **ClusterNode** class!

When some data (**numbers_**, bounding boxes, ...) are not available it is possible to build or update it:

```

void setBoundingBox();           //set the bounding box
void setRealBoundingBox();       //set the real bounding box
void updateRealBoundingBoxes();  //update recursively real bounding boxes
void updateNumbers();            //update numbers_ vector

```

When dealing with finite elements some additional data may be useful, in particular the link between dofs and elements. To avoid costly re-computation, it may be clever to store it. The following member functions do the job on demand when working with a `ClusterNode<FeDof>` or a `ClusterNode<Element>`:

```

const vector<number_t>& dofNumbers() const; //access to dofNumbers
const vector<Element*>& elements() const;   //access to elt pointers
vector<number_t> getDofNumbers(bool=false) const; //list of dof numbers
vector<Element*> getElements(bool=false) const; //list of elt numbers
void updateDofNumbers();                   //update recursively dofNumbers
void updateElements();                     //update recursively elements
number_t nbDofs() const;                   //number of dofs

```

Finally, there are some stuff to print and save to file node informations:

```

void printNode(std::ostream&, bool shift=true) const; //print node
void print(std::ostream&) const;                     //print tree from current node (recursive)
void printLeaves(std::ostream&) const;               //print leaves of the tree
void saveToFile(const string_t&, bool) const;        //save bounding boxes to file
friend ostream& operator<<(std::ostream&, const ClusterNode<T>&);

```

10.1.2 The ClusterTree class

The `ClusterTree<T>` class is the front end of `ClusterNode<T>` class which supports the tree structure of the cluster. Because the `ClusterTree<T>` class interfaces most data and functionalities of the `ClusterNode<T>` class, for explanation go to the previous section.

The `ClusterTree<T>` class manages the parameters and informations required or provided by the the `ClusterTree<T>` class. It handles the tree structure (the cluster) through the root node pointer:

```

template <typename T = FeDof> class ClusterTree
{
public:
    vector<T>* objects_;           //pointer to a list of objects
    ClusteringMethod method_;      //clustering method
    number_t maxInBox_;            //maximum number of objects in a leaf
    number_t depth_;              //depth of the tree
    bool storeNodeData_;          //store data on each node if true
    bool clearObjects_;           //if true deallocate vector objects_ when clear
    number_t nbNodes_;            //number of nodes (info)
    number_t nbLeaves_;           //number of leaves (info)
    bool withOverlap_;            //true if dof numbering of nodes overlap
    bool noEmptyBox_;             //do not keep empty boxes in tree
private:
    ClusterNode<T>* root_;        //root node of cluster
    ...

```

Note that the `ClusterTree` has `FeDof` object as template default argument. This is the standard cluster used by the `HMatrix` class presented in a next section.

This class provides a unique constructor, building the cluster from a list of geometric object, a clustering method, the maximal depth of the tree, the maximal number of objects in leaf node and a flag telling if empty leaf nod are removed from the tree :

```
ClusterTree(const vector<T>&, ClusteringMethod, number_t depth,
             number_t maxinleaf=0, bool store=true, bool noemptybox = true);
```



The copy constructor and assignment operator are the default ones, so the tree structure is not duplicated but only shared. Be cautious, when you use copy or assignment. In particular avoid destruction of **ClusterTree** object, except if you set the member data `clearObjects_` to `false` before destroy it.

Almost all the member functions are interface to their equivalent in **ClusterNode** class:

```
ClusterNode<T>* root();
void setOverlap();
void updateBoundingBoxes();           //update recursively bounding boxes
void updateRealBoundingBoxes();       //update recursively real bounding boxes
void updateNumbers();                 //update recursively Numbers_ vector
void updateDofNumbers();              //update recursively DofNumbers_ vector
void updateElements();               //update recursively Elements_ vector
void updateInfo();                   //update tree info (depth, ...)
number_t nbDofs() const;
void print(ostream&) const;           //print tree
void printLeaves(ostream&) const;    //print only leaves
void saveToFile(const string_t&,
                bool withRealBox= false) const; //save (real) bounding boxes to file
ostream& operator<<(ostream& out, const ClusterTree<T>&);
```

For instance, to generate a cluster of a disk mesh, write :

```
Disk disk(_center=Point(0.,0.),_radius=1.,_nnodes=16,_domain_name="Omega");
Mesh mesh(disk,_triangle,1,_gmsh);
Domain omg=mesh.domain("Omega");
Space V(omg,P0,"V",false);
ClusterTree<FeDof> clt(V.feSpace()->dofs,_regularBisection,10);
clt.saveToFile("cluster_disk.txt");
```

As the dofs of P0 approximation are virtually located at the centroids of triangles, clustering of **FeDof** is equivalent to the clustering of centroids. We show on the figure 10.2 the clustering of a disk mesh (1673 nodes) with maximum of 10 nodes by box, using regular method, boundingbox, cardinality bisection methods. Only boxes of leaves are displayed; different colors correspond to different node depths.

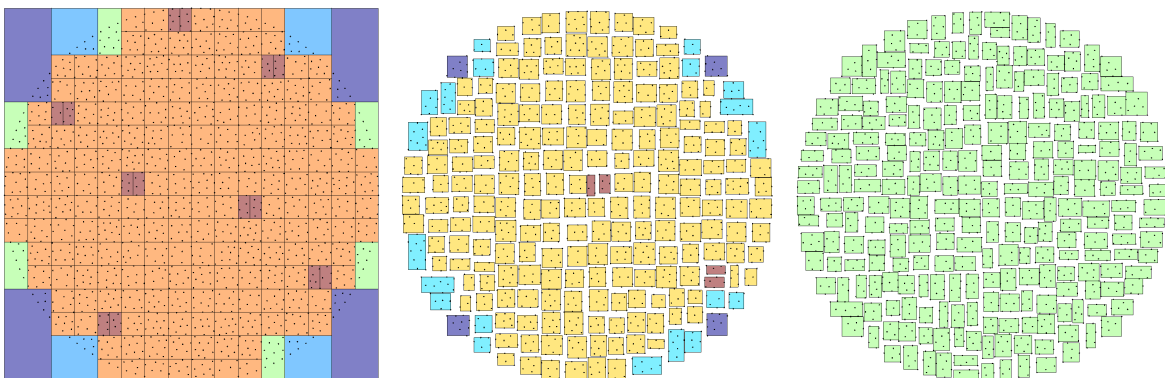


Figure 10.2: clustering of a disk mesh using regular, boundingbox and cardinality bisection methods

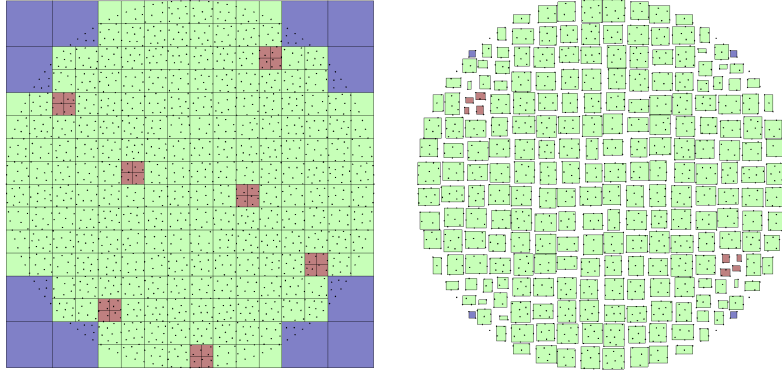


Figure 10.3: clustering of a disk mesh using `_uniformKdtree` and `_nonuniformKdtree` methods

library :	hierarchicalMatrix
header :	clusterTree.hpp
implementation :	clusterTree.cpp
unitary tests :	test_clusterNode.cpp
header dependencies :	space.h, config.h, utils.h

10.2 Approximate matrix

In the context of hierarchical matrix, some sub-matrices may be "compressed" to save memory and time computation. XLIFE++ provides a general class `ApproximateMatrix` to deal with real or complex approximate matrix that can be

- low rank matrix represented as the product UDV^* (`LowRankMatrix`)
- sinus cardinal decomposition (`SCDMatrix`), not yet available
- fast multipole representation (`FMMMatrix`), not yet available

10.2.1 The `ApproximateMatrix` class

The `ApproximateMatrix` class is a template abstract class (templated by the type of coefficients) that manages only the type of approximation, the name of the approximate matrix and defines all virtual functions that children has to implement:

```
template <typename T>
class ApproximateMatrix
{public :
MatrixApproximationType approximationType; //type of approximation
string_t name; //optional name useful for doc
virtual ~ApproximateMatrix() {}; //virtual destructor
virtual ApproximateMatrix<T>* clone() const=0; //creation of a clone
virtual number_t numberOfRows() const =0; //nb of rows
virtual number_t numberOfCols() const =0; //nb of columns
virtual number_t nbNonZero() const =0; //nb of non zeros coefficient
virtual number_t rank() const =0; //rank of matrix
virtual vector<T>& multMatrixVector(const vector<T>&, vector<T>&) const = 0;
virtual vector<T>& multVectorMatrix(const vector<T>&, vector<T>&) const = 0;
virtual real_t norm2() const =0; //Frobenius norm
virtual real_t norminf() const=0; //infinite norm
virtual LargeMatrix<T> toLargeMatrix() const = 0; //convert to LargeMatrix
virtual void print(std::ostream&) const =0; //print to stream
};
```

Up to now, the only children available is the templated `LowRankMatrix<T>` class.

10.2.2 The LowRankMatrix class

A low rank matrix is a matrix of the form

$$U D V^*$$

where U is a $m \times r$ matrix, V is a $n \times r$ matrix and D is a $r \times r$ diagonal matrix. The rank of a such matrix is at most r . So it is a low rank representation of a $m \times n$ matrix if r is small compared to m, n . Contrary to the class name suggests, some matrix may be not low rank representation. Think about the SVD (singular value decomposition) of a $m \times n$ matrix that can be handled by this class ($r = \min(m, n)$).

The `LowRankMatrix` class is derived from the `ApproximateMatrix` and handles the U, V matrices as `Matrix` objects of XLIFFE++ (dense matrix with major row access) and the diagonal matrix D as a `Vector` object of XLIFFE++:

```
template <typename T>
class LowRankMatrix : public ApproximateMatrix<T>
{public :
    Matrix<T> U_, V_;
    Vector<T> D_;
    ...
}
```

Different constructors are available:

```
LowRankMatrix(); //default constructor
//constructor from explicit matrices and vector
LowRankMatrix(const Matrix<T>&, const Matrix<T>&, const Vector<T>&,
               const string_t& na="");
LowRankMatrix(const Matrix<T>&, const Matrix<T>&, const string_t& na="");
//constructor from dimensions
LowRankMatrix(number_t m, number_t n, number_t r, const string_t& na="");
LowRankMatrix(number_t m, number_t n, number_t r, bool noDiag,
               const string_t& na="");
//constructor from matrix iterators and vector iterator
template <typename ITM, typename ITV>
LowRankMatrix(number_t m, number_t n, number_t r,
               ITM itmu, ITM itmv, ITV itd, const string_t& na="");
template <typename ITM>
LowRankMatrix(number_t m, number_t n, number_t r,
               ITM itmu, ITM itmv, const string_t& na="");
```

The `LowRankMatrix` class provides all virtual functions declared by the `ApproximateMatrix` class:

```
ApproximateMatrix<T>* clone() const;
number_t numberOfRows() const;
number_t numberOfCols() const;
number_t nbNonZero() const;
number_t rank() const;
bool hasDiag() const;
vector<T>& multMatrixVector(const vector<T>&, vector<T>&) const;
vector<T>& multVectorMatrix(const vector<T>&, vector<T>&) const;
LargeMatrix<T> toLargeMatrix() const;
real_t norm2() const;
real_t norminf() const;
void print(std::ostream&) const;
```

In addition, the class offers specific algebraic operations as member functions:


```

LowRankMatrix<T> svd(real_t eps) const;
LowRankMatrix<T>& operator+=(const LowRankMatrix<T>&);
LowRankMatrix<T>& operator-=(const LowRankMatrix<T>&);
LowRankMatrix<T>& operator*=(const T&);
LowRankMatrix<T>& operator/=(const T&);
LowRankMatrix<T>& add(const LowRankMatrix<T>&, const T&, real_t eps=0);

```

and as extern functions to the class (template declaration is omitted):

```

LowRankMatrix<T> combine(const LowRankMatrix<T>&, const T&,
                        const LowRankMatrix<T>&, const T&, real_t eps=0);
LowRankMatrix<T> operator+(const LowRankMatrix<T>&, const LowRankMatrix<T>&);
LowRankMatrix<T> operator-(const LowRankMatrix<T>&, const LowRankMatrix<T>&);
LowRankMatrix<T> operator-(const LowRankMatrix<T>&);
LowRankMatrix<T> operator+(const LowRankMatrix<T>&);
LowRankMatrix<T> operator*(const LowRankMatrix<T>&, const T&);
LowRankMatrix<T> operator/(const LowRankMatrix<T>&, const T&);
LowRankMatrix<T> operator*(const T&, const LowRankMatrix<T>&);

```

Note that the combination of a low rank matrix $L_1 = U_1 D_1 V_1$ of rank r_1 and a low rank matrix $L_2 = U_2 D_2 V_2$ of rank r_2 produces a low rank matrix $L = U D V$ of rank $r = r_1 + r_2$ because

$$L = a_1 L_1 + a_2 L_2 = \begin{bmatrix} U_1 & U_2 \end{bmatrix} \begin{bmatrix} a_1 D_1 & 0 \\ 0 & a_2 D_2 \end{bmatrix} \begin{bmatrix} V_1 \\ V_2 \end{bmatrix}.$$

When non zero, the `eps` argument in the `LowRankMatrix::add` and `combine` functions is used to re-compress the combined matrix with a truncated svd.

Finally, some compression methods based on SVD are implemented too. These compression methods "produce" a `LowRankMatrix` from a dense matrix stored in major column access and passed by a pointer to its first coefficient (template declaration is omitted):

```

void svdCompression(T* mat, number_t m, number_t n, number_t r,
                    LowRankMatrix<T>& lrm);
void svdCompression(T* mat, number_t m, number_t n, real_t eps,
                    LowRankMatrix<T>& lrm);
void rsvdCompression(T* mat, number_t m, number_t n, number_t r,
                     LowRankMatrix<T>& lrm);
void rsvdCompression(T* mat, number_t m, number_t n, real_t eps,
                     LowRankMatrix<T>& lrm);
void r3svdCompression(T* mat, number_t m, number_t n, real_t eps,
                      LowRankMatrix<T>& lrm,
                      number_t t = 0, number_t p = 0, number_t q = 0,
                      number_t maxit = 0)

```

The SVD compression functions do the full svd of the given matrix and then truncate the number of singular values and singular vectors keeping either the r largest singular values/vectors or the singular values greater than a given `eps`. The fact that this process leads to a good approximate matrix results from the Eckart–Young–Mirsky theorem.

As the full svd is an expansive algorithm time computation, some alternative methods based on the random svd are also available. Random svd consists in capturing the matrix range using only few gaussian random vectors and doing a svd on a smaller matrix. Again, two versions of random svd are available, one providing a low rank matrix of a given rank, the other one providing a low rank matrix with a control on the approximate error. This last method iterates on the rank, so it is more expansive. The `r3svd` is a more sophisticated iterative method (see <https://arxiv.org/ftp/arxiv/papers/1605/1605.08134.pdf>).



As svd and qr algorithms have not been implemented for the `Matrix` class, XLIfe++ uses the Eigen library that it is provided when installing XLIfe++.



svd, rsvd and aca compression does not work for matrix of matrices!

```

library : hierarchicalMatrix
header : ApproximateMatrix.hpp
implementation : ApproximateMatrix.cpp
unitary tests : test_HMatrix.cpp
header dependences : largeMatrix.h, config.h, utils.h

```

10.3 HMatrix

Hierarchical matrix is a tree representation of a matrix, each node representing a sub-matrix being either split in sub-matrices (tree nodes) or not if the node is a leaf of the tree. The hierarchical division in sub-matrices is based on hierarchical division of row and column matrix indices or dofs if it is a FE matrix (`ClusterTree` objects).

To manage this tree representation, three template classes are provided:

- the `HMatrixNode<T,I>` class dealing with a node of the matrix tree
- the `HMatrix<T,I>` front end class
- the `HMatrixEntry<I>` interface class that handles some pointers to `HMatrix<real_t,I>`, `HMatrix<complex_t,I>`,

These classes are templated by the type of matrix coefficients ($T=\text{real_t}$, complex_t , ...) and the type of row and column clustering ($I=\text{Point}$, FeDof , Element). Let us start by describing the `HMatrixNode` class supporting the main material.

10.3.1 The HMatrixNode class

```

template <typename T, typename I>
class HMatrixNode
{
public :
    HMatrixNode<T,I>* parent_;           //pointer to its parent, if 0 root node
    HMatrixNode<T,I>* child_;           //pointer to its first child, if 0 no child
    HMatrixNode<T,I>* next_;            //pointer to its brother, if 0 no brother
    ClusterNode<I>* rowNode_;            //row cluster node to access to row numbering
    ClusterNode<I>* colNode_;            //column cluster node to access to col numbering
    LargeMatrix<T>* mat_;                //pointer to a LargeMatrix
    ApproximateMatrix<T>* appmat_;       //pointer to an ApproximateMatrix
    bool admissible_;                    //block can be approximated
    number_t depth_;                     //depth of node in tree, root has 0 depth
    number_t rowSub_, colSub_;           //sub block numbering (i,j)
    bool isDiag_;                        //true if matrix node is located on the diagonal

```

Each node refers to its parent node, its first child node, its first brother node and access to its row and column numbering using some pointers to `ClusterNode` object. If node is a leaf (no child), it stores the node sub-matrix either as a `LargeMatrix` (pointer `mat_`) or as a `ApproximateMatrix` (pointer `appmat_`). If node is not a leaf, both the pointers `mat_` and `appmat_` are 0. Besides, `HMatrixNode` manages some additional useful informations that are set during construction of the tree such as `depth_`, `rowSub_`,

colSub_, isDiag_.

It manages also the particular data `admissible_` that tells from some geometrical rule if the sub-matrix attached to the node can be approximated, generally using compression methods. This property is mainly relevant in the context of BEM matrix where all interactions between distant dof sets can be reduced to few interactions. The choice of admissibility rule is governed by the `HMatrix` front end class using the enumerations:

```
enum HMatrixMethod {_standardHM, _denseHM};
enum HMAAdmissibilityRule {_boxesRule};
```

The `HMatrixNode` class has few constructors:

```
HMatrixNode(); //default constructor
HMatrixNode(HMatrixNode<T,I>*, HMatrixNode<T,I>*, HMatrixNode<T,I>*, number_t,
            ClusterNode<I>*, ClusterNode<I>*, number_t, number_t,
            LargeMatrix<T>*, ApproximateMatrix<T>* ); //full constructor
HMatrixNode(HMatrixNode<T,I>*, number_t); //node constructor
HMatrixNode(LargeMatrix<T>*, HMatrixNode<T,I>*,
            number_t); //LargeMatrix leaf constructor
HMatrixNode(ApproximateMatrix<T>*, HMatrixNode<T,I>*,
            number_t); //ApproximateMatrix leaf constructor
HMatrixNode(const HMatrixNode<T,I>&); //copy constructor
HMatrixNode<T,I>& operator=(const HMatrixNode<T,I>&); //assign operator
void copy(const HMatrixNode<T,I>&); //copy function
```

and some stuff related to destructor or cleaner:

```
~HMatrixNode(); //destructor
void clear(); //clear all except ClusterNode pointers
void clearMatrices(); //clear (deallocate) only matrices
```



Be cautious when copying `HMatrixNode` object, if allocated, the large matrix or the approximate matrix are copied, but not the `ClusterNode` objects (shared pointers!). In the same way, when a `HMatrixNode` object is cleared or deleted, matrix or approximate matrix are deleted if they have been allocated but the `ClusterNode` objects are not deallocated.

The most important function is

```
void divide(number_t rmin, number_t cmin, number_t maxdepth = 0,
            HMAAdmissibilityRule=_boxesRule, bool sym=false);
```

that creates recursively the sub-tree of the current node according to the admissibility rule given by the enumeration

```
enum HMAAdmissibilityRule {_boxesRule};
```

Up to now, only one rule is available, the standard boxes rule used in BEM computation telling that a `HMatrixNode` is admissible if the bounding box B_r of the row cluster node and the bounding box B_c of the column cluster node satisfy:

$$\text{diam}(B_r) \leq 2 * \eta * \text{dist}(B_r, B_c).$$



Note that in the case of a matrix having symmetry property (symmetric, skew-symmetric, self-adjoint or skew-adjoint), the division process may take it into account (argument `sym=true`). In that case, it does not generate sub-matrix nodes that are located above the diagonal, so saving memory.

Once the tree is built, some various member functions are available:

```

number_t numberOfRows() const;           //number of rows counted in T
number_t numberOfCols() const;          //number of columns counted in T
dimPair dimValues() const;              //dimensions of matrix values
void setClusterRow(ClusterNode<I>*);    //update row cluster node pointers
void setClusterCol(ClusterNode<I>*);    //update col cluster node pointers
number_t nbNonZero() const;              //number of T coefficients used
void getLeaves(list<HMatrixNode<T,I>* >&,
               bool = true) const;      //get leaves as a list

```

The `HMatrixNode` class implements all the stuff required by algebraic operations on `HMatrix` class:

```

vector<T>& multMatrixVectorNode(const vector<T>&, vector<T>&,
                               SymType symt=_noSymmetry) const;
vector<T>& multMatrixVector(const vector<T>&, vector<T>&,
                             SymType symt=_noSymmetry) const;
real_t norm2() const;                  //Frobenius norm
real_t norminf() const;                 //infinite norm

```

and by printing operations

```

void printNode(ostream&) const;         //print current node contents
void print(ostream&) const;             //print tree from current node
void printStructure(ostream&, number_t, number_t,
                   bool all=false, bool shift=false) const;

```

Finally, for performance measurement purpose, the class manages also some static data and static functions:

```

static bool counterOn;                  //flag to activate the operations counter
static number_t counter;                 //operations counter
static void initCounter(number_t n=0);   //init the counter and enable it
static void stopCounter();               //disable counter

```

10.3.2 The HMatrix class

The `HMatrix` class is the front end class of hierarchical matrices. It is templated by the type of matrix coefficients and the type of cluster objects. It manages the parameters of the matrix clustering, some general informations and handles the root node of the tree:

```

template <typename T, typename I>
class HMatrix
{private :
    HMatrixNode<T,I>* root_;             //root node of the tree representation
    ClusterTree<I>* rowCT_;              //row cluster tree
    ClusterTree<I>* colCT_;              //col cluster tree
public :
    string_t name;                       //optional name, useful for doc
    ValueType valueType_;                 //type of values (real, complex)
    StrucType strucType_;                 //structure of values (scalar, vector, matrix)
    HMatrixMethod method_;                 //method to define admissible block
    HMAdmissibilityRule admRule_;          //block admissible rule
    real_t eta_;                           //ratio in the admissibility criteria
    number_t rowmin_, colmin_;             //minimum size of block matrix
    SymType sym_;                           //type of symmetry
    number_t depth_;                       //maximal depth (info)
    number_t nbNodes;                      //number of nodes (info)
    number_t nbLeaves;                     //number of leaves (info)
    number_t nbAdmissibles;                //number of admissible blocks (info)
    number_t nbAppMatrices;                //number of approximate matrices (info)
    ...

```

The data member `depth`, `nbNodes`, `nbLeaves`, `nbAdmissibles`, `nbAppMatrices` are set by the division algorithm or computed on demand.

The class redefines the default constructor (initializing data members) and an explicit constructor filling the cluster parameters and building the tree structure recursively:

```
HMatrix();
HMatrix(ClusterTree<I>&, ClusterTree<I>&, number_t, number_t, number_t =0,
        const string_t& = "", SymType =_noSymmetry,
        HMatrixMethod =_standardHM, HMAmissibilityRule=_boxesRule, real_t =1.);
void buildTree(); //build the tree
void updateInfo(); //update tree info (depth, ...)
HMatrix(const HMatrix<T,I>&); //copy constructor
HMatrix<T,I>& operator=(const HMatrix<T,I>&); //assign operator
~HMatrix(); //destructor
void copy(const HMatrix<T,I>&); //copy all
void clear(); //clear all
void clearMatrices(); //deallocate matrix pointers
```

To preserve integrity of `HMatrix` object, the copy constructor and the assign operator do a hard copy of the tree and of the matrices attached to `HMatrixNode` objects but the row and col cluster trees are not copied ! When a `HMatrix` object is deleted or cleared, all the nodes of the tree and node matrices are deleted but not the row and col cluster trees!

The `HMatrix` class provides some stuff to get or extract various informations about the tree structure:

```
number_t numberOfRows() const; //number of rows counted in T
number_t numberOfCols() const; //number of cols counted in T
dimPair dimValues() const; //dimensions of values, (1,1) when scalar
number_t nbNonZero() const; //number of coefficients used
const ClusterTree<I>* rowTree() const; //access to row ClusterTree pointer
const ClusterTree<I>* colTree() const; //access to col ClusterTree pointer
void setClusterRow(ClusterTree<I>*); //change the row ClusterTree pointer
void setClusterCol(ClusterTree<I>*); //change the col ClusterTree pointer
pair<number_t, number_t> averageSize() const; //average size of leaves
number_t averageRank() const; //average rank of admissible leaves
list<HMatrixNode<T,I>* > getLeaves(bool = true) const; //get all leaves
```

It is possible to import a `LargeMatrix` in a `Hmatrix` only if the tree structure already exists and to export a `Hmatrix` to a `LargeMatrix` (dense storage).

```
void load(const LargeMatrix<T>&, HMAproximationMethod = _noHMAproximation);
LargeMatrix<T> toLargeMatrix(StorageType st=_dense, AccessType at=_row) const;
```

Some algebraic operations on `HMatrix` are available:

```
vector<T>& multMatrixVector(const vector<T>&, vector<T>&) const;
vector<T>& multMatrixVectorOmp(const vector<T>&, vector<T>&) const;
void addFELargeMatrix(const LargeMatrix<T>&); //add FE LargeMatrix to current
real_t norm2() const; //Frobenius norm
real_t norminf() const; //infinite norm
```

Note that the `multMatrixVector` function is recursive so it is not safe in multithread computation. This is the reason why there exists a non recursive function `multMatrixVectorOmp` that implements omp pragma to parallelize the computation.

The `Hmatrix` class provides the following print stuff:

```
void print(std::ostream&) const;
void printSummary(std::ostream&) const;
```

```

void printStructure(std::ostream&, bool all=false, bool =false) const;
void saveStructureToFile(const string_t&) const;
ostream& operator<<(ostream& os, const HMatrix<X,J>& hm);

```

As an illustration, we show on figure 10.4 the structure of a HMatrix based on the clustering of a mesh of a sphere, get by the following XLiFE++ code

```

Mesh meshd(Sphere(_center=Point(0.,0.,0.),_radius=1.,_nnodes=2,
                 _domain_name="Omega"),_triangle,1,_subdiv);
Domain omega=meshd.domain("Omega");
Space V(omega,P0,"V",false);
ClusterTree<FeDof> ct(V.feSpace()->dofs,_cardinalityBisection,10);
HMatrix<Real,FeDof> hm(ct,ct,nbox,nbox);
hm.saveStructureToFile("hmatrix.dat");

```

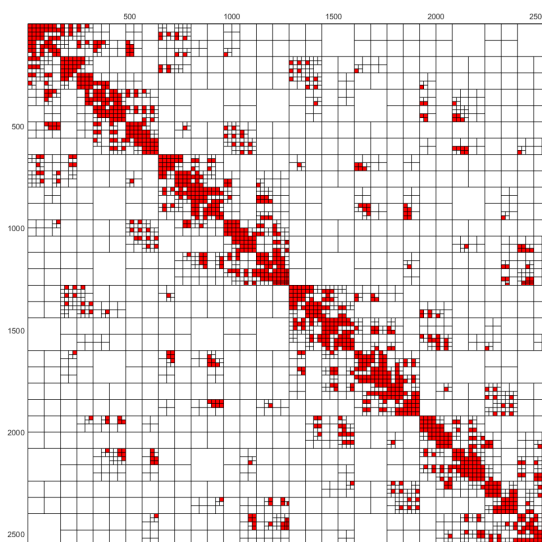


Figure 10.4: HMatrix with a sphere cluster, non admissible blocks in red

10.3.3 The HMatrixEntry class

The template `HMatrixEntry<I>` class is similar to the `MatrixEntry` class. In order to shadow the type of the HMatrix coefficients it handles pointers to `HMatrix<real_t,I>`, `HMatrix<complex_t,I>`, `HMatrix<Matrix<real_t>,I>` and `HMatrix<Matrix<complex_t>,I>`:

```

class HMatrixEntry
{
public :
  ValueType valueType_;           //entries value type
  StrucType strucType_;          //entries structure type
  HMatrix<real_t,I>* rEntries_p;  //pointer to real HMatrix
  HMatrix<complex_t,I>* cEntries_p; //pointer to complex HMatrix
  HMatrix<Matrix<real_t>,I>* rmEntries_p; //pointer to HMatrix of real matrix
  HMatrix<Matrix<complex_t>,I>* cmEntries_p; //pointer to HMatrix of cmplx matrix
  dimPair nbOfComponents;        //nb of rows and columns of matrix values
  ...

```

The following constructor/destructor stuff is available:

```

HMatrixEntry(ValueType, StrucType, ClusterTree<I>&, ClusterTree<I>&,
             number_t, number_t, number_t =1, number_t=1,
             SymType sy = _noSymmetry);
HMatrixEntry(const HMatrixEntry<I>&); //copy constructor
HMatrixEntry<I>& operator=(const HMatrixEntry<I>&); //assign operator
~HMatrixEntry() { clear(); } //destructor
void copy(const HMatrixEntry<I>&); //full copy of members
void clear(); //deallocate memory used

```

To preserve integrity of `HMatrixEntry` class, the copy constructor and assignment operator do a hard copy of `HMatrix` objects pointed as soon as the pointer is not null.

The class provides some useful functionalities:

```

void setClusterRow(ClusterTree<I>*&); //change the ClusterTree row pointer
void setClusterCol(ClusterTree<I>*&); //change the ClusterTree col pointer
template <typename T>
HMatrix<T, I>& getHMatrix() const //get HMatrix object
real_t norm2() const; //Frobenius norm
real_t norminf() const; //infinite norm
void print(ostream&) const; //print HMatrixEntry
void printSummary(ostream&) const; //print HMatrixEntry in brief

```

The member function `getHMatrix` has `real_t`, `complex_t`, `Matrix<real_t>` and `Matrix<complex_t>` specializations.

library :	hierarchicalMatrix
header :	HMatrix.hpp
implementation :	HMatrix.cpp
unitary tests :	test_HMatrix.cpp
header dependences :	ApproximateMatrix.hpp, ClusterTree.hpp, largeMatrix.h ,config.h, utils.h

10.3.4 The HMatrixIM class

In order to inform the code that `HMatrix` has to be used, a special class of integration method has been developed : `HMatrixIM` . It inherits from the `DoubleIM` class inheriting itself from the `IntegrationMethod` class. Attaching such integration method to a BEM bilinear form, `HMatrix` computation algorithm will be involved.

```

class HMatrixIM : public DoubleIM
{public:
    ClusterTree<FeDof>* rowCluster_; //row cluster pointer
    ClusterTree<FeDof>* colCluster_; //col cluster pointer
    mutable bool deletePointers_; //flag enabling pointers deallocation
    ClusteringMethod clusterMethod; //clustering method
    HMApproximationMethod hmAppMethod; //type of approximation of admissible blocks
    number_t minRowSize, minColSize; //minimum row/col size of leaf matrix
    number_t maxRank; //maximal rank of approximate matrices
    real_t epsRank; //precision used in compression
    IntegrationMethod* intgMethod; //real integration method
    ...
};

```

This class provides some constructors

```

HMatrixIM(ClusteringMethod clm, number_t minRow, number_t minCol,
          HMApproximationMethod hmap, number_t maxr, IntegrationMethod& im);
HMatrixIM(ClusteringMethod clm, number_t minRow, number_t minCol,

```

```

    HMAproximationMethod hmap, real_t epsr, IntegrationMethod& im);
HMatrixIM(HMAproximationMethod hmap, number_t maxr, IntegrationMethod& im,
    ClusterTree<FeDof>& rowC, ClusterTree<FeDof>& colC);
HMatrixIM(HMAproximationMethod hmap, real_t epsr, IntegrationMethod& im,
    ClusterTree<FeDof>& rowC, ClusterTree<FeDof>& colC);

```

The two last constructors allow to load some row/column clusters whereas the two first ones provide the parameters to build them. When building **HMatrixIM** object from clustering parameters, the `deletePointers` flag is true, so destroying the **HMatrixIM** object induces the deallocation of the cluster pointers. When building **HMatrixIM** object from cluster pointers, these pointers will not be deallocated by destructor.

The class has also a clear method to delete the cluster pointers and a destructor that clears the cluster pointers if `deletePointers` is true:

```

~HMatrixIM();
void clear();

```

The **HMatrixIM** class is used as following:

```

Mesh meshd(Sphere(_center=Point(0.,0.,0.),_radius=1.,_nnodes=9,
    _domain_name="Omega"),_triangle,1,_subdiv);
Domain omega=meshd.domain("Omega");
Space W(omega,P0,"V",false);
Unknown u(W,"u"); TestFunction v(u,"v");
Kernel G=Laplace3dKernel();
SauterSchwabIM ssim(5,5,4,3,2.,4.);
HMatrixIM him(_cardinalityBisection, 20, 20,_acaplus,0.00001,ssim);
BilinearForm alf=intg(omega,omega,u*G1*v,him);
TermMatrix A(alf,"A");

```



Because the compression methods do not work yet for matrix of matrices, **HMatrix** is not fully operational when dealing with some problems with vector unknown.

library :	hierarchicalMatrix
header :	HMatrix.hpp
implementation :	HMatrix.cpp
unitary tests :	test_HMatrix.cpp
header dependences :	ApproximateMatrix.hpp, ClusterTree.hpp, largeMatrix.h ,config.h, utils.h

11

The *term* library

This library addresses the `Term` class and its inherited classes that handle the numerical representations of linear and bilinear forms involved in problems to be solved, say vectors and matrices (`LargeMatrix` class).

The `Term` class is an abstract class collecting general data (name, computing informations) and has four children :

- the `TermVector` class to represent linear forms involving multiple unknowns,
- the `SuTermVector` class to represent linear forms involving only one unknown,
- the `TermMatrix` to represent bilinear forms involving multiple pairs of unknowns,
- the `SuTermMatrix` to represent bilinear forms involving only one pair of unknowns,

`TermVector` class is linked to a `LinearForm` object (defined on multiple unknowns) and handles a map of `SuTermVector` object indexed by unknown. In a same way, `TermMatrix` class is linked to a `BiLinearForm` object (defined on multiple unknown pairs) and handles a map of `SuTermMatrix` object indexed by unknown pair. `SuTermVector` and `SuTermMatrix` are related to `SuLinearForm` and `SuBiLinearForm` and carries the numerical representations as vector or matrix. This is a block representation.

For instance, consider the following two unknowns $(u, p) \in V \times H$ problem:

$$\begin{cases} a(u, v) + b(p, v) &= f(v) \\ c(u, q) + d(p, q) &= g(q) \end{cases}$$

where a, b, c, d are bilinear forms defines respectively on $V \times V, H \times V, V \times H, H \times H$ and f, g are linear forms defined on V and H . The block representation is

$$\begin{bmatrix} \mathbf{A} & \mathbf{B} \\ \mathbf{C} & \mathbf{D} \end{bmatrix} \begin{pmatrix} U \\ P \end{pmatrix} = \begin{pmatrix} F \\ G \end{pmatrix}$$

matrices being stored with their own storages. In order to shadow the type of coefficients (real, complex, scalar, vector, matrix) the vector and matrix representations are managed through the `VectorEntry` class (see `utils` library) and `MatrixEntry` class (see `largeMatrix` library).

In certain circumstances the block representation is not well adapted (matrix factorisation, essential condition to apply). So the representation is moved to a scalar global representation, block representation may be or not kept.

This library collects the main algorithms of vector and matrix computation, in particular

- computation from linear and bilinear form (see the `computation` subdirectory)
- computation from explicit vector, matrix or function (nodal computation)
- linear combination of `TermVector` or `TermMatrix` (see `LcTerm` class)
- the elimination process due to essential conditions (in relation with `essentialConditions` library)

- the interface to solvers (direct solvers, iterative solvers or eigen solvers defined in *largeMatrix*, *solvers*, *eigenSolvers* libraries)
- some import and export facilities (see the *ioTerms* subdirectory)

TermMatrix and *TermVector* are end user classes, therefore the functions that interact with users has to be simply designed!

11.1 The Term class

The *Term* class is the abstract basis class of *TermVector*, *TermMatrix*, *SuTermVector*, *SuTermMatrix* classes. It collects general informations:

```
class Term
{
protected :
    String name_;           //term name
    TermType termType_;     //type of term
    ComputingInfo computingInfo_; //computing information
public :
    Parameters params;      //a free zone to store additional data
    static std::vector<Term *> theTerms;
}
```

where *TermType* enumerates the child types :

```
enum TermType {_termUndef ,_termVector ,_termMatrix ,_suterminVector ,_suterminMatrix };
```

and *ComputingInfo* collect various computing informations:

```
class ComputingInfo
{
public :
    bool isComputed;
    bool noAssembly;
    bool multithreading;
    bool useGpu;
    StorageType storageType;           //_cs ,_skyline ,_dense
    AccessType storageAccess;         //_row ,_col ,_dual ,_sym
    EliminationMethod elimination;    //_noElimination , _pseudoElimination ,
    _realElimination
};
```

Using a static vector of *Term* pointers, the class maintains a list of all terms.

This class provides a default constructor, declared *protected* to forbid instantiation by user, and some basic functions (mainly accessors)

```
Term(const String& na="", ComputingInfo ci=ComputingInfo()); //protected
virtual ~Term();
const String& name() const;
String& name();
TermType termType() const;
TermType& termType();
ComputingInfo computingInfo() const;
ComputingInfo& computingInfo();
bool& computed();
bool computed() const;
virtual void compute()=0;
```



```

virtual void clear()=0;
virtual void print(std::ostream&) const=0;
friend std::ostream& operator<<(std::ostream&,const Term&);
static void clearGlobalVector();
};

```

Finally, some general functions (to compute and clear terms) are provided:

```

void compute(Term&);
void compute(Term&, Term&);
...
void clear(Term&);
void clear(Term&, Term&);
...

```

library :	term
header :	Term.hpp
implementation :	Term.cpp
unitary tests :	test_TermVector.cpp
header dependencies :	essentialConditions.h, config.h, utils.h

11.2 The TermVector class

The **TermVector** class carries numerical representation of a linear form and more generally any vector attached to an approximation space:

```

class TermVector : public Term
{
protected :
    LinearForm linForm_;
    map<const Unknown*, SuTermVector*> suTerms_;
    VectorEntry* entries_p;
    VectorEntry* scalar_entries_p;
    vector<DofComponent> cdofs_;

```

The linear form is a multiple unknowns form, may be reduced to a single unknown form (see *form* library). This linear form is void if **TermVector** is not explicitly construct from a linear form (linear combination, nodal construction). The map *suTerms_* carries each vector block related to an unknown. When there is only one element in map, it means that the **TermVector** is actually a single unknown term vector. The *scalar_entries_p* pointer may be allocated to concatenate the block vectors. When unknowns are vector unknowns, the representation may be not suited in certain cases (direct solver), so it is possible to create its scalar representation using the *scalar_entries_p* (**VectorEntry** pointer). The **VectorEntry** object created has to be of real scalar or complex scalar type. In that case, a non block row numbering consisting in vector of **DofComponent** (Unknown pointer, dof number, component number) is also defined (*cdofs_*).

The **TermVector** class provides useful constructors and assign operators designed for end users :

```

//default constructor
TermVector(const String& na = "", bool noass = false);

//from linear form with options and no essential conditions
TermVector(const LinearForm&, const string_t& na="");
TermVector(const LinearForm&, TermOption, const string_t& na="");
TermVector(const LinearForm&, TermOption, TermOption, const string_t& na="");

```

```

TermVector(const LinearForm&, TermOption, TermOption, TermOption,
           const string_t& na="");

//from linear form with options and one essential condition
TermVector(const LinearForm&, const EssentialConditions&,
           const string_t& na="");
TermVector(const LinearForm&, const EssentialConditions&, TermOption,
           const string_t& na="");
TermVector(const LinearForm&, const EssentialConditions&, TermOption,
           TermOption, const string_t& na="");
TermVector(const LinearForm&, const EssentialConditions&, TermOption,
           TermOption, TermOption, const string_t& na="");

void build(const LinearForm &, const EssentialConditions*,
           const vector<TermOption>&, const string_t&);

//from value or function (interpolation)
template <typename T>
    TermVector(const Unknown&, const Domain&, const T&, const String& na = "");

//copy like
TermVector(const TermVector&, const String& na = "");
template <typename T>
    TermVector(const TermVector&, const T&, const String& na);
TermVector(const SuTermVector &, const String& na="");
TermVector(SuTermVector *, const String& na="");
void copy(const TermVector&, const String &);

//constructor from explicit function of TermVector's
TermVector(const TermVector&, funSR1_t& f, const string_t& na="");
TermVector(const TermVector&, TermVector&, funSR2_t& f, const string_t& na="");
TermVector(const TermVector&, funSC1_t& f, const string_t& na="");
TermVector(const TermVector&, TermVector&, funSC2_t& f, const string_t& na="");
    single unknown

//constructor from symbolic function of TermVector's
TermVector(const TermVector&, const SymbolicFunction& , const string_t& na="");
TermVector(const TermVector&, const TermVector&, const SymbolicFunction& fs,
const string_t& na="");
TermVector(const TermVector&, const TermVector&, const TermVector&,
const SymbolicFunction& fs, const string_t& na="");

//constructor of a vector TermVector from scalar TermVector's (single unknown, same
space)
TermVector(const Unknown&, const TermVector&, const TermVector&, const string_t&
="");
TermVector(const Unknown&, const TermVector&, const TermVector&, const TermVector&,
const string_t& ="");
TermVector(const Unknown&, const TermVector&, const TermVector&, const TermVector&,
const TermVector&, const string_t& ="");

//from linear combination
TermVector(const LcTerm&);
//assignment
TermVector& operator=(const TermVector&);
TermVector& operator=(const LcTerm&);
//insert SuTermVector in TermVector
void insert(const SuTermVector &);
void insert(SuTermVector *);
void insert(const Unknown*, SuTermVector*);
//destructor, clear
virtual ~TermVector();
void clear();

```

When `TermVector` is constructed from linear form, it is automatically computed except if it has set to be not computed using a the `TermOption _notCompute` in its construction.

`TermOptions` are managed as an enumeration :

```
enum TermOption
{
    //general
    _compute,                //default behaviour
    _notCompute,             //default behaviour
    _assembled,              //default behaviour
    _unassembled,           //not yet available
    //only for TermMatrix
    _nonSymmetricMatrix,     //to force symmetry property
    _symmetricMatrix,
    _selfAdjointMatrix,
    _skewSymmetricMatrix,
    _skewAdjointMatrix,
    _csRowStorage,           //to force storage
    _csColStorage,
    _csDualStorage,
    _csSymStorage,
    _denseRowStorage,
    _denseColStorage,
    _denseDualStorage,
    _skylineSymStorage,
    _skylineDualStorage,
    _pseudoReduction,        //default behaviour
    _realReduction,          //not yet available
    _penalizationReduction   //not yet available
};
```

Constructors involving `SuTermVector` class are reserved to developers. End users should not have to acces to `SuTermVector` class, but we decide to let this access free to advanced users. Constructors involving `LcTerm` class are implicitly used by users when they write combination of terms in a symbolic way (see `LcTerm` class).

The class provides some useful accessors (for users or developers):

```
//user accessors
const LinearForm& linearForm() const;
Number nbOfUnknowns() const;
bool isSingleUnknown() const;
set<const Unknown*> unknowns() const;
ValueType valueType() const;
Number size() const;
Number nbDofs() const;
Number nbDofs(const Unknown&) const;
const vector<DofComponent>& cdofs() const;
vector<DofComponent>& cdofs();
const Dof& dof(const Unknown&, Number ) const;

//developer accessors
cit_mustv begin() const;
it_mustv begin();
cit_mustv end() const;
it_mustv end();
SuTermVector* firstSut() const;
SuTermVector* firstSut();
const Unknown* unknown() const;
VectorEntry& entries();
const VectorEntry* entries() const;
VectorEntry& scalar_entries();
const VectorEntry* scalar_entries() const;
VectorEntry* actual_entries() const;
```

it_mustv and *cit_mustv* are iterator aliases:

```
typedef map<const Unknown*, SuTermVector*>::iterator it_mustv;
typedef map<const Unknown*, SuTermVector*>::const_iterator cit_mustv;
```

Access to a block of **TermVector** say a **SuTermVector** is proposed in two ways : either using **subVector** functions returning reference or pointer to **SuTermVector** objects or using **operator()** returning a new **TermVector** object, the **SuTermVector** object being copied. This last method is addressed to end users. For particular purpose, is is also possible to reinterpret a **TermVector** as a raw vector or a raw vector of vectors (say a **Vector<S>** or a **Vector<Vector<S> >** with S a Real or a Complex).

```
const SuTermVector& subVector() const;
SuTermVector& subVector(const Unknown*);
const SuTermVector& subVector(const Unknown*) const;
SuTermVector& subVector(const Unknown&);
const SuTermVector& subVector(const Unknown&) const;
SuTermVector* subVector_p(const Unknown*);
const SuTermVector* subVector_p(const Unknown*) const;
TermVector operator()(const Unknown&) const; //user usage
template<typename T>
    Vector<T>& asVector(Vector<T>&) const;
```

The main computing operations are related to the construction of **TermVector** from linear form and construction from linear combination **TermVector**:

```
void compute();
void compute(const LcTerm&); //implicit
TermVector& operator+=(const TermVector&);
TermVector& operator-=(const TermVector&);
template<typename T>
    TermVector& operator*=(const T&);
template<typename T>
    TermVector& operator/=(const T&);
```

Note that the evaluation of **TermVector** defined by a value or a function is done during creation of **TermVector**. The algebraic operators $+=$, $-=$, $*=$, $/=$ do the computation, it is not delayed as it is the case for $+$, $-$, $*$ and $/$ operators. The linear combination is based on **LcTerm** class which manages a list of pairs of **Term** and coefficient. Algebraic operators on **TermVector**'s produce **LcTerm** object carrying the linear combination that will be computed by **TermVector** constructor or assignment operator:

```
const TermVector& operator+(const TermVector&);
LcTerm operator-(const TermVector&);
LcTerm operator+(const TermVector&, const TermVector&);
LcTerm operator-(const TermVector&, const TermVector&);
LcTerm operator+(const LcTerm&, const TermVector&);
LcTerm operator-(const LcTerm&, const TermVector&);
LcTerm operator+(const TermVector&, const LcTerm&);
LcTerm operator-(const TermVector&, const LcTerm&);
template<typename T>
    LcTerm operator*(const TermVector&, const T&);
template<typename T>
    LcTerm operator*(const T&, const TermVector& )
template<typename T>
    LcTerm operator/(const TermVector&, const T&)
```

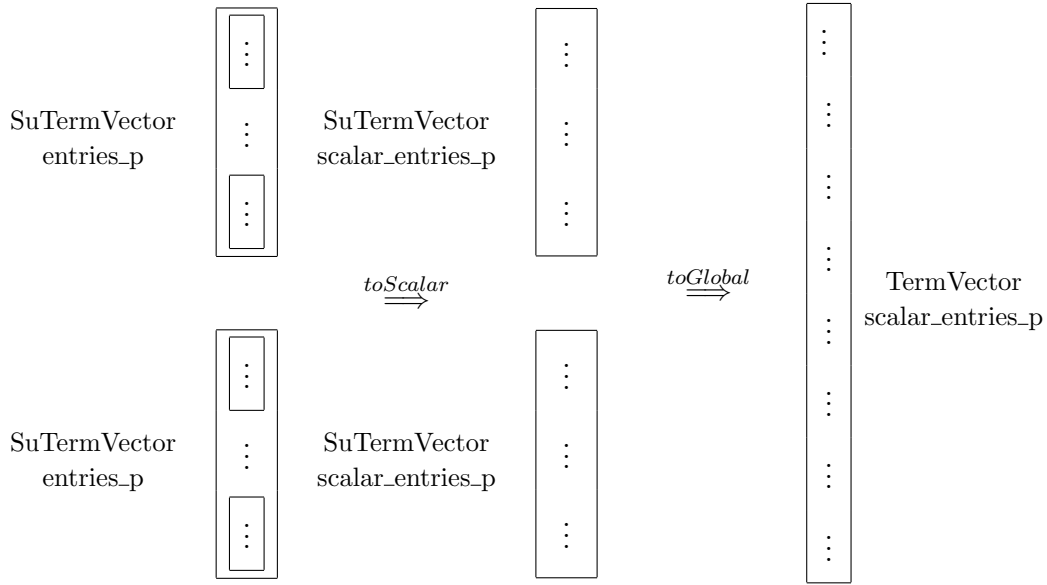
To manage different representation of **TermVector**, the following function are provided :

```

void toScalar(bool keepEntries=false);
void toVector(bool keepEntries=false);
void toGlobal(bool keepSuTerms=false);
void toLocal(bool keepGlobal=false);
void adjustScalarEntries(const vector<DofComponent>&);

```

Note the difference between *toScalar* and *toGlobal*. *toScalar* forces the scalar representation of **SuTermVector**'s and *toGlobal* go to non block representation, allocating the *scalar_entries_p* pointer of **TermVector**. *toGlobal* operation induces *toScalar* operation. *toLocal* reconstructs block representation from global representation (inverse of *toGlobal*) and *toVector* returns to vector representation (inverse of *toScalar*).



In some circumstances, to avoid recomputation, it may be useful to change the unknowns of a **TermVector** without changing its content, with an optional management of components of unknown:

```

void changeUnknown(const Unknown&, const Unknown&,
                  const Vector<Number>& = Vector<Number>());
void changeUnknown(const Unknown&,
                  const Vector<Number>& = Vector<Number>());
void changeUnknown(const Unknown&, Number); //shortcut
void changeUnknown(const Unknown&, Number, Number); //shortcut
void changeUnknown(const Unknown&, Number, Number, Number); //shortcut

```

The principle of **changeUnknown** functions is the following :

- when two unknowns (u,v) are specified, the unknown u is replaced by the unknown v and the reference to unknown u does no longer exists in **TermVector**
- when a renumbering components vector (rc) is given too, components are re-settled along this vector
- when v is not specified, only components are re-settled

Here are some exemples :

- u a scalar unknown, v a scalar unknown, replace unknown u by unknown v
- u a scalar unknown, v a 3-vector unknown, rc=[2] leads to $\text{sutv} = [0 \text{ sutv } 0]$

- u a 4-vector unknown, v a 2-vector unknown, rc=[2,4] leads to $\text{sutv} = [0 \text{ sutv1 } 0 \text{ sutv2}]$
- u a 3-vector unknown, rc=[3,0,1], re-settle components as $[\text{sutu3 } 0 \text{ sutu1}]$



Be cautious that there is no check of the space consistency! So use it with care.

It is possible to address values of a **TermVector** by unknown and index, to get it or to set it:

```
Value getValue(const Unknown&, Number) const;
void setValue(const Unknown&, Number, const Value&);
void setValue(const Unknown&, Number, const Real&);
void setValue(const Unknown&, Number, const Complex&);
void setValue(const Unknown&, Number, const vector<Real>&);
void setValue(const Unknown&, Number, const vector<Complex>&);
```

The **Value** class handles different type of values, in particular **Real** and **Complex** values.

It is also possible to change some values of a **TermVector** by specifying the unknown that it is concerned, the geometrical part where values will be changed and the values that can be given either by a constant, a function or a **TermVector**:

```
template <typename T>
void setValue(const Unknown&, const GeomDomain&, const T&);
void setValue(const Unknown&, const GeomDomain&, const TermVector&);
void setValue(const Unknown&, const TermVector&);
```

In all **setValue** functions, the unknown may be omitted. In that case, the first unknown is assumed.

Most powerful functions are functions that can evaluate any function of approximation space (V_h) (representing by a **TermVector** U) at any point, using space interpolation :

$$u_h(x) = \sum_{i=1,n} U_{u,i} w_i(x)$$

where $U_{u,i}$ is the i^{th} components related to the unknown u of **TermVector** U and $(w_i)_{i=1,n}$ the basis functions of approximation space V_h .

```
Value evaluate(const Unknown&, const Point&) const;
template<typename T>
T& operator()(const vector<Real>&, T&) const;
template<typename T>
T& operator()(const Unknown&, const vector<Real>&, T&) const;
```

Some additional computational tools are available:

```
TermVector& toAbs();
TermVector& toReal();
TermVector& toImag();
TermVector& toComplex();
TermVector& toConj();
TermVector mapTo(const GeomDomain&, const Unknown&) const;
TermVector onDomain(const GeomDomain&) const;
Real norm2() const;
Complex maxValAbs() const;
```

and the class proposes two basic output functions:

```
void print(ostream&) const;
void saveToFile(const String&, bool encode=false) const;
```

Besides, some of the functionalities are also provides as extern functions:

```
//linear combination
const TermVector& operator+(const TermVector&);
LcTerm operator-(const TermVector&);
LcTerm operator+(const TermVector&, const TermVector&);
LcTerm operator-(const TermVector&, const TermVector&);
LcTerm operator+(const LcTerm&, const TermVector&);
LcTerm operator-(const LcTerm&, const TermVector&);
LcTerm operator+(const TermVector&, const LcTerm&);
LcTerm operator-(const TermVector&, const LcTerm&);
template<typename T>
LcTerm operator*(const TermVector&, const T&);
template<typename T>
LcTerm operator*(const T&, const TermVector&);
template<typename T>
LcTerm operator/(const TermVector&, const T&)

//inner, hermitian product, norm, ...
TermVector abs(const TermVector&);
TermVector real(const TermVector&);
TermVector imag(const TermVector&);
TermVector conj(const TermVector&);
Complex innerProduct(const TermVector&, const TermVector&);
Complex hermitianProduct(const TermVector&, const TermVector&);
Complex operator|(const TermVector&, const TermVector&);
Real norm(const TermVector&, Number l=2);
Real norm1(const TermVector&);
Real norm2(const TermVector&);
Real norminfy(const TermVector&);

//output functions
void print(ostream&) const;
void saveToFile(const String&, bool encode=false) const;
void saveToFile(const String&, const list<const TermVector*>&, IOFormat iof=_raw);
void saveToFile(const String&, const TermVector&, IOFormat iof=_raw);
void saveToFile(const String&, const TermVector&, const TermVector&,
               IOFormat iof=_raw);
void saveToFile(const String&, const TermVector&, const TermVector&,
               const TermVector&, IOFormat iof=_raw);
void saveToFile(const String&, const TermVector&, const TermVector&,
               const TermVector&, const TermVector&, IOFormat iof=_raw);
```

Available export format are

- `_raw` : export only the values in a raw format
- `_vtk` : export mesh and values displayable with Paraview
- `_vtu` : export mesh and values displayable with Paraview
- `_matlab` : export mesh and values as an executable Matlab file
- `_xyzv`: export mesh nodes and values as x y z v1 v2 ...

Integral representation is a particular operation mixing numerical quadrature and interpolation :

$$u(x_i) = \int_{\Gamma} opk(K)(x_i, y) aop opu(p)(y) dy \quad i = 1, n.$$

where K is a kernel, opk an operator on kernel (`OperatorOnKernel`), p a layer potential, opu an operator on unknown (`OperatorOnUnknown`) and aop an algebraic operator.


```
template<typename T>
Vector<T>& integralRepresentation(const vector<Point>& xs,
    const LinearForm& lf, const TermVector& U, Vector<T>& val)
TermVector integralRepresentation(const Unknown&, const GeomDomain&,
    const LinearForm&, const TermVector&);
```

For instance, the single layer integral representation on a list of points is computed as follows:

```
Vector<Point> Points;
...
Vector<Real> val;
integralRepresentation(Points, intg(gamma, G(_y) * u), U, val);
```

or if the list of points is the list of nodes of a domain:

```
TermVector IR=integralRepresentation(Omega, intg(gamma, G(_y) * u), U, val);
```

For most of functions of TermVector, the algorithms travel the map of SuTermVector, calling SuTermVector's functions. For instance, the compute function looks like:

```
void TermVector::compute()
{
    for(it_mustv it = suTerms_.begin(); it != suTerms_.end(); it++)
    {
        if(!it->second->computed()) it->second->compute();
    }
    computed() = true;
}
```

library :	term
header :	TermVector.hpp
implementation :	TermVector.cpp
unitary tests :	test_TermVector.cpp
header dependences :	SuTermVector.hpp, termUtils.hpp, Term.hpp, form.h, config.h, utils.h

11.3 The SuTermVector class

The SuTermVector class carries numerical representation of a single linear form and more generally any vector attached to an approximation space.

```
class SuTermVector : public Term
{
protected :
    SuLinearForm* sulf_p;
    const Unknown* u_p;
    mutable Space* space_p;
    std::vector<Space*> subspaces;
    VectorEntry* entries_p;
    VectorEntry* scalar_entries_p;
    std::vector<DofComponent> cdofs_;
}
```

The sulf_p contains a pointer to a SuLinearForm (single unknown linear form). It may be null, that means there is no explicit linear form attached to the SuTermVector.

The space_p pointer points to the largest subspace involved in SuTermVector. This pointer should not

be null because it carries the dofs numbering of vector. For this reason too, the largest subspace has to be correctly updated during any operation on `SuTermVector`. The `Space*` vector `subspaces` contains the subspaces (as subspace of largest space) attached to basic linear forms defined in the `SuLinearForm` pointer (if defined), see `buildSubspaces` function.



Do not confused the `space_p` pointer and `u_p->space_p` pointer that specifies the whole space. They may be same!

The numerical representation of vector is defined in the `entries_p` `VectorEntry` pointer as a real/complex vector or a real/complex vector of vectors, regarding the value type (real or complex) and the structure of the unknown (scalar or vector). If required, vector of vectors representation may be expanded to a vector of scalars stored in the `scalar_entries_p` `VectorEntry` pointer. In that case the `cdofs_` vector of `DofComponent` gives the numbering (in `DofComponent`) of entries. Note that if `SuTermVector` is of scalar type, `scalar_entries_p = entries_p`.

The `SuTermVector` class provides one constructor from linear form, some constructors from constant value, functions or symbolic functions, one constructor from linear combination and some copy constructors:

```
//constructor from linear form
SuTermVector(SuLinearForm* sulf = 0, const String& na = "", bool noass = false);
//constructors from constant value or function (no linear form)
SuTermVector(const String&, const Unknown*, Space*, ValueType vt = _real, Number n
    = 0, Dimen nv = 0, bool noass = false); //vector of zeros
template <typename T>
SuTermVector(const Unknown&, const GeomDomain&, const Vector<T>&,
    const String&, bool);
template <typename T>
SuTermVector(const Unknown&, const GeomDomain&, const T&,
    const String&, bool);
SuTermVector(const Unknown&, const GeomDomain&, funSR_t&,
    const String& na="", bool noass=false);
SuTermVector(const Unknown&, const GeomDomain&, funSC_t&,
    const String& na="", bool noass=false);
SuTermVector(const Unknown&, const GeomDomain&, funVR_t&,
    const String& na="", bool noass=false);
SuTermVector(const Unknown&, const GeomDomain&, funVC_t&,
    const String& na="", bool noass=false);
//constructor from explicit function of SuTermVector's
SuTermVector(const SuTermVector&, funSR1_t& f, const string_t& na="");
SuTermVector(const SuTermVector&, const SuTermVector&, funSR2_t& f, const string_t&
    na="");
SuTermVector(const SuTermVector&, funSC1_t& f, const string_t& na="");
SuTermVector(const SuTermVector&, const SuTermVector&, funSC2_t& f, const string_t&
    na="");
//constructor from symbolic function of SuTermVector's
SuTermVector(const SuTermVector&, const SymbolicFunction&, const string_t& na="");
SuTermVector(const SuTermVector&, const SuTermVector&, const SymbolicFunction&,
    const string_t& na="");
SuTermVector(const SuTermVector&, const SuTermVector&, const SuTermVector&,
    const SymbolicFunction&, const string_t& na="");
//constructor of a vector SuTermVector from scalar SuterminVector's
SuTermVector(const Unknown&, const SuTermVector&, const SuTermVector&, const
    string_t& = "");
SuTermVector(const Unknown&, const std::list<const SuTermVector*>&, const string_t&
    = "");
//constructor from linear combination (no linear form)
SuTermVector(const LcTerm&);
//copy constructors and assign operator
```

```

SuTermVector(const SuTermVector&);
template<typename T>
SuTermVector(const SuTermVector&, const T&);
SuTermVector& operator=(const SuTermVector&);
//other useful stuff
virtual ~SuTermVector();
void copy(const SuTermVector&); //full copy
void initFromFunction(const Unknown&, const GeomDomain&, const Function&,
                     const String& na="", bool noass=false);
void clear();

```

A lot of useful accessors and shortcuts to some properties are proposed:

```

string_t name();
void name(const string_t&);
SuLinearForm* sulfp() const;
SuLinearForm& sulfp();
VectorEntry& entries();
const VectorEntry* entries() const;
TermType termType() const;
ValueType valueType() const;
StrucType strucType() const;
const Unknown* up() const;
const Unknown& up();
number_t nbOfComponents() const;
const Space* spacep() const;
Space& spacep();
set<const Space*> unknownSpaces() const;
number_t nbDofs() const;
number_t size() const;
VectorEntry& scalar_entries();
const VectorEntry* scalar_entries() const;
VectorEntry* actual_entries() const;
const vector<DofComponent>& cdofs() const;
vector<DofComponent>& cdofs();
const Dof& dof(number_t n) const;
const GeomDomain* domain() const;

```

`SuTermVector` may be modified by the following operators and functions:

```

SuTermVector& operator+=(const SuTermVector&);
SuTermVector& operator-=(const SuTermVector&);
template<typename T>
SuTermVector& operator*=(const T&);
template<typename T>
SuTermVector& operator/=(const T&);
SuTermVector& merge(const SuTermVector&);
SuTermVector& toAbs();
SuTermVector& toReal();
SuTermVector& toImag();
SuTermVector operator()(const ComponentOfUnknown&) const;
SuTermVector& toComplex();
complex_t maxValAbs() const;
void setValue(number_t, const Value&);
void setValue(const Value&, const GeomDomain&);
void setValue(const Function&, const GeomDomain&);
void setValue(const SuTermVector&, const GeomDomain&);
void setValue(const SuTermVector&);

```

The operator `()` extracts unknown component term vector as a `SuTermVector` when `Unknown` is a vector unknown. For instance if `u` is a vector unknown, `T(u[2])` returns a `SuTermVector` containing the

elements of T corresponding to second component.

Some values can be extracted or computed from:

```
Value getValue(number_t) const;
SuTermVector* onDomain(const GeomDomain& dom) const;
template<typename T>
T& operator()(const vector<real_t>&, T&) const;
SuTermVector* interpolate(const Unknown&, const GeomDomain&);
Value evaluate(const Point&) const;
template<typename T>
Vector<T>& asVector(Vector<T>&) const;
```

SuTermVector provides real computation algorithms, in particular FE computation ones:

```
void buildSubspaces();
void compute(); //!< compute from linear form
void compute(const LcTerm&);
template<typename T, typename K>
void computeFE(const SuLinearForm&, Vector<T>&, K&);
template<typename T, typename K>
void computeIR(const SuLinearForm&, Vector<T>&, K& vt, const vector<Point>&,
               const Vector<T>&, const vector<Vector<real_t>>* nxs=0);
```

Some particular member functions allow to change the internal representation:

```
void toScalar(bool keepVector=false);
void toVector(bool keepEntries=false);
void extendTo(const SuTermVector&);
void extendTo(const Space&);
void extendScalarTo(const vector<DofComponent>&, bool useDual = false);
SuTermVector* mapTo(const GeomDomain&, const Unknown&, bool =true) const;
void changeUnknown(const Unknown&, const Vector<number_t>&);
void adjustScalarEntries(const vector<DofComponent>&);
```

Some extern function are provided to compute norms, ...

```
Complex innerProduct(const SuTermVector&, const SuTermVector&);
Complex hermitianProduct(const SuTermVector&, const SuTermVector&);
Real norm1(const SuTermVector&);
Real norm2(const SuTermVector&);
Real norminfy(const SuTermVector&);
```

Finally, there are some print and export stuff:

```
void print(ostream&) const;
void print(ostream&, bool r, bool h) const; // raw and header option
void saveToFile(const String&, bool encode=false) const;

//extern save functions
void saveToFile(const string_t&, const Space*, const list<SuTermVector*>&,
               IOFormat iof=_raw, bool withDomName=true);
void saveToMsh(ostream&, const Space*, const list<SuTermVector*>&,
               vector<Point>, splitvec_t, const GeomDomain*);
void saveToMtlb(ostream&, const Space*, const list<SuTermVector*>&,
               vector<Point>, splitvec_t, const GeomDomain*);
void saveToVtk(ostream&, const Space*, const list<SuTermVector*>&,
               vector<Point>, splitvec_t, const GeomDomain*);
void saveToVtkVtu(ostream&, const Space*, const list<SuTermVector*>&,
                 vector<Point>, splitvec_t, const GeomDomain*);
pair<vector<Point>, map<number_t, number_t>> ioPoints(const Space* sp);
splitvec_t ioElementsBySplitting(const Space*, map<number_t, number_t>);
```

library :	term
header :	SuTermVector.hpp
implementation :	SuTermVector.cpp
unitary tests :	test_TermVector.cpp
header dependencies :	Term.hpp, LcTerm.hpp, termUtils.hpp, form.h, config.h, utils.h

11.4 The TermVectors class

The **TermVectors** is simply an encapsulation of a list of **TermVector**. It implements a small interface to **std::vector<TermVector>** and it is addressed to beginners.

```
class TermVectors: public std::vector<TermVector>
{
public :
    TermVectors(Number n=0);
    TermVectors(const std::vector<TermVector>& vs);
    const TermVector& operator()(Number n) const;
    TermVector& operator()(Number n);
    void print(std::ostream&) const;
};
std::ostream& operator<<(std::ostream&, const TermVectors&);
```

library :	term
header :	TermVector.hpp
implementation :	TermVector.cpp
unitary tests :	test_TermVector.cpp
header dependencies :	Term.hpp, LcTerm.hpp, termUtils.hpp, form.h, config.h, utils.h

11.5 The KernelOperatorOnTermVector class

To deal with integral representation, say for instance

$$v(x) = \int_{\Gamma} K(x, y) \varphi(y)$$

where φ will be represented by a **TermVector** object F, the user can define the linear form

$$\int_{\Gamma} K(x, y) u(y)$$

and then call the **integralRepresentation** function to link its **TermVector** object F to the linear form. To make the user life easier, XLiFE++ provides a small class **KernelOperatorOnTermVector** that relates a **OperatorOnKernel** and a **TermVector**. So, writing **intg(gamma, K*F)** will be interpreted as the linear form acting to the **TermVector** F by handling in the back a **KernelOperatorOnTermVector**:

```
class KernelOperatorOnTermvector
{
protected:
    OperatorOnKernel opker_; //Kernel operator
    AlgebraicOperator aop_; //operation (*,|,%,^) between kernel and TermVector
    const TermVector* tv_; //TermVector involved
public:
```

```

bool termAtLeft;           //if true : opker aop tv else : tv aop opker
}

```

The class provides one basic constructor and some accessors:

```

KernelOperatorOnTermvector(const OperatorOnKernel&, AlgebraicOperator,
                           const TermVector&, bool atleft);
AlgebraicOperator algop();
AlgebraicOperator& algop();
const OperatorOnKernel& opker() const;
const TermVector* termVector();
ValueType valueType() const;;
bool xnormalRequired() const;
bool ynormalRequired() const;

```

Besides, extern functions to the class manage the various operation involving kernel and TermVector:

```

KernelOperatorOnTermvector operator*(const TermVector&, const Kernel&);
KernelOperatorOnTermvector operator|(const TermVector&, const Kernel&);
KernelOperatorOnTermvector operator^(const TermVector&, const Kernel&);
KernelOperatorOnTermvector operator%(const TermVector&, const Kernel&);
KernelOperatorOnTermvector operator*(const Kernel&, const TermVector&);
...
KernelOperatorOnTermvector operator*(const TermVector&, const OperatorOnKernel&);
...
KernelOperatorOnTermvector operator*(const OperatorOnKernel&, const TermVector&);
...
OperatorOnUnknown toOperatorOnUnknown(const KernelOperatorOnTermvector&);

```

Finally, to build integral involving `KernelOperatorOnTermvector` some pseudo-constructors are provided

```

std::pair<LinearForm, const TermVector*>
intg(const GeomDomain&, const KernelOperatorOnTermvector&, const IntegrationMethod&);
std::pair<LinearForm, const TermVector*>
intg(const GeomDomain&, const KernelOperatorOnTermvector&, const
     IntegrationMethods&);
std::pair<LinearForm, const TermVector*>
intg(const GeomDomain&, const KernelOperatorOnTermvector&, QuadRule = _defaultRule,
     number_t = 0);

```

library :	term
header :	KernelOperatorOnTermVector.hpp
implementation :	KernelOperatorOnTermVector.cpp
unitary tests :	test_operator.cpp
header dependences :	OperatorOnKernel.hpp, TermVector.hpp, config.h, utils.h

11.6 The TermMatrix class

The `TermMatrix` class carries numerical representation of a bilinear form and more generally any matrix attached to a pair of unknowns. It follows a similar organization to `TermVector` class but manages row and column information:

```

class TermMatrix : public Term
{protected :
BilinearForm bilinForm_;           //bilinear form if exists
std::map<uvPair, SuTermMatrix*> suTerms_; //list of SuTermMatrix

```

```

MatrixEntry* scalar_entries_p;           //scalar entries representation
std::vector<DofComponent> cdofs_c;      //column scalar dofs
std::vector<DofComponent> cdofs_r;      //row scalar dofs
SetOfConstraints* constraints_u_p;       //constraints to apply to row
SetOfConstraints* constraints_v_p;       //constraints to apply to column
MatrixEntry* rhs_matrix_p;              //matrix used by right hand side correction
}

```

The bilinear form is a multiple unknown pairs form, may be reduced to a single unknown pair form (see [form](#) library). This bilinear form is void if [TermMatrix](#) is not explicitly construct from a bilinear form (linear combination, nodal construction, ...). In particular, this bilinear form is not updated when [TermMatrix](#) objects are combined.

The map `suTerms_` carries each matrix block related to a pair of unknowns. When there is only one element in map, it means that the [TermMatrix](#) is actually a single unknown pair term matrix.

As block representation may be too restrictive in certain cases, it is possible to create its non block representation in `scalar_entries_p` ([MatrixEntry](#) pointer). The [MatrixEntry](#) object created has to be of real scalar or complex scalar type. In that case, non block row and non block column numbering, consisting in vectors of [DofComponent](#) (Unknown pointer, dof number, component number), are also defined (`cdofs_r` and `cdofs_c`).

Essential conditions (conditions on space functions) induce special treatments in computation of [TermMatrix](#). It is the reason why they are handled with [SetOfConstraints](#) objects which are the algebraic representations of essential conditions (see [essentialConditions](#) for details). [SetOfConstraints](#) on row may differ from [SetOfConstraints](#) on column (it is often the same). In case of non homogeneous essential condition, the part of matrix that is eliminated contributes to the right hand side; this part is stored into `rhs_matrix_p` [MatrixEntry](#) pointer. Obviously, the pointers `constraints_u_p`, `constraints_v_p`, `rhs_matrix_p` are null pointer whenever no essential conditions are imposed.

The [TermMatrix](#) class provides useful constructors and assign operators designed for end users :

```

TermMatrix(const string_t & na="?");

//constructors with no essential boundary conditions and options
TermMatrix(const BilinearForm&, const string_t& na="");
TermMatrix(const BilinearForm&, TermOption, const string_t& na="");
TermMatrix(const BilinearForm&, TermOption, TermOption, const string_t& na="");
...
//constructor with one essential boundary condition (on u and v)
TermMatrix(const BilinearForm&, const EssentialConditions&, const string_t& na="");
TermMatrix(const BilinearForm&, const EssentialConditions&, TermOption,
            const string_t& na="");
...
//constructor with one essential boundary condition and explicit reduction method
TermMatrix(const BilinearForm&, const EssentialConditions&, const ReductionMethod&,
            const string_t& na="");
TermMatrix(const BilinearForm&, const EssentialConditions&, const ReductionMethod&,
            TermOption, const string_t& na="");
...
//constructor with two essential boundary conditions (on u and v)
TermMatrix(const BilinearForm&, const EssentialConditions&,
            const EssentialConditions&, const string_t& na="");
TermMatrix(const BilinearForm&, const EssentialConditions&,
            const EssentialConditions&, TermOption, const string_t& na="");
...
//constructor with two essential boundary conditions and explicit reduction method
TermMatrix(const BilinearForm&, const EssentialConditions&, const
            EssentialConditions&,

```



```

        const ReductionMethod&, const string_t& na="");
TermMatrix(const BilinearForm&, const EssentialConditions&, const
    EssentialConditions&,
        const ReductionMethod&, TermOption, const string_t& na="");
...
//effective constructor
void build(const BilinearForm& blf, const EssentialConditions* ecu,
    const EssentialConditions* ecv, const std::vector<TermOption>&,
    const string_t& na);

//copy constructors
TermMatrix(const TermMatrix &, const string_t& na="");
// diagonal TermMatrix from a TermVector or a vector
TermMatrix(const TermVector &, const string_t& na="");
template<typename T>
TermMatrix(const Unknown&, const GeomDomain&, const Vector<T> &,
    const string_t& na="");
//TermMatrix from a SuTermMatrix
TermMatrix(const SuTermMatrix &, const string_t& na=""); //full copy
TermMatrix(SuTermMatrix *, const string_t& na=""); //pointer copy
//TermMatrix of the form opker(xi,yj)
TermMatrix(const Unknown&, const GeomDomain&, const Unknown&,
    const GeomDomain&, const OperatorOnKernel&,
    const string_t& na="");
// TermMatrix from a linear combination
TermMatrix(const LcTerm&,const string_t& na="");

//assign operator
TermMatrix& operator=(const TermMatrix&);
TermMatrix& operator=(const LcTerm&);

```

The `ReductionMethod` class allows the user to specify the reduction method he wants, one of the enumeration `ReductionMethodType`: `_pseudoReduction`, `_realReduction`, `_penalizationReduction`, `_dualReduction`. It is also possible to provide an additional real/complex parameter (say alpha) : the diagonal coefficient of the pseudo-eliminated part or the penalization coefficient.

```

ReductionMethodType method; //type of reduction method
complex_t alpha; //diagonal or penalization coefficient
ReductionMethod(ReductionMethodType= _noReduction, const complex_t& = 1.);
void print(std::ostream& out) const;
friend :ostream& operator << (ostream&, const ReductionMethod&);

```



When reduction method is not specified, the pseudo reduction with 1 as diagonal coefficient is used.
Only the pseudo reduction method is available!



When `TermMatrix` is constructed, it is automatically computed except if it has set to be not computed using a the `TermOption` `_notCompute` in its construction.

Developers may used other construction/destruction stuff:

```

void initFromBlf(const BilinearForm &, const String& na = "",
    bool noass = false);
void initTermVector(TermVector&, ValueType, bool col=true);
template<typename T>
void initTermVectorT(TermVector&,const T&, bool col=true);
void insert(const SuTermMatrix &);
void insert(SuTermMatrix *);
void copy(const TermMatrix&);
void clear();
virtual ~TermMatrix();

```

This class provides a lot of accessors and properties (shortcuts to other class accessors):

```
bool isSingleUnknown() const;
std::set<const Unknown*> rowUnknowns() const;
std::set<const Unknown*> colUnknowns() const;
Number nbTerms() const;
ValueType valueType() const;
AccessType storageAccess() const;
void setStorage(StorageType, AccessType);
StorageType storageType() const
bool hasConstraints() const;
FactorizationType factorization() const;
SymType globalSymmetry() const;
TermMatrix operator()(const Unknown&,const Unknown&) const;

//stuff reserved to developers
cit_mustm begin() const;
it_mustm begin();
cit_mustm end() const;
it_mustm end();
std::map<uvPair, SuTermMatrix*>& suTerms();
MatrixEntry*& scalar_entries();
const MatrixEntry* scalar_entries() const;
MatrixEntry*& rhs_matrix();
const MatrixEntry* rhs_matrix() const;
const std::vector<DofComponent>& cdofsr() const;
const std::vector<DofComponent>& cdofsc() const;
std::vector<DofComponent>& cdofsr();
std::vector<DofComponent>& cdofsc();
SuTermMatrix& subMatrix(const Unknown*,const Unknown*);
const SuTermMatrix& subMatrix(const Unknown*,const Unknown*) const;
SuTermMatrix* subMatrix_p(const Unknown*,const Unknown*);
const SuTermMatrix* subMatrix_p(const Unknown*,const Unknown*) const;

TermVector& getRowCol(number_t, AccessType, TermVector&) const;
TermVector row(number_t) const;
TermVector column(number_t) const;
template <typename T>
LargeMatrix<T>& getLargeMatrix(StrucType =_undefStrucType) const;
template <typename T>
HMatrix<T,FeDof>& getHMatrix(StrucType =_undefStrucType) const;
```



The access `operator()` returns a `SuTermMatrix` as a `TermMatrix` (obviously a single unknown one). Note that it is a full copy. To avoid copy, use `subMatrix` member functions.

`it_mustm` and `cit_mustm` are iterator aliases:

```
typedef std::map<uvPair, SuTermMatrix*>::iterator it_mustm;
typedef std::map<uvPair, SuTermMatrix*>::const_iterator cit_mustm;
```

It is also possible to get row or column (index starts from 1) of the `TermMatrix` if it is a single unknown one and to access to the `LargeMatrix` or the `HMatrix` object that really stores the matrix values:

```
TermVector& getRowCol(number_t, AccessType, TermVector&) const;
TermVector row(number_t) const;
TermVector column(number_t) const;
template <typename T>
LargeMatrix<T>& getLargeMatrix(StrucType =_undefStrucType) const;
template <typename T>
HMatrix<T,FeDof>& getHMatrix(StrucType =_undefStrucType) const;
```




The `setStorage` function allows the user to impose the matrix storage.

The class proposes to users interfaces to computation algorithms (FE computation and algebraic operations):

```
void compute();
void compute(const LcTerm&, const String& na="");

template<typename T> TermMatrix& operator*=(const T&);
template<typename T> TermMatrix& operator/=(const T&);
TermMatrix& operator+=(const TermMatrix&);
TermMatrix& operator-=(const TermMatrix&);
friend TermVector& multMatrixVector(const TermMatrix&, const TermVector&,
    TermVector&);
friend TermVector& multVectorMatrix(const TermMatrix&, const TermVector&,
    TermVector&);
friend TermVector operator*(const TermMatrix&, const TermVector&);
friend TermVector operator*(const TermVector&, const TermMatrix&);
friend TermMatrix operator*(const TermMatrix&, const TermMatrix&);
```

The computation of `TermMatrix` is a quite intricate process in case of essential conditions to apply to bilinear form. The implementation of `compute()` looks like (some details are omitted):

```
void TermMatrix::compute()
{
    //compute suterms
    for(it_mustm it = suTerms_.begin(); it != suTerms_.end(); it++)
        it->second->compute();
    if(constraints_u_p==0 && constraints_v_p==0) {computed() = true; return;}
    //deal with essential conditions
    bool global=false;
    if(constraints_u_p!=0) global=constraints_u_p->isGlobal();
    if(constraints_v_p!=0 && !global) global=constraints_u_p->isGlobal();
    if(global) toGlobal(_noStorage, _noAccess, _noSymmetry, false);
    else toScalar();
    switch(computingInfo_.elimination)
    {
        case _noElimination : break;
        case _pseudoElimination : pseudoElimination(); break;
    }
    computed() = true;
}
```

The principle of pseudo-elimination consists in combining some rows and columns, and "eliminating" some rows and columns (in fact set to 0 except diagonal coefficient). Two cases are to be considered:

- case of essential conditions which do not couple unknowns
- case of essential conditions which couples unknowns (global constraints)

In the first case, the pseudo-elimination process may be performed on each matrix of `SuTermMatrix` involving unknowns concerned by essential conditions whereas in the second case, the process must be performed on the whole matrix of `TermMatrix`. If no such representation exists, we have to create it (`toGlobal` function). See the `pseudoElimination()` function.

This class offers many interfaces to factorization tools and direct solvers :

```
//developers stuff
```

```

void initTermVector(TermVector&, ValueType, bool col=true);
template<typename T>
void initTermVectorT(TermVector&, const T&, bool col=true);

void toScalar(bool keepEntries=false);
void toGlobal(StorageType, AccessType, SymType=noSymmetry, bool keepSuTerms=false);
void toSkyline();
void mergeNumbering(std::map<const Unknown*, std::list<SuTermMatrix* > >&,
    std::map<SuTermMatrix*, std::vector<Number > >&, std::vector<DofComponent>&,
    AccessType);
void addIndices(std::vector<std::set<Number> >&, MatrixStorage*, const
    std::vector<Number>&, const std::vector<Number>&);
void mergeBlocks();
friend TermVector prepareLinearSystem(TermMatrix&, TermVector&, MatrixEntry*&,
    VectorEntry*&, bool toScal=false);
void pseudoElimination();

//direct solver member functions
void ldltSolve(const TermVector&, TermVector&);
void ldltstarSolve(const TermVector&,TermVector&);
void luSolve(const TermVector&,TermVector&);
void umfpackSolve(const TermVector&,TermVector&);
void sorSolve(const TermVector& V, TermVector& R, const Real w, SorSolverType sType
    );
void sorUpperSolve(const TermVector& V, TermVector& R, const Real w );
void sorDiagonalMatrixVector(const TermVector& V, TermVector& R, const Real w );
void sorLowerSolve(const TermVector& V, TermVector& R, const Real w );
void sorDiagonalSolve(const TermVector& V, TermVector& R, const Real w );

```

End users have to use the following external functions:

```

//factorization tool
void ldltFactorize(TermMatrix&, TermMatrix&);
void ldltstarFactorize(TermMatrix&, TermMatrix&);
void luFactorize(TermMatrix&, TermMatrix&);
void umfpackFactorize(TermMatrix&, TermMatrix&);
void factorize(TermMatrix&, TermMatrix&, FactorizationType ft=_noFactorization);

//direct solver (one right hand side)
TermVector factSolve(TermMatrix&, const TermVector&);
TermVector ldltSolve(TermMatrix&, const TermVector&);
TermVector ldltstarSolve(TermMatrix&, const TermVector&);
TermVector luSolve(TermMatrix&, const TermVector&);
TermVector gaussSolve(TermMatrix&, const TermVector&, bool keepA = false);
TermVector umfpackSolve(TermMatrix&, const TermVector&, bool keepA = false);
TermVector magmaSolve(TermMatrix& A, const TermVector& B, bool useGPU=false,
    bool keepA=false);
TermVector lapackSolve(TermMatrix& A, const TermVector& B, bool keepA=false);
TermVector directSolve(TermMatrix&, const TermVector&, bool keepA = false);
TermVector schurSolve(TermMatrix&, const TermVector&, const Unknown&, const
    Unknown&,
    bool keepA = false);

//direct solver (few right hand sides)
TermVectors factSolve(TermMatrix&, const vector<TermVector>&);
TermVectors ldltSolve(TermMatrix&, const vector<TermVector>&, TermMatrix&);
TermVectors ldltstarSolve(TermMatrix&, const vector<TermVector>&, TermMatrix&);
TermVectors luSolve(TermMatrix&, const vector<TermVector>&, TermMatrix&);
TermVectors gaussSolve(TermMatrix&, const vector<TermVector>&,bool keepA = false);
TermVectors umfpackSolve(TermMatrix&, const vector<TermVector>&,bool keepA = false);
TermVectors directSolve(TermMatrix&, const vector<TermVector>&,bool keepA = false);

// direct solver (TermMatrix right hand side)

```

```
TermMatrix factSolve(TermMatrix&, TermMatrix&);
TermMatrix directSolve(TermMatrix&, TermMatrix&, KeepStatus=_keep);
TermMatrix inverse(TermMatrix&);
```

`factorize()`, `factSolve()` and `directSolve` functions are general functions that determine automatically the adapted methods to apply.



to use `umfpackSolve`, `magmaSolve` or `lapackSolve`, libraries `umfpack`, `magma`, `lapack-blas` have to be installed and activated. `directSolve` choose `umfPack` solver when matrix is sparse and `umfpack` available, and `Lapack` solver when matrix is dense and `lapack` available. `umfpack` solver is always faster than `XLiFE++` solvers but `lapack` solver may be slower than the `XLiFE++` gauss solver if non optimized `lapack-blas` libraries are used

Note that the `factSolve` and `directSolve` functions can use a `TermMatrix` as right hand side. In that case, they compute $A^{-1}B$ using a factorization of A . The `inverse()` function computes the inverse of a `TermMatrix` using the `directSolve` function with the identity matrix as right hand side. These functions provide some `TermMatrix` that are stored as column dense storage. So be care with memory usage!



Up to now, the usage of `factSolve`, `directSolve` functions with a `TermMatrix` as right hand side and `inverse()` is restricted to single unknown `TermMatrix`.

In a same way, interfaces to various iterative solvers are provided :

```
//! general iterative solver
TermVector iterativeSolveGen(IterativeSolverType isType, TermMatrix& A,
    TermVector& B, const TermVector& X0, const PreconditionerTerm& P,
    const real_t tol, const number_t iterMax, const real_t omega,
    const number_t krylovDim, const number_t verboseLevel);
//! general front end for iterativeSolve
TermVector iterativeSolve(TermMatrix& A, TermVector& B, const TermVector& X0,
    const PreconditionerTerm& P, const std::vector<Parameter>& ps);

//! user front-end for iterative solver
inline TermVector iterativeSolve(TermMatrix& A, TermVector& B, const TermVector&
    X0, const PreconditionerTerm& P, 0 to 5 Parameter);
inline TermVector iterativeSolve(TermMatrix& A, TermVector& B, const
    PreconditionerTerm& P, 0 to 6 Parameter);
inline TermVector iterativeSolve(TermMatrix& A, TermVector& B, const TermVector&
    X0, 0 to 6 Parameter);
inline TermVector iterativeSolve(TermMatrix& A, TermVector& B, 0 to 6 Parameter);

//! user front-end for BiCG solver
inline TermVector bicgSolve(TermMatrix& A, TermVector& B, const TermVector& X0,
    const PreconditionerTerm& P, 0 to 4 Parameter);
inline TermVector bicgSolve(TermMatrix& A, TermVector& B, const PreconditionerTerm&
    P, 0 to 4 Parameter);
inline TermVector bicgSolve(TermMatrix& A, TermVector& B, const TermVector& X0, 0
    to 4 Parameter);
inline TermVector bicgSolve(TermMatrix& A, TermVector& B, 0 to 4 Parameter);

//! user front-end for BiCGStab solver
inline TermVector bicgStabSolve(TermMatrix& A, TermVector& B, const TermVector& X0,
    const PreconditionerTerm& P, 0 to 4 Parameter);
inline TermVector bicgStabSolve(TermMatrix& A, TermVector& B, const
    PreconditionerTerm& P, 0 to 4 Parameter);
inline TermVector bicgStabSolve(TermMatrix& A, TermVector& B, const TermVector& X0,
```

```

    0 to 4 Parameter);
inline TermVector bigStabSolve(TermMatrix& A, TermVector& B, 0 to 4 Parameter);

///! user front-end for CG solver
inline TermVector cgSolve(TermMatrix& A, TermVector& B, const TermVector& X0, const
    PreconditionerTerm& P, 0 to 4 Parameter);
inline TermVector cgSolve(TermMatrix& A, TermVector& B, const PreconditionerTerm&
    P, 0 to 4 Parameter);
inline TermVector cgSolve(TermMatrix& A, TermVector& B, const TermVector& X0, 0 to
    4 Parameter);
inline TermVector cgSolve(TermMatrix& A, TermVector& B, 0 to 4 Parameter);

///! user front-end for CGS solver
inline TermVector cgsSolve(TermMatrix& A, TermVector& B, const TermVector& X0,
    const PreconditionerTerm& P, 0 to 4 Parameter);
inline TermVector cgsSolve(TermMatrix& A, TermVector& B, const PreconditionerTerm&
    P, 0 to 4 Parameter);
inline TermVector cgsSolve(TermMatrix& A, TermVector& B, const TermVector& X0, 0 to
    4 Parameter);
inline TermVector cgsSolve(TermMatrix& A, TermVector& B, 0 to 4 Parameter);

///! user front-end for GMRes solver
inline TermVector gmresSolve(TermMatrix& A, TermVector& B, const TermVector& X0,
    const PreconditionerTerm& P, 0 to 4 Parameter);
inline TermVector gmresSolve(TermMatrix& A, TermVector& B, const
    PreconditionerTerm& P, 0 to 4 Parameter);
inline TermVector gmresSolve(TermMatrix& A, TermVector& B, const TermVector& X0, 0
    to 4 Parameter);
inline TermVector gmresSolve(TermMatrix& A, TermVector& B, 0 to 4 Parameter);

///! user front-end for QMR solver
inline TermVector qmrSolve(TermMatrix& A, TermVector& B, const TermVector& X0,
    const PreconditionerTerm& P, 0 to 4 Parameter);
inline TermVector qmrSolve(TermMatrix& A, TermVector& B, const PreconditionerTerm&
    P, 0 to 4 Parameter);
inline TermVector qmrSolve(TermMatrix& A, TermVector& B, const TermVector& X0, 0 to
    4 Parameter);
inline TermVector qmrSolve(TermMatrix& A, TermVector& B, 0 to 4 Parameter);

```

iterativeSolve(...) are the front end of all other iterative methods.

iterativeSolveGen(...) are the front end of all iterative methods.

To compute eigenvalues, internal solvers or interface to arpack solvers are available:

```

///! general eigen solver
EigenElements eigenSolveGen(TermMatrix* pA, TermMatrix* pB, number_t nev, string_t
    which, real_t tol, EigenComputationalMode eigCompMode,
    complex_t sigma, const char mode, EigenSolverType
    solver);

///! general eigen solver
EigenElements eigenInternGen(TermMatrix* pA, TermMatrix* pB, number_t nev, string_t
    which, real_t tol, EigenComputationalMode eigCompMode,
    complex_t sigma, bool isShift);

///! general front-end for generalized eigen solver
EigenElements eigenSolve(TermMatrix& A, TermMatrix& B, std::vector<Parameter> ps);

///! user front-end for eigen solver
EigenElements eigenSolve(TermMatrix& A, TermMatrix& B, 0 to 10 Parameter);
EigenElements eigenSolve(TermMatrix& A, 0 to 10 Parameter);

///! user front-end for internal eigen solver

```

```

inline EigenElements eigenInternSolve(TermMatrix& A, TermMatrix& B, 0 to 9
    Parameter);
inline EigenElements eigenInternSolve(TermMatrix& A, 0 to 9 Parameter);

//! user front-end for external Arpack eigen solver
inline EigenElements arpackSolve(TermMatrix& A, TermMatrix& B, 0 to 9 Parameter);
inline EigenElements arpackSolve(TermMatrix& A, 0 to 9 Parameter);

```

The **EigenElements** class handles the list of eigenvalues and the list of eigenvectors. See the **TermVector** section.



arpack solver are available only if the arpack lib is installed.

When addressing integral representations, say

$$r(x) = \int_{\Sigma} op(K)(x, y) * op(u)(y) dy$$

one can be interested in the matrix with coefficients

$$r_{ij} = \int_{\Sigma} op(K)(x_i, y) * op(w_j)(y) dy$$

where $(x_i)_i$ are some points and $(w_j)_j$ some shape functions related to the unknown u . Special functions, named **integralFunction** will produce such matrices embedded in a **TermMatrix** object:

```

TermMatrix integralRepresentation(const Unknown&, const GeomDomain&,
    const LinearForm&);
TermMatrix integralRepresentation(const GeomDomain&, const LinearForm&);
TermMatrix integralRepresentation(const std::vector<Point>&, const LinearForm&);

```

When no domain is explicitly passed, one shadow **PointsDomain** object will be created from the list of points given. When no unknown is explicitly passed, one (minimal) space and related shadow unknown will be created to represent the row unknown.

These functions call the fundamental computation function:

```

template<typename T, typename K>
void SuTermMatrix::computeIR(const SuLinearForm&f, T*, K&, const
    std::vector<Point>& xs);

```

Finally some print and export functions are provided:

```

//! as member function
void print(std::ostream&) const;
void viewStorage(std::ostream&) const;
void saveToFile(const String&, StorageType, bool=false) const;
//! as non member function
void saveToFile(const TermMatrix&, const String&, StorageType,
    bool enc = false);

```

library :	term
header :	TermMatrix.hpp
implementation :	TermMatrix.cpp
unitary tests :	test_TermMatrix.cpp
header dependences :	SuTermMatrix.hpp, termUtils.hpp, Term.hpp, form.h, config.h, utils.h

11.7 The SuTermMatrix class

The **SuTermMatrix** class carries numerical representation of a single unknown pair bilinear form and more generally any matrix attached to a pair of unknowns (row unknown and row column).



By convention, 'u' refers to column (unknown of bilinear form) and 'v' to row (test function in bilinear form).

```
class SuTermMatrix : public Term
{protected :
    SuBilinearForm* sublf_p;           //bilinear form through pointer
    const Unknown* u_p;               //column unknown
    const TestFunction* v_p;          //row unknown
    mutable Space* space_u_p;         //pointer to u-space
    mutable Space* space_v_p;         //pointer to v-space
    std::vector<Space*> subspaces_u;   //u-subspaces involved
    std::vector<Space*> subspaces_v;   //v-subspaces involved
    MatrixEntry* entries_p;           //matrix as LargeMatrix
    MatrixEntry* scalar_entries_p;     //scalar matrix as LargeMatrix
    std::vector<DofComponent> cdofs_u; //component u-dof list
    std::vector<DofComponent> cdofs_v; //component v-dof list
    MatrixEntry* rhs_matrix_p;        //correction matrix when essential cond.
}
```

The **sublf_p** contains a pointer to a **SuBiLinearForm** (single unknown pair bilinear form). It may be null, that means there is no explicit bilinear form attached to the **SuTermMatrix**.

The **space_u_p** (resp. **space_v_p**) pointer points to the largest subspace involved in **SuTermMatrix** columns (resp. rows). These pointer should never be null because they carry the row and column dofs numbering. For this reason too, the largest subspaces have to be correctly updated during any operation on **SuTermMatrix**. The **Space*** vector **subspaces_u** (resp. **subspaces_v**) contains the subspaces (as subspace of largest space) attached to basic bilinear forms defined in the **SuBiLinearForm** pointer (if defined), see **buildSubspaces** function.



Do not confused the **space_u_p** pointer and **u_p->space_p** pointer that specifies the whole space. They may be same!

The numerical representation of matrix is defined in the **entries_p** **MatrixEntry** pointer as a real/complex matrix or a real/complex matrix of matrices, regarding the value type (real or complex) and the structure of the unknowns (scalar or vector). If required, matrix of matrices representation may be expanded to a matrix of scalars stored in the **scalar_entries_p** **MatrixEntry** pointer. In that case, the **cdofs_u** and **cdofs_v** vectors of **DofComponent** give the row and column numbering (in **DofComponent**) of entries. Note that if **SuTermMatrix** is of scalar type, **scalar_entries_p** = **entries_p**.

In case of essential conditions applied to, **rhs_matrix_p** **MatrixEntry** pointer may be allocated to store the eliminated part of **SuTermMatrix**. Note that, **SuTermMatrix** class does not manage **SetOfConstraints** objects, the treatment of essential condition being driven by **TermMatrix**!

The **SuTermMatrix** class also handles HMatrix (hierarchical matrix):

```
HMatrixEntry<FeDof>* hm_entries_p; //hierarchical matrix
HMatrixEntry<FeDof>* hm_scalar_entries_p; //scalar hierarchical matrix
ClusterTree<FeDof>* cluster_u;      //column cluster tree
ClusterTree<FeDof>* cluster_v;      //row cluster tree
```

HMatrix stuff is described in chapter 10. Let's remember that the hierarchical matrix are built from the clustering of row and column indices (**cluster_u**, **cluster_v**). There are two HMatrixEntry pointers, one to handle any matrix (scalar or matrix), the other one to handle the scalar representation of a

matrix of matrices if it is required. When HMatrix is a scalar one, both the pointers are the same.

The `SuTermMatrix` class provides some constructors from bilinear form, a constructor from `SuTermVector` (diagonal matrix), a constructor from linear combination and a copy constructor:

```
// basic constructor
SuTermMatrix(const Unknown*, Space*, const Unknown*, Space*, MatrixEntry*, const
    String& na="");
//constructor from bilinear form
SuTermMatrix(SuBilinearForm* sublf = 0, const String& na="", bool noass=false);
SuTermMatrix(SuBilinearForm* sublf = 0, const String& na="",
    ComputingInfo cp= ComputingInfo());
SuTermMatrix(SuBilinearForm*, const Unknown*, const Unknown*, Space*, Space*,
    const std::vector<Space*>&, const std::vector<Space*>&,
    const String&, MatrixEntry* =0);
//constructor of diagonal matrix
SuTermMatrix(SuTermVector &, const String& na="");
SuTermMatrix(const Unknown*, Space*, const Unknown*, Space*, SuTermVector&,
    StorageType=_noStorage, AccessType=_noAccess, const String& na="");
void diagFromSuTermVector(const Unknown*, Space*, const Unknown*,Space*,
    SuTermVector&,
        StorageType=_noStorage, AccessType=_noAccess,
        const String& na="");
//constructor from linear combination
SuTermMatrix(const LcTerm&,const String& na="");
//copy constructor and assign operator
SuTermMatrix(const SuTermMatrix&,const String& na="");
void copy(const SuTermMatrix&);
SuTermMatrix& operator=(const SuTermMatrix&);
//destructor, clear
virtual ~SuTermMatrix();
void clear();
```

There is no specific constructor for `SuTermMatrix` of type `Hmatrix`. It is induced by some properties of the bilinear form; more precisely by a particular choice of integration method.

It provides a lot of accessors and property functions:

```
TermType termType() const;
ValueType valueType() const;
StrucType strucType() const;
Space& space_u() const;
Space& space_v() const;
Space* space_up() const;
Space* space_vp() const;
Space*& space_up();
Space*& space_vp();
const Unknown* up() const;
const Unknown* vp() const;
const Unknown*& up();
const Unknown*& vp();
MatrixEntry*& entries();
const MatrixEntry* entries() const;
MatrixEntry*& scalar_entries();
const MatrixEntry* scalar_entries() const;
MatrixEntry*& rhs_matrix();
const MatrixEntry* rhs_matrix() const;
const std::vector<DofComponent>& cdofsu() const;
const std::vector<DofComponent>& cdofsv() const;
std::vector<DofComponent>& cdofsu();
std::vector<DofComponent>& cdofsv();
SymType symmetry() const;
```

```

StorageType storageType() const;
MatrixStorage* storage() const;
void setStorage(StorageType, AccessType);
void toStorage(StorageType, AccessType);

```

The main computation functions and related stuff follow:

```

void compute();
void compute(const LcTerm&, const String& na="");
template<unsigned int CM>
void compute(const std::vector<SuBilinearForm>&, ValueType, StrucType);
template<typename T, typename K>
void computeIR(const SuLinearForm&f, T*, K&, const std::vector<Point>&);

void buildSubspaces();
void setStorage(StorageType, AccessType);
void toStorage(StorageType, AccessType);
void addDenseToStorage(const std::vector<SuBilinearForm>&, MatrixStorage*) const;
void toScalar(bool keepEntries=false);
std::vector<SuBilinearForm> getSublfs(ComputationType, ValueType&) const;
void updateStorageType(const std::vector<SuBilinearForm>&, std::set<Number>&,
                      std::set<Number>&, StorageType&) const;

```

`SuTermMatrix` may be modified by the following algebraic operators:

```

template<typename T>
SuTermMatrix& operator*=(const T&);
template<typename T>
SuTermMatrix& operator/=(const T&);
SuTermMatrix& operator+=(const SuTermMatrix&);
SuTermMatrix& operator-=(const SuTermMatrix&);
friend SuTermVector operator*(const SuTermMatrix&, const SuTermVector&);
friend SuTermMatrix operator*(const SuTermMatrix&, const SuTermMatrix&);

```

Finally, there are some print and export stuff:

```

void print(std::ostream&) const;
void print(std::ostream&, bool) const;
void viewStorage(std::ostream&) const;
void saveToFile(const String&, StorageType, bool=false) const;

```

Most of the previous functions works when `SuTermMatrix` handles a `HMatrix`, but some not. As there is no check, be cautious! Besides, there are some specific functions related to `Hmatrix` representation:

```

bool hasHierarchicalMatrix() const;
HMatrix<T, FeDof>& getHMatrix(StrucType str=_undefStrucType) const;

```

library :	<code>term</code>
header :	<code>SuTermMatrix.hpp</code>
implementation :	<code>SuTermMatrix.cpp</code>
unitary tests :	<code>test_TermMatrix.cpp</code>
header dependencies :	<code>Term.hpp</code> , <code>LcTerm.hpp</code> , <code>termUtils.hpp</code> , <code>form.h</code> , <code>config.h</code> , <code>utils.h</code>

11.8 The LcTerm class

The `LcTerm` class is a template class handling a linear combination of terms of type `TermVector`, `TermMatrix`, `SuTermVector` or `SuTermMatrix`. It is automatically involved as a result whenever any linear combination of `Term`'s is handled:


```
TermVector tv1, tv2;
...
out<<norm2(tv1-tv2);
```

In this example `tv1-tv2` produces a `LcTerm<TermVector>` that is cast implicitly to a `TermVector` because the definition of a constructor of `TermVector` from a `LcTerm<TermVector>`. Then any functions defined on `TermVector` may be applied, the `norm2` function in this example.

The `LcTerm` class inherits from `vector<pair<const TT *, complex_t >>`

```
template <typename TT>
class LcTerm : public vector<pair<const TT *, complex_t> >
{
public:
    string_t nametype;
    typedef typename vector<pair<const TT *, complex_t> >::iterator iterator;
    typedef typename vector<pair<const TT *, complex_t> >::const_iterator
        const_iterator;
```

It has template constructors and insertion functions (`push_back`):

```
template<typename T>
    LcTerm(const TT*,const T&);
    LcTerm(const TT*,const T&, const TT*, const T&);
    LcTerm(const TT&,const T&);
    LcTerm(const TT&,const T&, const TT&, const T&);
    LcTerm(const vector<const TT *>&, const vector<T>&);
    void push_back(const Term*,const T&);
    void push_back(const Term&,const T&);
```

and to avoid some `std::vector` member functions are overloaded:

```
number_t size() const ;
const_iterator begin();
const_iterator end();
iterator begin();
iterator end();
```

It supports the following algebraic operations:

```
template<typename T>
    LcTerm<TT>& operator*=(const T&);
    LcTerm<TT>& operator/=(const T&);
    LcTerm<TT>& operator+=(const LcTerm<TT>&);
    LcTerm<TT>& operator-=(const LcTerm<TT>&);
```

and provides the following utilities

```
ValueType coefType() const;
void print(ostream &) const;
template <typename TX>
friend ostream& operator<<(ostream &,const LcTerm<TX>&)
```

Note that `TermVector` and `TermMatrix` classes have functions that creates some `LcTerm` objects.

library :	term
header :	LcTerm.hpp
implementation :	
unitary tests :	
header dependences :	Term.hpp, config.h, utils.h

11.9 The SymbolicTermMatrix class

Sometimes, numerical methods involve more complex combinations of `TermMatrix` than linear combinations, for instance $\mathbb{A} = \mathbb{M} + \mathbb{K}\mathbb{M}^{-1}\mathbb{K}^t$ where \mathbb{M} and \mathbb{K} are sparse matrices. Generally, it is not a good idea to compute \mathbb{A} because the result is a dense matrix. The purpose of the `SymbolicTermMatrix` class is to describe as symbolic this matrix, not to compute it but to compute $\mathbb{A}X$ with X a `TermVector`. A `SymbolicTermMatrix` object is a node of the binary tree with the following data

- a symbolic operation (one of id, +, -, *, /, inv, tran, conj, adj)
- a `TermMatrix` object (pointer to), may be 0
- up to two `SymbolicTermMatrix` objects (pointer to)
- a coefficient (complex scalar) applied to the operation

Using these data, the symbolic form of the \mathbb{A} matrix is presented on figure 11.1. On a node, there is either a `TermMatrix` (leaves) or a pair of `SymbolicTermMatrix`.

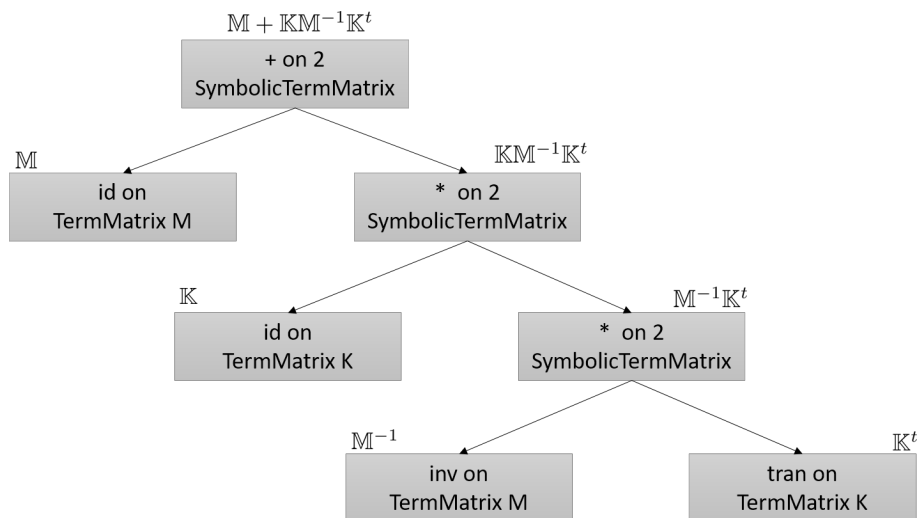


Figure 11.1: Tree representation of matrix $\mathbb{M} + \mathbb{K}\mathbb{M}^{-1}\mathbb{K}^t$

So the `SymbolicTermMatrix` class has the following data members

```

class SymbolicTermMatrix
{
public:
    SymbolicTermMatrix* st1_, *st2_;
    const TermMatrix* tm_;
    complex_t coef_;
    SymbolicOperation op_;
    bool delMat_;
}
  
```

the following constructors and destructor stuff

```

SymbolicTermMatrix();
SymbolicTermMatrix(const TermMatrix&, const complex_t& =complex_t(1.,0.));
SymbolicTermMatrix(SymbolicOperation, SymbolicTermMatrix*, SymbolicTermMatrix*=0);
SymbolicTermMatrix(LcTerm<TermMatrix>& lc);
~SymbolicTermMatrix();
void add(LcTerm<TermMatrix>&, std::vector<std::pair<const TermMatrix*, complex_t>
>::iterator);
SymbolicTermMatrix(const SymbolicTermMatrix&);
SymbolicTermMatrix& operator=(const SymbolicTermMatrix&);

```

The `delMat` variable is a flag telling the `TermMatrix` can be deleted because one of the constructors creates a copy of the original one.

```

SymbolicTermMatrix();
SymbolicTermMatrix(const TermMatrix&, const complex_t& =complex_t(1.,0.));
SymbolicTermMatrix(SymbolicOperation, SymbolicTermMatrix*, SymbolicTermMatrix*=0);
SymbolicTermMatrix(LcTerm<TermMatrix>& lc);
~SymbolicTermMatrix();
void add(LcTerm<TermMatrix>&, std::vector<std::pair<const TermMatrix*, complex_t>
>::iterator);
SymbolicTermMatrix(const SymbolicTermMatrix&);
SymbolicTermMatrix& operator=(const SymbolicTermMatrix&);

```

The class offers classic print stuff :

```

string_t asString() const;
void print(std::ostream&) const;
void print(PrintStream& os) const {print(os.currentStream());}
friend std::ostream& operator<<(std::ostream& ,const SymbolicTermMatrix&)

```

Besides a lot of operator are provided :

```

SymbolicTermMatrix& operator *(const TermMatrix&,SymbolicTermMatrix&);
SymbolicTermMatrix& operator *(SymbolicTermMatrix&, const TermMatrix&);
SymbolicTermMatrix& operator +(const TermMatrix&,SymbolicTermMatrix&);
SymbolicTermMatrix& operator +(SymbolicTermMatrix&, const TermMatrix&);
SymbolicTermMatrix& operator -(const TermMatrix&,SymbolicTermMatrix&);
SymbolicTermMatrix& operator -(SymbolicTermMatrix&, const TermMatrix&);
SymbolicTermMatrix& operator +(LcTerm<TermMatrix>&,SymbolicTermMatrix&);
SymbolicTermMatrix& operator +(SymbolicTermMatrix&, LcTerm<TermMatrix>&);
SymbolicTermMatrix& operator -(LcTerm<TermMatrix>&,SymbolicTermMatrix&);
SymbolicTermMatrix& operator -(SymbolicTermMatrix&, LcTerm<TermMatrix>&);
SymbolicTermMatrix& operator *(LcTerm<TermMatrix>&,SymbolicTermMatrix&);
SymbolicTermMatrix& operator *(SymbolicTermMatrix&, LcTerm<TermMatrix>&);
SymbolicTermMatrix& operator *(SymbolicTermMatrix&, SymbolicTermMatrix&);
SymbolicTermMatrix& operator +(SymbolicTermMatrix&, SymbolicTermMatrix&);
SymbolicTermMatrix& operator -(SymbolicTermMatrix&, SymbolicTermMatrix&);
SymbolicTermMatrix& conj(SymbolicTermMatrix&);
SymbolicTermMatrix& adj(SymbolicTermMatrix&);
SymbolicTermMatrix& tran(SymbolicTermMatrix&);
SymbolicTermMatrix& operator *(SymbolicTermMatrix&, const complex_t&);
SymbolicTermMatrix& operator *(const complex_t&, SymbolicTermMatrix&);
SymbolicTermMatrix& operator /(SymbolicTermMatrix&, const complex_t&);
SymbolicTermMatrix& inv(const TermMatrix&);
SymbolicTermMatrix& inv(SymbolicTermMatrix&);

```

Because some syntaxes may be ambiguous,the operator `~` is overloaded in order to move a `TermMatrix` to a `SymbolicTermMatrix`:

```

SymbolicTermMatrix& operator~(const TermMatrix&):

```

The most important functions are function that compute recursively the matrix vector product $\mathbb{A}X$:

```
TermVector operator*(const SymbolicTermMatrix&, const TermVector&);
TermVector multMatrixVector(const SymbolicTermMatrix&, const TermVector&);
TermVector operator*(const TermVector&, const SymbolicTermMatrix&);
TermVector multVectorMatrix(const TermVector&, const SymbolicTermMatrix&);
```

library :	term
header :	SymbolicTermMatrix.hpp
implementation :	SymbolicTermMatrix.cpp
unitary tests :	unit_TermMatrix.cpp
header dependences :	TermMatrix.hpp, config.h, utils.h

11.10 The computation algorithms

11.10.1 Matrix computation

In this section, we explain how the computation of matrix from bilinear form works. This computation depends on the type of the bilinear form (single integral term, double integral term, ...) and the types of unknown and test function. This section addresses only the computation of **SuTermMatrix**.

The principle of computation, implemented in **SuTermMatrix::compute()**, consists in 4 steps:

- collect basic bilinear forms along their domain and their computation type (see **SuTermMatrix::getSuBlfs()** function.):

_FEComputation	single integral on a domain	u,v in FE spaces
_FEextComputation	single integral on a boundary domain with non tangential derivatives	u,v in FE spaces
_IEComputation	double integral on a domains pair	u,v in FE space, standard Kernel
_SPComputation	single integral on a domain	u,v in SP spaces
_FESPComputation	single integral on a domain	one in SP space, other in FE space
_IESPComputation	double integral on a domains pair	u,v in FE space, TensorKernel
_IEHMComputation	double integral on a domains pair	u,v in FE space, HMatrix

- construct a storage consistent with all collections to compute (imposed by user or chosen between compressed sparse storage or dense storage)
- create **MatrixEntry** with the right structure and value types
- for each collection, call the computation function regarding its computation type

This last step looks like in **SuTermMatrix::compute()**:

```
if (FESublfs.size() > 0) compute<_FEComputation>(FESublfs, vt, str);
if (FEextSublfs.size() > 0) compute<_FEextComputation>(FEextSublfs, vt, str);
if (SPSublfs.size() > 0) compute<_SPComputation>(SPSublfs, vt, str);
if (FESPSublfs.size() > 0) compute<_FESPComputation>(FESPSublfs, vt, str);
if (IESublfs.size() > 0) compute<_IEComputation>(IESublfs, vt, str);
if (IESPSublfs.size() > 0) compute<_IESPComputation>(IESPSublfs, vt, str);
```

where all specializations of `compute<unsigned int CM>` are implemented in the *XXMatrixComputation.hpp* files included add the end of the *SuTermMatrix.hpp* file. All the computation algorithms mainly work as follows:

```

retry general data (spaces, subspaces, domains, ...)
loop on element on domain 1
  retry element data (local to global numbering)
  (loop on element on domain 2)      only for IE and IESP computation
  (retry element data)
  compute shapevalues
  loop on basic bilinear form
    loop on quadrature points
      compute operator on unknowns
      add in elementary matrix using tensorial operations
      assembly in global matrix

```



For sake of simplicity, evaluation of operators returns always a **scalar vector**, even if the unknowns are vector ones. So this vector has to be correctly reinterpreted when it is added in elementary block matrix.

When there is a HMatrix computation that requires a specific representation (HMatrixEntry), this one is achieved after all the computations requiring a MatrixEntry. Then the MatrixEntry (if it exists one) is merged in the HMatrixEntry. Obviously, anything cannot be merged to HMatrixEntry!

For IE computation, some specialized function are devoted to particular integration method adapted to singular kernels (Sauter-Schwab method for instance). They are implemented in particular files (*SauterSchwabbIM.hpp*, *LenoirSallesM.hpp*, *DuffyIM.hpp*), see the *src/term/computation* directory.

Up to now, most of the computation functions have been parallelized (multi-threading with omp).

11.10.2 Understanding TermMatrix operations

The following figure sketches out how the operations are processed in the class hierarchy from [TermMatrix](#) to storage classes that implement the computation algorithms:

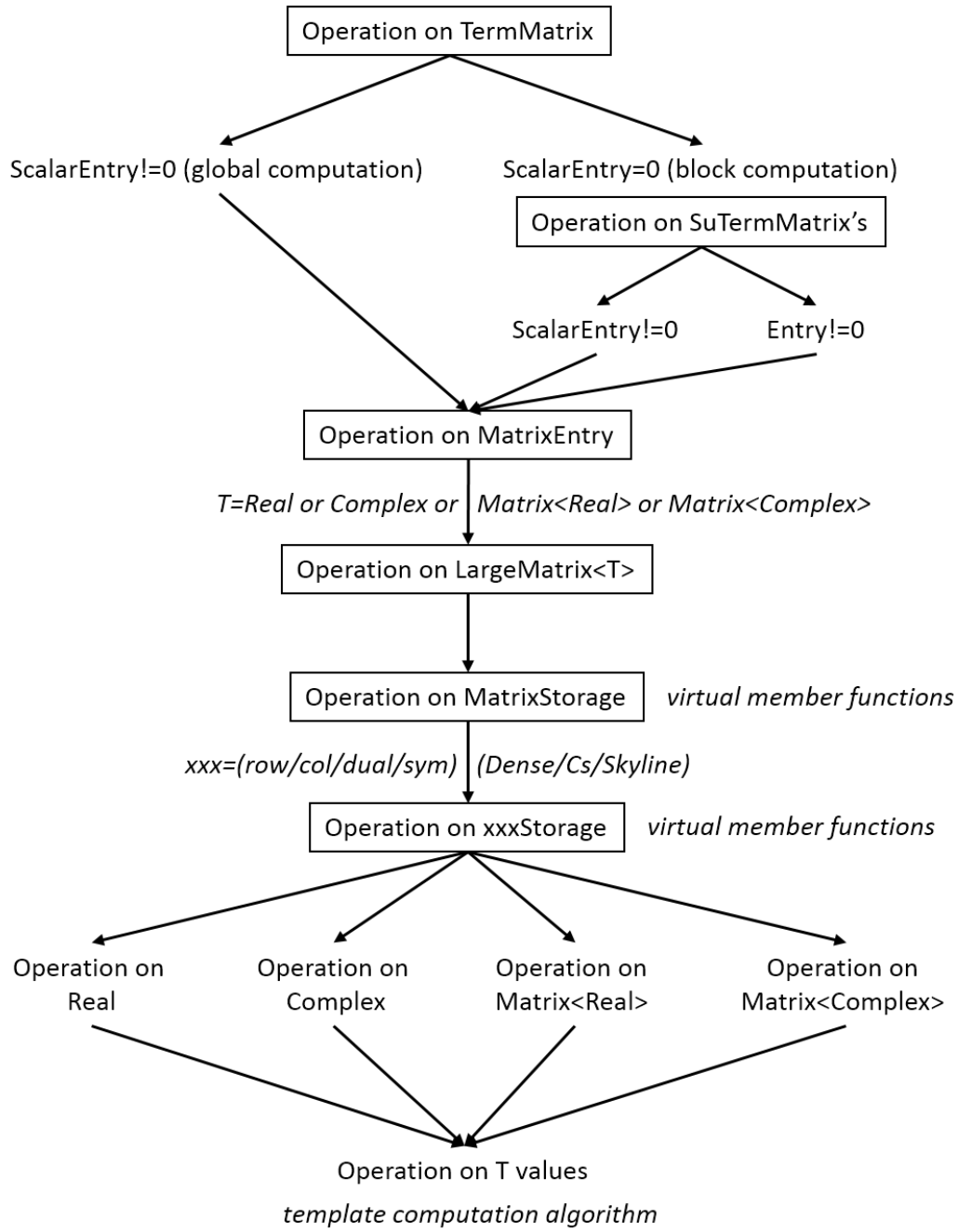


Figure 11.2: The process of a TermMatrix operation

11.10.3 Vector computation

For the moment, the `SuTermVector` class provides

- the `computeFE` function that computes single integral on a FE space
- the `computeIR` function that computes integral representation

These functions are implemented in the `FeVectorComputation.hpp` file included by the `SuTermVector.hpp` file. It works as matrix computation but it is simpler.

11.11 The Projector class

The **Projector** class is intended to handle projection from one FE space, say V to an other FE space, say W . The projection $w \in W$ of a $v \in V$ is based on a bilinear form, say $a(.,.)$, defined on both spaces :

$$a(w, \tilde{w}) = a(v, \tilde{w}) \quad \forall \tilde{w} \in W.$$

Let $(w_i)_{i=1,n}$ a basis of W and $(v_i)_{i=1,m}$ a basis of V , the above problem is equivalent to the matrix problem:

$$\mathbb{A} W = \mathbb{B} V$$

where $\mathbb{A}_{ij} = a(w_j, w_i)$, $\mathbb{B}_{ij} = a(v_j, w_i)$, $v = \sum_{i=1,m} V_i v_i$ and $w = \sum_{i=1,n} W_i w_i$. The bilinear form should be symmetric and positive on the space W in order to the matrix \mathbb{A} be invertible. The most simple example is the L^2 projection related to the bilinear form:

$$a(w, \tilde{w}) = \int_{\Omega} w \tilde{w} d\Omega.$$

As a consequence, the **Projector** class manages the following data:

```
class Projector
{
public:
    ProjectorType projectorType;    //!< type of projection
    string_t name;                  //!< name of projection
protected:
    Space* V_, *W_;                 //!< spaces involved
    Unknown* u_V, *u_W;             //!< internal unknowns
    const GeomDomain* domain_;      //!< domain involved in bilinear forms
    BilinearForm* a_V, *a_W;        //!< bilinear forms used by projector
    mutable TermMatrix* A_, *B_;    //!< matrix a(W,W) and a(W,V)
    mutable TermMatrix* invA_B;     //!< matrix inv(A)*B if allocated
}
```

where the projector type is one of the following:

```
enum ProjectorType {_noProjectorType=0, _userProjector, _L2Projector, _H1Projector,
                    _H10Projector};
```

Because, bilinear forms in XLIFF++ are linked to unknowns/spaces, the class manages two bilinear forms and their matrix representation. The unknowns used by the **Projector** are not some user unknowns but are specific to the class. Finally, it can allocate a **TermMatrix** object representing the matrix $\mathbb{A}^{-1} \mathbb{B}$.

Besides, in order to save time and memory, all the projectors that have been handled (explicitely or implicitely) are listed in a static vector:

```
static std::vector<Projector*> theProjectors;
```

Projector are constructed from spaces, a projector type or a bilinear form and an optional name. When dealing with the restriction of a space to a domain, the domain has to be given. When dealing with projection of vector unknown, the sizes have to be specified.

```
Projector(Space& V, Space& W, ProjectorType pt=_L2Projector,
          const string_t& na="");
Projector(Space& V, Space& W, const GeomDomain&, ProjectorType pt=_L2Projector,
          const string_t& na="");
Projector(Space& V, Space& W, BilinearForm& a, const string_t& na="");
Projector(Space& V, dimen_t nbcV, Space& W, dimen_t nbcW,
          ProjectorType pt=_L2Projector, const string_t& na="");
```

```

Projector(Space& V, dimen_t nbcV, Space& W, dimen_t nbcW, const GeomDomain&,
          ProjectorType pt=_L2Projector, const string_t& na="");
Projector(Space& V, dimen_t nbcV, Space& W, dimen_t nbcW, BilinearForm& a,
          const string_t& na="");
~Projector();
void init(dimen_t nbc_V, dimen_t nbc_W);

```

The `init()` function do really the job of construction. In particular, it computes the matrix `A_` and `B_` and do a factorization of `A_`.

Note that the `TermMatrix` is not allocated at the construction. To allocate it, one has to call the member function:

```

TermMatrix& asTermMatrix(const Unknown&, const Unknown&,
                        const string_t& = "invA_B");

```

that returns a reference to the `TermMatrix` `invA_B` and deletes the `TermMatrix` `A_` and `B_`.

Once constructed, projectors can process `TermVector`'s by using one of the three operator functions:

```

TermVector operator()(TermVector& V, const Unknown& u) const;
TermVector operator()(const TermVector& V) const;
TermVector& operator()(TermVector& V, TermVector& PV) const;

```

Because an unknown is always linked to any `TermVector`, the user can pass the unknown either in a explicit way or in a implicit way when the `TermVector` result is passed to the operator. When no unknown is passed to, the unknown of the projection will be the first unknown linked to the space `W`. If the `TermMatrix` `invA_B` is allocated, the projection does the product `invA_B * V`, if not the projection does the product `B_ * V` and solve the linear system `A_ * X = B_ * V` using the `factSolve` functions.

Projection can also be processed by using the operator `*`:

```

friend TermVector operator*(const Projector& P, const TermVector& V);

```

that only calls `P(V)`.

Users can compute the projection of a `TermVector` without instanciating a `Projector` object:

```

TermVector projection(const TermVector& V, Space& W, ProjectorType pt=_L2Projector);

```

In that case, a `Projector` object is created in the back.

Two functions deal with the list of all projectors:

```

friend Projector& findProjector(Space& V, Space& W,
                               ProjectorType pt=_L2Projector);
static void clearGlobalVector();

```

The `findProjector` creates a new projector if projector has not been found.

Finally, there are some print stuff:

```

void print(ostream&) const;
friend ostream& operator<<(ostream&, const Projector&);

```

library :	term
header :	Projector.hpp
implementation :	Projector.cpp
unitary tests :	unit_Projector.cpp
header dependences :	Term.hpp, config.h, utils.h

To take into account linear essential conditions (conditions acting on unknown or test function spaces), XLIFE++ proposes a general approach based on an abstract algebraic representation of essential conditions, say

$$\mathbb{C}U = G$$

where U represents the vector of components of a function u in space $\mathcal{U} = \left\{ \sum_{i=1,n} U_i w_i \right\}$, $(w_i)_{i=1,n}$ the basis of discrete space, \mathbb{C} a $p \times n$ matrix and G a p vector. In other words, we consider the discrete constrained space :

$$\tilde{\mathcal{U}} = \left\{ \sum_{i=1,n} U_i w_i, \mathbb{C}U = G \right\}. \quad (12.1)$$

Such representation may be also used for constraints on test function :

$$\tilde{\mathcal{V}} = \left\{ \sum_{i=1,m} V_i \tau_i, \mathbb{D}V = 0 \right\}. \quad (12.2)$$

Both may exist in the same time and may be different. Nevertheless, constraints on test functions are always homogeneous ones! In usual cases, the test function constraints are the linear part of unknown constraints ($\mathbb{C} = \mathbb{D}$).

12.1 A general process

The first step consists in writing essential conditions given in a symbolic form (see documentation of [EssentialConditions](#) class) as a constraints system, details being exposed in documentation of [Constraints](#) class). As constraints may be redundant, the constraints system has to be reduced to a minimal system, eliminating redundant conditions or identifying conflicts in constraints. This is achieved using QR factorization. Once the constraints systems are reduced, the main work consists to use them in matrix problems we deal with. As we will see, it is possible to consider different techniques : *real reduction* (some rows and columns of system are deleted), *pseudo reduction* (some rows and columns of system are replaced), *penalization* (some rows and columns of system are penalized) and *duality* (constraints system and new dual unknowns are introduced in the problem).

Reinterpretation of a linear system with constraints

We consider a general discrete variational problem :

$$\text{find } u \in \tilde{\mathcal{U}} \text{ such that } a(u, v) = l(v) \quad \forall v \in \tilde{\mathcal{V}}, \quad (12.3)$$

where the spaces $\tilde{\mathcal{U}}$ and $\tilde{\mathcal{V}}$ are given by (12.1-12.2), $a(.,.)$ a bilinear form on $\mathcal{U} \times \mathcal{V}$ and $l(.)$ a linear form on \mathcal{V} .

We assume in the following, that the constraints systems $\mathbb{C}U = G$ and $\mathbb{D}V = 0$ are reduced, so the $m \times n$ matrix \mathbb{C} and the $p \times q$ matrix \mathbb{D} have the following representation :

$$\mathbb{C} = [\mathbb{E} \ \mathbb{R}] \text{ and } \mathbb{D} = [\mathbb{F} \ \mathbb{S}] \quad (12.4)$$

where \mathbb{E} is a $m \times m$ upper triangular matrix (invertible), \mathbb{R} a $m \times n - m$ matrix (may be null, $n \geq m$), \mathbb{F} a $p \times p$ upper triangular matrix (invertible) and \mathbb{S} a $p \times q - p$ matrix (may be null, $q \geq p$). The column indices \mathcal{E} (resp. \mathcal{F}) of \mathbb{E} (resp. \mathbb{F}) correspond to the eliminated unknowns (resp. test functions) and the column indices \mathcal{R} (resp. \mathcal{S}) of \mathbb{R} (resp. \mathbb{S}) correspond to reduced unknowns (resp. test functions). Introducing $\mathbb{X} = \mathbb{E}^{-1} \mathbb{R}$ and $\mathbb{Y} = \mathbb{F}^{-1} \mathbb{S}$, the constraints read :

$$U_{\mathcal{E}} + \mathbb{X} U_{\mathcal{R}} = \mathbb{E}^{-1} G \text{ and } V_{\mathcal{F}} + \mathbb{Y} V_{\mathcal{S}} = 0 \quad (12.5)$$

where $U_{\mathcal{E}}$ (resp. $U_{\mathcal{R}}$) denotes the unknown components restricted to indices \mathcal{E} (resp. \mathcal{R}) and $V_{\mathcal{F}}$ (resp. $V_{\mathcal{S}}$) denotes the test function components restricted to indices \mathcal{F} (resp. \mathcal{S}).

The space $\tilde{\mathcal{U}}$ may be reinterpreted by eliminating the components $U_{\mathcal{E}}$:

$$\begin{aligned} \sum_{i=1,n} U_i w_i &= \sum_{i \in \mathcal{E}} U_i w_i + \sum_{i \notin \mathcal{E}} U_i w_i \\ &= - \sum_{i \in \mathcal{E}} \sum_{j \in \mathcal{R}} \mathbb{X}_{ij} U_j w_i + \sum_{j \notin \mathcal{E}} U_j w_j + \sum_{i \in \mathcal{E}} (\mathbb{E}^{-1} G)_i w_i \\ &= \sum_{j \notin \mathcal{E}} u_j \left(w_j - \sum_{i \in \mathcal{E}} \mathbb{X}_{ij} w_i \right) + \sum_{i \in \mathcal{E}} (\mathbb{E}^{-1} G)_i w_i \quad (\text{setting } \mathbb{X}_{ij} = 0 \ \forall i, \ \forall j \notin \mathcal{E}). \end{aligned}$$

Considering the functions

$$\tilde{w}_j = w_j - \sum_{i \in \mathcal{E}} \mathbb{X}_{ij} w_i \quad j \notin \mathcal{E}, \quad (12.6)$$

the vector space $\tilde{\mathcal{U}}_0$ associated to $\tilde{\mathcal{U}}$ is generated by the functions \tilde{w}_j :

$$\tilde{\mathcal{U}}_0 = \left\{ \sum_{i \notin \mathcal{E}} U_i \tilde{w}_i \right\}.$$

In a same way, the vector space $\tilde{\mathcal{V}}$ is generated by the functions :

$$\tilde{\tau}_j = \tau_j - \sum_{i \in \mathcal{F}} \mathbb{Y}_{ij} w_i \quad j \notin \mathcal{F}, \quad (12.7)$$

and

$$\tilde{\mathcal{V}} = \left\{ \sum_{i \notin \mathcal{F}} U_i \tilde{\tau}_i \right\}.$$

Setting $g = \sum_{i \in \mathcal{E}} (\mathbb{E}^{-1} G)_i w_i$, the problem (12.3) is equivalent to

$$\text{find } \tilde{u} \in \tilde{\mathcal{U}}_0 \text{ such that } a(\tilde{u}, \tilde{v}) = l(\tilde{v}) - a(g, \tilde{v}) \quad \forall v \in \tilde{\mathcal{V}} \quad (12.8)$$

equivalent to

$$\text{find } \tilde{u} = \sum_{j \notin \mathcal{E}} \tilde{U}_j \tilde{w}_j \text{ such that } \sum_{j \notin \mathcal{E}} \tilde{U}_j a(\tilde{w}_j, \tilde{\tau}_i) = l(\tilde{\tau}_i) - a(g, \tilde{\tau}_i) \quad \forall i \notin \mathcal{F}. \quad (12.9)$$

The system (12.9) may be explicitated by using (assuming τ_i is a real function) :

$$a(\tilde{w}_j, \tilde{\tau}_i) = a(w_j, \tau_i) - \sum_{k \in \mathcal{E}} \mathbb{X}_{kj} a(w_k, \tau_i) - \sum_{k \in \mathcal{F}} \bar{\mathbb{Y}}_{ki} a(w_j, \tau_k) + \sum_{k \in \mathcal{E}} \sum_{l \in \mathcal{F}} \mathbb{X}_{kj} \bar{\mathbb{Y}}_{li} a(w_k, \tau_l) \quad (12.10)$$

and

$$l(\tilde{\tau}_i) = l(\tau_i) - \sum_{k \in \mathcal{F}} \bar{\mathbb{Y}}_{ki} l(\tau_k). \quad (12.11)$$

Finally, from the knowledge of all $a(w_j, \tau_i)$ and $l(\tau_i)$ corresponding to the computation without essential conditions, it is possible to construct the system (12.9). In terms of matrix, the different operations are mainly row or column combinations. XLiFE++ implements this general scheme. In a simple case, the Dirichlet problem for instance, the previous expression are trivial because $\mathbb{X} = \mathbb{Y} = 0$, $\tilde{w}_j = w_j$, $\forall j \notin \mathcal{E}$ and $\tau_j = w_j$.

When condition is not homogeneous ($g \neq 0$), there is a contribution from g and a part of matrix representing $a(.,.)$ (columns with index in \mathcal{E}). In order to avoid additional reduction process when an other data g is considered, it is possible to keep in memory this part of matrix.

Overview of classes involved

To describe essential conditions, XLiFE++ uses :

- some classes used to describe (bi)linearform : `Unknown DifferentialOperator OperatorOnUnknown`
- the class `LcOperatorOnUnknown` which handles linear combination of `OperatorOnUnknown` and allow to associate geometric domains (`GeomDomain`). The general syntaxes look like (\diamond represent any algebraic operation consistant with objects involved) :

$$a_1 * op_1(f_1 \diamond u_1 \diamond g_1) | dom_1 + a_2 * op_2(f_2 \diamond u_2 \diamond g_2) | dom_2 + \dots$$

$$(a_1 * op_1(f_1 \diamond u_1 \diamond g_1) + a_2 * op_2(f_2 \diamond u_2 \diamond g_2) + \dots) | dom$$

This class supports a lot of expressions but just a few ones could be used as essential condition ! This class belongs to the `Operator` library.

- To transform a previous expression as an essential condition, it is required to give a right hand side :

$$\text{LcOperatorOnUnknown} = \text{constant or LcOperatorOnUnknown} = \text{function}$$

it produces an object of `EssentialCondition` class.

- Concatenating `EssentialCondition` objects using the operator `&` produces a set of essential conditions managed by `EssentialConditions` class.

All these classes are user classes!

`Constraints` is the core class implementing most of the algorithms : transformation of symbolic conditions into constraints system, reduction of constraints system, reduction or pseudo-reduction of matrix, right hand side correction.

12.2 The EssentialCondition class

The `EssentialCondition` class handles the symbolic representation of an essential condition in the form $expression(u_1 | dom_1, u_2 | dom_2, \dots) = f$ where $expression(\dots)$ is a `LcOperatorOnUnknown` object and f is a `Function` object.

```
class EssentialCondition
{
    protected :
    LcOperatorOnUnknown ecTerms_; // combination of operator on unknown
    Function *fun_p;             // function defining data of EC
                                // (=0 when EC is homogeneous)
    EcType type_;                // type of essential condition
}
```

The type of essential condition are listed in the *EcType* enumeration:

```
enum EcType {_undefEcType,      //undefined type
             _DirichletEc,      //one unknown, one domain
             _transmissionEc,   //two unknowns, one domain
             _periodicEc,       //one unknown, two domains
             _crackEc,          //two unknowns, two domains
             _meanEc            //intg_D(opu)=f
};
```

The class proposes basic constructors which do full copy of *LcOperatorOnUnknown* and *Function* objects:

```
EssentialCondition()
EssentialCondition(const LcOperatorOnUnknown&);
EssentialCondition(const LcOperatorOnUnknown&, const Function&);
EssentialCondition(Domain&, const LcOperatorOnUnknown&);
EssentialCondition(Domain&, const LcOperatorOnUnknown&, const Function&);
EssentialCondition(const EssentialCondition&);
EssentialCondition& operator=(const EssentialCondition&);
```

some accessors or shortcuts :

```
const std::vector<OpuValPair>& bcTerms() const;
const Function& fun() const;
Function& funp();
it_opuval begin();
cit_opuval begin() const;
it_opuval end();
cit_opuval end() const;
Number size() const;
EcType type() const;
```

some function giving properties

```
bool isSingleUnknown() const;
Dimen nbOfUnknowns() const;
const Unknown* unknown() const;
std::set<const Unknown*> unknowns() const;
bool isSingleDomain() const;
Domain* domain() const;
std::set<Domain*> domains() const;
Dimen nbOfDomains() const;
Number nbTerms() const;
Complex coefficient() const;
std::vector<Complex> coefficients() const;
std::set<DiffOpType> diffOperators() const;
bool nbOfDifOp() const;
DiffOpType diffOperator() const;
bool isHomogeneous() const;
String name() const;
```

some functions to set type and domain:

```
EcType setType(); //set the type of the Essential condition
void setDomain(Domain& dom) //affect domain to EC
EssentialCondition& operator |(Domain&) //affect domain of EC
```

When affect the domain using *setDomain* function or *|* operator, all the domains defined in *EcTerms* are reset to the new domain. This domain affectation is also done by the non member function:

```
EssentialCondition on(const Domain&, const EssentialCondition&);
```

and printing facilities:

```
void print(std::ostream&) const;
friend std::ostream& operator<<(std::ostream&, const EssentialCondition&);
```

To construct essential condition on the fly, the = operator of classes `Unknown`, `OperatorOnUnknown` and `LcOperatorOnUnknown` are overloaded:

```
EssentialCondition LcOperatorOnUnknown::operator = (const Real &);
EssentialCondition LcOperatorOnUnknown::operator = (const Complex &);
EssentialCondition LcOperatorOnUnknown::operator = (const Function &);
EssentialCondition OperatorOnUnknown::operator = (const Real &);
EssentialCondition OperatorOnUnknown::operator = (const Complex &);
EssentialCondition OperatorOnUnknown::operator = (const Function &);
EssentialCondition Unknown::operator = (const Real &);
EssentialCondition Unknown::operator = (const Complex &);
EssentialCondition Unknown::operator = (const Function &);
```

All these member functions are implemented in the file *EssentialCondition.cpp*.

library :	<code>essentialCondition</code>
header :	<code>EssentialCondition.hpp</code>
implementation :	<code>EssentialCondition.cpp</code>
unitary tests :	<code>test_EssentialCondition.cpp</code>
header dependences :	<code>Operator.h</code> , <code>config.h</code> , <code>utils.h</code>

12.3 The EssentialConditions class

The `EssentialConditions` class manages a list of `EssentialCondition`. It inherits from `list<EssentialConditions>` with no additional member data:

```
class EssentialConditions : public std::list<EssentialCondition>
```

As it is a user class, it mainly interfaces function of `list` class

```
//constructors and assign
EssentialConditions() {}
EssentialConditions(const EssentialCondition&);
EssentialConditions(const EssentialCondition&, const EssentialCondition&);
EssentialConditions(const EssentialConditions&);
EssentialConditions& operator = (const EssentialConditions&);
//utilities
bool coupledUnknowns() const;
std::set<const Unknown*> unknowns() const;
const Unknown* unknown() const;
void print(std::ostream&) const;
friend std::ostream& operator << (std::ostream&, const EssentialConditions &);
```

It overloads the & operator to concatenate `EssentialCondition`:

```
EssentialConditions operator & (const EssentialCondition&, const
    EssentialCondition&);
EssentialConditions operator & (const EssentialCondition&, const
    EssentialConditions&);
EssentialConditions operator & (const EssentialConditions&, const
    EssentialCondition&);
```

```
EssentialConditions operator & (const EssentialConditions&, const
EssentialConditions&);
```

```
library : essentialCondition
header : EssentialConditions.hpp
implementation : EssentialConditions.cpp
unitary tests : test_EssentialCondition.cpp
header dependences : EssentialCondition.hpp, Operator.h, config.h, utils.h
```

12.4 The Constraints class

The `Constraints` class handles the algebraic representation $\mathbb{C}U = G$ of a set of essential conditions (`EssentialConditions`):

```
class Constraints
{
protected :
MatrixEntry* matrix_p;           //constraints matrix
VectorEntry* rhs_p;              //constraints right hand side
public :
EssentialConditions conditions_;  //list of essential conditions
std::vector<DofComponent> cdofsr_; //row DofComponent's of matrix_p
std::vector<DofComponent> cdofsc_; //col DofComponent's of matrix_p
std::map<DofComponent, Number> elcdofs_; //map of eliminated cdofs
std::map<DofComponent, Number> recdofs_; //map of reduced cdofs
bool reduced;                    //true if reduced system
bool local;                      //true if local conditions
bool symmetric;                  //true if keep symmetry
bool isId;                       //true if Id matrix
}
```

Even if unknowns involved in constraints system are vector unknowns, the representation uses scalar unknown numbering (`cdofsc_` vector). The `cdofsr_` handles row numbering, each row corresponding to one constraint equation linked to an "artificial" test function. The `elcdofs_` map carries the eliminated index set \mathcal{E} and `recdofs_` map carries the reduced index set \mathcal{R} . `Constraints` is processed in two steps :

- construction of the constraint system from the list of essential conditions : each condition are translated in its matrix representation, then all the matrix representations are merged in a large one
- reduction of the large constraints system using QR algorithm with removing the redundant constraints and identifying the conflicting constraints

A constraint is local (`local = true`) if it involves only one dof. Dirichlet conditions are local and transmission or periodic conditions are not. The class proposes mainly the constructor from a list of essential conditions and usual copy and delete stuff (full copy):

```
Constraints(MatrixEntry* =0, VectorEntry* =0);
Constraints(const EssentialCondition&);
Constraints(const Constraints&);
Constraints& operator=(const Constraints&);
void copy(const Constraints&);
~Constraints();
void clear();
```

some accessors and property functions :

```

const MatrixEntry* matrixp() const;
const VectorEntry* rhsp() const;
bool coupledUnknowns() const;
const Unknown* unknown() const;
std::set<const Unknown*> unknowns() const;

```

The fundamental functions are the building functions

```

bool createDirichlet(const EssentialCondition&); //create from Dirichlet cond.
void createLf(const EssentialCondition&); //create from linear form cond.
void createNodal(const EssentialCondition&); //create general condition
MatrixEntry* constraintsMatrix(const OperatorOnUnknown&, const GeomDomain*,
    Space*, const complex_t&, vector<Point>&, const Function*,
    vector<DofComponent>&); //create submatrix of a constraint
void concatenateMatrix(MatrixEntry&, std::vector<DofComponent>&, const MatrixEntry&,
    const std::vector<DofComponent>&); //concatenate 2 constraints matrices
void reduceConstraints(real_t asZero); //reduce constraints matrix
template <typename T>
void buildRhs(const Function*, vector<Point>&, const T&); //tool building rhs

```

Functions *createXXX(const EssentialCondition&)* build the matrix representation for *XXX* condition. Now, there exist Dirichlet, transmission and periodic type. It is quite easy to add new translation functions to take into account new ones. The *reduceConstraints* function processes the reduction of any constraints sytem using QR algorithm. The *asZero* variable is used to round to zero very small values induced by rounding errors in QR algorithm. This function modifies the current matrix and right hand side vector. So the original representation is lost after reduction!

To build the matrix representation of a set of essential conditions, two cases have to be considered:

- no condition that couples two unknowns (independent conditions); in that case one constraints system is built for each unknown
- there exists one coupling condition (coupled conditions); in that case a unique constraints system for all unknowns is built

For instance, the conditions (u_1, u_2 two unknowns):

$$u_1 = 0|_{\Sigma} \quad u_1 = 0|_{\Gamma} \quad u_2 = 0|_{\Gamma}$$

gives two constraints system, one merging u_1 conditions and an other one for u_2 . While the conditions :

$$u_1 = 0|_{\Sigma} \quad u_1 = 0|_{\Gamma} \quad u_2 - u_1 = g|_{\Gamma}$$

gives a global constraints system merging all conditions.

The merging and building process is done by an external functions that return a list of **Constraints** indexed by unknown (**Unknown** pointer). In case of coupled conditions, the list contains only one **Constraints** indexed by 0.

```

map<const Unknown*, Constraints*> mergeConstraints(vector<Constraints*>&);
map<const Unknown*, Constraints*> buildConstraints(const EssentialConditions&);

```

The *buildConstraints* function looks like (iterator declaration omitted):

```

map<const Unknown*, Constraints*> buildConstraints(const EssentialConditions& ecs){
map<const Unknown*, Constraints*> mconstraints
vector<Constraints*> constraints(ecs.size());
itc=constraints.begin();

```



```

//translate conditions into constraints
for (ite=ecs.begin(); ite!=ecs.end(); ite++, itc++)
    *itc=new Constraints(*ite);
//merge constraints
mconstraints = mergeConstraints(constraints);
//reduce constraints
for (itm=mconstraints.begin(); itm!=mconstraints.end(); itm++)
    itm->second->reduceConstraints();
return mconstraints;}

```

The `Constraints` class provides important tools that take into account essential conditions in linear systems:

```

void pseudoColReduction(MatrixEntry*, vector<DofComponent>&,
                        vector<DofComponent>&, MatrixEntry*&);
void pseudoRowReduction(MatrixEntry*, vector<DofComponent>&,
                        vector<DofComponent>&);
friend void extendStorage(MatrixEntry*, vector<DofComponent>&,
                          vector<DofComponent>&, const Constraints*, const Constraints*);
void extractCdofs(const vector<DofComponent>&, map<DofComponent, Number>&,
                  map<DofComponent, Number>&, bool useDual=false) const;
friend void appliedRhsCorrectorTo(VectorEntry*, vector<DofComponent>&,
                                  MatrixEntry*, const Constraints*, const Constraints*,
                                  ReductionMethod);

```

The functions *pseudoColReduction* and *pseudoRowReduction* performs the reduction of system matrix according to relation 12.10. The *pseudoColReduction* also stores the part of matrix system to be memorized to process right hand side correction. If case of non homogeneous constraints, the function *appliedRhsCorrectorTo* modifies the system right hand side according to relation 12.11. Regarding unknown and test function constraints, *extendStorage* function extends the storage of a matrix. This extension occurs when constraints are not local. *extractCdofs* function is a facility to localized eliminated or reduced rows/columns in matrix. All of these functions are called by compute functions of `TermMatrix` class.

In order to deal with constraints acting on different matrix blocs, the `SetOfConstraints` class is provided. It manages mainly a map relating unknown and constraint:

```

class SetOfConstraints : public map<const Unknown*, Constraints*>
{
public:
SetOfConstraints(const Unknown*, Constraints*); //from a constraints pointer
SetOfConstraints(const map<const Unknown*, Constraints*>& mc); //from pointers
SetOfConstraints(const EssentialConditions&); //from EssentialConditions
SetOfConstraints(const SetOfConstraints&); //copy constructor with full copy
SetOfConstraints& operator = (const SetOfConstraints&); //assignment with copy
~SetOfConstraints(); //destructor deallocating constraints pointers
void clear(); //deallocate constraints pointer and reset the map

Constraints* operator()(const Unknown*) const; //access operator from unknown
bool isGlobal() const; //true if a unique global constraint
bool isReduced() const; //true if all constraints have been reduced
void print(std::ostream&) const; //print utility
friend std::ostream& operator<<(std::ostream&, const SetOfConstraints&);
};

```

The `SetOfConstraints` objects are very important because they are used by the `TermMatrix` objects when they have to deal with essential conditions!

library :	essentialCondition
header :	Constraints.hpp
implementation :	Constraints.cpp
unitary tests :	test_Constraints.cpp
header dependences :	EssentialCondition.hpp, EssentialConditions.hpp, LargeMatrix.h, config.h, utils.h

13 The *solvers* library

The *solvers* library comes with some iterative solver methods which target a linear system. These solvers can be divided into two kinds of group. One is for symmetric linear system and another is for a non-symmetric system. Each solver uses *largeMatrix* and *Vector* as inputs and returns the result in a *Vector*. The type of problem, real or complex, needs specified also as another input of a solver. In some cases, a solver can take advantage of a *Preconditioner* as an extra input to decrease the conditioner number of the system. A *Preconditioner*, which is also a *LargeMatrix* involved with matrix factorization, is only available in the type of *SkylineStorage* and *CsStorage*.

All solvers in the library are based on the class *IterativeSolver*, which provides some basic properties and methods. Specific iterative algorithms are implemented in the inherited class for two cases: with and without a preconditioner.

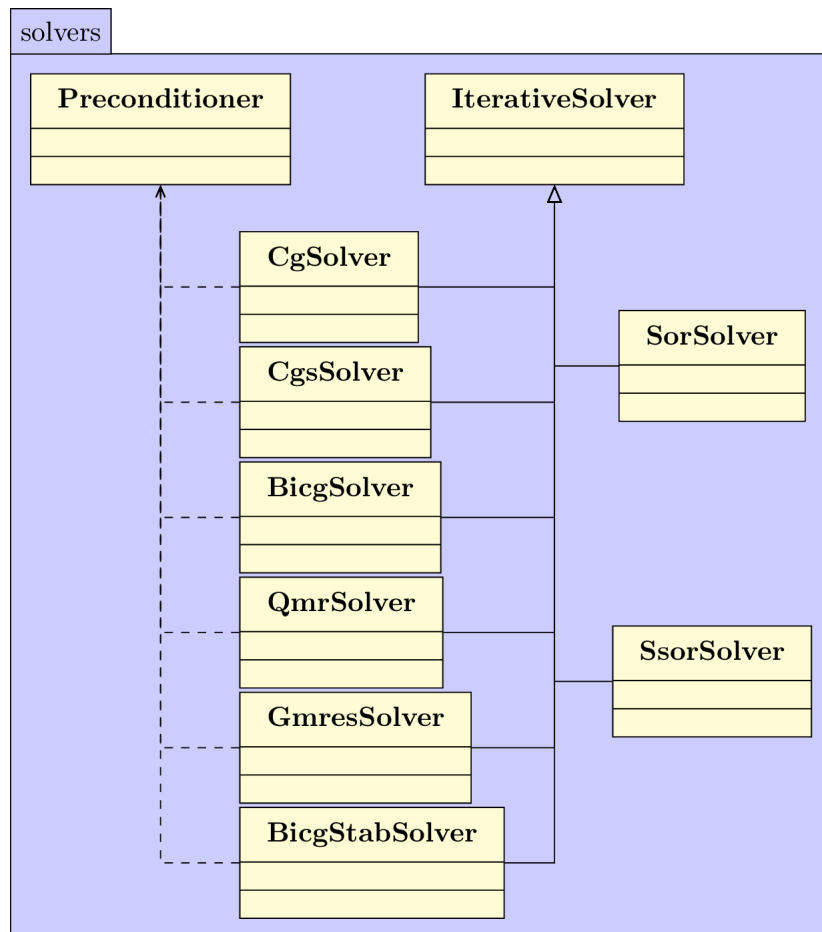


Figure 13.1: The *solvers* library main class diagram

13.1 The IterativeSolver class

As the base class of all other solver classes, the `IterativeSolver` provides some basic properties and methods for all descendants.

```
class IterativeSolver
{
public:
    ///! Constructor by name
    IterativeSolver(const String&, const Number vb = 0);
    ///! Full constructor
    IterativeSolver(const String&, const Number maxOfIt,
                    const Real eps, const Number vb = 0, const bool prec = true);
    ///! Destructor
    virtual ~IterativeSolver();
    ///! Iterative solver name for documentation purposes
    String name() { return name_; }
    ///! Reset Solver
    void resetSolver();
protected:
    ///! Define a default value for the maximum number of iterations
    Number maximumOfIterations(const size_t nbRows);
    ///! Define a default value for krylov dimension
    Number krylovDimension(Number kd, const size_t nbRows)
    ...
}
```

A solver object can be constructed in several ways: With default values or with user-input values. Information of solver can be printed out with some auxiliary functions.

```
void printHeader(const size_t) const;
void printHeader(const size_t, const String&) const;
void printHeader(const size_t, const Real) const;
void printHeader(const size_t, const Number) const;
void printHeader(const size_t, const Number, const String&) const;
void printOutput() const;
void printIteration() const;
```

In order to avoid invalid cast from complex to real and vice versa, some inline functions are provided.

```
inline void assign(Real& x, const Real& y) { x = y; }
inline void assign(Complex& x, const Real& y) { x = y; }
inline void assign(Complex& x, const Complex& y) { x = y; }
inline void assign(Real& x, const Complex& y) { x = y.real(); }
```

library :	solvers		
header :	IterativeSolver.hpp		
implementation :	IterativeSolver.cpp		
unitary tests :	test_BicgSolver.hpp,	test_BicgStabSolver.hpp,	test_CgSolver.hpp,
	test_CgsSolver.hpp,	test_GmresSolver.hpp,	test_QmrSolver.hpp,
	test_SorSolver.hpp, test_SsorSolver.hpp		
header dependences :	Preconditioner.hpp, config.h, utils.h		

13.2 The Preconditioner class

In some cases, the rate of convergence of an iterative solver can increase as a result of preconditioning. A preconditioner is basically a special matrix which can be factorized. There exist several types of matrix-factorization for `Preconditioner`: LU, LDLt, LDL*, Diag and SOR. Because of constraints of factorizing, only matrices of `SkylineStorage` and `CsStorage` can be used as a preconditioner.

In detail, a preconditioner is based on a template class `Mat`, on which all the preconditioned-solver are processed.

```
template<class Mat>
class Preconditioner
{
public:
    Mat* precondMatrix_p; // pointer to precondition matrix
private:
    //! Type of preconditioner
    PreconditionerType type_;
    //! Relaxation parameter for SSOR preconditioners
    Real omega_;
public:
    Preconditioner()
        : precondMatrix_p(0), type_(noPreconditioner), omega_(0.) { }

    Preconditioner(PreconditionerType pt, Mat& A, const Real w = 1.)
        : precondMatrix_p(&A), type_(pt), omega_(w) { }

    Preconditioner(Mat& A, PreconditionerType pt, const Real w = 1.)
        : precondMatrix_p(&A), type_(pt), omega_(w) { }
    ~Preconditioner() {};
    //! Return type of preconditionner
    static String name(PreconditionerType pt)
    {
        String s("");
        switch ( pt )
        {
            case _lu:          s = "LU"; break;
            case _ldlt:        s = "LDLt"; break;
            case _ldlStar:     s = "LDL*"; break;
            case _ssor:        s = "SSOR"; break;
            case _diag:        s = "Diagonal"; break;
            //case myPreconditioner: s = "User Supplied"; break;
            default: break;
        }
        return s;
    }
    ...
}
```



A preconditioner can only be templated with a specific kind of matrix that can be decomposed. There are only two types of storage, with which matrix can be factorized: *SkylineStorage* with LU, LDLt, LDL* factorization; *CsStorage* with SOR and DIAG factorization.

There are several constructors:

```
Preconditioner()
    : precondMatrix_p(0), type_(noPreconditioner), omega_(0.) { }

Preconditioner(PreconditionerType pt, Mat& A, const Real w = 1.)
    : precondMatrix_p(&A), type_(pt), omega_(w) { }

Preconditioner(Mat& A, PreconditionerType pt, const Real w = 1.)
    : precondMatrix_p(&A), type_(pt), omega_(w) { }
```

The `PreconditionerType` is an enumeration.

```
enum PreconditionerType { noPreconditioner, _lu, _ldlt, _ldlStar, _ssor, _diag,
    _myPreconditioner };
```

library :	solvers
header :	Preconditioner.hpp
implementation :	Preconditioner.hpp
unitary tests :	test_LargeMatrixSkylineStorage.cpp, test_LargeMatrixCsStorage.cpp
header dependencies :	config.h, utils.h

13.3 Solvers

Each solver class in the *solvers* library implements a specific iterative method to solve a linear system: $A \cdot X = B$ in two cases: Real or Complex. The algorithm of all solvers except Sor and Sor can work with and without a *Preconditioner*. Each solver takes *largeMatrix* *matA* and *Vector* *vecB* as inputs and a *Vector* *vecX* as output whose value is initialized with the input *Vector* *vecX0*.

A solver without a preconditioner is invoked with the overloading operator():

```
template<class Mat, class VecB, class VecX>
VecX operator()(Mat& matA, VecB& vecB, VecX& vecX0, ValueType solType)
```

Meanwhile, solver with a preconditioner *pc* can be used with the call

```
template<class Mat, class VecB, class VecX, class Prec>
VecX operator()(Mat& matA, VecB& vecB, VecX& vecX0, Prec& pc, ValueType solType)
```

User must specify the type of solver *solType*, which can be *_real* or *_complex*.

The algorithm corresponds to without-preconditioner solver

```
template<typename K, class Mat, class VecB, class VecX>
void algorithm(Mat& matA, VecB& vecB, VecX& vecX, VecX& vecR)
```

and to a preconditioned one

```
template<typename K, class Mat, class VecB, class VecX, class Prec>
void algorithm(Mat& matA, VecB& vecB, VecX& vecX, VecX& vecR, Prec& pc)
```

13.3.1 The BicgSolver class

This class implements the Biconjugate Gradient method to solve a linear system. There are several basic constructors calling *IterativeSolver* constructors

```
class BicgSolver : public IterativeSolver
{
public:
    ///! Default Constructor
    BicgSolver()
        : IterativeSolver("BiCG") {}

    ///! Full constructor
    BicgSolver(const Real eps,
               const Number maxOfIt = defaultMaxIterations,
               const Number vb = 0)
        : IterativeSolver("BiCG", maxOfIt, eps, vb) {}
    ...
}
```

library :	solvers
header :	
implementation :	
unitary tests :	test_BicgSolver.hpp
header dependencies :	IterativeSolver.hpp, Preconditioner.hpp

13.3.2 The BicgStabSolver class

This class implements the Biconjugate Gradient Stabilized method to solve a linear system. There are several basic constructors calling **IterativeSolver** constructors

```
class BicgStabSolver : public IterativeSolver
{
public:
    ///! Default Constructor
    BicgStabSolver() : IterativeSolver("BiCGStab") {}

    ///! Full Constructor
    BicgStabSolver(const Real eps,
                  const Number maxOfIt = defaultMaxIterations,
                  const Number vb = 0)
        : IterativeSolver("BiCGStab", maxOfIt, eps, vb) {}
    ...
}
```

library :	solvers
header :	
implementation :	
unitary tests :	test_BicgStabSolver.hpp
header dependencies :	IterativeSolver.hpp, Preconditioner.hpp

13.3.3 The CgSolver class

This class implements the Conjugate Gradient method to solve a linear system. There are several basic constructors calling **IterativeSolver** constructors

```
class CgSolver : public IterativeSolver
{
public:
    ///! Default constructor
    CgSolver() : IterativeSolver("CG") {}

    ///! Full constructor
    CgSolver(const Real eps,
             const Number maxOfIt = defaultMaxIterations,
             const Number vb = 0)
        : IterativeSolver("CG", maxOfIt, eps, vb) {}
    ...
}
```

library :	solvers
header :	
implementation :	
unitary tests :	test_CgSolver.hpp
header dependencies :	IterativeSolver.hpp, Preconditioner.hpp

13.3.4 The CgsSolver class

This class implements the Conjugate Gradient Squared method to solve a linear system. There are several basic constructors calling `IterativeSolver` constructors

```
class CgsSolver : public IterativeSolver
{
public:
    //! Default constructor
    CgsSolver()
        : IterativeSolver("CGS") {}

    //! Full constructor
    CgsSolver(const Real eps,
              const Number maxOfIt = defaultMaxIterations,
              const Number vb = 0)
        : IterativeSolver("CGS", maxOfIt, eps, vb) {}
    ...
}
```

library :	solvers
header :	
implementation :	
unitary tests :	test_CgsSolver.hpp
header dependencies :	IterativeSolver.hpp, Preconditioner.hpp

13.3.5 The GmresSolver class

This class implements the Generalized Minimal Residual method to solve a linear system. Different from other solvers, the `GmresSolver` has more constructors: user can specify the Krylov dimension in the constructor.

```
class GmresSolver : public IterativeSolver
{
private:
    Number krylovDim_; // << Dimension of Krylov space

public:
    //! Default Constructor
    GmresSolver()
        : IterativeSolver("GMRes"), krylovDim_(defaultKrylovDimension) {}

    //! Constructor with Krylov dimension
    GmresSolver(const Number kd)
        : IterativeSolver("GMRes", defaultMaxIterations,
                          theDefaultConvergenceThreshold,
                          krylovDim_(kd)) {}

    //! Full constructor
    GmresSolver(const Number kd, const Real eps,
              const Number maxOfIt = defaultMaxIterations,
              const Number vb = 0)
        : IterativeSolver("GMRes", maxOfIt, eps, vb), krylovDim_(kd) {}
    ...
}
```

library :	solvers
header :	
implementation :	
unitary tests :	test_GmresSolver.hpp
header dependencies :	IterativeSolver.hpp, Preconditioner.hpp

13.3.6 The QmrSolver class

This class implements the Quasi-Minimal Residual method to solve a linear system. There are several basic constructors calling **IterativeSolver** constructors

```
class QmrSolver : public IterativeSolver
{
public:
    ///! Constructor
    QmrSolver()
        : IterativeSolver("QMR") {}

    ///! Full Constructor
    QmrSolver(const Real eps,
              const Number maxOfIt = defaultMaxIterations,
              const Number vb = 0)
        : IterativeSolver("QMR", maxOfIt, eps, vb) {}
    ...
}
```

library :	solvers
header :	
implementation :	
unitary tests :	test_QmrSolver.hpp
header dependencies :	IterativeSolver.hpp, Preconditioner.hpp

13.3.7 The SorSolver class

This class implements the Successive Over Relaxation method to solve a linear system. For this solver, user can have more choices to specify the relaxation factor *omega* in the constructor. Different from other solvers above, **SorSolver** doesn't work with a preconditioner.

```
class SorSolver : public IterativeSolver
{
private:
    Real omega_; // relaxation factor of SOR method

public:
    ///! Default Constructor
    SorSolver()
        : IterativeSolver("SOR"), omega_(1.) {}

    ///! Constructor with omega
    SorSolver(const Real w)
        : IterativeSolver("SOR"), omega_(w) {}

    ///! Full constructor
    SorSolver(const Real w, const Real eps,
              const Number maxOfIt = defaultMaxIterations,
              const Number vb = 0)
        : IterativeSolver("SOR", maxOfIt, eps, vb), omega_(w) {}
}
```

...

library : solvers
header :
implementation :
unitary tests : test_SorSolver.hpp
header dependences : IterativeSolver.hpp, Preconditioner.hpp

13.3.8 The SsorSolver class

This class implements the Symmetric Successive Over Relaxation method to solve a linear system. Like [SorSolver](#), the [SsorSolver](#) only works without a preconditioner and user can specify the relaxation factor *omega* in the constructor.

```
class SsorSolver : public IterativeSolver
{
private:
    Real omega_; //relaxation factor of SOR method

public:
    //! Default Constructors
    SsorSolver() : IterativeSolver("SSOR"), omega_(1.) {}

    //! Constructor with omega
    SsorSolver(const Real w) : IterativeSolver("SSOR"), omega_(w) {}

    //! Full constructor
    SsorSolver(const Real w, const Real eps,
               const Number maxOfIt = defaultMaxIterations,
               const Number vb = 0)
        : IterativeSolver("SSOR", maxOfIt, eps, vb), omega_(w) {}
    ...
}
```

library : solvers
header :
implementation :
unitary tests : test_SsorSolver.hpp
header dependences : IterativeSolver.hpp, Preconditioner.hpp

Besides some built-in direct solvers for sparse linear system, XLIFFE++ provides an interface to UMFPACK, a set of routines for solving non symmetric sparse linear systems, $Ax = b$, using the Unsymmetric MultiFrontal method. More information can be found at <http://www.cise.ufl.edu/research/sparse/umfpack/>.

The aim of *umfpackSupport* is to make ease of the UMFPACK for XLIFFE++'s users, allowing them to solve the linear system $Ax = b$ with a simple call. Instead of invoking "multi-argument" UMFPACK solver, user can make call of them as any built-in iterative solver in the context of XLIFFE++.

14.1 The UMFPACK wrappers

Written in ANSI/ISO C, UMFPACK library consists of 32 user-callable routines. Nearly all these routines come in four versions, with different sizes of integers and for real or complex floating-point numbers:

- real double precision, **int** integers
- complex double precision, **int** integers
- real double precision, **SuiteSparse_long** integers
- complex double precision, **SuiteSparse_long** integers

Only the interfaces to the first two versions are implemented in XLIFFE++. However, more versions can be easily added in the need of user.

Because callable UMFPACK routines are written in C, so as to call them correctly from the C++ environment of XLIFFE++, we must make these routines "extern C". And it's the role of *UmfPackWrappers*. Only some primary routines of UMFPACK are made interface; however, it's not difficult to include others.

```
#ifndef __cplusplus
extern "C" {
#endif

int DISYMBOLIC(int n_row, int n_col, const int Ap[], const int Ai[], const double
Ax[], void** Symbolic, const double Control[UMFPACK_CONTROL], double
Info[UMFPACK_INFO]);
int DINUMERIC(const int Ap[], const int Ai[], const double Ax[], void* Symbolic,
void** Numeric, const double Control[UMFPACK_CONTROL], double
Info[UMFPACK_INFO]);
int DISOLVE(int sys, const int Ap[], const int Ai[], const double Ax[], double X[],
const double B[], void* Numeric, const double Control[UMFPACK_CONTROL], double
Info[UMFPACK_INFO]);
void DIFREESYMBOLIC(void** Symbolic);
void DIFREENUMERIC(void** Numeric);
...

int ZISYMBOLIC(int n_row, int n_col, const int Ap[], const int Ai[], const double
Ax[], const double Az[], void** Symbolic, const double
Control[UMFPACK_CONTROL], double Info[UMFPACK_INFO]);
int ZINUMERIC(const int Ap[], const int Ai[], const double Ax[], const double Az[],
void* Symbolic, void** Numeric, const double Control[UMFPACK_CONTROL], double
Info[UMFPACK_INFO]);
```

```

int ZISOLVE(int sys, const int Ap[], const int Ai[], const double Ax[], const
double Az[], double Xx[], double Xz[], const double Bx[], const double Bz[],
void* Numeric, const double Control[UMFPACK_CONTROL], double
Info[UMFPACK_INFO]);
void ZIFREESYMBOLIC(void** Symbolic);
void ZIFREENUMERIC(void** Numeric);
...

```

Routine names beginning with DI correspond to the version of real double precision, **int** integers, the ones with ZI imply the versions of complex double precision, **int** integers.

14.2 The UMFPACK class

The **UMFPACK** class plays a role of high-level wrapper of routines provided by UmfPackWrappers:

- ***SYMBOLIC**: performing a symbolic decomposition on the sparsity of a matrix
- ***NUMERIC**: performing a numeric decomposition of a matrix
- ***FREESYMBOLIC**: freeing symbolic object created by ***SYMBOLIC**
- ***FREENUMERIC**: freeing numeric object created by ***NUMERIC**
- ***GETLUNZ**: returning the number of non-zeros in lower matrix L and upper matrix U in LU decomposition
- ***GETNUMERIC**: retrieving lower matrix L, upper matrix U, permutation P,Q, and scale factor R
- ***GETDET**: returning determinant of original matrix

The **""** corresponds to DI-real double precision, or ZI-complex double precision.

All these routines are organized under templated "class-like" interface.

```

template<typename OrdinalType, typename ScalarType>
class UMFPACK
{
public:
    typedef typename NumTraits<ScalarType>::magnitudeType MagnitudeType;

    //! Default Constructor.
    inline UMFPACK(void) {}

    //! Destructor.
    inline ~UMFPACK(void) {}

    OrdinalType symbolic(OrdinalType nRow, OrdinalType nCol, const OrdinalType
        Ap[], const OrdinalType Ai[], const ScalarType Ax[], void** Symbolic, const
        MagnitudeType Control[UMFPACK\_CONTROL], MagnitudeType Info[UMFPACK\_INFO]);

    OrdinalType numeric(const OrdinalType Ap[], const OrdinalType Ai[], const
        ScalarType Ax[], void* Symbolic, void** Numeric, const MagnitudeType
        Control[UMFPACK\_CONTROL], MagnitudeType Info[UMFPACK\_INFO]);

    void freeSymbolic(void** Symbolic);
    void freeNumeric(void** Numeric);

    OrdinalType solve(OrdinalType sys, const OrdinalType Ap[], const OrdinalType
        Ai[], const ScalarType Ax[], ScalarType X[], const ScalarType B[], void*
        Numeric, const MagnitudeType Control[UMFPACK\_CONTROL], MagnitudeType
        Info[UMFPACK\_INFO]);

```

```

OrdinalType getLunz(OrdinalType* lnz, OrdinalType* unz, OrdinalType* nRow,
    OrdinalType* nCol, OrdinalType* nzUdiag, void* Numeric);

OrdinalType getNumeric(OrdinalType Lp[], OrdinalType Lj[], ScalarType Lx[],
    OrdinalType Up[], OrdinalType Ui[], ScalarType Ux[], OrdinalType P[],
    OrdinalType Q[], ScalarType Dx[], OrdinalType* doRecip, ScalarType Rs[],
    void* Numeric);

OrdinalType getDeterminant(ScalarType* Mx, ScalarType* Ex, void* NumericHandle,
    ScalarType UserInfo[UMFPACK_INFO]);
...

```

One advantage of this organization is to allow the specialization of code corresponding to the two versions: real double precision and complex double precision with **int** integer.

```

template<
class UMFPACKint, double>
{
    public:
        inline UMFPACK(void) {}
        inline ~UMFPACK(void) {}

        int symbolic(int nRow, int nCol, const int Ap[], const int Ai[], const double
            Ax[], void** Symbolic, const double Control[UMFPACK_CONTROL], double
            Info[UMFPACK_INFO]);

        int numeric(const int Ap[], const int Ai[], const double Ax[], void* Symbolic,
            void** Numeric, const double Control[UMFPACK_CONTROL], double
            Info[UMFPACK_INFO]);

        void freeSymbolic(void** Symbolic);

        void freeNumeric(void** Numeric);

        int solve(int sys, const int Ap[], const int Ai[], const double Ax[], double
            X[], const double B[], void* Numeric, const double
            Control[UMFPACK_CONTROL], double Info[UMFPACK_INFO]);
        ...
}

```

The specialization of complex version comes with an extra method for the linear system $Ax = b$ with A complex double precision and b real double precision.

```

template<
class UMFPACKint, std::complex<double> >
{
    public:
        inline UMFPACK(void) {}
        inline ~UMFPACK(void) {}

        int symbolic(int nRow, int nCol, const int Ap[], const int Ai[], const
            std::complex<double> Ax[], void** Symbolic, const double
            Control[UMFPACK_CONTROL], double Info[UMFPACK_INFO]);

        int numeric(const int Ap[], const int Ai[], const std::complex<double> Ax[],
            void* Symbolic, void** Numeric, const double Control[UMFPACK_CONTROL],
            double Info[UMFPACK_INFO]);

        void freeSymbolic(void** Symbolic);

        void freeNumeric(void** Numeric);
}

```

```

int solve(int sys, const int Ap[], const int Ai[], const std::complex<double>
Ax[], std::complex<double> X[], const std::complex<double> B[], void*
Numeric, const double Control[UMFPACK_CONTROL], double Info[UMFPACK_INFO]);

// Solve in case AX=B with A complex, B real
int solveCR(int sys, const int Ap[], const int Ai[], const std::complex<double>
Ax[], std::complex<double> X[], const double Bx[], const double Bz[], void*
Numeric, const double Control[UMFPACK_CONTROL], double Info[UMFPACK_INFO]);
...

```

```

library : umfpackSupport
header : UmfPack.hpp
implementation : UmfPack.cpp
unitary tests : test_UmfPackSolver.cpp
header dependences : utils.h

```

14.3 The UmfPackSolver class

The class, as its name, serves for solving the sparse linear system $Ax = b$. Being similar to other solvers of XLIFF++, this solver can be provoked with a call of operator(). For example:

```

UmfPackSolver umfpackSolver;
// With A largeMatrix; b and x are std::vector
umfpackSolver(A,b,x);

```

By calling the solver in this way, the storage and values of the `largeMatrix` *A* are extracted and these extractions are converted to the format of UMFPACK including: column pointer, row index and values. These three are inputs to underlying UMFPACK routines. Because extraction and conversion between `largeMatrix` of XLIFF++ and UMFPACK format are expensive, in some cases, it's better to directly make use of the conversion and it can be easily done by calling operator() with more inputs

```

// Solve AX=B with UmfPack
// \param[in] colPointer column pointer of CSC-like Umfpack format (after calling
// function extract2UmfPack of largeMatrix)
// \param[in] rowIdx row index of CSC-like Umfpack format (after calling function
// extract2UmfPack of largeMatrix)
// \param[in] values values of CSC-like Umfpack format (after calling function
// extract2UmfPack of largeMatrix)
// \param[in] vecB vector B (vector B should be std::vector)
// \param[in] vecX vector X (vector X should be std::vector)
template<class ScalarTypeMat, class VecB, class VecX>
void operator()(int size, const std::vector<int>& colPointer, const
std::vector<int>& rowIdx, const std::vector<ScalarTypeMat>& values, const VecB&
vecB, VecX& vecX, UmfPackComputationMode sys=A)

```

One of a technical issue on working with different types of scalar value, in this case, there are Real and Complex values, is the static dispatching. For dispatching at runtime, we can use simple *if-else* statements or the switch statement. However, the *if-else* statement requires both branches to compile successfully, even when the condition tested by *if* is known at compile time. To overcome this problem, we use a simple technique that is initially described in Alexandrescu, mapping integral Constants to Types:

```

template <int v>
struct Int2Type
{
enum { value = v };
};

```

`Int2Type` generates a distinct type for each distinct constant integral value passed. This is because different template instantiations are distinct types; thus, `Int2Type<0>` is different from `Int2Type<1>`, and so on. In addition, the value that generates the type is "saved" in the enum member value. For a linear system $Ax = b$, we can divide into four cases depending to the scalar type of A and b , and corresponding to each case, there is a `Int2Type`:

- A real and b real; `Int2Type<0>`
- A real and b complex; `Int2Type<1>`
- A complex and b real; `Int2Type<2>`
- A complex and b complex; `Int2Type<3>`

Because each case above needs processing in a different way, we define an internal class of `UmfPackSolver` to take care of it.

```
// Auxiliary internal class to process 4 cases of equation AX = B:
//   + A:real, B: real
//   + A:real, B:complex
//   + A:complex, B: real
//   + A:complex, B:complex
template<typename ScalarTypeMat, typename VecB, typename VecX>
class SolverIntern
{
public:
    typedef typename NumTraits<ScalarTypeMat>::magnitudeType MagType;
    typedef typename Conditional<NumTraits<ScalarTypeMat>::IsComplex,
        ScalarTypeMat, typename VectorTrait<VecB>::Type>::type ScalarType;

    SolverIntern() {}
    ...
};
```

For example, the case of A real and b real.

```
// Solve AX=B in case A Real, B Real
void solve(int sys, int nRows, int nCols, const std::vector<int>& colPointer, const
    std::vector<int>& rowIdx, const std::vector<ScalarTypeMat>& values, const VecB&
    vecB, VecX& vecX, Int2Type<0>)
{
    void* symbolic;
    void* numeric;
    double* null = (double*) NULL;

    UMFPACKint, ScalarType> umfPack;

    (void)umfPack.symbolic(nRows, nCols, &(colPointer[0]),
        &(rowIdx[0]),&(values[0]),&symbolic,null,null);
    (void)umfPack.numeric(&(colPointer[0]), &(rowIdx[0]), &(values[0]), symbolic,
        &numeric, null,null);
    (void)umfPack.solve((int)sys, &(colPointer[0]), &(rowIdx[0]),&(values[0]),
        &vecX[0], &vecB[0], numeric, null, null);
    umfPack.freeSymbolic(&symbolic);
    umfPack.freeNumeric(&numeric);
}
```

library :	<code>umfpackSupport</code>
header :	<code>UmfPackSolver.hpp</code>
implementation :	
unitary tests :	<code>test_UmfPackSolver.cpp</code>
header dependences :	<code>utils.h</code> , <code>UmfPack.hpp</code>

14.4 The UmfPackLU class

UMFPACK library allows not only to solve the linear system $Ax = b$ but also to retrieve more information. There are cases where users may wish to do more with the LU factorization of a matrix rather than solve a linear system. And the `UmfPackLU` class is for this purpose. Not like to the "non-templated" `UmfPackSolver` class, the `UmfPackLU` depends on the type of input matrix.

```
template<typename MatrixType>
class UmfPackLU
{
public:
    typedef typename MatrixType::ScalarType ScalarType;

    struct LUMatrixType {
        std::vector<int> outerIdx;
        std::vector<int> innerIdx;
        std::vector<ScalarType> values;
    };

public:
    UmfPackLU() { init(); }
    UmfPackLU(const MatrixType& matrix)
    {
        init();
        compute(matrix);
    }
    ...
}
```

Apart from solving the linear system $Ax = b$, this class provides some methods to work with LU factorization.

```
inline const LUMatrixType& matrixL() const
{
    if (extractedDataAreDirty_) extractData();
    return (lMatrix_);
}

inline const LUMatrixType& matrixU() const
{
    if (extractedDataAreDirty_) extractData();
    return (uMatrix_);
}

inline const std::vector<int>& permutationP() const
{
    if (extractedDataAreDirty_) extractData();
    return (pPerm_);
}

inline const std::vector<int>& permutationQ() const
{
    if (extractedDataAreDirty_) extractData();
    return (qPerm_);
}

inline const std::vector<Real>& scaleFactorR() const
{
    if (extractedDataAreDirty_) extractData();
    return (rScaleFact_);
}
```



```
}
```

All these methods above return the result of LU factorization. $PAQ = LU$. The matrix **U** is returned in compressed column form (with sorted columns). The matrix **L** is returned in compressed row form (with sorted rows). One remarkable thing is that the compressed column (row) form is similar to CSC (CSR) of XLIFF++. These two matrix are returned in form of struct **LUMatrixType**

```
struct LUMatrixType
{
    std::vector<int> outerIdx;
    std::vector<int> innerIdx;
    std::vector<ScalarType> values;
};
```

The *outerIdx* corresponds to the *colPointer* of CSC and *rowPointer* of CSR, while the *innerIdx* corresponds to the *rowIndex* of CSC and *colIndex* of CSR. Remember that, not like CSC or CSR, the *values* of **LUMatrixType** struct doesn't contain the "first-position" zero (0).

The permutations **P** and **Q** are represented as permutation vectors, where $P[k] = i$ means that row *i* of the original matrix is the *k*-th row of **PAQ**, and where $Q[k] = j$ means that column *j* of the original matrix is the *k*-th column of **PAQ**.

The vector **R** is the scale factor in the LU factorization. The first value of *scaleFactorR* defines how the scale factors **Rs** are to be interpreted. If it's one(1), then the scale factors *scaleFactorR[i+1]* are to be used by multiplying row *i* of matrix **A** by *scaleFactorR[i]*. Otherwise, the entries in row *i* are to be divided by *scaleFactorR[i+1]*.

Before retrieving all these above values, a matrix at least needs factorizing. It can be done in two ways, by constructor

```
UmfPackLU(const MatrixType& matrix)
{
    init();
    compute(matrix);
}
```

or by the method **compute**

```
/* Computes the sparse Cholesky decomposition of \a largeMatrix
 */
void compute(const MatrixType& matrix)
{
    analyzePattern(matrix);
    factorize(matrix);
}
```

In this function, a symbolic decomposition on the sparsity of a matrix is performed by **analyzePattern** then a numeric decomposition performed by **factorize**.

After the matrix is factorized, its factorization can be used to solve the linear system $Ax = b$ with :

```
template<typename VectorScalarType>
void solve(const std::vector<VectorScalarType>& b, std::vector<typename
    Conditional<NumTraits<ScalarType>::IsComplex, ScalarType,
    VectorScalarType>::type>& x) const;
```

One advantage of **UmfPackLU** over **UmfPackSolver** is that for a same matrix **A**, it only needs to do factorization once then all these factorizations can be used over and over again to solve the linear system $Ax = b$. Of course, it doesn't have the flexibility of "non-templated" **UmfPackSolver**, it needs template arguments in order to define type of matrix. Depending on a specific demand, one can be a better choice than another!!!

library :	umfpackSupport
header :	UmfPackLU.hpp
implementation :	
unitary tests :	test_UmfPackSolver.cpp
header dependences :	utils.h, UmfPack.hpp

The *eigenSolvers* library comes with built-in solvers of eigen problem which deal with sparse matrices. The user interface of eigen solvers is similar to the *solvers*: all solvers are invoked with operator().

To calculate eigenvalues and eigenvector, at the moment, nearly all finite-element libraries rely on a well-known large scale eigen solver ARPACK. Although this library is efficient and robust in computing varieties of eigen problem, it relies largely on other libraries: BLAS and LAPACK. The fact that in order to use ARPACK, users must already have these two installed, as well as all these libs are in Fortran, can make programming an efficient-C++ finite-element application a harsh work.

So as to ease user out of the complexities and dependencies, XLiFE++ supplies an intern eigensolver which is capable of solving large scale eigen problems with several algorithms. Two goals motivated the development of this intern eigensolver: independence and simplicity. The intention of *independence*, as mentioned above, is to free XLiFE++ from other Fortran libraries. The concept of *simplicity* drives the development of this part to allow users to call this eigen solver like other solvers (e.x: GmResSolver, ...) in XLiFE++. This is encouraged by promoting code modularization and multiple level of access to solvers and data. In the following, we outline the intern eigen solver structure. In particular, we present

- *eigenCore* the eigen solver of *dense* matrix
- *eigenSparse* the eigen solver of *sparse* matrix based on the *eigenCore*

15.1 *eigenCore* sub-library

Most of the standard eigenvalue algorithms exploit projection processes in order to extract approximate eigenvectors from a given subspace. The basic idea of a projection method is to extract an approximate eigenvector from a specified low-dimensional subspace. If these approximation can be extracted, a small matrix eigenvalue problem is obtained. The numerical solution of the small $m \times m$ eigenvalue problem will be treated by the *eigenCore* library. The *eigenCore* library is organized in multi-level, each of which utilizes subroutines provided by the lower one.

- *utils* : the lowest level provides various basic operations
- *decomposition* : the second level implements several decomposition algorithms
- *eigenSolver* : the highest level solves different eigen problem

15.1.1 *utils* part

This part contains several classes which work on different basic operations.

The `VectorEigenDense` class

Class `VectorEigenDense` is derived directly from `Vector`. Besides taking advantages of all functionalities of `Vector`, this class supplements some particular functions to calculate Householder transformation.

```
template<typename K>
class VectorEigenDense : public Vector<K>
{
public:
    typedef typename VectorEigenDense::type_t Scalar;
    typedef typename NumTraits<Scalar>::RealScalar RealScalar;
    typedef typename VectorEigenDense::it_vk it_vk;
    typedef typename VectorEigenDense::cit_vk cit_vk;

    VectorEigenDense() : Vector<K>(), acType_(-col) {}
    VectorEigenDense(const Dimen l) : Vector<K>(l, K()), acType_(-col) {}
    ...
}
```

This class offers some additional functions to calculate House holder values.

```
void makeHouseHolderInPlace(K& tau, Real& beta);
void makeHouseHolderInPlace(K& tau, Real& beta, Number tail);
void makeHouseHolder(VectorEigenDense<K>& essential, K& tau, Real& beta) const;
```

Note that in the future, we may improve calculation performance of eigensolver for *dense* matrix by changing this class `VectorEigenDense`. By using the same block of allocated data, we can gain more efficiencies. However, for the moment, it's sufficient to have this class inherited from class `Vector`

library :	eigenSolvers
header :	
implementation :	
unitary tests :	test_EigenSolverUtils.cpp
header dependences :	Traits.hpp

The MatrixEigenDense class

Class `MatrixEigenDense` inherits from the fundamental class `Matrix`. Basically, this class reuses all functionalities of its father in addition to some methods to support computing House holder, Cholesky composition and Jacobian rotation.

```
template<typename K>
class MatrixEigenDense : public Matrix<K>
{
public:
    typedef typename MatrixEigenDense<K>::type_t Scalar;
    typedef typename NumTraits<Scalar>::RealScalar RealScalar;
    typedef typename std::vector<K>::iterator it_vk;
    typedef typename std::vector<K>::const_iterator cit_vk;

    MatrixEigenDense() : Matrix<K>(), col_(1) {}
    MatrixEigenDense(const Dimen r) : Matrix<K>(r,r), col_(r) {}
    ...
}
```

Householder computation can be invoked with

```
void applyHouseholderOnTheLeft(const VectorEigenDense<K>& essential, const K& tau);
void applyHouseholderOnTheRight(const VectorEigenDense<K>& essential, const K& tau);
```

Cholesky decomposition can be calculated by

```
MatrixEigenDense<K> cholesky() const;
```

or one can call Jacobian rotation with

```
template<typename OtherScalar>
inline void applyOnTheLeft(Index p, Index q, const JacobiRotation<OtherScalar>& j);
template<typename OtherScalar>
inline void applyOnTheRight(Index p, Index q, const JacobiRotation<OtherScalar>& j);
```

Similar to class `VectorEigenDense`, in the future, this class may be improved to make *eigenCore* more efficient.

library :	eigenSolvers
header :	
implementation :	
unitary tests :	test_EigenSolverUtils.cpp
header dependencies :	Traits.hpp

The JacobiRotation class

Class `JacobiRotation` represents a *Jacobian* or *Givens* rotation, 2D rotation in a plane defined by its cosine and sine.

```
template<typename Scalar>
class JacobiRotation
{
public:
    typedef typename NumTraits<Scalar>::RealScalar RealScalar;

    /** Default constructor without any initialization. */
    JacobiRotation() {}

    /** Construct a planar rotation from a cosine-sine pair (\a c, \s). */
    JacobiRotation(const Scalar& c, const Scalar& s) : c_(c), s_(s) {}
}
```

```

Scalar& c() { return c_; }
Scalar c() const { return c_; }
Scalar& s() { return s_; }
Scalar s() const { return s_; }
...

```

The class provide two basic operations: making a Jacobi rotation and applying this rotation on both the right and left side of a selfadjoint 2×2 matrix; making a Givens rotation and applying this rotation on the left side of a vector.

```

template<typename Scalar>
bool JacobiRotation<Scalar>::makeJacobi(RealScalar x, Scalar y, RealScalar z);
template<typename Scalar>
void JacobiRotation<Scalar>::makeGivens(const Scalar& p, const Scalar& q, Scalar*
z);

```

```

library : eigenSolvers
header :
implementation :
unitary tests : test_EigenSolverUtils.cpp
header dependences : Traits.hpp

```

The NumTraits class

NumTraits greatly facilitate the management of the sort of extra parameters that come up during the implementation of eigensolver algorithm.

```

template<typename K>
struct NumTraits {};

template<>
struct NumTraits<Real> {
    enum {
        IsComplex = 0
    };
    static inline bool isReal() { return true;}
    static inline bool isComplex() { return false;}
    static inline Real zero() { return Real(0.0);}
    static inline Real one() { return Real(1.0);}
    static inline Real imag(const Real& v) { return v;}
    static inline Real real(const Real& v) { return v;}
    static inline Real value(const Real& v) { return v;}
    static inline Real abs2(const Real& v) { return v*v;}
    static inline Real epsilon() { return std::numeric_limits<Real>::epsilon(); }
    typedef Real RealScalar;
    typedef Complex ComplexScalar;
};

template<>
struct NumTraits<Complex> {
    enum {
        IsComplex = 1
    };
    static inline bool isReal() {return false;}
    static inline bool isComplex() { return true;}
    static inline Complex zero() { return Complex(0.0,0.0);}
    static inline Complex one() { return Complex(1.0,0.0);}
    static inline Real imag(const Complex& v) { return v.imag();}

```

```

static inline Real real(const Complex& v) { return v.real(); }
static inline Complex value(const Complex& v) { return v; }
static inline Real abs2(const Complex& v) { return v.real() * v.real() +
    v.imag() * v.imag(); }
typedef Real RealScalar;
typedef Complex ComplexScalar;
};

```

```

library : eigenSolvers
header :
implementation :
unitary tests : test_EigenSolverUtils.cpp
header dependencies : utils.h

```

15.1.2 *decomposition* part

All classes of this section utilize functions provided from the lower level - `utils` - to implement various decomposition algorithms, which play an important role in solving different eigen problems.

The Tridiagonalization class

Class `Tridiagonalization` performs a tridiagonal decomposition of a self-adjoint matrix A such that: $A = QTQ^*$ where Q is unitary and T a real symmetric tridiagonal matrix. A tridiagonal matrix is a matrix which has non-zero elements only on the main diagonal and the first diagonal below and above it. This class is used in `SelfAdjointEigenSolver` to compute the eigenvalues and eigenvectors of a selfadjoint matrix.

```

template<typename _MatrixType>
class Tridiagonalization
{
public:

    /** \brief Synonym for the template parameter \p _MatrixType. */
    typedef _MatrixType MatrixType;

    typedef typename MatrixType::type_t Scalar;
    typedef typename NumTraits<Scalar>::RealScalar RealScalar;
    typedef VectorEigenDense<Scalar> CoeffVectorType;
    typedef VectorEigenDense<RealScalar> DiagonalType;
    ...
    Tridiagonalization(Dimen size);
    Tridiagonalization(const MatrixType& matrix);
    ...

```

This class comes with default constructor with size of the matrix whose tridiagonal decomposition will be computed. In fact, this parameter is just a hint. After being created, object of this class can be used to calculate any square *dense* matrix with

```
Tridiagonalization& compute(const MatrixType& matrix);
```

The unitary matrix Q can only be invoked after calling the *compute*:

```
MatrixType matrixQ();
```

”Householder coefficients” can be retrieved by

```
inline CoeffVectorType householderCoefficients() const;
```

We don't need the full-form of matrix T in computation of eigenvalues; however its real diagonal and its real sub-diagonal are essential in calculating eigenvalues of the matrix. These two values can be retrieved with:

```
DiagonalReturnType diagonal() const;
SubDiagonalReturnType subDiagonal() const;
```

For example, the following 5×5 real symmetric matrix

$$A = \begin{bmatrix} 1.36 & -0.816 & 0.521 & 1.43 & -0.144 \\ -0.816 & -0.659 & 0.794 & -0.173 & -0.406 \\ 0.521 & 0.794 & -0.541 & 0.461 & 0.179 \\ 1.43 & -0.173 & 0.461 & -1.43 & 0.822 \\ -0.144 & -0.406 & 0.179 & 0.822 & -1.37 \end{bmatrix}$$

after being tridiagonalized, has diagonal and sub-diagonal vectors

diagonal = (1.36 -0.659 -0.541 -1.43 -1.37)

sub-diagonal = (-0.816 0.794 0.461 0.822)

library :	eigenSolvers
header :	
implementation :	
unitary tests :	test_EigenSolverDecomposition.cpp
header dependencies :	Traits.hpp, VectorEigenDense.hpp, HouseHolderSequence.hpp

The `HessenbergDecomposition` class

Class `HessenbergDecomposition` performs an Hessenberg decomposition of a matrix A . In the real case, the Hessenberg decomposition consists of an orthogonal matrix Q and a Hessenberg matrix H such that $A = QHQT$. An orthogonal matrix is a matrix whose inverse equals its transpose ($Q^{-1} = Q^T$). A Hessenberg matrix has zeros below the subdiagonal, so it is almost upper triangular. The Hessenberg decomposition of a complex matrix is $A = QHQQ^*$ with Q unitary (that is, $Q^{-1} = Q^*$).

```
template<typename _MatrixType>
class HessenbergDecomposition
{
public:

    /** \brief Synonym for the template parameter \p _MatrixType. */
    typedef _MatrixType MatrixType;

    /** \brief Scalar type for matrices of type #MatrixType. */
    typedef typename MatrixType::type_t Scalar;

    /** \brief Type for vector of Householder coefficients.
     *
     * This is column vector with entries of type #Scalar. The length of the
     * vector is one less than the size of #MatrixType,
     */
    typedef VectorEigenDense<Scalar> CoeffVectorType;

    /** \brief Return type of matrixQ() */
    typedef typename
        HouseholderSequence<MatrixType, CoeffVectorType>::ConjugateReturnType
        HouseholderSequenceType;

    /** \brief Default constructor; the decomposition will be computed later.
```



```

*
* \param [in] size The size of the matrix whose Hessenberg decomposition
* will be computed.
*
* The default constructor is useful in cases in which the user intends to
* perform decompositions via compute(). The \p size parameter is only
* used as a hint. It is not an error to give a wrong \p size, but it may
* impair performance.
*
* \sa compute() for an example.
*/
HessenbergDecomposition(Number size)
: matrix_(size, size),
  isInitialized_(false)
{
    if (size > 1)
        hCoeffs_.resize(size - 1);
}
...

```

We can use the constructor which computes the Hessenberg decomposition at construction time.

```
HessenbergDecomposition(const MatrixType&);
```

Once the decomposition is computed, we can use the `matrixH()` and `matrixQ()` functions to construct the matrices H and Q in the decomposition.

```
MatrixType matrixQ();
MatrixType matrixH() const;
```

Alternatively, calling the function `compute()` to compute the Hessenberg decomposition of a given matrix. then we can use these two functions above

```
HessenbergDecomposition& compute(const MatrixType& matrix);
```

For instance, the following 5×5 real symmetric matrix

$$A = \begin{bmatrix} 1.36 & -0.816 & 0.521 & 1.43 & -0.144 \\ -0.816 & -0.659 & 0.794 & -0.173 & -0.406 \\ 0.521 & 0.794 & -0.541 & 0.461 & 0.179 \\ 1.43 & -0.173 & 0.461 & -1.43 & 0.822 \\ -0.144 & -0.406 & 0.179 & 0.822 & -1.37 \end{bmatrix}$$

has Hessenberg decomposition

$$H = \begin{bmatrix} 1.36 & 1.7328972849 & 0 & 0 & 0 \\ 1.7328972849 & -1.1933420269 & -0.96533126903 & -9.1846580094e-17 & 2.021583842e-16 \\ 0 & -0.96533126903 & -1.2812029745 & 0.21380585197 & 1.9428902931e-16 \\ 0 & 0 & 0.21380585197 & -1.6885671008 & 0.34719629211 \\ 0 & 0 & 0 & 0.34719629211 & 0.16311210222 \end{bmatrix}$$

and unitary matrix

$$Q = \begin{bmatrix} 1 & 0 & 0 & 0 & 0 \\ 0 & -0.47088769029 & 0.12629876626 & -0.67227759101 & -0.55709626223 \\ 0 & 0.30065255716 & -0.19453390601 & 0.43672469898 & -0.82524913607 \\ 0 & 0.82520759451 & 0.045097442878 & -0.56297063107 & -0.0079192904815 \\ 0 & -0.083097827699 & -0.97168482632 & -0.20094388887 & 0.092438643767 \end{bmatrix}$$

library :	eigenSolvers
header :	
implementation :	
unitary tests :	test_EigenSolverDecomposition.cpp
header dependencies :	Traits.hpp, VectorEigenDense.hpp

The RealSchur class

Class `RealSchur` performs a real Schur decomposition of a square matrix. Given a real square matrix A , this class computes the real Schur decomposition: $A = UTU^T$ where U is a real orthogonal matrix and T is a real quasi-triangular matrix. An orthogonal matrix is a matrix whose inverse is equal to its transpose, $U^{-1} = U^T$. A quasi-triangular matrix is a block-triangular matrix whose diagonal consists of 1-by-1 blocks and 2-by-2 blocks with complex eigenvalues. The eigenvalues of the blocks on the diagonal of T are the same as the eigenvalues of the matrix A , and thus the real Schur decomposition is used in `RealEigenSolver` to compute the eigendecomposition of a matrix.

```
template<typename _MatrixType>
class RealSchur
{
public:
    typedef _MatrixType MatrixType;
    typedef typename MatrixType::type_t Scalar;
    typedef typename NumTraits<Scalar>::ComplexScalar ComplexScalar;

    typedef VectorEigenDense<ComplexScalar> EigenvalueType;
    typedef VectorEigenDense<Scalar> ColumnVectorType;

    /** \brief Default constructor.
     *
     * \param [in] size Positive integer, size of the matrix whose Schur
     * decomposition will be computed.
     *
     * The default constructor is useful in cases in which the user intends to
     * perform decompositions via compute(). The \p size parameter is only
     * used as a hint. It is not an error to give a wrong \p size, but it may
     * impair performance.
     *
     * \sa compute() for an example.
     */
    RealSchur(Index size)
        : matT_(size, size),
          matU_(size, size),
          hess_(size),
          maxIterations_(40),
          isInitialized_(false),
          matUisUptodate_(false),
          info_(noConvergence)
    { }
    ...
}
```

We can use the constructor which computes the real Schur decomposition at construction time.

```
RealSchur(const MatrixType& matrix, bool computeU = true);
```

Once the decomposition is computed, we can use the `matrixU()` and `matrixT()` functions to retrieve the matrices U and T in the decomposition.

```
MatrixType matrixU() const;
MatrixType matrixT() const;
```

Alternatively calling the function `compute()` to compute the real Schur decomposition of a given matrix, after that we are able to use these two functions above.

The implementation of Schur decomposition of real square matrix is adapted from <http://math.nist.gov/javanumerics/jama/>, whose code is based on EISPACK. The Schur decomposition is computed by first reducing the matrix to *Hessenberg* form using the class `HessenbergDecomposition`. The *Hessenberg* matrix is then reduced to triangular form by performing *FrancisQR* iterations with implicit double shift.

There are several auxiliary functions in order to implement this algorithm :

```
Scalar computeNormOff();
Index findSmallSubdiagEntry(Index iu, Scalar norm);
void splitOffTwoRows(Index iu, bool computeU, Scalar exshift);
void computeShift(Index iu, Index iter, Scalar& exshift, Vector3s& shiftInfo);
void initFrancisQRStep(Index il, Index iu, const Vector3s& shiftInfo, Index& im,
    Vector3s& firstHouseholderVector);
void performFrancisQRStep(Index il, Index im, Index iu, bool computeU, const
    Vector3s& firstHouseholderVector);
```

For instance, the following 5×5 non symmetric matrix

$$A = \begin{bmatrix} 1.8779 & -0.5583 & -0.2099 & 0.2696 & 0.1097 \\ 0.9407 & -0.3114 & -1.6989 & 0.4943 & 1.1287 \\ 0.7873 & -0.57 & 0.6076 & -1.4831 & -0.29 \\ -0.8759 & -1.0257 & -0.1178 & -1.0203 & 1.2616 \\ 0.3199 & -0.9087 & 0.6992 & -0.447 & 0.4754 \end{bmatrix}$$

has the quasi-triangular matrix in the Schur decomposition

$$H = \begin{bmatrix} 1.6552531332 & -0.30767059895 & -0.45435806468 & 1.5602779282 & 0.68358491675 \\ 0 & 1.1891303949 & 1.9306398993 & 0.96785347442 & 0.57970343469 \\ 0 & -1.3969331077 & 0.22387492903 & -0.78380738142 & 1.27402357 \\ 0 & 0 & 0 & 0.092450608769 & -0.5563858236 \\ 0 & 0 & 0 & 0 & -1.5315090659 \end{bmatrix}$$

and the orthogonal matrix in the Schur decomposition.

$$Q = \begin{bmatrix} 0.72665944996 & 0.28908942548 & -0.090578168382 & 0.61658006012 & 0.0042394246225 \\ 0.13365823926 & 0.75252664471 & -0.11720311204 & -0.52512343394 & -0.35545454525 \\ 0.50462706182 & -0.59105620177 & -0.20754951129 & -0.34476029525 & -0.4838053679 \\ -0.1068784212 & 0.010284495663 & 0.747282602 & 0.23512571367 & -0.61217305049 \\ 0.43362431945 & -0.026033604674 & 0.61363746549 & -0.41222645012 & 0.51459752796 \end{bmatrix}$$

library :	eigenSolvers
header :	
implementation :	
unitary tests :	test_EigenSolverDecomposition.cpp
header dependences :	Jacobi.hpp, VectorEigenDense.hpp, HessenbergDecomposition.hpp

15.1.3 *eigenSolver* main part

In this section, there are solvers for three kinds of eigen problem :

- Hermitian eigenproblem
- Generalized Hermitian eigenproblem
- Non-Hermitian (real) eigenproblem

The following presents eigen solver classes corresponding to each kind of eigen problem.

The SelfAdjointEigenSolver class

Class `SelfAdjointEigenSolver` computes eigenvalues and eigenvectors of self-adjoint matrices. A matrix A is selfadjoint if it equals its adjoint. For real matrices, this means that the matrix is symmetric: it equals its transpose.

This class computes the eigenvalues and eigenvectors of a self-adjoint matrix. These are the scalars λ and vectors v such that $Av = \lambda v$. The eigenvalues of a self-adjoint matrix are always real. If D is a diagonal matrix with the eigenvalues on the diagonal, and V is a matrix with the eigenvectors as its columns, then $A = VDV^{-1}$ (for self-adjoint matrices, the matrix V is always invertible). This is called the eigendecomposition. The implemented algorithm exploits the fact that the matrix is self-adjoint, making it faster and more accurate than the general purpose eigenvalue algorithms implemented in `RealEigenSolver`.

```
template<typename _MatrixType>
class SelfAdjointEigenSolver
{
public:

    typedef _MatrixType MatrixType;

    /** \brief Scalar type for matrices of type \p _MatrixType. */
    typedef typename MatrixType::type_t Scalar;

    /** \brief Real scalar type for \p _MatrixType.
     *
     * This is just \c Scalar if #Scalar is real (e.g., \c float or
     * \c double), and the type of the real part of \c Scalar if #Scalar is
     * complex.
     */
    typedef typename NumTraits<Scalar>::RealScalar RealScalar;

    /** \brief Type for vector of eigenvalues as returned by eigenvalues().
     *
     * This is a column vector with entries of type #RealScalar.
     * The length of the vector is the size of \p _MatrixType.
     */
    typedef VectorEigenDense<RealScalar> RealVectorType;
    typedef Tridiagonalization<MatrixType> TridiagonalizationType;

    /** \brief Default constructor for fixed-size matrices.
     *
     * The default constructor is useful in cases in which the user intends to
     * perform decompositions via compute().
     */
    SelfAdjointEigenSolver()
        : eivec_(),
          eivalues_(),
          subdiag_(),
          maxIterations_(30),
          isInitialized_(false)
    { }
    ...
}
```

We can use the constructor which computes the eigenvalues and eigenvectors at construction time.

```
SelfAdjointEigenSolver(const MatrixType& matrix, Int options = _computeEigenvector);
```

Once the eigenvalue and eigenvectors are computed, they can be retrieved with

```
const RealVectorType& eigenvalues() const;
const MatrixType& eigenvectors() const;
```

Alternatively, we can call the function `compute()` to compute the eigenvalues and eigenvectors of a given matrix.

```
SelfAdjointEigenSolver& compute(const MatrixType& matrix, Int options =
    _computeEigenvector);
```

This implementation uses a symmetric QR algorithm. The matrix is first reduced to tridiagonal form using the `Tridiagonalization` class. The tridiagonal matrix is then brought to diagonal form with implicit symmetric QR steps with Wilkinson shift.

For instance, the following 5×5 real symmetric matrix

$$A = \begin{bmatrix} 1.36 & -0.816 & 0.521 & 1.43 & -0.144 \\ -0.816 & -0.659 & 0.794 & -0.173 & -0.406 \\ 0.521 & 0.794 & -0.541 & 0.461 & 0.179 \\ 1.43 & -0.173 & 0.461 & -1.43 & 0.822 \\ -0.144 & -0.406 & 0.179 & 0.822 & -1.37 \end{bmatrix}$$

has eigenvalues and corresponding (column) eigenvector

eigenvalues = (-2.64925 -1.76682 -0.742793 0.22723 2.29163)

eigenvectors = $\begin{bmatrix} -0.32648814917 & -0.098193931162 & 0.34709885718 & -0.010999922777 & 0.87359305479 \\ -0.20806598152 & -0.64229394812 & 0.22733262857 & 0.66233712946 & -0.23194058392 \\ 0.050094522003 & 0.62960023585 & -0.16376394055 & 0.73987891396 & 0.16387387324 \\ 0.72069515152 & -0.39678270513 & -0.40156098976 & 0.11452959441 & 0.38573789934 \\ -0.57289010572 & -0.15486595502 & -0.79985773898 & -0.025508690809 & 0.085967242812 \end{bmatrix}$

library :	eigenSolvers
header :	
implementation :	
unitary tests :	test_EigenSolverInternCore.cpp
header dependences :	VectorEigenDense.hpp, Tridiagonalization.hpp

The GeneralizedSelfAdjointEigenSolver class

Class `GeneralizedSelfAdjointEigenSolver` computes eigenvalues and eigenvectors of the generalized self-adjoint eigen problem.

This class solves the generalized eigenvalue problem $Av = \lambda Bv$. In this case, the matrix A should be self-adjoint and the matrix B should be positive definite.

```
template<typename _MatrixType>
class GeneralizedSelfAdjointEigenSolver : public SelfAdjointEigenSolver<_MatrixType>
{
    typedef SelfAdjointEigenSolver<_MatrixType> Base;
public:
    typedef _MatrixType MatrixType;

    /** \brief Default constructor for fixed-size matrices.
     *
     * The default constructor is useful in cases in which the user intends to
     * perform decompositions via compute().
     */
    GeneralizedSelfAdjointEigenSolver() : Base() {}

    /** \brief Constructor, pre-allocates memory for dynamic-size matrices.
```

```

*
* \param [in] size Positive integer, size of the matrix whose
* eigenvalues and eigenvectors will be computed.
*
* \sa compute() for an example
*/
GeneralizedSelfAdjointEigenSolver(Index size)
: Base(size)
{}
...

```

We can use the constructor which computes the eigenvalues and eigenvectors at construction time.

```

GeneralizedSelfAdjointEigenSolver(const MatrixType& matA, const MatrixType&
matB,
                                int options = _computeEigenvector | _Ax_lBx)
: Base(matA.numOfCols());

```

Alternatively, we can call the function `compute()` to compute the eigenvalues and eigenvectors of a given matrix.

```

GeneralizedSelfAdjointEigenSolver& compute(const MatrixType& matA, const
MatrixType& matB,
                                           int options =
                                           _computeEigenvector | _Ax_lBx);

```

Eigenvalues and eigenvectors can be retrieved with same functions of `SelfAdjointEigenSolver`

```

const RealVectorType& eigenvalues() const;
const MatrixType& eigenvectors() const;

```

For instance, the following 5×5 real symmetric matrix A and positive definite matrix B

$$A = \begin{bmatrix} 1.36 & -0.816 & 0.521 & 1.43 & -0.144 \\ -0.816 & -0.659 & 0.794 & -0.173 & -0.406 \\ 0.521 & 0.794 & -0.541 & 0.461 & 0.179 \\ 1.43 & -0.173 & 0.461 & -1.43 & 0.822 \\ -0.144 & -0.406 & 0.179 & 0.822 & -1.37 \end{bmatrix}$$

$$B = \begin{bmatrix} 1.1981 & 1.4405 & 1.5367 & 1.0971 & 1.1276 \\ 1.4405 & 2.1207 & 1.9115 & 1.3836 & 1.642 \\ 1.5367 & 1.9115 & 2.4828 & 1.7702 & 1.7543 \\ 1.0971 & 1.3836 & 1.7702 & 1.4359 & 1.2428 \\ 1.1276 & 1.642 & 1.7543 & 1.2428 & 1.424 \end{bmatrix}$$

has eigenvalues and corresponding (column) eigenvector

eigenvalues = (-40.0667 -12.6587 -3.08602 0.0522151 40.3448)

$$\text{eigenvectors} = \begin{bmatrix} -2.5241477328 & -0.01326907506 & 0.83333293087 & -0.040348533316 & 4.3939678718 \\ 3.7607674197 & 0.3488243247 & 1.1073133113 & -0.3174355818 & -4.3303408407 \\ 3.0396247156 & 2.615639864 & -0.34676637691 & -0.36756457169 & -3.9258019948 \\ 0.45843793898 & -2.0051996759 & -0.65863554474 & -0.039469282032 & 1.4762595546 \\ -6.6376554636 & -1.9381010232 & -0.97022016252 & 0.070809133211 & 4.9718019236 \end{bmatrix}$$

library :	eigenSolvers
header :	
implementation :	
unitary tests :	test_EigenSolverInternCore.cpp
header dependences :	Tridiagonalization.hpp

The RealEigenSolver class

Class `RealEigenSolver` computes eigenvalues and eigenvectors of general real matrices.

The eigenvalues and eigenvectors of a matrix A are scalars λ and vectors v such that $Av = \lambda v$. If D is a diagonal matrix with the eigenvalues on the diagonal, and V is a matrix with the eigenvectors as its columns, then $AV = VD$. The matrix V is almost always invertible, in which case we have $A = VDV^{-1}$. The eigenvalues and eigenvectors of the matrix may be complex, even when the matrix is real. However, we can choose real matrices V and D satisfying $AV = VD$, just like the eigendecomposition, if the matrix D is not required to be diagonal, but if it is allowed to have blocks of the form

$$\begin{bmatrix} u & v \\ -v & u \end{bmatrix}$$

(where u and v are real numbers) on the diagonal. These blocks correspond to complex eigenvalue pairs $u \pm iv$. We call this variant of the eigendecomposition the pseudo-eigendecomposition.

```
template<typename _MatrixType>
class RealEigenSolver
{
public:

    /** \brief Synonym for the template parameter \p _MatrixType. */
    typedef _MatrixType MatrixType;

    /** \brief Scalar type for matrices of type #MatrixType. */
    typedef typename MatrixType::type_t Scalar;
    typedef typename NumTraits<Scalar>::RealScalar RealScalar;
    typedef VectorEigenDense<Scalar> ColumnVectorType;
    typedef VectorEigenDense<Scalar> VectorType;

    /** \brief Complex scalar type for #MatrixType.
     *
     * This is \c std::complex<Scalar> if #Scalar is real (e.g.,
     * \c float or \c double) and just \c Scalar if #Scalar is
     * complex.
     */
    typedef typename NumTraits<Scalar>::ComplexScalar ComplexScalar;
    typedef VectorEigenDense<ComplexScalar> ComplexVectorType;

    /** \brief Type for vector of eigenvalues as returned by eigenvalues().
     *
     * This is a column vector with entries of type #ComplexScalar.
     * The length of the vector is the size of #MatrixType.
     */
    typedef VectorEigenDense<ComplexScalar> EigenvalueType;

    /** \brief Type for matrix of eigenvectors as returned by eigenvectors().
     *
     * This is a square matrix with entries of type #ComplexScalar.
     * The size is the same as the size of #MatrixType.
     */
    typedef MatrixEigenDense<ComplexScalar> EigenvectorsType;

    /** \brief Default constructor.
     *
     * The default constructor is useful in cases in which the user intends to
     * perform decompositions via RealEigenSolver::compute(const MatrixType&,
     * bool).
     *
     * \sa compute() for an example.
     */
    RealEigenSolver() : eivec_(), eivalues_(), isInitialized_(false), realSchur_(),
        matT_() {}
};
```

```

/**
 * \brief Default constructor with memory preallocation
 *
 * Like the default constructor but with preallocation of the internal data
 * according to the specified problem \a size.
 * \sa RealEigenSolver()
 */
RealEigenSolver(Index size)
: eivec_(size, size),
  eivalues_(size),
  isInitialized_(false),
  eigenvectorsOk_(false),
  realSchur_(size),
  matT_(size, size)
{}
...

```

Same as other two eigensolvers, we can use the constructor which computes the eigenvalues and eigenvectors at construction time.

```
RealEigenSolver(const MatrixType& matrix, bool computeEigenvectors = true);
```

Once the eigenvalue and eigenvectors are computed, they can be retrieved with

```
const EigenvalueType& eigenvalues() const;
EigenvectorsType eigenvectors() const;
```

Alternatively, we can call the function `compute()` to compute the eigenvalues and eigenvectors of a given matrix.

```
RealEigenSolver& compute(const MatrixType& matrix, bool computeEigenvectors = true);
```

This implementation first reduces the matrix to real Schur form using the `RealSchur` class. The Schur decomposition is then used to compute the eigenvalues and eigenvectors.

For instance, the following 5×5 non symmetric matrix

$$A = \begin{bmatrix} 1.8779 & -0.5583 & -0.2099 & 0.2696 & 0.1097 \\ 0.9407 & -0.3114 & -1.6989 & 0.4943 & 1.1287 \\ 0.7873 & -0.57 & 0.6076 & -1.4831 & -0.29 \\ -0.8759 & -1.0257 & -0.1178 & -1.0203 & 1.2616 \\ 0.3199 & -0.9087 & 0.6992 & -0.447 & 0.4754 \end{bmatrix}$$

has eigenvalues and corresponding (column) eigenvector

eigenvalues = ((1.65525,0) (0.706503,1.56973) (0.706503,-1.56973) (0.0924506,0) (-1.53151,0))

$$\text{eigenvectors} = \begin{bmatrix} (0.727, 0) & (0.177, -0.001) & (0.177, 0.001) & (-0.230, 0) & (-0.041, 0) \\ (0.133, 0) & (0.534, -0.218) & (0.534, 0.218) & (-0.664, 0) & (-0.392, 0) \\ (0.504, 0) & (-0.444, 0.0814) & (-0.444, -0.0814) & (-0.336, 0) & (-0.587, 0) \\ (-0.106, 0) & (0.011, 0.457) & (0.0118, -0.457) & (0.135, 0) & (-0.695, 0) \\ (0.433, 0) & (-0.036, 0.469) & (-0.036, -0.469) & (-0.611, 0) & (-0.121, 0) \end{bmatrix}$$

library :	eigenSolvers
header :	
implementation :	
unitary tests :	test_EigenSolverInternCore.cpp
header dependences :	VectorEigenDense.hpp, MatrixEigenDense.hpp, RealSchur.hpp

15.2 *eigenSparse* sub-library

The *eigenSparse* provides tools that are useful for solving a wide variety of eigenvalue problems. The solvers currently implemented within *eigenSparse* compute a partial eigen-decomposition for the generalized eigenvalue problem

$$Ax = Bx\lambda$$

We assume that the matrices A and B are large, possibly sparse, and that only their application to a block of vectors is required.

This section outlines the *eigenSparse*. Three subsections describe the *eigenSparse* operator/multivector interface, the eigensolver, and the various implementations provided by the *eigenSparse*.

The lib *eigenSparse* is highly independent of other packages of XLiFE++ except for *eigenCore* which provides essential functions in performing the dense arithmetic for Rayleigh-Ritz methods.

15.2.1 The Operator/Multivector Interface

eigenSparse utilizes traits classes to define interfaces for the scalar field, multivectors, and matrix operators. This allows generic programming techniques to be used when developing numerical algorithms in the *eigenSparse*. *eigenSparse*'s eigensolver is comprised of abstract numerical interfaces that are all implemented using templates and the functionality of the template arguments is provided through their corresponding trait classes. Most classes in *eigenSparse* accept three template parameters:

- a scalar type, describing the field over which the vectors and operators are defined;
- a multivector type over the given scalar field, providing a data structure that denotes a collection of vectors; and
- an operator type over the given scalar field, providing linear operators used to define eigenproblems and preconditioners.

We note that the *eigenSparse* was designed to support block methods, defined as those that apply A or B to a collection of vectors (a multivector). Algorithms in *eigenSparse* are developed at a high-level, where the underlying linear algebra objects are opaque. The choice in linear algebra is made through templating, and access to the functionality of the underlying objects is provided via the traits classes *MultiVecTraits* and *OperatorTraits*.

These classes define opaque interfaces, specifying the operations that multivector and operator classes must support in order to be used in *eigenSparse* without exposing low-level details of the underlying data structures.

The *MultiVecTraits* and *OperatorTraits* classes specify the operations that the multivector and operator type must support in order for them to be used by *eigenSparse*.

The *OperatorTraits* class only needs to provide one method, described in the Table, that applies an operator to a multivector. This interface defines the only interaction required from an operator, even though the underlying operator may be a matrix, spectral transformation, or preconditioner.

OperatorTraits<ST,MV,OP>	
Method name	Description
<code>apply(A,X,Y)</code>	Applies the operator A to the multivector X, placing the result in the multivector Y

The methods defined by the *MultiVecTraits* class, listed in following table, are the nesseary creational and arithmetic ones. The creational methods generate empty or populated multivectors from a previously created multivector. The populated multivectors can be a deep copy, where the object contains the storage for the multivector entries, or a shallow copy, where the object has a view of another multivector's

storage. A shallow copy is useful when only a subset of the columns of a multivector is required for computation, which is a situation that commonly occurs during the generation of a subspace.

MultiVecTraits<ST,MV>	
<i>Method name</i>	<i>Description</i>
<code>clone(X,numvecs)</code>	Creates a new multivector from X with <i>numvecs</i> vectors
<code>cloneCopy(X,index)</code>	Creates a new multivector with a copy of the contents of a subset of the multivector X (deep copy)
<code>cloneView(X,index)</code>	Creates a new multivector that shares the selected contents of a subset of the multivector X (shallow copy)
<code>getVecLength(X)</code>	Returns the vector length of the multivector X
<code>getNumberVecs(X)</code>	Returns the number of vectors in the multivector X
<code>mvTimesMatAddMv(alpha,X,D,beta,Y)</code>	Applies a dense matrix D to multivector X and accumulates the result into multivector Y: $Y \leftarrow \alpha XD + \beta Y$
<code>mvAddMv(alpha,X,beta,Y)</code>	Performs multivector: $Y \leftarrow \alpha X + \beta Y$
<code>mvTransMv(alpha,X,Y,D)</code>	Computes the dense matrix $D \leftarrow \alpha X^H Y$
<code>mvDot(X,Y,d)</code>	Computes the corresponding dot products: $d[i] \leftarrow \bar{a}_i y_i$
<code>mvScale(X,d)</code>	Scales the i-th column of a multivector X by $d[i]$
<code>mvNorm(X,d)</code>	Computes the 2-norm of each vector of X : $d[i] \leftarrow \ x_i\ _2$
<code>setBlock(X,Y,index)</code>	Copies the vectors in X to a subset of vectors in Y
<code>mvInit(X,alpha)</code>	Replaces each entry in the multivector X with a scalar α
<code>mvRandom(X)</code>	Replaces the entries in the multivector X by random scalars
<code>mvPrint(X)</code>	Print the multivector X

The arithmetic methods defined by the `MultiVecTraits` are essential to the computations required by the Rayleigh-Ritz method and the general eigen-iteration. The `mvTimesMatAddMv` and `mvAddMv` methods, for instance, are necessary for updating the approximate eigenpairs and their residuals. The `mvDot` and `mvTransMv` methods are required by the orthogonalization procedure of the eigen-iteration. The `mvScale` and `mvNorm` methods are necessary, at the very least, for the computation of approximate eigenpairs and for some termination criteria of the eigen-iteration. Deflation and locking of converged eigenvectors necessitates the `setBlock` method in many cases. Initialization of the bases for the eigen-iteration requires methods such as `mvRandom` and `mvInit`. The ability to perform error checking and debugging is supported by methods that give dimensional attributes (`getVecLength`, `getNumberVecs`) and allow the users to print out a multivector (`mvPrint`).

Calling methods of `MultiVecTraits` and `OperatorTraits` requires that specializations of these traits classes have been implemented for given template arguments. XLIFE++ provides the following classes that can be used with these traits classes:

- `MultiVectorAdapter`, an extended version of `Vector`, allowing the process on block of vector.
- `LargeMatrixAdapter`, a wrapper of `LargeMatrix` in order to take advantage of multiplication matrix-vector of sparse matrix

The MultiVectorAdapter class

This class provides the implementations of the interfaces along with abstract `MultiVec` class. By inheriting directly from `MultiVec` class, the implementation works via run-time polymorphism. This option isn't optimal from performance point of view because of the dispatching cost during run-time. In the near future, we can improve it by implementing a specialization of `MultiVecTraits` class, which allows compile-time polymorphism.

By using the pointer to each vector, `MultiVectorAdapter` class allows to create different objects, which share a same view of memory (shallow copy). This feature is essential when we want only to work on specific column vectors of a multivector.

```

template <class ScalarType>
class MultiVecAdapter : public MultiVec<ScalarType>
{
public:
    typedef std::vector<ScalarType>* VectorPointer;

    ///! Constructor for a \c numberVecs vectors of length \c length.
    MultiVecAdapter(const Number length, const Dimen numberVecs) :
        length_(length),
        numberVecs_(numberVecs)
    {
        check();

        data_.resize(numberVecs);
        ownership_.resize(numberVecs);

        // Allocates memory to store the vectors.
        for (Dimen v = 0 ; v < numberVecs_ ; ++v)
        {
            data_[v] = new std::vector<ScalarType>(length);
            ownership_[v] = true;
        }

        // Initializes all elements to zero.
        mvInit(0.0);
    }
    ...

```

The LargeMatrixAdapter class

This class is simply a wrapper of `LargeMatrix` class in order to take advantage of consuming-time matrix-vector multiplication provided by XLiFE++. The method `apply` implements multiplication of a `LargeMatrix` with a block of vector. Noting that it's only a way to implement application of an operator to vector. Depending on specific eigenproblem, we can create a new class with a different approach to apply operator; for example, operator in case of spectral transformation to find the not-well separated eigenvalues.

```

template<typename ScalarType>
class LargeMatrixAdapter : public Operator<ScalarType>
{
private:
    typedef LargeMatrix<ScalarType> LMatrix;
public:
    LargeMatrixAdapter() { lMatrix_p = NULL; }
    LargeMatrixAdapter(LMatrix* p) { lMatrix_p = p; }

    LargeMatrixAdapter& operator=(LMatrix matrix) { lMatrix_p = &matrix; return *this; }

    ///! Destructor.
    virtual ~LargeMatrixAdapter() {};

    void apply(const MultiVec<ScalarType>& x, MultiVec<ScalarType>& y) const
    {
        for (int i = 0; i < x.getNumberVecs(); ++i) {
            multMatrixVector(*lMatrix_p, *(x[i]), *(y[i]));
        }
    }
private:
    LMatrix* lMatrix_p;

```

```
};
```

The LargeMatrixAdapterInverse class

Similar to [LargeMatrixAdapter](#), this class is simply a wrapper of [LargeMatrix](#) in order to take advantage of consuming-time matrix-vector multiplication provided by XLiFE++. The class operates on an already factorized matrix to carry out the multiplication of its inverse (if invertible) with a vector. Instead of invoking the function [multMatrixVector](#), the method [apply](#) calls the routine [multInverMatrixVector](#) of class [LargeMatrix](#) with the type of factorization.

```
template<typename MatrixType, typename ScalarType>
class LargeMatrixAdapterInverse : public Operator<ScalarType>
{
private:
    typedef const MatrixType LMatrix;
public:
    // LargeMatrixAdapterInverse() : lMatrix_p(NULL), fac_(_noFactorization) {}
    LargeMatrixAdapterInverse(LMatrix* p, FactorizationType fac) : lMatrix_p(p),
        fac_(fac) {}

    LargeMatrixAdapterInverse& operator=(LMatrix matrix) { lMatrix_p = &matrix;
        fac_ = _noFactorization; return (*this); }

    //! Destructor.
    virtual ~LargeMatrixAdapterInverse() {};

    void apply(const MultiVec<ScalarType>& x, MultiVec<ScalarType>& y) const
    {
        for (int i = 0; i < x.getNumberVecs(); ++i) {
            multInverMatrixVector(*lMatrix_p, *(x[i]), *(y[i]), fac_);
        }
    }

private:
    LMatrix* lMatrix_p;
    FactorizationType fac_;
};
```

The LargeMatrixAdapterInverseGeneralized class

On some specific eigen problems, it is better to do spectral transformation, especially, to find not well-separated eigenvalues. This class serves for this purpose. As indicated by its name, this class works mainly for the spectral transformation of generalized eigen problem. The first operation is multiplication of the one matrix with a vector. The vector result is multiplied by the inverse of the second matrix which is provided in a form of factorized one. Both operations are done in the routine [apply](#).

```
template<typename MatrixType, typename ScalarType>
class LargeMatrixAdapterInverseGeneralized : public Operator<ScalarType>
{
private:
    typedef const MatrixType LMatrix;
public:
    LargeMatrixAdapterInverseGeneralized(LMatrix* pA, LMatrix* pB,
        FactorizationType fac) : lMatrixA_p(pA), lMatrixB_p(pB), fac_(fac) {}

    // LargeMatrixAdapterInverseGeneralized& operator=(LMatrix matrix) { lMatrix_p
    = &matrix; fac_ = _noFactorization; return (*this); }
```

```

    //! Destructor.
    virtual ~LargeMatrixAdapterInverseGeneralized() {};

    void apply(const MultiVec<ScalarType>& x, MultiVec<ScalarType>& y) const
    {
        Vector<ScalarType> vecTemp(x.getVecLength());
        for (int i = 0; i < x.getNumberVecs(); ++i) {
            multMatrixVector(*lMatrixB_p, *(x[i]), vecTemp);
            multInverMatrixVector(*lMatrixA_p, vecTemp, *(y[i]), fac_);
        }
    }

private:
    LMatrix* lMatrixA_p;
    LMatrix* lMatrixB_p;
    FactorizationType fac_;
};

```

15.2.2 *eigenSparse* framework

eigenSparse is organized with a multi-tiered access for eigensolver algorithms. Users have the choice of interfacing at one of two levels: either working at a high-level with a eigensolver manager or working at a low-level directly with an eigensolver.

Consider as an example the block Davidson iteration.

Block Davidson Algorithm

Require: Set an initial guess \mathbf{V} and $\mathbf{H} = \mathbf{I}$

- 1: **for** $iter = 1$ to $iter_{max}$ **do**
 - 2: **repeat**
 - 3: Compute \mathbf{M} -orthonormal basis \mathbf{V} for $[\mathbf{V}, \mathbf{H}]$
 - 4: Project \mathbf{A} onto \mathbf{V} : $\hat{A} = \mathbf{V}^H \mathbf{A} \mathbf{V}$
 - 5: Compute selected eigenpairs (\mathbf{Q}, Γ) of \hat{A} : $\hat{A} \mathbf{Q} = \mathbf{Q} \Gamma$
 - 6: Compute Ritz vectors: $\mathbf{X} = \mathbf{V} \mathbf{Q}$
 - 7: Compute residuals: $\mathbf{R} = \mathbf{A} \mathbf{X} - \mathbf{M} \mathbf{X} \Gamma$
 - 8: Precondition the residuals: $\mathbf{H} = \mathbf{N} \mathbf{R}$
 - 9: Check convergence and possibly terminate iteration
 - 10: **until** the matrix \mathbf{V} is not expandable
 - 11: Restart \mathbf{V}
 - 12: **end for**
-

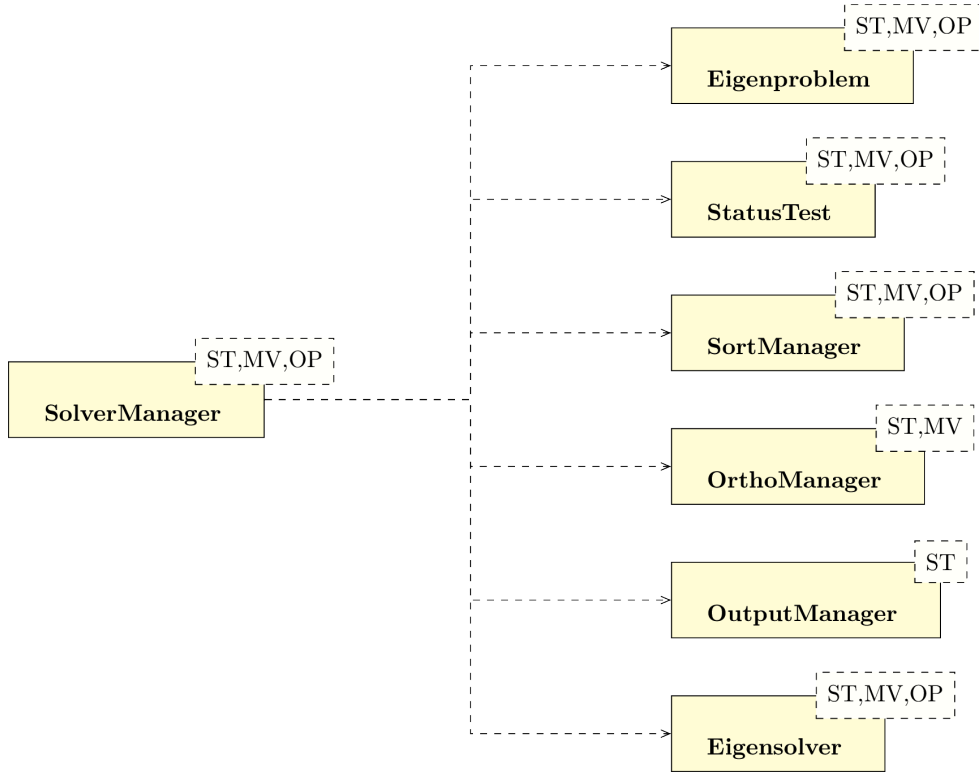


Figure 15.1: **SolverManager** class collaboration graph

The linear operators \mathbf{A} and \mathbf{M} define the eigenproblem to be solved. The linear operator \mathbf{N} is a preconditioner for the problem, and its application to a block of vectors is required. Example preconditioners \mathbf{N} include the inverse of the diagonal of \mathbf{A} (Jacobi preconditioner), an algebraic multigrid preconditioner for \mathbf{A} , or a preconditioned iteration that approximates the solution of a linear set of equations with \mathbf{A} , e.g., the preconditioned conjugate gradient algorithm.

The *multivector* \mathbf{V} , \mathbf{X} , and \mathbf{R} serve for some purpose. The column-vectors for matrix \mathbf{V} form the basis for the Rayleigh-Ritz approximation conducted at Step 5-6. We make the specific choice that these vectors are orthonormal with respect to the inner product induced by the Hermitian positive-definite matrix \mathbf{M} , but this is not a requirement.

The block Davidson eigensolver as described above is useful for examining some of the functionality provided by *eigenSparse*. The algorithm above highlights three levels of functionality. The first level is given by Steps 1-12 that constitute the eigensolver strategy to solve problem (2). The second level is given by Steps 2-10 that form the eigen-iteration. The third level consists of computational steps that can be implemented in a variety of manners, so encouraging modularization. Step 3 requires an orthonormalization method. The decisions involved in Step 5 require a determination of the eigenvalues and invariant subspace of interest, as well as a definition of accuracy. To check convergence in Step 9, several criteria are possible. For instance, a norm induced by a matrix other than \mathbf{M} may be employed. The restarting of this eigensolver (Step 11) can be performed in a variety of ways and therefore need not be tightly coupled to the eigensolver. Each of these mechanisms provide opportunity for decoupling functionality that need not be implemented in a specific manner.

Looking deeper into the algorithm, the iteration repeats until the basis \mathbf{V} is full (in which case it is time to restart) or some stopping criterion has been satisfied. Many valid stopping criteria exist, as well as many different methods for restarting the basis. Both of these, however, are distinct from the essential iteration as described above. A user wanting to perform block Davidson iterations could ask the solver to perform these iterations until a user-specified stopping criterion was satisfied or the basis was full, at which time the user would perform a restart. This allows the user complete control over the stopping criteria and the restarting mechanism, and leaves the eigensolver responsible for a relatively simple bit

of state and behavior.

The eigensolvers (encapsulating an iteration and the associated state) are derived classes of the abstract base class `Eigensolver`. The goals of this class are three-fold:

- to define an interface used for checking the status of a solver by a status test;
- to contain the essential iteration associated with a particular eigensolver iteration;
- to contain the state associated with that iteration.

The status tests, assembled to describe a specific stopping criterion and queried by the eigensolver during the iteration, are represented as subclasses of `StatusTest`. The communication between status test and eigensolver occurs inside of the `iterate()` method provided by each `Eigensolver`. This code generally takes the form:

```
SomeEigensolver::iterate() {  
  while ( statustest.checkStatus(this) != _passed ) {  
    //  
    // perform eigensolver iterations  
    //  
  }  
}
```

Each `StatusTest` provides a method, `checkStatus()`, which through queries to the methods provided by `Eigensolver`, determines whether the solver meets the criteria defined by that particular status test. After a solver returns from `iterate()`, the caller has the option of accessing the state associated with the solver and re-initializing the solver with a new state.

While this method of interfacing with the solver is powerful, it can be tedious. This method requires that user construct a number of support classes, in addition to managing call to `Eigensolver::iterate()`. The `SolverManager` class was developed to address this complaint. A solver manager is a class that wraps around an eigensolver, providing additional functionality while also handling lower-level interaction with the eigensolver that a user may not wish to handle. Solver managers are intended to be easy to use, while still providing the features and flexibility needed to solve real-world eigenvalue problems. For example, the `BlockDavidsonSolMgr` takes only two arguments in its constructor: an `Eigenproblem` specifying the eigenvalue problem to be solved and a `Parameters` of options specific to this solver manager. The solver manager instantiates an eigensolver, along with the status tests and other support classes needed by the eigensolver. To solve the eigenvalue problem, the user simply calls the `solve()` method of the solver manager. The solver manager performs repeated calls to the eigensolver `iterate()` method, performs restarts and locking, and places the final solution into the eigenproblem.

15.2.3 *eigenSparse* classes

The following lists and briefly describes the current classes of *eigenSparse*.

The `EigenProblem` class

`EigenProblem` is a container for the components of an eigenvalue problem, as well as the solutions. By requiring eigen problems to derive from `EigenProblem`, *eigenSparse* defines a minimum interface that can be expected of all eigenvalue problems by the classes that will work with the problems (e.g., eigensolvers and status testers).

```
/*!  
 * \class xlifepp::EigenProblem  
 * \brief This provides a basic implementation for defining standard or  
 * generalized eigenvalue problems.  
 */  
template<class ScalarType, class MV, class OP>
```



```

class EigenProblem
{
public:

    ///! Empty constructor - allows xlifepp::EigenProblem to be described at a later
    time through "Set Methods".
    EigenProblem();

    ///! Standard Eigenvalue Problem Constructor.
    EigenProblem(const SmartPtr<const OP>& Op, const SmartPtr<MV>& initVec);

    ///! Generalized Eigenvalue Problem Constructor.
    EigenProblem(const SmartPtr<const OP>& Op, const SmartPtr<const OP>& B, const
        SmartPtr<MV>& initVec);

    ///! Copy Constructor.
    EigenProblem(const Eigenproblem<ScalarType, MV, OP>& Problem);

    ///! Destructor.
    ~EigenProblem() {};
    ...
}

```

Both the eigen problem and the eigensolver in *eigenSparse* are templated on the scalar type, the multivector type and the operator type. Before declaring an eigen problem, users must choose classes to represent these entities. Having done so, they can begin to specify the parameters of the eigen problem. The `EigenProblem` class defines set methods for the parameters of the eigen problem. These methods are:

- `setOperator` - set the operator for which the eigenvalues will be computed
- `setA` - set the A operator for the eigenvalue problem $Ax = Mx\lambda$
- `setM` - set the M operator for the eigenvalue problem $Ax = Mx\lambda$
- `setPrec` - set the preconditioner for the eigenvalue problem
- `setInitVec` - set the initial iterate
- `setAuxVecs` - set the auxiliary vectors, a subspace used to constrain the search space for the solution
- `setNEV` - set the number of eigenvalues to be computed
- `setHermitian` - specify whether the problem is Hermitian

In addition to these *set* methods, `EigenProblem` defines a method `setProblem()` that gives the class the opportunity to perform any initialization that may be necessary before the problem is handed off to an eigensolver, in addition to verifying that the problem has been adequately defined.

For each of the *set* methods listed above, there is a corresponding *get* function. These are the functions used by eigensolvers and solver managers to get the necessary information from the eigenvalue problem. In addition, there are two methods for storing and retrieving the results of the eigenvalue computation:

```

const EigenSolverSolution & getSolution();
void setSolution(const EigenSolverSolution & sol);

```

In order to verify if all the parameters are set, `EigenProblem` provide a method `isProblemSet()`

The EigenSolverSolution structure

The `EigenSolution` structure was developed in order to facilitate setting and retrieving of solution data.

```
/// Struct for storing an eigenproblem solution.
template <class ScalarType, class MV>
struct EigenSolverSolution {
    /// The computed eigenvectors
    SmartPtr<MV> evecs;
    /// An orthonormal basis for the computed eigenspace
    SmartPtr<MV> espace;
    /// The computed eigenvalues
    std::vector<ValueEigenSolver<ScalarType>> evals;
    /// \brief An index into evecs to allow compressed storage of eigenvectors for
    real, non-Hermitian problems.
    *
    *  index has length numVecs, where each entry is 0, +1, or -1. These have the
    following interpretation:
    *  - index[i] == 0: signifies that the corresponding eigenvector is stored
    as the i column of evecs. This will usually be the
    *  case when ScalarType is complex, an eigenproblem is Hermitian, or a
    real, non-Hermitian eigen problem has a real eigenvector.
    *  - index[i] == +1: signifies that the corresponding eigenvector is stored
    in two vectors: the real part in the i column of evecs and the
    *  <i>positive</i> imaginary part in the i+1 column of evecs.
    *  - index[i] == -1: signifies that the corresponding eigenvector is stored
    in two vectors: the real part in the i-1 column of evecs and the
    *  <i>negative</i> imaginary part in the i column of evecs
    */
    std::vector<int> index;
    /// The number of computed eigenpairs
    int numVecs;

    EigenSolverSolution() : evecs(), espace(), evals(), index(), numVecs(0) {}
};
```

All *eigenSparse* solver managers place the results of the computation in the `EigenProblem` class using an `EigenSolverSolution` structure. The number of eigen solutions computed is given by field `numVecs`. The eigenvalues are always stored as two real values, even when templated on a complex data type or when the eigenvalues are real. Similarly, the eigen space can always be represented by a multivector of width `numVecs`, even for non-symmetric eigen problems over the real field. The storage scheme for eigenvectors requires more finesse.

When solving real symmetric eigen problems, the eigenvectors can always be chosen to be real, and therefore can be stored in a single column of a real multivector. When solving eigenproblems over a complex field, whether Hermitian or non-Hermitian, the eigenvectors may be complex, but the multivector is defined over the complex field, so that this poses no problem. However, real non-symmetric problems can have complex eigenvectors, which prohibits a one-for-one storage scheme using a real multivector. Fortunately, the eigenvectors in this scenario occur as complex conjugate pairs, so the pair can be stored in two real vectors. This permits a compressed storage scheme, which uses an index vector stored in the `EigenSolverSolution`, allowing conjugate pair eigenvectors to be easily retrieved from `evecs`. The integers in `EigenSolverSolution::index` take one of three values: $\{0, +1, -1\}$. These values allow the eigenvectors to be retrieved as follows:

- $index[i] = 0$: the i -th eigenvector is stored uncompressed in column i of `evecs`
- $index[i] = +1$: the i -th eigenvector is stored compressed, with the real component in column i of `evecs` and the positive complex component stored in column $i + 1$ of `evecs`
- $index[i] = -1$: the i -th eigenvector is stored compressed, with the real component in column $i - 1$ of `evecs` and the negative complex component stored in column i of `evecs`

Because this storage scheme is only required for non-symmetric problems over the real field, all other eigenproblems will result in an index vector composed entirely of zeroes. For the real non-symmetric case, the +1 index will always immediately precede the corresponding -1 index.

The EigenSolver class

The `EigenSolver` class defines the basic interface that must be met by any eigen solver class in *eigenSparse*. The specific eigensolvers are implemented as derived classes of `EigenSolver`. There are two eigen solvers currently implemented in *eigenSparse*

- `BlockDavidson` : A block Davidson solver for Hermitian eigenvalue problems
- `BlockKrylovSchur` : A block Krylov Schur solver for Hermitian or non-Hermitian eigenvalue problems.

The class `EigenSolver`, like `EigenProblem`, is templated on the scalar type, multivector type and operator type. The options for the eigen solver are passed through the constructor, defined by `EigenSolver` to have the following form:

```
template<class ScalarType , class MV, class OP>
class EigenSolver {
public:

    ///! name Constructors/Destructor//{

    ///! Default Constructor.
    EigenSolver() {} ;

    ///! Basic Constructor.
    /*! This constructor, implemented by all xlifepp eigensolvers, takes an
        xlifepp::Eigenproblem,
        xlifepp::SortManager, xlifepp::OutputManager, and Parameters as input. These
        four arguments are sufficient enough for constructing any xlifepp::EigenSolver
        object.
    */
    EigenSolver( const SmartPtr<EigenProblem<ScalarType,MV,OP> >& problem ,
                const SmartPtr<SortManager<ScalarType> >& sorter ,
                const SmartPtr<OutputManager<ScalarType> >& printer ,
                const SmartPtr<StatusTest<ScalarType,MV,OP> >& tester ,
                const SmartPtr<OrthoManager<ScalarType,MV> >& ortho ,
                Parameters& params );

    ...
}
```

These classes are used as follows:

- **problem** - the eigenproblem to be solved; the solver will get the problem operators from here.
- **sorter** - the sort manager selects the significant eigenvalues
- **printer** - the output manager dictates verbosity level in addition to processing output streams
- **tester** - the status tester dictates when the solver should quit `iterate()` and return to the caller
- **ortho** - the orthogonalization manager defines the inner product and other concepts related to orthogonality, in addition to performing these computations for the solver
- **params** - the parameter list specifies eigensolver-specific options; see the documentation for a list of options support by individual solvers

In addition to specifying an iteration, an eigensolver also specifies a concept of state, i.e. the current data associated with the iteration. After declaring an Eigensolver object, the state of the solver is in an uninitialized state. For most solver, to be initialized mean to be in a valid state, containing all of the information necessary for performing eigen solver iterations.

Eigensolver provides two methods concerning initialization: *isInitialized()* indicates whether the solver is initialized or not, and *initialize()* (with no arguments) instructs the solver to initialize itself using random data or the initial vectors stored in the eigenvalue problem.

To ensure that solvers can be used as efficiently as possible, the user needs access to the state of the solver. To this end, each eigensolver provides low-level methods for getting and setting the state of the solver:

- **getState()** - returns a solver-specific structure with read-only pointers to the current state of the solver.
- **initialize(...)** - accepts a solver-specific structure enabling the user to initialize the solver with a particular state.

The combination of these two methods, along with the flexibility provided by status tests, allows the user almost total control over eigen solver iterations.

Moreover, each eigen solver provides two significant types of methods: status methods and solver- specific state methods. The status methods are defined by the Eigensolver abstract base class and represent the information that a generic status test can request from any eigensolver. A list of these methods is given in following table

<i>Method name</i>	<i>Description</i>
getNumIters	Get the current number of iterations.
getRitzValues	Get the most recent Ritz values.
getRitzVectors	Get the most recent Ritz vectors.
getRitzIndex	Get the Ritz index needed for indexing compressed Ritz vectors.
getResNorms	Get the most recent residual norms, with respect to the OrthoManager.
getRes2Norms	Get the most recent residual 2-norms.
getRitzRes2Norms	Get the most recent Ritz residual 2-norms.
getCurSubspaceDim	Get the current subspace dimension.
getMaxSubspaceDim	Get the maximum subspace dimension.
getBlockSize	Get the block size.

The SolverManager class

Using *eigenSparse* by interfacing directly with eigensolvers is extremely powerful, but can be tedious. Solver managers provide a way for users to encapsulate specific solving strategies inside of an easy-to-use class. Novice users may prefer to use existing solver managers, while advanced user may prefer to write custom solver managers. **SolverManager** defines only two methods: a constructor accepting an **EigenProblem** and a parameter list of solver manager-specific options; and a **solve()** method, taking no arguments and returning either **_converged** or **_unconverged**. Consider the following example code:

```
// Create an eigen problem
SmartPtr<EigenProblem<ST,MV,OP> > problem =
// Create parameter list to pass into the solver manager
Parameters pl;
pl.set("Verbosity", (int)verbosity);
pl.set("Which", which);
pl.set("Orthogonalization", ortho);
pl.set("Block Size", blockSize);
pl.set("Num Blocks", numBlocks);
pl.set("Maximum Restarts", maxRestarts);
```

```
// Create a solver manager
BlockDavidsonSolMgr<ST,MV,OP> bdSolverMan(problem, pl);
// Solve the eigenvalue problem
ComputationInfo returnCode = bdSolverMan.solve();
// Get the solution from the problem
EigenSolverSolution<ST,MV> sol = problem->getSolution();
```

As has been stated before, the goal of the solver manager is to create an eigensolver object, along with the support objects needed by the eigensolver. Another purpose of many solver managers is to manage and initiate the repeated calls to the underlying solver's `iterate()` method. For solvers that build a Krylov subspace to some maximum dimension (e.g., `BlockKrylovSchur` and `BlockDavidson`), the solver manager will also assume the task of restarting the solver when the subspace is full. This is something for which multiple approaches are possible. Also, there may be substantial flexibility in creating the support classes (e.g., sort manager, status tests) for the solver. An aggressive solver manager could even go so far as to construct a preconditioner for the eigenvalue problem.

These examples are meant to illustrate the flexibility that specific solver managers may have in implementing the `solve()` routine. Some of these options might best be incorporated into a single solver manager, which takes orders from the user via the parameter list given in the constructor. Some of these options may better be contained in multiple solver managers, for the sake of code simplicity. It is even possible to write solver managers that contain other solvers managers; motivation for something like this would be to select the optimal solver manager at runtime based on some expert knowledge, or to create a hybrid method which uses the output from one solver manager to initialize another one.

The StatusTest class

the purpose of the `StatusTest` should be clear: to give the user or solver manager flexibility in stopping the eigen solver iterations in order to interact directly with the solver.

Many reasons exist for why a user would want to stop the solver from iterating:

- some convergence criterion has been satisfied and it is time to quit;
- some part of the current solution has reached a sufficient accuracy to removed from the iteration ("locking");
- the solver has performed a sufficient or excessive number of iterations.

These are just some commonly seen reasons for ceasing the iteration, and each of these can be so varied in implementation/parametrization as to require some abstract mechanism controlling the iteration.

Below is a list of status tests:

- `StatusTestCombo` - this status test allows for the boolean combination of other status tests, creating near unlimited potential for complex status tests.
- `StatusTestOutput` - this status test acts as a wrapper around another status test, allowing for printing of status information on a call to `checkStatus()`
- `StatusTestMaxIters` - this status test monitors the number of iterations performed by the solver; it can be used to halt the solver at some maximum number of iterations or even to require some minimum number of iterations.
- `StatusTestResNorm` - this status test monitors the residual norms of the current iterate.
- `StatusTestWithOrdering` - this status test also monitors the residual norms of the current iterate, but only considers the residuals along with a set of auxiliary eigenvalues.

The SortManager class

The purpose of a sort manager is to separate the eigensolver classes from the sorting functionality required by those classes. This satisfies the flexibility principle sought by *eigenSparse*, by giving users the opportunity to perform the sorting in whatever manner is deemed to be most appropriate. *eigenSparse* defines an abstract class **SortManager** with two methods, one for sorting real values and one for sorting complex values:

```
// Sorting with real value
virtual void sort(std::vector<MagnitudeType>& evals, SmartPtr<std::vector<int>>
    perm = _smPtrNull, int n = -1) const = 0;
// Sorting with complex value
virtual void sort(std::vector<MagnitudeType>& rEvals,
    std::vector<MagnitudeType>& iEvals,
    SmartPtr<std::vector<int>> perm = _smPtrNull,
    int n = -1) const = 0;
```

The BasicSort class

The concrete derived class implements sorting scheme for sorting real values and one for sorting complex values.

```
template<class MagnitudeType>
class BasicSort : public SortManager<MagnitudeType>
{
public:
    /*! \brief Parameter list driven constructor

    This constructor accepts a paramter list with the following options:
    - \c "Sort Strategy" - a \c string specifying the desired sorting strategy.
    See setSortType() for valid options.
    */
    BasicSort(Parameters& pl);

    /*! \brief String driven constructor

    Directly pass the string specifying sort strategy. See setSortType() for
    valid options.
    */
    BasicSort(const String& which = "LM");

    //! Destructor
    virtual ~BasicSort();
    ...
}
```

By default, sort strategy Largest Magnitude (LM) is used. So as to change the strategy, user can easily specify it with **setSortType** method.

The OrthoManager class

Orthogonalization and orthonormalization are commonly performed computations in iterative eigensolvers; in fact, for some eigensolvers, they represent the dominant cost. Different scenarios may require different approaches (e.g, Euclidean inner product versus M inner product, orthogonal projections versus oblique projections). Combined with the plethora of available methods for performing these computations, Anasazi has left as much leeway to the users as possible.

Orthogonalization of multivectors is performed by derived classes of the abstract class **OrthoManager** that provides five methods:

- **innerProd(X,Y,Z)** - performs the inner product defined by the manager.

- `norm(X)` - computes the norm induced by `innerProd()`.
- `project(X,C,Q)` - given an orthonormal basis `Q`, projects `X` onto to the space perpendicular to `colspan(Q)`, optionally returning the coefficients of `X` in `Q`.
- `normalize(X,B)` - returns an orthonormal basis for `colspan(X)`, optionally returning the coefficients of `X` in the computed basis.
- `projectAndNormalize(X,C,B,Q)` - computes an orthonormal basis for subspace `colspan(X)` - `colspan(Q)`, optionally returning the coefficients of `X` in `Q` and the new basis.

It should be noted that a call to `projectAndNormalize()` is not necessarily equivalent to a call to `project()` followed by `normalize()`. This follows from the fact that, for some orthogonalization managers, a call to `normalize()` may augment the column span of a rank- deficient multivector in order to create an orthonormal basis with the same number of columns as the input multivector. In this case, the code

```
orthoman.project(X,C,Q);
orthoman.normalize(X,B);
```

could result in an orthonormal basis `X` that is not orthogonal to the basis in `Q`. XLIFF++ provides two orthogonalization managers:

- `BasicOrthoManager` - performs orthogonalization using multiple steps of classical Gram-Schmidt.
- `SVQBOrthoManager` - performs orthogonalization using the SVQB orthogonalization technique described by Stathapoulos and Wu

The core function of these orthogonalization managers is `findBasis`, which find an operator-orthonormal basis for a span, with the option of extending the subspace.

The OutputManager class

The output manager exists to provide flexibility with regard to the verbosity of the eigen solver. Each output manager has two primary concerns: what output is printed and where the output is printed to. When working with the output manager, output is classified into one of the following message types:

```
enum MsgEigenType
{
    _errorsEigen = 0,           /*!< Errors [ always printed ] */
    _warningsEigen = 0x1,      /*!< Internal warnings */
    _iterationDetailsEigen = 0x2, /*!< Approximate eigenvalues, errors */
    _orthoDetailsEigen = 0x4,   /*!< Orthogonalization/orthonormalization
                                details */
    _finalSummaryEigen = 0x8,   /*!< Final computational summary */
    _timingDetailsEigen = 0x10,  /*!< Timing details */
    _statusTestDetailsEigen = 0x20, /*!< Status test details */
    _debugEigen = 0x40         /*!< Debugging information */
};
```

Output manager are subclasses of the abstract base class `OutputManager`. This class provides the following output-related methods:

```
template <class ScalarType>
class OutputManager {
public:
    ...
    /*! Find out whether we need to print out information for this message type.
    /*! This method is used by the solver to determine whether computations are
        necessary for this message type.
```

```

*/
virtual bool isVerbosity(MsgEigenType type) const = 0;

///! Send output to the output manager.
virtual void print(MsgEigenType type, const String output) = 0;

///! Create a stream for outputting to.
virtual std::ostream& stream(MsgEigenType type) = 0;
...

```

XLiFE++ provides a single output manager, `BasicOutputManager`. This class accepts an output stream from the user. The output corresponding to the verbosity level of the manager is sent to this stream.

The SolverUtils class

Most of the large-scale eigenvalue algorithms exploit projection processes so as to extract approximate eigenvectors from a given subspace. Inside this subspace, a small matrix eigenvalue problem is obtained. XLiFE++ provides `SolverUtils` class, a collection of utilities necessary for the solvers. These utilities include sorting, orthogonalization, projecting/solving local eigen systems,

```

template<class ScalarType, class MV, class OP>
class SolverUtils
{
public:
    typedef typename NumTraits<ScalarType>::RealScalar MagnitudeType;
    typedef typename NumTraits<ScalarType>::Scalar SCT;
    typedef MatrixEigenDense<SCT> MatrixDense;

    ///! name Constructor/Destructor//{

    ///! Constructor.
    SolverUtils();

    ///! Destructor.
    virtual ~SolverUtils() {};
    ...

```

In fact, all utilities along with this class make use of functions of `eigenCore`. The most important function of this class is `directSolver`, a routine for computing eigen pairs of Hermitian pencil. This method is heavily used during the projection of vectors into a subspace and normalization of these vector with a specified norm.

15.2.4 Examples

The following gives sample code for solving a symmetric eigenvalue problem using the Block Davidson solver manager.

The first step in solving an eigenvalue problem is to define the eigenvalue problem. However, before that, we need to make sure define the operator and the multivector.

```

// Load LargeMatrix from file
LargeMatrix<Real> rMatSym(dataPathTo(rMatrixSym), _dense, 782, _cs, _symmetric);
// Define the operator (in this case, it's LargeMatrix)
SmartPtr< const LargeMatrixAdapter<ST> > K(new LargeMatrixAdapter<ST>(&rMatSym));
// Sometimes, we need a preconditioner to be able to solve the eigenvalue problem
SmartPtr< const LargeMatrixAdapter<ST> > Prec(new LargeMatrixAdapter<ST>(&matPrec));
// Define multivector
SmartPtr<MV> ivec = SmartPtr<MultiVecAdapter<ST> >(new
    MultiVecAdapter<ST>(rMatSym.nbCols, blockSize));
// Initialize the multivector with random values

```



```
ivec->mvRandom();
```

Everything is fine then eigenvalue problem is defined as follow:

```
SmartPtr<EigenProblem<ST,MV,OP> > problem = SmartPtr<EigenProblem<ST,MV,OP> >(new
    EigenProblem<ST,MV,OP>(K,ivec));
// Inform the eigenProblem that the operator cK is symmetric
problem->setHermitian(true);
// Set the number of eigenvalues requested
problem->setNEV(nev);
```

The second step is to specify parameters for the solver manager, depending on algorithm (Block Davidson or Block Krylov Schur), we can have a different parameters. Nevertheless, there are some essential ones to be specified

```
const int nev = 4; //6
int blockSize = 5; //6;
int numBlocks = 5; //14;
int maxRestarts = 25; //8;
Real tol = 1.0e-6;
String which("SM"), ortho("SVQB");
bool locking = true;
bool insitu = false;
MsgEigenType verbosity = _errorsEigen;

Parameters pl;
pl.set("Verbosity", (int)verbosity);
pl.set("Which", which);
pl.set("Orthogonalization", ortho);
pl.set("Block Size", blockSize);
pl.set("Num Blocks", numBlocks);
pl.set("Maximum Restarts", maxRestarts);
pl.set("Convergence Tolerance", tol);
pl.set("Use Locking", locking);
pl.set("Locking Tolerance", tol/10);
pl.set("In Situ Restarting", insitu);
```

Till now, we have all the information needed to declare the solver manager and solve the problem:

```
BlockDavidsonSolMgr<ST,MV,OP> bdSolverMan(problem, pl);
ComputationInfo returnCode = bdSolverMan.solve();
```

The return value of the solver indicate whether the algorithm succeeds or not.(i.e: whether the requested number of eigenpairs were found to a sufficient accuracy (as defined by the solver manager). Output from `solve()` routine in this example might look as follows:

BlockDavidson Eigen Solver Test : Standard symmetric matrix test with Smallest Magnitude

Direct residual norms computed	
Eigenvalue	Residual (M)
-1.29891e-06	8.63079e-08
-1.29973e-06	6.27415e-08
-1.30048e-06	7.99246e-08
-1.33992e-06	7.09023e-07

The *arpackppSupport* library contains the interface of XLIFFE++ with ARPACK library through ARPACK++. Because of commonality and stability, ARPACK library is chosen as the external eigen solver.

ARPACK is a well known collection of FORTRAN subroutines designed to compute a few eigenvalues and eigenvectors of large scale sparse matrices and pencils. It implements a variant of the Arnoldi process for finding eigenvalues called Implicit restarted Arnoldi method (IRAM) and is capable of solving a great variety of problems from single precision positive definite symmetric problems to double precision complex non-Hermitian generalized problems. ARPACK++ is the C++ an interface of ARPACK.

XLIFFE++ offers an interface to ARPACK++ through a well-organized collection of classes which hide all the complexity of ARPACK++. User can easily call an eigen solver like a call to an iterative solver. Moreover, this interface offers advanced users a flexible way to make use of all features of ARPACK++ with a minimum effort by modifying some classes. This package is organized into 2 layers:

- **Users Interface** wraps all ARPACK++ interface and provides simple call prototype to user
- **Arpack++ Interface** is collection of classes interacting directly with ARPACK++

In order to use ARPACK++, ARPACK must be already installed. Arpack++ package can be found at <http://www.ime.unicamp.br/~chico/arpack++/>. However, because of the deprecation of this version, a patch at <http://reuter.mit.edu/index.php/software/arpackpatch/> needs applied to make sure a correct compilation.

The following is organized in "bottom-up", the ARPACK++ interface is described then Users Interface.

16.1 ARPACK++ interface layer

ARPACK++ interface is a group of class which is an interface between XLIFFE++ and ARPACK++. Each class of this group corresponds to a specific eigen problem with a computational mode.

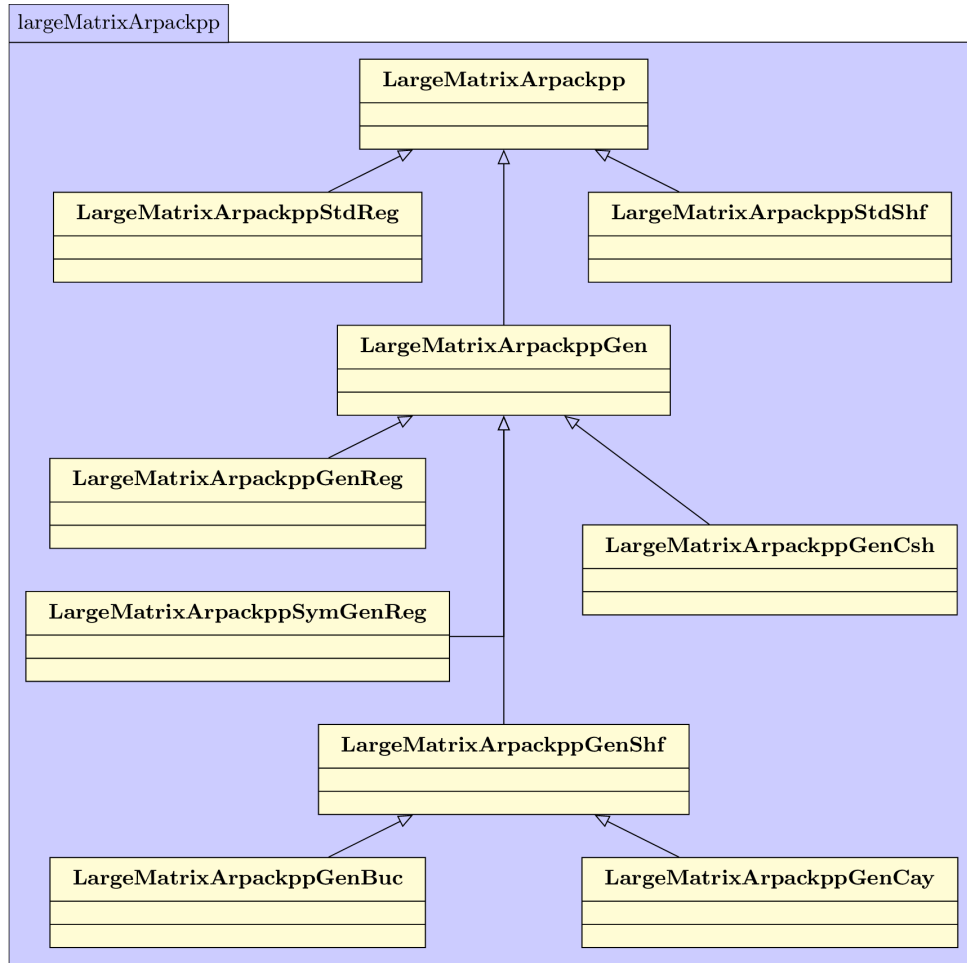


Figure 16.1: Hierarchy of LargeMatrixArpackpp

`LargeMatrixArpackpp` is an interface class between XLIFE++ and ARPACK++. This class is an abstract base class so that all classes to be used in the program are derived from this one. This class defines some basic properties and methods which are used in all derived classes.

```

template<typename K, class Mat>
class LargeMatrixArpackpp
{
public:
    LargeMatrixArpackpp(const Mat& matA, const Number sizeProblem);
    LargeMatrixArpackpp(const Mat& matA, const Number sizeProblem, const String
        name);
    virtual ~LargeMatrixArpackpp();
    virtual void multOPx (K* x, K* y) = 0;
    String kindOfFactorization() const {return factName_;}

protected:
    const Mat* matA_p;    //< Left hand-side matrix A, generally stiffness matrix
        (given data)
    Number size_;        //< Size of problem
    String name_;        //< Name of problem
    String factName_;    //< Name of the algorithm used to compute the
        factorization pointed to by fact_p
    Mat* fact_p;        //< Factorization of matB, matAsI or matAsB in order to speed up
        linear systems solving.

protected:
    void (Mat::* solver_)(std::vector<K>& b, std::vector<K>& x);

```

```

void checkDims(const Mat* mat);
void factorize(const Mat* mat);
void bzMultAx(K* x, K* y);          //< Matrix-vector product  $y \leftarrow A * x$ 
void array2Vector(const K*, std::vector<K>&);    //< Convert array to Vector
void vector2Array(std::vector<K>&, K*);          //< Convert Vector to Array
}; // end of Class LargeMatrixArpackpp

```

```

library :   eigenSolvers
header :
implementation :
unitary tests : test_EigenSolverArpackpp.hpp
header dependences :

```

16.1.1 The LargeMatrixArpackppStdReg class

`LargeMatrixArpackppStdReg` class is an interface between XLIFF++ and ARPACK++ designed to enable the use of Arpack++'s own classes for standard problems in regular mode. This class allows XLIFF++ users to solve real symmetric, non-symmetric and complex standard eigen problem in regular mode. This class holds informations of matrices and some methods related to product matrix-vector. In general, only matrix-vector product $y \leftarrow Ax$ is needed.

```

template<typename K, class Mat>
class LargeMatrixArpackppStdReg: public LargeMatrixArpackpp<K, Mat>
{
public:
    LargeMatrixArpackppStdReg(const Mat& matA, const Number sizeProblem);
    virtual ~LargeMatrixArpackppStdReg() {}
    virtual void multOPx(K* x, K* y);    //< Matrix-vector product  $y \leftarrow A * x$ 

private:
    //! Duplicating such object is disabled
    LargeMatrixArpackppStdReg(const LargeMatrixArpackppStdReg&);
    LargeMatrixArpackppStdReg& operator=(const LargeMatrixArpackppStdReg&);
}; // end of Class LargeMatrixArpackppStdReg

```

```

library :   eigenSolvers
header :
implementation :
unitary tests : test_EigenSolverArpackpp.hpp
header dependences : LargeMatrixArpackpp.hpp

```

16.1.2 The LargeMatrixArpackppStdShf class

`LargeMatrixArpackppStdShf` class is an interface between XLIFF++ and ARPACK++ designed to enable the use of ARPACK++'s own classes for standard problems in shift and inverse mode. This class allows XLIFF++ users to solve real symmetric, non-symmetric and complex standard eigen problem in shift and inverse mode. This class holds informations of matrices and some methods related to product matrix-vector. In general, we need to provide matrix-vector product $y \leftarrow (A - \sigma I)^{-1}x$. However, thanks to some built-in special solvers of XLIFF++, only $(A - \sigma I)$ is enough.

```

template<typename K, class Mat>
class LargeMatrixArpackppStdShf: public LargeMatrixArpackpp<K, Mat>
{

```

```

public:
    LargeMatrixArpackppStdShf(const Mat& matA, const Number sizeProblem, const K
        sigma);
    virtual ~LargeMatrixArpackppStdShf();
    virtual void multOPx (K* x, K* y);    //< Matrix-vector product  $y \leftarrow$ 
         $inv(A - sigma * Id) * x$ 
private:
    const Mat* matAsIp; //<  $A - sigma * Id$  matrix for shift and invert mode
        in standard problems
    bool localAlloc_;    //< flag to indicate if dynamic memory is allocated to
        store the matrix  $(A - sigma * Id)$ 

    //! Duplicating such object is disabled
    LargeMatrixArpackppStdShf(const LargeMatrixArpackppStdShf&);
    LargeMatrixArpackppStdShf& operator=(const LargeMatrixArpackppStdShf&);
}; // end of Class LargeMatrixArpackppStdShf

```

```

library :   eigenSolvers
header :
implementation :
    unitary tests :   test_EigenSolverArpackpp.hpp
header dependences :   LargeMatrixArpackpp.hpp

```

16.1.3 The LargeMatrixArpackppGen class

`LargeMatrixArpackppGen` is an abstract class which plays a role of an interface between XLIFE++ and ARPACK++ designed to enable the use of ARPACK++'s own classes for generalized eigen problem $Ax = \lambda Bx$. All classes for generalized problem are derived from this one.

```

template<typename K, class Mat>
class LargeMatrixArpackppGen: public LargeMatrixArpackpp<K, Mat>
{
public:
    LargeMatrixArpackppGen(const Mat& matA, const Mat& matB, const Number
        sizeProblem);
    LargeMatrixArpackppGen(const Mat& matA, const Mat& matB, const Number
        sizeProblem, const String name);
    virtual ~LargeMatrixArpackppGen() {}
    virtual void multBx (K* x, K* y); //<  $Matrix\text{-}vector\ product\ y \leftarrow B * x$ 
        required for all derived classes

protected:
    const Mat* matB_p; //< Right hand-side matrix  $B$ , e.g. mass matrix (given data)
    Mat* matAsB_p; //!<  $Matrix\ (A - sigma * B)$ 
    bool localAlloc_; //< flag to indicate if dynamic memory is allocated to
        store  $A - sigma * B$ ,

private:
    //! Duplicating such object is disabled
    LargeMatrixArpackppGen(const LargeMatrixArpackppGen&);
    LargeMatrixArpackppGen& operator=(const LargeMatrixArpackppGen&);
}; // end of Class LargeMatrixArpackppGen

```

library :	eigenSolvers
header :	
implementation :	
unitary tests :	test_EigenSolverArpackpp.hpp
header dependences :	LargeMatrixArpackpp.hpp

16.1.4 The LargeMatrixArpackppGenReg class

`LargeMatrixArpackppGenReg` class is interface classes between XLIFE++ and ARPACK++ designed to enable the use of ARPACK++'s own classes for generalized problems in regular mode except for real symmetric generalize problem. This class holds informations of matrices and some methods related to product matrix-vector. In general, we need to provide matrix-vector product $w \leftarrow (B)^{-1}Ax$ and $z \leftarrow Bx$

```
template<typename K, class Mat>
class LargeMatrixArpackppGenReg: public LargeMatrixArpackppGen<K, Mat>
{
public:
    LargeMatrixArpackppGenReg(const Mat& matA, const Mat& matB, const Number
        sizeProblem);
    virtual ~LargeMatrixArpackppGenReg() {};
    virtual void multOPx (K* x, K* y); //< Matrix-vector product y <- inv(B)*A * x

private:
    //! Duplicating such object is disabled
    LargeMatrixArpackppGenReg(const LargeMatrixArpackppGenReg&);
    LargeMatrixArpackppGenReg& operator=(const LargeMatrixArpackppGenReg&);
}; // end of Class LargeMatrixArpackppGenReg
```

library :	eigenSolvers
header :	
implementation :	
unitary tests :	test_EigenSolverArpackpp.hpp
header dependences :	LargeMatrixArpackppGen.hpp

16.1.5 The LargeMatrixArpackppSymGenReg class

`LargeMatrixArpackppSymGenReg` is basically not different from class `LargeMatrixArpackppGenReg` except on how the calculation results are returned. Maybe in the future, we don't need this class anymore.

```
template<typename K, class Mat>
class LargeMatrixArpackppSymGenReg: public LargeMatrixArpackppGen<K, Mat>
{
public:
    LargeMatrixArpackppSymGenReg(const Mat& matA, const Mat& matB, const Number
        sizeProblem);
    virtual ~LargeMatrixArpackppSymGenReg() {}
    virtual void multOPx (K* x, K* y); //< Matrix-vector product y <- inv(B)*A * x

private:
    //! Duplicating such object is disabled
    LargeMatrixArpackppSymGenReg(const LargeMatrixArpackppSymGenReg&);
    LargeMatrixArpackppSymGenReg& operator=(const LargeMatrixArpackppSymGenReg&);
}; // end of Class LargeMatrixArpackppSymGenReg
```

library :	eigenSolvers
header :	
implementation :	
unitary tests :	test_EigenSolverArpackpp.hpp
header dependencies :	LargeMatrixArpackppGen.hpp

16.1.6 The LargeMatrixArpackppGenShf class

Class `LargeMatrixArpackppGenShf` is interface between XLIFE++ and ARPACK++ designed to enable the use of ARPACK++'s own classes for generalized problems in shift and invert mode. All classes for generalized problem in relevant shift and invert mode (including Buckling and Cayley) are derived from this one.

```
template<typename K, class Mat>
class LargeMatrixArpackppGenShf: public LargeMatrixArpackppGen<K, Mat>
{
public:
    LargeMatrixArpackppGenShf(const Mat& matA, Mat& matB, const Number sizeProblem,
        const K sigma, const String name = "Generalized Shift and Invert");
    virtual ~LargeMatrixArpackppGenShf() {}
    virtual void multOPx (K* x, K* y); //< Matrix-vector product y <- inv(A-sigma B)*x

private:
    //! Duplicating such object is disabled
    LargeMatrixArpackppGenShf(const LargeMatrixArpackppGenShf&);
    LargeMatrixArpackppGenShf& operator=(const LargeMatrixArpackppGenShf&);
}; // end of Class LargeMatrixArpackppGenShf
```

library :	eigenSolvers
header :	
implementation :	
unitary tests :	test_EigenSolverArpackpp.hpp
header dependencies :	LargeMatrixArpackppGen.hpp

16.1.7 The LargeMatrixArpackppGenBuc class

Class `LargeMatrixArpackppGenBuc` is derived from class `LargeMatrixArpackppGenShf` to solve generalized problems Buckling mode. Two products matrix-vector need provided: $w \leftarrow (A - \sigma B)^{-1}x$ and $z \leftarrow Ax$

```
template<typename K, class Mat>
class LargeMatrixArpackppGenBuc: public LargeMatrixArpackppGenShf<K, Mat>
{
public:
    LargeMatrixArpackppGenBuc(const Mat& matA, Mat& matB, const Number sizeProblem,
        const K sigma);
    virtual ~LargeMatrixArpackppGenBuc() {}
    virtual void multBx (K* x, K* y); //< Matrix-vector product y <- A * x

private:
    //! Duplicating such object is disabled
    LargeMatrixArpackppGenBuc(const LargeMatrixArpackppGenBuc&);
    LargeMatrixArpackppGenBuc& operator=(const LargeMatrixArpackppGenBuc&);
}; // end of Class LargeMatrixArpackppGenBuc
```

library :	eigenSolvers
header :	
implementation :	
unitary tests :	test_EigenSolverArpackpp.hpp
header dependencies :	LargeMatrixArpackppGen.hpp

16.1.8 The LargeMatrixArpackppGenCay class

Class `LargeMatrixArpackppGenCay` is derived from class `LargeMatrixArpackppGenShf` to solve generalized problems in Cayley mode. Three products matrix-vector need provided: $y \leftarrow (A - \sigma B)^{-1}z$, $w \leftarrow Az$ and $u \leftarrow Bz$

```
template<typename K, class Mat>
class LargeMatrixArpackppGenBuc: public LargeMatrixArpackppGenShf<K, Mat>
{
public:
    LargeMatrixArpackppGenBuc(const Mat& matA, Mat& matB, const Number sizeProblem,
        const K sigma);
    virtual ~LargeMatrixArpackppGenBuc() {}
    virtual void multBx (K* x, K* y); //< Matrix-vector product  $y \leftarrow A * x$ 

private:
    //! Duplicating such object is disabled
    LargeMatrixArpackppGenBuc(const LargeMatrixArpackppGenBuc&);
    LargeMatrixArpackppGenBuc& operator=(const LargeMatrixArpackppGenBuc&);
}; // end of Class LargeMatrixArpackppGenBuc
```

library :	eigenSolvers
header :	
implementation :	
unitary tests :	test_EigenSolverArpackpp.hpp
header dependencies :	LargeMatrixArpackppGen.hpp

16.1.9 The LargeMatrixArpackppGenCsh class

Class `LargeMatrixArpackppGenCsh` is derived from class `LargeMatrixArpackppGen` to solve real non-symmetric generalized problem in complex shift and invert mode. Three products matrix-vector need provided: $y \leftarrow (A - \sigma B)^{-1}z$, $w \leftarrow Az$ and $u \leftarrow Bz$

```
template<typename K, class Mat>
class LargeMatrixArpackppGenCsh : public LargeMatrixArpackppGen<K, Mat>
{
public:
    LargeMatrixArpackppGenCsh(const Mat& matA, Mat& matB, const Number sizeProblem,
        Complex sigma, const char part);
    virtual ~LargeMatrixArpackppGenCsh() {};
    //!
        Matrix-vector product  $y \leftarrow \text{real}(\text{inv}(A - \text{sigma}*B))$  if part = 'R'
                         $y \leftarrow \text{imag}(\text{inv}(A - \text{sigma}*B))$  if part = 'I'
    */
    void virtual multOPx (K* x, K* y);
    void multAx (K* x, K* y); //< Matrix-vector product  $y \leftarrow A * x$ 
private:
    //! Duplicating such object is disabled
    LargeMatrixArpackppGenCsh(const LargeMatrixArpackppGenCsh&);
```

```

    LargeMatrixArpackppGenCsh& operator=(const LargeMatrixArpackppGenCsh&);
}; // end of Class LargeMatrixArpackppGenCsh

```

```

library :   eigenSolvers
header :
implementation :
unitary tests :   test_EigenSolverArpackpp.hpp
header dependences :   LargeMatrixArpackppGen.hpp

```

16.2 Users interface layer

User interface contains classes which wrap all ARPACK++ interface layer and provide user a simple prototype. All eigen solvers are invoked by operator(), which is similar to other iterative solvers of *solvers* library. There are two principal classes, each of which solves a specific eigen problem

- **EigenSolverStdArpackpp** for standard eigen problem $Ax = \lambda x$
- **EigenSolverGenArpackpp** for generalized eigen problem $Ax = \lambda Bx$

For each type of eigen problem, input matrix A decides the value type (Real or Complex) of eigenvalue and/or eigenvector found. There are three cases corresponding to input matrix A: real symmetric, complex and real non-symmetric. For the first two cases, the eigenvalue and/or eigenvector found have the same value type of matrix A (i.e: Real if matrix A is real symmetric and Complex if matrix A is complex). For the real non-symmetric case, eigenvalue and/or eigenvector can be complex. Based on value type of input matrix and output eigenvalue (and/or eigenvector), an operator () can be invoked to solve an eigen problem correctly. There are three prototypes of operator() corresponding to three cases:

- **real symmetric case**: input matrix A is real symmetric and output eigenvalues/eigenvectors are real
- **real non-symmetric case**: input matrix A is real non-symmetric and output eigenvalues/eigenvectors are complex
- **complex case**: input matrix A is complex and output eigenvalues/eigenvectors are complex

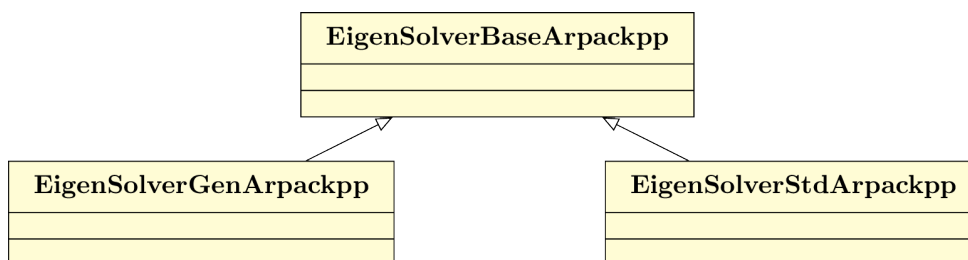


Figure 16.2: **EigenSolverBaseArpackpp** class

16.2.1 The EigenSolverBaseArpackpp class

The **EigenSolverBaseArpackpp** is the base class which provides some methods to construct object and retrieve eigenvalue/eigenvector returned by ARPACK++. All other wrapper classes are derived from this one.


```

class EigenSolverBaseArpackpp
{
public:
    EigenSolverBaseArpackpp(const String& nam);
    EigenSolverBaseArpackpp(const String& nam, const Number maxOfIt, const Real
        epsilon);
    virtual ~EigenSolverBaseArpackpp();
    // Iterative solver name for documentation purposes
    String name() { return name_; }
    // Some functions to extract Eigenvalue and EigenVector returned from Arpack
    // Extract EigenValue and EigenVector
    template<class ArProblem, class EigValue, class EigVector>
    Number extractEigenValueVector(ArProblem& arProb,
        EigValue& eigValue, EigVector& eigVector,
        Number nev, Number size, bool isReturnVector);

    // Extract EigenValue and EigenVector for non-symmetric problem
    template<class ArProblem, class EigValue, class EigVector>
    Number extractEigenValueVectorNonSym(ArProblem& arProb,
        EigValue& eigValue, EigVector& eigVector,
        Number nev, Number size, bool
            isReturnVector);

    // Set some parameters with user-defined value
    template<class ArProblem>
    void setParameters(ArProblem& arProb);
protected:
    // Extract result EigenVector
    template<typename K, class ArProblem>
    void extractEigenVector(ArProblem& arProb, std::vector<K>& result, Number size,
        Number idx);

    // Extract result EigenVector of non-symmetric problem
    template<class ArProblem>
    void extractEigenVectorNonSym(ArProblem& arProb, std::vector<Complex>& result,
        Number size, Number idx, bool isNegative, bool isNegativeLeft);
    ...
};

```

16.2.2 The EigenSolverStdArpackpp class

The `EigenSolverStdArpackpp` class enables the use of ARPACK++ to solve the standard eigen problem $Ax = \lambda x$ in all computational modes. Moreover, this class simplifies the post-processing stage, the found eigenvalues/eigenvectors are returned directly to user. However, advanced user can easily customize the some lines of code to get more features of ARPACK++ (e.g Schur vector, etc, ...). There are three prototypes of operator () corresponding to three cases: real symmetric, complex and real non-symmetric.

- Real symmetric:

```

template<class Mat>
Number operator()(Mat& matA, Vector<Real>& eigValue, std::vector<Vector<Real>>&
    eigVector, Number nev, EigenComputationalMode compMode, Real sigma, const char*
    calMode = "LM")

```

- Real non-symmetric:

```

template<class Mat>
Number operator()(Mat& matA, Vector<Complex>& eigValue, std::vector<Vector<Complex>
    >& eigVector, Number nev, EigenComputationalMode compMode, Complex sigma, const
    char* calMode = "LM")

```

- Complex:

```
template<class Mat>
Number operator()(Mat& matA, Vector<Complex>& eigValue, std::vector<Vector<Complex>
>& eigVector, Number nev, EigenComputationalMode compMode, Real sigma, const
char* calMode = "LM")
```



If `compMode` isn't `_regular`, only `largeMatrix` with storage of `skylineStorage` can be used as matrix input.

The `operator()` returns a value specifying the number of eigenvalues/eigenvectors being have found by ARPACK++. This value is equal or less than the input `nev`, which determines the number of eigenvalues/eigenvectors which user needs. `sigma` determines shift value in Shift mode (also in Buckling and Cayley mode). By default, `nev` eigenvalues with largest magnitude are turned; however, the function can return eigenvalues with smalles magnitude by changing `calMode` to "SM".

The `EigenComputationalMode` is enumeration:

```
enum EigenComputationalMode {_regular, _buckling, _cayley, _shiftInv,
_cplxShiftInv};
```

In order to solve eigen problem with ARPACK++, it is necessary to define a ARPACK++ problem then link this problem to XLIFE++ through ARPACK++ interface layer (i.e `LargeMatrixArpackpp`, etc, ...). There are three ARPACK++ classes which are used to define a standard eigen problem: `ARSymStdEig`, `ARNonSymStdEig`, `ARCompStdEig`.

The following functions define ARPACK++ problem and make link to XLIFE++ via ARPACK++ interface layer.

```
template<typename K, class MatArpackpp, class Arpackpp, class Mat, class EigValue,
class EigVector>
Number regMode(Mat& matA, Number size, Number nev, EigValue& eigValue, EigVector&
eigVector, const char* calMode);
Number shfInvMode(Mat& matA, Number size, Number nev, K sigma, EigValue& eigValue,
EigVector& eigVector, const char* calMode);
Number regModeNonSym(Mat& matA, Number size, Number nev, EigValue& eigValue,
EigVector& eigVector, const char* calMode);
Number shfInvModeNonSym(Mat& matA, Number size, Number nev, K sigma, EigValue&
eigValue, EigVector& eigVector, const char* calMode);
```



Users can easily customize the parameters of ARPACK++ problem corresponding to their need.

library :	eigenSolvers	
header :		
implementation :		
unitary tests :	test_EigenSolverArpackpp.hpp	
header dependences :	EigenSolverBaseArpackpp.hpp,	LargeMatrixArpackppStdReg.hpp,
	LargeMatrixArpackppStdShf.hpp	

16.2.3 The EigenSolverGenArpackpp class

The `EigenSolverStdArpackpp` class enables the use of ARPACK++ to solve the generalized eigen problem $Ax = \lambda Bx$ in all computational modes. Moreover, this class simplifies the post-processing stage, the found eigenvalues/eigenvectors are returned directly to user. However, advanced user can easily customize several lines of code to get more features of ARPACK++ (e.g Schur vector, etc, ...). There

are three prototypes of operator () corresponding to three cases: real symmetric, complex and real non-symmetric.

- Real symmetric:

```
template<class Mat>
Number operator()(const Mat& matA, Mat& matB, Vector<Real>& eigValue,
    std::vector<Vector<Real>>& eigVector, Number nev, EigenComputationalMode
    compMode, Real sigmaR, const char* calMode = "LM");
```

- Real non-symmetric:

```
template<class Mat>
Number operator()(const Mat& matA, Mat& matB, Vector<Complex>& eigValue,
    std::vector<Vector<Complex>>& eigVector, Number nev, EigenComputationalMode
    compMode, Complex sigma, const char* calMode = "LM");
```

- Complex:

```
template<class Mat>
Number operator()(const Mat& matA, Mat& matB, Vector<Complex>& eigValue,
    std::vector<Vector<Complex>>& eigVector, Number nev, EigenComputationalMode
    compMode, Real sigmaR, const char* calMode = "LM");
```

- Real non-symmetric with complex shift:

```
template<class Mat>
Number operator()(const Mat& matA, Mat& matB, Vector<Complex>& eigValue,
    std::vector<Vector<Complex>>& eigVector, Number nev, EigenComputationalMode
    compMode, const char mode, Complex sigma, const char* calMode = "LM");
```



If compMode isn't `_regular`, only largeMatrix with storage of skylineStorage can be used as matrix input.

Having matrix B as an additional input, the operator() of `EigenSolverGenArpackpp` class is called with the same parameters as the case of `EigenSolverStdArpackpp`. Moreover, it has one more prototype which is served for real non-symmetric case with complex shift value. Besides the same parameters like other prototypes, it is necessary for user to specify which part of complex shift value: real or imaginary, to use in the shift mode by parameter `mode`. This input can be 'R' for real part or 'I' for imaginary part of complex shift value `sigma`. Same as `EigenSolverStdArpackpp`, it needs to define ARPACK++ problems and link them to XLiFE++. `ARSymGenEig`, `ARNonSymGenEig` and `ARCompGenEig` class are used to define ARPACK++ generalized problem.

The following functions define *Arpack++* problem and make link to XLiFE++ via ARPACK++ interface layer.

```
template<typename K, class MatArpackpp, class Arpackpp, class Mat, class EigValue,
    class EigVector>
Number regMode(const Mat& matA, const Mat& matB, Number size, Number nev, EigValue&
    eigValue, EigVector& eigVector, const char* calMode);
Number regModeNonSym(const Mat& matA, const Mat& matB, Number size, Number nev,
    EigValue& eigValue, EigVector& eigVector, const char* calMode);
Number symShfBucMode(const Mat& matA, Mat& matB, Number size, Number nev, K sigma,
    const char mode, EigValue& eigValue, EigVector& eigVector, const char* calMode);
Number symCaleyMode(const Mat& matA, Mat& matB, Number size, Number nev, K sigma,
    EigValue& eigValue, EigVector& eigVector, const char* calMode);
Number shfMode(const Mat& matA, Mat& matB, Number size, Number nev, K sigma,
    EigValue& eigValue, EigVector& eigVector, const char* calMode);
```

```

Number shfModeNonSym(const Mat& matA, Mat& matB, Number size, Number nev, K sigma,
    EigValue& eigValue, EigVector& eigVector, const char* calMode);
Number complexShfInvMode(const Mat& matA, Mat& matB, Number size, Number nev, Real
    sigmaR, Real sigmaI, const char mode, EigValue& eigValue, EigVector& eigVector,
    const char* calMode);

```



Users can easily customize the parameters of ARPACK++ problem corresponding to their need.

library :	eigenSolvers	
header :		
implementation :		
unitary tests :	test_EigenSolverArpackpp.hpp	
header dependences :	EigenSolverBaseArpackpp.hpp,	LargeMatrixArpackppGenReg.hpp,
	LargeMatrixArpackppGenShf.hpp	

16.2.4 Examples

The following matrices and vectors are used for all examples below.

```

const Number rowNum = 20;
const Number colNum = 20;
const Number nev = 15;
const String rMatrixDataSym("rSym20.data");
const String rMatrixDataSymPosDef("rSymPos20.data");
const String rMatrixDataNoSym("rnonSym20.data");
const String cMatrixDataSym("c20.data");
const String cMatrixData("c20b.data");
Vector<Real> rEigValue(colNum);
Vector<Complex> cEigValue(colNum);
std::vector<Vector<Real>> rEigVector(nev, rEigValue);
std::vector<Vector<Complex>> cEigVector(nev, cEigValue);

LargeMatrix<Real> rMatSkylineSym(dataPathTo(rMatrixDataNoSym), _dense, rowNum,
    colNum, _skyline, _sym);
LargeMatrix<Real> rMatSkylineSymSym(dataPathTo(rMatrixDataSym), _dense, rowNum,
    _skyline, _symmetric);

// A trick to make matrix A and matrix B have the same storage (just for testing
// purpose)
LargeMatrix<Real> rMatPosDefTemp(dataPathTo(rMatrixDataSymPosDef), _dense, rowNum,
    _skyline, _symmetric);
LargeMatrix<Real> rMatPosDefSym(rMatSkylineSymSym);
for (Number i = 1; i < rowNum + 1; i++)
    for (Number j = 1; j < colNum + 1; j++)
        { rMatPosDefSym(i, j) = rMatPosDefTemp(i, j); }

// Positive definite in real non-symmetric case
LargeMatrix<Real> rMatPosDefNoSymTemp(dataPathTo(rMatrixDataSymPosDef), _dense,
    rowNum, colNum, _skyline, _sym);
LargeMatrix<Real> rMatPosDefNoSym(rMatSkylineSym);
for (Number i = 1; i < rowNum + 1; i++)
    for (Number j = 1; j < colNum + 1; j++)
        { rMatPosDefNoSym(i, j) = rMatPosDefNoSymTemp(i, j); }

// Complex Large Matrix
LargeMatrix<Complex> cMatSkylineSymSym(dataPathTo(cMatrixDataSym), _dense, rowNum,
    _skyline, _symmetric);

Real sigma = 10.0;

```

```
Complex csigma(-700, 20.0);
EigenSolverStdArpackpp eigenSolverStd;
EigenSolverGenArpackpp eigenSolverGen;
```

Standard eigenvalue problem

Real symmetric case

```
// Regular mode
if ( 0 != eigenSolverStd(rMatSkylineSymSym, rEigValue, rEigVector, nev, _regular,
    sigma) ) {
    out << "Eigenvalue regular mode (Skyline Sym) : " << rEigValue << std::endl;
    out << "Eigenvector regular mode (Skyline Sym) : " << rEigVector << std::endl;
}
// Shift and invert mode
if ( 0 != eigenSolverStd(rMatSkylineSymSym, rEigValue, rEigVector, nev, _shiftInv,
    sigma) ) {
    out << "Eigenvalue shift and inverse mode : " << rEigValue << std::endl;
    out << "Eigenvalue shift and inverse mode : " << rEigVector << std::endl;
}
```

Real non -symmetric case

```
// Regular mode
if ( 0 != eigenSolverStd(rMatSkylineSym, cEigValue, cEigVector, nev, _regular,
    sigma) ) {
    out << "Eigenvalue regular mode (Skyline Sym) : " << cEigValue << std::endl;
    out << "Eigenvector regular mode (Skyline Sym) : " << cEigVector << std::endl;
}
// Shift and invert mode
if ( 0 != eigenSolverStd(rMatSkylineSym, cEigValue, cEigVector, nev, _shiftInv,
    sigma) ) {
    out << "Eigenvalue shift and inverse mode (Skyline Sym) : " << cEigValue <<
        std::endl;
    out << "Eigenvalue shift and inverse mode (Skyline Sym): " << cEigVector <<
        std::endl;
}
```

Complex case

```
// Regular mode
if ( 0 != eigenSolverStd(cMatSkylineSymSym, cEigValue, cEigVector, nev, _regular,
    csigma) ) {
    out << "Eigenvalue regular mode (Skyline Sym) : " << cEigValue << std::endl;
    out << "Eigenvector regular mode (Skyline Sym) : " << cEigVector << std::endl;
}
// Shift and invert mode
if ( 0 != eigenSolverStd(cMatSkylineSymSym, cEigValue, cEigVector, nev, _shiftInv,
    csigma) ) {
    out << "Eigenvalue shift and inverse mode (Skyline Sym) : " << cEigValue <<
        std::endl;
    out << "Eigenvector shift and inverse mode (Skyline Sym) : " << cEigVector <<
        std::endl;
}
```

Generalized eigenvalue problem

Real symmetric case

```

// Regular mode
if ( 0 != eigenSolverGen(rMatSkylineSymSym, rMatPosDefSym, rEigValue, rEigVector,
nev, _regular, sigma) ) {
    out << "Eigenvalue regular mode (Skyline Sym):" << rEigValue << std::endl;
    out << "Eigenvalue regular mode (Skyline Sym): " << rEigVector << std::endl;
}
// Shift and invert mode
if ( 0 != eigenSolverGen(rMatSkylineSymSym, rMatPosDefSym, rEigValue, rEigVector,
nev, _shiftInv, sigma) ) {
    out << "Eigenvalue shift and inverse mode (Skyline Sym) :" << rEigValue <<
std::endl;
    out << "Eigenvalue shift and inverse mode (Skyline Sym) : " << rEigVector <<
std::endl;
}
// Buckling mode
if ( 0 != eigenSolverGen(rMatPosDefSym, rMatSkylineSymSym, rEigValue, rEigVector,
nev, _buckling, sigma) ) {
    out << "Eigenvalue buckling mode (Skyline Sym) (positive definite):" <<
rEigValue << std::endl;
    out << "Eigenvalue buckling mode (Skyline Sym) (positive definite): " <<
rEigVector << std::endl;
}
// Cayley mode
if ( 0 != eigenSolverGen(rMatSkylineSymSym, rMatPosDefSym, rEigValue, rEigVector,
nev, _cayley, sigma) ) {
    out << "Eigenvalue cayley mode (Skyline Sym):" << rEigValue << std::endl;
    out << "Eigenvalue cayley mode (Skyline Sym): " << rEigVector << std::endl;
}

```

Real non -symmetric case

```

// Regular mode
if ( 0 != eigenSolverGen(rMatSkylineSym, rMatPosDefSym, cEigValue, cEigVector, nev,
_regular, sigma) ) {
    out << "Eigenvalue regular mode (Skyline Dual) : " << cEigValue << std::endl;
    out << "Eigenvector regular mode (Skyline Dual) : " << cEigVector << std::endl;
}
// Shift and invert mode
if ( 0 != eigenSolverGen(rMatSkylineSym, rMatPosDefNoSym, cEigValue, cEigVector,
nev, _shiftInv, sigma) ) {
    out << "Eigenvalue shift and inverse mode :" << cEigValue << std::endl;
    out << "Eigenvalue shift and inverse mode: " << cEigVector << std::endl;
}
// Complex shift with real part
if ( 0 != eigenSolverGen(rMatSkylineSym, rMatPosDefNoSym, cEigValue, cEigVector,
nev, _cplxShiftInv, 'R', csigma) ) {
    out << "Eigenvalue complex shift and inverse mode (Real part) :" << cEigValue
<< std::endl;
    out << "Eigenvalue complex shift and inverse mode (Real part): " << cEigVector
<< std::endl;
}
// Complex shift with imaginary part
if ( 0 != eigenSolverGen(rMatSkylineSym, rMatPosDefNoSym, cEigValue, cEigVector,
nev, _cplxShiftInv, 'I', csigma) ) {
    out << "Eigenvalue complex shift and inverse mode (Imaginary part):" <<
cEigValue << std::endl;
    out << "Eigenvalue complex shift and inverse mode (Imaginary part): " <<
cEigVector << std::endl;
}

```

Complex case

```

// Regular mode

```

```

if ( 0 != eigenSolverGen(cMatSkylineSymSym, cMatPosDef, cEigValue, cEigVector, nev,
    _regular, csigma)) {
    out << "Eigenvalue regular mode (Skyline Sym) : " << cEigValue << std::endl;
    out << "Eigenvector regular mode (Skyline Sym) : " << cEigVector << std::endl;
}
// Shift and invert mode
if ( 0 != eigenSolverGen(cMatSkylineSymSym, cMatPosDef, cEigValue, cEigVector, nev,
    _shiftInv, csigma, "LM")) {
    out << "Eigenvalue shift and inverse mode (Skyline Sym) : " << cEigValue <<
        std::endl;
    out << "Eigenvector shift and inverse mode (Skyline Sym) : " << cEigVector <<
        std::endl;
}

```

Besides making use of class **EigenSolverStdArpackpp** and class **EigenSolverGenArpackpp**, we can call ARPACK++ problem object directly. However, it's only recommended for advanced users!! The following examples describe how to use ARPACK++ object directly.

```

//1. Create interface object
LargeMatrixArpackppStdReg<Real, LargeMatrix<Real> > intMat(rMatSkylineSymSym,
    size);

//2. Create Arpack++ problem object
ARSymStdEig<Real, LargeMatrixArpackppStdReg<Real, LargeMatrix<Real> > >
    arProb(size, nev, &intMat, &LargeMatrixArpackppStdReg<Real, LargeMatrix<Real>
    >::multOPx, (char *)"LM");

//3. Compute eigenvalues
Int nconv = arProb.FindEigenvalues();

for (Int i = 0; i < nconv; ++i) {
    out << arProb.Eigenvalue(i) << " ";
}
out << "\n";

//4. Change some parameters
arProb.ChangeNev(nev-2);
arProb.ChangeTol(1.e-6);

nconv = arProb.FindEigenvectors();

for (Int i = 0; i < nconv; ++i) {
    out << arProb.Eigenvalue(i) << " ";
}
out << "\n";

//4. Retrieve eigenVector
Real* eigenVec_p = 0;
for (Int i = 0; i < nconv; ++i) {
    eigenVec_p = arProb.RawEigenvector(i);
    out << "Eigenvector [" << i << "] :";
    for (Int j = 0; j < size; j++) {
        out << *eigenVec_p << " ";
        eigenVec_p++;
    }
    out << "\n";
}

out << "\n";

```


OpenMP is a shared-memory application programming interface (API) whose features are based on prior efforts to facilitate shared-memory parallel programming. OpenMP is intended to be suitable for implementation on a broad range of SMP architectures, even for uniprocessor computers also. OpenMP can be added to a sequential program in Fortran, C, or C++ to describe how the work is to be shared among threads that will execute on different processors or cores and to order accesses to shared data as needed. The appropriate insertion of OpenMP features into a sequential program will allow applications to benefit from shared-memory parallel architectures with minimal modification to the code. Because of these advantages, OpenMP has been added to **XLiFE++** in order to exploit its considerable parallelism in heavy computation parts. In a typical Finite Element package, assembling the stiffness matrices costs most of the total runtime then solving linear equation systems.

17.1 Sparse matrix-vector multiplication

In a Finite Element package, after repeatedly setting up the stiffness matrices, it needs to solve a linear equation system, which can be done either in two ways (1) by direct methods or (2) iterative methods. In direct methods, the factorization of matrix costs nearly most of the runtime, meanwhile, in iterative methods, matrix-vector multiplication plays an important role. Since the matrices arising from the discretisation are sparse, an appropriate matrix storage format, sparse format is used. **XLiFE++** provides a plenty types of sparse matrix, which can serve for different purposes. Although all the sparse matrix can be taken for parallelized sparse-matrix-vector-multiplication (SpMV); among them, CSR (compressed storage row) seems to be the best candidate.

The form of SpMV is $y = Ax$, where A is a sparse matrix, x and y are dense vectors. x is source vector and y is destination vector.

In the following, the algorithms of parallelization of matrix-vector multiplication for CSR and CSC (compressed storage column) are mentioned, on which parallelized matrix-vector multiplication of other sparse format (CSDual, CSSym, SkylineDual, SkylineSym) are based.

17.1.1 SpMV of CSR

Because CSR is a well-known format, only a small recall about its structure is presented.

A matrix A is an $m \times n$ matrix, and the number of non-zero elements is nz , CSR format needs to store three arrays:

- `values[nz]` store the value of each non-zero element in matrix A
- `colIndex[nz]` stores the column index of each element in `val[nz]` array
- `rowPointer[m+1]` stores the index of the first non-zero element of each row and `rowPointer[m] = nz`

For instance, the following 5×6 matrix

$$A = \begin{bmatrix} 11 & 12 & 0 & 14 & 0 & 16 \\ 0 & 22 & 23 & 0 & 0 & 26 \\ 31 & 32 & 33 & 0 & 0 & 0 \\ 0 & 0 & 43 & 44 & 0 & 0 \\ 51 & 0 & 53 & 54 & 55 & 0 \end{bmatrix}$$

is stored as follow in XLiFE++ by the three following *std::vector*:

```
values = ( 0 11 12 14 16 22 23 26 31 32 33 43 44 51 53 54 55 )
colIndex = ( 0 1 3 5 1 2 5 0 1 2 2 3 0 2 3 4 )
rowPointer = ( 0 4 7 10 12 16 )
```

In XLiFE++, different from "standard", the array *values* of CSR format contains one more element 0 at the first position.

Since it must store the location information explicitly as well as the value of each non-zero element, extra communication time is need to access these location data. As a matrix of CSR format is store row by row, the simplest way to parallelize the matrix-vector multiplication using OpenMP is to assign each thread of parallel region to work with one row. However, considering the problem of load balancing, scheduling overhead and synchronization overheads, it's better to apply the *row partitioning* scheme. The matrix will be partitioned into blocks of row by the number of threads.

```
// Assign each thread an approximately equal number of non-zero
Number nnzEachThread = std::floor(nnz/numThread);

// Index of non-zero in each range for each thread
Number nnzRangeIdx = nnzEachThread;

// Lower and upper bound position for each thread
std::vector<std::vector<Number>::const_iterator> itThreadLower(numThread);
std::vector<std::vector<Number>::const_iterator> itThreadUpper(numThread);

itp = itpb;
itThreadLower[0] = itpb;
for (int i = 0; i < numThread; i++) {
    itThreadLower[i] = itp;
    nnzRangeIdx = *itp + nnzEachThread;
    itpLower = std::lower_bound(itp, itpe, nnzRangeIdx);
    itpUpper = std::upper_bound(itpLower, itpe, nnzRangeIdx);
    itp = ((nnzRangeIdx - *(itpUpper - 1)) < (*itpUpper - nnzRangeIdx)) ? itpUpper - 1 :
        itpUpper;
    itThreadUpper[i] = itp;
}
itThreadUpper[numThread-1] = itpe;
```

Each block is assigned to a thread and has an approximately equal number of non-zero elements. By partitioning this way, each thread operates on its own part of the *values*, *colIndex* and *rowPointer*. All threads access element of the source vector *x*. Since accesses on *x* are read-only, there is no invalidation traffic. One advantage of *row partitioning* is that each thread works on its own part of destination vector *y*, therefore, there is no need of synchronization among threads.

```
#pragma omp parallel for default(none)\
    private(i, itr, iti, itie, itim, itpLower, itpUpper) \
    shared(numThread, itbLower, itbUpper, itpb, itib, itm,
        itvb, itrb) \
    schedule(dynamic,1)
for(i = 0; i < numThread; i++) {
    itpLower = *(itbLower+i);
    itpUpper = *(itbUpper+i);
    for (; itpLower != itpUpper; itpLower++) {
        itr = itrb + (itpLower - itpb);
        *itr *= 0;
        iti = itib + *(itpLower);
        itie = itib + *(itpLower + 1);
        itim = itm + *(itpLower);

        while(iti != itie) {
            *itr += *(itim) * *(itvb + *iti); iti++; itim++;
        }
    }
}
```

```

    }
  }
}

```

However, the coarse-grained approach sometimes can not assure the load balancing of SpMV of a very large matrix (e.g with million non-zero elements), and a **granularity factor** is a solution to make it more fine-grained. Each block of row of a thread is divided into smaller pieces, which not only make SpMv more fine-grained but also allow a thread to take work from each other.

```

const Number GRANULARITY = 16;
numThread *= GRANULARITY;

```

library :	largeMatrix
header :	RowCsStorage.hpp
implementation :	RowCsStorage.cpp
unitary tests :	test_OpenMP.cpp
header dependences :	CsStorage.hpp, config.h, utils.h

17.1.2 SpMV of CSC

A matrix A is an $m \times n$ matrix, and the number of non-zero elements is nz , CSC format needs to store three arrays:

- `values[nz]` store the value of each non-zero element in matrix A
- `rowIndex[nz]` stores the row index of each element in `val[nz]` array
- `colPointer[m+1]` stores the index of the first non-zero element of each column and `colPointer[m] = nz`

For instance, the following 5×6 matrix

$$A = \begin{bmatrix} 11 & 12 & 0 & 14 & 0 & 16 \\ 0 & 22 & 23 & 0 & 0 & 26 \\ 31 & 32 & 33 & 0 & 0 & 0 \\ 0 & 0 & 43 & 44 & 0 & 0 \\ 51 & 0 & 53 & 54 & 55 & 0 \end{bmatrix}$$

is stored as follow in XLiFE++ by the three following `std::vector`:

```

values = ( 0 11 31 51 12 22 32 23 33 43 53 14 44 54 55 16 26 )
rowIndex = ( 0 2 4 0 1 2 1 2 3 4 0 3 4 4 0 1 )
colPointer = ( 0 3 6 10 13 14 16 )

```

In XLiFE++, different from "standard", the array `values` of CSC format contains one more element 0 at the first position

Because the "column-like" storage of CSC is contrast to the natural manner of "row-processing" matrix-vector multiplication, the approach using OpenMP to parallelize matrix-vector multiplication of CSC is to limit the affect of this "column-like" characteristic.

Since a CSC matrix is stored in column by column, the simplest way to parallelize the matrix-vector multiplication using OpenMP is to use each thread to process a column, one by one; then the result of each thread is contributed to the destination vector by a "reduction-like" operation. Obviously, this approach is not efficient. Not only each thread processes non-adjacent column, which causes cache misses in cases of many threads; but also the problem of load-balancing can happen because of the difference in the number of non-zero elements in each column. An improved approach is to divide the matrix into

groups of column, each of which has an approximately equal number of non-zero elements. Because all the columns in each group are adjacent, it limits the effect of cache miss. Moreover, load-balancing hardly happens thanks to equal number of non-zero elements processed by each thread. The following code implements the approach described above.

```
// Lower and upper bound position for each thread
std::vector<std::vector<Number>::const_iterator> itThreadLower(numThread);
std::vector<std::vector<Number>::const_iterator> itThreadUpper(numThread);

std::vector<Number>::const_iterator itpLower, itpUpper;
// Find the smallest and largest rowIdxs corresponding to each thread
std::vector<Number> sRowIdx(numThread);
std::vector<Number> lRowIdx(numThread);

itp = itpb;
itThreadLower[0] = itpb;
for (int i = 0; i < numThread; i++) {
    itThreadLower[i] = itp;
    nnzRangeIdx = *itp + nnzEachThread;
    itpLower = std::lower_bound(itp, itpe, nnzRangeIdx);
    itpUpper = std::upper_bound(itpLower, itpe, nnzRangeIdx);
    itp = ((nnzRangeIdx - *(itpUpper - 1)) < (*itpUpper - nnzRangeIdx)) ? itpUpper - 1 :
        itpUpper;
    itThreadUpper[i] = itp;
}
itThreadUpper[numThread - 1] = itpe;

std::vector<std::vector<Number>::const_iterator>::const_iterator itbLower, itbUpper;
itbLower = itThreadLower.begin();
itbUpper = itThreadUpper.begin();
```

The serial codes above split a CSC matrix into blocks of column, each of which has nearly the same number of non-zero elements. After that, each of thread will process a block of column, store the result into its temporary vector and write this vector into the destination vector by a "reduction-like" operation. One remark is that temporary vector of each thread have necessarily the same size as the destination vector. It's is the trade-off of this method.

After finishing its own matrix-vector multiplication, each thread writes its result into the destination vector. Since array reduction isn't supported in OpenMP C++ (some projects have tried to make this feature available but it seems not to be ready in the near future!!). The only way, at the moment, is to use critical section. And we can easily realize that it is coarse-grained.

```
typedef typename IterationVectorTrait<ResIterator>::Type ResType;
#pragma omp parallel \
    private(i, tid, itr, iti, itie, itim, itpLower, itpUpper, \
            itv) \
    shared(numThread, itbLower, itbUpper, itpb, itib, itm, itvb, \
           itrb, nRows)
{
    tid = omp_get_thread_num();
    std::vector<ResType> resTemp(nRows);
    typename std::vector<ResType>::iterator itbResTemp = resTemp.begin(),
        iteResTemp = resTemp.end(), itResTemp = itbResTemp;

    #pragma omp for nowait schedule(dynamic,1)
    for(i = 0; i < numThread; i++) {
        itpLower = *(itbLower+i);
        itpUpper = *(itbUpper+i);
        for (; itpLower != itpUpper; itpLower++) {
            itv = itvb + (itpLower - itpb);
            iti = itib + *(itpLower);
            itie = itib + *(itpLower + 1);
```

```

        itim = itm + *(itpLower);

        while(iti != itie) {
            itResTemp = itbResTemp + (*iti);
            *itResTemp += *itim * *(itv);
            iti++; itim++;
        }
    }
}
#pragma omp critical (updateResult)
{
    for (itResTemp = itbResTemp; itResTemp != iteResTemp; itResTemp++) {
        itr = (itr + (itResTemp - itbResTemp));
        *itr += *itResTemp;
    }
}
}

```

```

library : largeMatrix
header : ColCsStorage.hpp
implementation : ColCsStorage.cpp
unitary tests : test_OpenMP.cpp
header dependences : CsStorage.hpp, config.h, utils.h

```

17.2 Sparse matrix factorization

As mentioned above, one way to solve a linear equation system is to use direct methods, which heavily depends on matrix factorization. Firstly, the costly runtime factorization is done, secondly, the factorized matrix is used to solve the linear equation system $Ax = b$. Till now, XLiFE++ has supported some popular factorization methods: LU, LDLt and LDL*; all of them have a parallelized version with OpenMP. Because of factorization algorithm, only skyline storage is suitable to be factorized.

Because LDLt can be considered to be a special case of LU, the following only mentions the parallelized LU factorization.

The idea behind multi-threaded factorization is simple: Instead of implementing the factorization element by element as in the serial version, we make the algorithm work on block by block. For each iteration, block on diagonal is processed then block on the row and column corresponding to this diagonal block. First of all, the diagonal block is calculated

```

#pragma omp single
diagBlockSolverParallel(k*blockSize, blockSizeRow[k], it_rownx,
                        k*blockSize, blockSizeCol[k], it_colnx,
                        it_fd, it_fl, it_fu,
                        it_md, it_ml, it_mu);

```

After that, the block of row and column can be computed in the same time

```

#pragma omp for nowait
for (jj = k+1; jj < nbBlockCol; jj++) {
    #pragma omp task untied firstprivate(k, jj) \
        shared(blockSize, blockSizeRow, blockSizeCol, it_rownx,
                it_colnx, it_fl, it_fu, it_mu)
    upperBlockSolverParallel(k*blockSize, blockSizeRow[k], it_rownx,
                            jj*blockSize, blockSizeCol[jj], it_colnx,
                            it_fl, it_fu, it_mu);
}

```

```

#pragma omp for
  for (ii = k+1; ii < nbBlockRow; ii++) {
    #pragma omp task untied firstprivate(k, ii) \
      shared(blockSize, blockSizeRow, blockSizeCol, it_rownx,
             it_colnx, it_fd, it_fl, it_fu, it_ml)
    lowerBlockSolverParallel(ii*blockSize, blockSizeRow[ii], it_rownx,
                           k*blockSize, blockSizeCol[k], it_colnx,
                           it_fd, it_fl, it_fu, it_ml);
  }
}

```

In these code, we use one of the latest feature of OpenMP, directive task, which has been only supported since OpenMP 3.0. By using this directive, we can take advantage of taskification of the code and prevent the potential of load balancing problem.

One common problem on working with blocking-algorithm is the block size. It is impossible to find out a optimal value for the blockSize that is able to change the performance of program largely. After some experiments with typical sparse matrices of XLiFE++, one simple formula is provided to specify blockSize.

```

const Number BLOCKFACTOR = 0.05*std::min(nbOfRows(),nbOfColumns());
Number numBlockMin = BLOCKFACTOR;
Number blockSize = std::floor(diagonalSize()/numBlockMin);
Number nbBlockRow = std::ceil(nbOfRows()/blockSize);
Number nbBlockCol = std::ceil(nbOfColumns()/blockSize);
Number numBlock = std::min(nbBlockRow, nbBlockCol);

std::vector<Number> blockSizeRow(nbBlockRow, blockSize), blockSizeCol(nbBlockCol,
    blockSize);
blockSizeRow[nbBlockRow-1] = nbRows_ - (nbBlockRow-1)*blockSize;
blockSizeCol[nbBlockCol-1] = nbCols_ - (nbBlockCol-1)*blockSize;

```

Again, this value is only "good" for a certain group of sparse matrices. It maybe needs changing in the future.



External libraries

A.1 Installation and use of BLAS and LAPACK libraries

Using UMFPACK or ARPACK means using BLAS and LAPACK libraries. XLiFE++ offers the ability to choose your BLAS/LAPACK installation :

- Using BLAS/LAPACK installed with UMFPACK or ARPACK
- Using default BLAS/LAPACK installed on your computer
- Using standard BLAS/LAPACK libraries, such as OPENBLAS.

To do so, you just have to use **XLIFEPP_LAPACK_LIB_DIR** and/or **XLIFEPP_BLAS_LIB_DIR** to set the directory containing LAPACK and BLAS libraries:

```
cmake [...] -DXLIFEPP_BLAS_LIB_DIR=path/to/Blas/library/directory
            -DXLIFEPP_LAPACK_LIB_DIR=path/to/Lapack/library/directory path/to/CMakeLists.txt
```



It is useless to use **XLIFEPP_LAPACK_LIB_DIR** and/or **XLIFEPP_BLAS_LIB_DIR** options if you do not activates configuration with UMFPACK or ARPACK

A.2 Installation and use of UMFPACK library

The prerequisite to make use of UMFPACK is to have it installed or at least its libraries are compiled. The *umfpackSupport* can be linked with or without UMFPACK in case of none of its functions is invoked. Otherwise, any try to use its provided functions can lead to a linkage error. Details to compile and install UMFPACK can be found at <http://www.cise.ufl.edu/research/sparse/umfpack/>. All the steps will be described below supposing UMFPACK already installed or compiled in the user's system.

In order to make use of UMFPACK routines, user must configure CMAKE with options: **XLIFEPP_ENABLE_UMFPACK**, **XLIFEPP_UMFPACK_INCLUDE_DIR** and **XLIFEPP_UMFPACK_LIB_DIR**.

```
cmake -DXLIFEPP_ENABLE_UMFPACK=ON
      -DXLIFEPP_UMFPACK_INCLUDE_DIR=path/to/UMFPACK/include/directory
      -DXLIFEPP_UMFPACK_LIB_DIR=path/to/UMFPACK/library/directory
      path/to/CMakeLists.txt
```



Theoretically, UMFPACK does not need to use BLAS/LAPACK, but as it is highly recommended by UMFPACK (for accuracy reasons), XLiFE++ demand that you use BLAS/LAPACK.

UMFPACK is provided by SUITESPARSE. When looking how UMFPACK is compiled, it seems that it can depend (maybe in the same way as BLAS/LAPACK) from other libraries provided by SUITESPARSE.

In this case, you have a simpler way to define paths: by using **XLIFEPP_SUITESPARSE_HOME_DIR**. When given alone, it consider the given directory as the home directory containing every library provided by SUITESPARSE with a specific tree structure. If it is not the case, you can use more specific options of the form **XLIFEPP_XXX_INCLUDE_DIR**

and **XLIFEPP_XXX_LIB_DIR**, where XXX can be AMD, COLAMD, CAMD, CCOLAMD, CHOLMOD, SUITESPARSECONFIG or UMFPACK.

A.3 Installation and use of ARPACK++ library

ARPACK++ distribution can be obtained from the following URL: <http://www.ime.unicamp.br/~chico/arpack++/>. However, because of the deprecation of this version, a patch at <http://reuter.mit.edu/index.php/software/arpackpatch/> needs applied to make sure a correct compilation. Users of Unix-like system can follow the instructions on <https://help.ubuntu.com/community/Arpack%2B%2B> to make the patch. And this patch is often not enough to compile correctly with recent compilers. So XLIFE++ contains its own patched release of ARPACK++, used by default.

Because ARPACK++ is an interface to the original ARPACKFORTRAN library, this library must be available when installing the C++ code. Although FORTRAN files are not distributed along with ARPACK++, they can be easily downloaded from Netlib and are even available under some Unix-like systems.

The BLAS and LAPACK routines are required by the ARPACKFORTRAN, so make sure these two libraries are installed in the system. Like ARPACK, these two libraries are available under some Unix-like systems. By default, the intern eigensolver of XLIFE++ is used for calculating eigenvalues and eigenvectors. To make use of ARPACK++, users must configure CMAKE with options : **XLIFEPP_ENABLE_ARPACK** and **XLIFEPP_ARPACK_LIB_DIR**.

The current directory is the root directory containing XLIFE++ source code. To enable ARPACK, we use the command:

```
cmake -DXLIFEPP_ENABLE_ARPACK=ON
      -DXLIFEPP_ARPACK_LIB_DIR=path/to/arpack/libraries/directory
      path/to/CMakeLists.txt
```

After configuring, we can make the library

```
make
```



Documentation policy

The documentation policy of the library is based on three complementary documentations:

- a online documentation compliant with DOXYGEN syntax,
- a user documentation describing main features of the library with some examples,
- a developer documentation describing as possible all the features and given additional explanations.

It is the responsibility and the duty of the developers to write these documentations **in English**.

B.1 Inline documentation

The inline documentation is compliant with DOXYGEN syntax and the html documentation is automatically generated with majors updates by the source control manager.

This inline documentation follows a documentation convention available on the development page of the XLiFE++ website: <http://uma.ensta-paristech.fr/XLiFE++/> . We will sum it up now in the following:

B.1.1 General points

1. The reference language of the documentation is English.
2. Function or operation declarations are documented with brief (one-line, one-sentence) comments and only brief comments.
3. Function or operation definitions (inline or not) are documented by detailed comments.

B.1.2 Documentation of a file

Here is the structure of the comment you have to write to document a file (header or source) :

```
/*!  
    brief comment  
  
    \file Parameter.hpp  
    \author T. Anderson  
    \date 25 jan 2011  
    \since 3 mar 2007  
  
    next part of detailed comment  
*/
```

This comment is dedicated to present the content of the file: list of classes, list of external functions, ... It is written at the top of the file before everything else.

Some additional remarks:

- Thanks to the QT_AUTOBRIEF = YES parameter, the first line of the comment is interpreted as the brief comment. Therefore, the brief comment has to be placed before everything else. There is another solution with the macro `\brief`:


```

/*!
  \file Parameter.hpp
  \author T. Anderson
  \date 25 jan 2011
  \since 3 mar 2007

  \brief brief comment

  next part of detailed comment
*/

```

- The `\date` macro is the last modification date.
- The `\since` macro is the file creation date.
- When there are several authors, you will use the `\authors` macro:

```

/*!
  brief comment

  \file Parameter.hpp
  \authors T. Anderson, R. Cypher
  \date 25 jan 2011
  \since 3 mar 2007

  next part of detailed comment
*/

```

In this case, authors are separated by commas.

The file documentation rules are:

1. The file documentation is written at the top of the file, before everything else,
2. The file documentation follows one of the previous syntaxes, with the 4 pieces of information file name, author(s), creation date and last modification date.

B.1.3 Documentation of a class, a typedef, an enum or a struct

1. A class documentation is written just before the class declaration/definition in the header file.
2. The structure of the class documentation is as follows:

```

/*!
  \class A
  brief comment associated to A

  next part of detailed comment associated to A
*/
class A {
...
};

```

3. To document a typedef, you will use the `\typedef` macro instead of the `\class` macro.
4. To document a struct, you will use the `\struct` macro instead of the `\class` macro.
5. To document an enum, you will use the `\enum` macro instead of the `\class` macro.

B.1.4 Documentation of attributes

1. Attributes can be documented according to one of the following syntax:

```
class A {
    public :
        //! comment associated to i
        int i;
        double j; //!< comment associated to j
        /*!
            comment associated to k
        */
        B k;
        ...
}
```

B.1.5 Documentation of functions and constructors

1. Functions and operations declarations are documented with a brief comment, and only a brief comment. The syntax, shown here for operations but valid for external functions, is one of the following :

```
class A {
    public :
        ...
        //! brief comment associated to declaration of f
        void f();

        /*!
            brief comment associated to declaration of g
        */
        void g();

        void h(); //!< comment associated to declaration of h
};
```

2. Functions and operations definitions (inline or not) are documented with detailed comments. If the declaration has not been commented first, the first line of the comment will be the brief comment, as seen in previous sections. The syntax, shown here for inline operations but valid for the others types, is one of the following:

```
class A {
    public :
        B b;
        ...
        //! comment associated to definition of f
        void f() {
            ...
        }
        void g() //! comment associated to definition of g
        {
            ...
        }
        A() //! comment associated to definition of composite constructor
        : B()
        {
            ...
        }

        /*!
            comment associated to definition of h
        */
        void h() {
```

```

    ...
}
};

```

You may notice that for the function `g`, the comment marking is `/*!` and not `/*!<` and that for the constructor in the same case, the comment is placed before components constructor calls.

Let's see how to document function arguments.

B.1.6 Documentation of arguments

1. In the case only of a function definition (or an operation definition), you may want to document arguments and return value. The syntax is one of the following :

```

/*!
    comment of function f
*/
int f (double d, /*!< comment of d
           int i /*!< comment of i
        ) {
    ...
}
/*!
    comment of function g
    \param d comment of d
    \param i comment of i
    \return comment of return value
*/
int g (double d, int i) {
    ...
}

```

B.2 User and developer documentation

User documentation as well developer documentation have to be written **in L^AT_EX format**. Each tex file concerns a package of files of the library, namely a topics of the library, for instance Parameters. For each package `xxx`, two files have to be written: the user documentation file, named `xxx.tex` placed in subdirectory `doc/pdf/inputs/dev/` and the developer documentation file, also named `xxx.tex`, but placed in the subdirectory `doc/pdf/inputs/usr/`. In these files, there are only latex section items (one or fewer) and subsection items, no chapter items. All these files are collected in the main tex files: `user_documentation.tex` and `dev_documentation.tex` in chapter environment, each chapter corresponding to a folder of the library.

For instance, the Parameters user documentation (`Parameters.tex`) looks like:

```

\section{Parameters classes}

In order to attached some user's data to anything (in particular functions), two
classes
(Parameter and Parameters) are proposed. The {\class Parameter} class handles a
single data
of type {\emph{integer, real, complex, string}} or {\emph{void *}} with the possibility
to name
the parameter. The {\class Parameters} class handles a list of {\class Parameter}
objects.\\

\subsection{The {\class Parameter} object}

```

```

It is easy to define a parameter by its constructor or the assignment operation:
...
\subsection{The {\class Parameters} object: List of parameters}
...

```

The Parameters developer documentation (*Parameters.tex*) looks like:

```

\section{Parameters}

It may be useful to have a data structure to store freely any kind of parameters
(values,
string, ...) with a very simple interface for the end user. In particular, such
structure
could be attached to user functions (see chap. \ref{chapfunctions}). This is
achieved with
two classes: the {\class Parameter} class which define one parameter and the
{\class Parameters} class which manages a list of {\class Parameter} objects.

\subsection{The {\class Parameter} class}

The aim of the {\class Parameter} class is to encapsulate in the same structure, a
data of
integer type, real type, complex type, string type and pointer type (void pointer)
with the
capability to name the parameter. Thus, this class proposes as private members:
...
\subsection{The {\class Parameters} class}
...

\displayInfos{library=utils, header=Parameters.hpp, implementation=Parameters.cpp,
test=test\_Parameters.cpp, header dep={config.h, String.hpp}, stl dep={map,
iostream,
sstream}}

```

- The end of a file has to be exposed the file dependences with the following form:

```

library : utils
header : Parameters.hpp
implementation : Parameters.cpp
unitary tests : test_Parameters.cpp
header dependences : config.h, String.hpp

```

For this purpose, you have to use the `\displayInfos` macro.

- To handle some C++ lines, you have to use the environment *lstlisting*:

```

\begin{lstlisting}
class Parameter
{...};
\end{lstlisting}

```

```

class Parameter
{...};

```

or if you want to ignore language colors:

```
\begin{lstlisting}[language={}]
class Parameter
{...};
\end{lstlisting}
```

```
class Parameter
{...};
```

- When you name a class, for example `Messages`, you have to use the `\class` macro:

The `{\class Messages}` class

- When you name a sublibrary of XLiFE++ (a subdirectory of `src`), you have to use the `\lib` macro:

The `{\lib utils}` library

- When you name a function, you have to use the `\cmd` macro:

The `{\cmd findDomain}` function

- When you name the library XLiFE++, you have to use the `\xlfepp` macro. For external libraries of softwares, such as GMSH or ARPACK++, you have to use `\gmsh` or `\arpackpp` macros. The full available list is : `\xlfepp`, `\gmsh`, `\arpack`, `\arpackpp`, `\blas`, `\lapack`, `\umfpack`, `\eispack`, `\melina`, `\melinapp`, `\montjoie`, `\paraview`, `\freefem`, `\doxygen`, `\cmake`, `\git` and `\convmesh`.
- When you add a package documentation in *user_documentation.tex* or *dev_documentation.tex*, you have to use the `\inputDoc` with the basename of the file in argument, and to add it in the main itemize of the library:

```
\chapter{The {\lib utils} library}
```

The `{\lib utils}` library collects all the general useful classes and functionalities of the `\xlfepp` library. It is an independent library. It addresses:

```
\begin{itemize}
\item {\class String} capabilities (mainly additional functions to the
{\class std::string} class)
\item {\class Point} class to deal with point of any dimension
\item {\class Vector} class to deal with numerical vectors (say real or complex
vectors)
\item {\class Matrix} class to deal with numerical matrices with dense storage
(small
matrices)
\item {\class Parameter} classes to deal with general set of user parameters
\item {\class Function} class encapsulating user functions
\item {\class Messages} classes to handle error, warning and info messages
\end{itemize}

\inputDoc{String}
\inputDoc{Point}
\inputDoc{Vector}
\inputDoc{Matrix}
\inputDoc{Parameters}
\inputDoc{Functions}
\inputDoc{Messages}
```

- When you add a general documentation in *user_documentation.tex* or *dev_documentation.tex*, you have to use the `\inputDoc*` with the basename of the file in argument, and to add it in the main itemize of the library:

```
\chapter{Tests policy}
```

In order to maintain backward compatibility of the code during its development, it is mandatory to have various test functions. Some of them are local test functions, says low level tests, and concerns "exhaustive" tests of functionalities of a class or a collection of classes or utilities. Others are global test functions, say high level test functions, involving a lot of classes of the library, for instance, solving a physical problem from the mesh up to the results.

```
\inputDoc*{tests_policy}
```



Tests policy

In order to maintain backward compatibility of the code during its development, it is mandatory to have various test functions. Some of them are local test functions, say low level tests, and concerns "exhaustive" tests of functionalities of a class or a collection of classes or utilities. Others are global test functions, say high level test functions, involving a lot of classes of the library, for instance, solving a physical problem from the mesh up to the results.

C.1 Test families and test functions

C.1.1 About test families

There are several families of tests:

unit: Unitary tests are aimed to test each function of each class separately. In fact, it is easier to say it than to do it. But with a very large set of unitary tests, one can avoid most of bugs.

sys: System tests are aimed to test some user applications but with size of the linear systems not too big.

dev: Developer tests are aimed to test new functionalities someone is developing. These tests are often like system tests, but are supposed to be deleted as soon as functionalities are fully developed. This family can also be used to check error convergence and cputime for larger problems.

ext: External tests are aimed to test wrappers to external libraries, such as ARPACK and UMFPACK.

Each family correspond to a sub-directory of *tests*.

C.1.2 Definition of a test function

A test function is a usual c++ function with a meaningful name **beginning by the family name it belongs to** defined in a file with the same name. The test function takes an optional boolean as input argument and returns a string (String class). For instance, for the unitary tests of the class Function, the file *unit_Function.cpp* in the *unit* directory looks like:

```
/*! \file unit_Function.cpp
    \author XXX
    \since YYY
    \date ZZZ
*/
#include "xlife++-libs.h"
#include "testUtils.h"

String unit_Function(bool check=true)
{String report="unit_Function : OK";
...
return report;
}
```

and for the system test SauterSchwabIM:

```

/*! \file sys_SauterSchwabIM.cpp
    \author
    \since
    \date

*/
#include "xlife++-libs.h"
#include "testUtils.h"

String sys_SauterSchwabIM (bool check=true)
{String report="sys_SauterSchwabIM : OK";
  ...
  return report;
}

```

The string report has to contain information about the errors occurring during the test in order to easily locate them. There is no writing format of the errors except **the first line of the report** which has to be of the form:

- if the test succeeds: **name_of_the_test : OK**,
- if the test fails: **name_of_the_test : n error(s)** with $n > 0$ the number of errors.

It is up to the developers to write meaningful tests and more exhaustive tests of the class functionalities as possible.

C.1.3 Development of a test function

There are mainly two approaches to make a test:

- internal value testing: compare results to expected results inside the function (value comparison),
- external value testing: comparing results to reference results get previously and stored in a file (ascii comparison).

The first one is better than the second one for two reasons: in one hand it avoids false positive answers due to change of comment or variable answers such as the value of a pointer and in the other hand it allows to distinguish some variations due to rounding errors which are probably meaningless errors. Note that it is possible to use the file comparison if the file contains only numerical values or if you read it in a way to recover meaningful values.

For instance, the internal value testing SauterSchwabIM test looks like:

```

String sys_SauterSchwabIM (bool check=true)
{String report="sys_SauterSchwabIM : OK";
  ...
  real_t nSol= ... //compute the norm of the solution
  real_t nSolExpected=0.12345678; //expected norm get previously
  real_t diff=abs(nSol-nSolExpected);
  if (diff>tol) //tol is a tolerance (10*epsMachine)
  {report="sys_SauterSchwabIM : 1 error";
    report+="\n * error 1 : difference between norm of the solution are different,";
    report+=" ' difference="+tostring(diff);
    return report;
  }
  if (diff!=0) //almost the same
  {report="sys_SauterSchwabIM : 1 error";
    report+="\n * error 1 : norm of the solution are almost the same,";
    report+=" difference="+tostring(diff);
    report+=" possibly a rounding error";
  }
}

```



```

return report;
}

```

and the external value testing looks like:

```

String sys_SauterSchwabIM(bool check=true)
{String report="sys_SauterSchwabIM : OK";
...
real_t nSol= ... //compute the norm of the solution
std::stringstream out;
out<<"norm of the solution ="<<nSol;
if(!check) //save the result in a file
{std::ofstream fout("sys_SauterSchwabIM.res");
fout<<out;
fout.close();
report="sys_SauterSchwabIM : results save to file sys_SauterSchwabIM.res";
}
else //compare the result
{ifstream fin("sys_SauterSchwabIM.res");
String resf;
fin>>resf;
if(resf!=out) //lines are different
{report="test_Laplace3D : 1 error";
report+="\n * error 1 : difference between the lines";
report+="\n ref : "+resf;
report+="\n new : "+out;
}
return report;
}
}

```

This second approach has the advantage to have a unique file to construct the checked values (check=false) and to performs the regression test (check=true) but it is more sensitive. In order to help the developers, some useful functions are proposed in the *testUtils.cpp* file:

```

String saveResToFile(std::stringstream& result, const String& rootname);
String diffResFile(std::stringstream& result, const String& rootname);

```

where *result* is a stringstream containing all the output of a test and *rootname* is the name of the test function. The function *saveResToFile* write the results of the test to the file *rootname.res* and the function *diffResFile* compares the results of the test with the file, line by line. Both, the functions produce a compliant report. Using the function, the previous example becomes:

```

String sys_SauterSchwabIM(bool check=true)
{String report="sys_SauterSchwabIM : OK";
...
real_t nSol= ... //compute the norm of the solution
std::stringstream out;
out<<"norm of the solution ="<<nSol;
String rootname="sys_SauterSchwabIM";
if(check) return diffResFile(out, rootname);
else return saveResToFile(out, rootname);
}

```

The external value testing is rather dedicated to the tests of classes and functionalities which involve a lot of basic tests.

C.1.4 Includes

As you may have noticed in previous listings, a test file, whatever its family always includes 2 headers: *xlife++-libs.h* and *testUtils.h*.

C.2 Test of a new version of the library

This section addresses the process of global test involved before a minor or major upgrade of the library. It concerns the administrators of the library.

C.2.1 Compilation of tests

When compiling tests, each individual test is compiled, but also one global test per family. There is also a specific global test running both unit and sys test families. When compiling a global test, CMAKE sources all files in the corresponding directory prefixed by the family. If a source file does not respect this naming convention, it is ignored.

C.2.2 Running tests

The administrators have a special tool to run a lot of tests, *xlifepp_test_runner.rb*. This is a Ruby script that runs one executable without compiling it if it does not exist.

```
xlifepp_test_runner.rb deals with all automatic tasks such as updating doxygen,  
generating  
commit history and generating snapshots and releases
```

SYNOPSIS:

```
xlifepp_test_runner.rb -c [-n] <test>  
xlifepp_test_runner.rb -e [-n] <test>  
xlifepp_test_runner.rb -h
```

DESCRIPTION:

XLiFE++ is a C++ library with a large set of tests. Each test can be launched in 2 modes :

- the edit mode (default in XLiFE++ executables) makes tests generating results file, so called res file
- the check mode (default in xlifepp_test_runner.rb) makes tests comparing results to the current res file

As XLiFE++ compilation is based on CMake, it is not a good way to define running targets, because of generators (to Eclipse, XCode, Visual Studio, CodeBlocks, ...). On the one hand, the lesser targets there are the easier XLiFE++ is to compile and run, on the other hand targets in generators are not just compilation targets: you can run them, and pass parameters to them. However, XLiFE++ developers using cmake in command-line mode must have an easy way to run tests.

That's what xlifepp_test_runner.rb is for !!!

OPTIONS:

-h, --help	shows the current help
-c, --check	test will be run in check mode (default)
-cxx, --cxx-compiler	sets the compiler to use
-e, --edit	test will be run in edit mode
<test> sys_1d, ...)	name of the test (Examples: all, unit_Mesh_gmsh,
-n	activates dry-run mode
-v	activates verbose level (same as --verbose-level
1)	
--verbose-level <num> none.	allows to set the verbose level. Default is 0,



Compiling and linking policy

D.1 General purpose

With XLiFE++, we want to develop a C++ library, that we can use easily and at list on Windows, Unix and Mac OS platforms. The last two are very similar, whereas the Windows platform is a very different working environment as far as compiling is concerned. Generally, developing code on Windows means the use of an IDE, such as Visual Dev C++, Eclipse, CodeBlocks, ... On Unix and Mac OS, you can use a "basic" text editor and Gnu Makefile. You can also use IDE, such as XCode on Mac OS. As we do not want to impose an IDE, the multi-platform solution comes with CMAKE!!!

D.2 CMAKE and IDE generators

D.2.1 CMAKE and Windows

It is very easy to have a IDE generator with CMAKE. The GUI was very well thought. The most important part is to give the path to the CMakeLists.txt file.

D.2.2 CMAKE and Unix or Mac OS

If you does not want to use an IDE, you just have to execute the following command :

```
cmake PATH_TO_CMakeLists.txt
```

D.2.3 New files and IDE generators

When you change the name of some source files or delete some source files or create source files, you have to rerun CMAKE to take them into account. If you modify file contents, it is unnecessary. This means you have to rebuild the IDE generator you use. A new file will not be detected and compiled.

D.3 CMAKE and XLiFE++

Whatever your OS or IDE choice, you may have to go to the `build` directory to launch CMAKE. As a result, every file generated by CMAKE will be in this directory, even for a IDE generator. The `CMakeLists.txt` file is in the XLiFE++ root directory.

How does the `CMakeLists.txt` work ?

D.3.1 Building libraries

First, it lists all files in the `src` directory with extension `.hpp` or `.cpp` and extracts the names of the direct sub-directories of `src`. This list is the list of the libraries to compile.

Then, for each library, it lists all files with extension `.hpp` or `.cpp` and builds targets for the library generation. The name of the lib is `libXXX.a` where `XXX` is the name of the directory in the lib list and is supposed to be placed in the `lib` directory. More precisely, it will be in subdirectories of `lib` whose names are built according to the compiler used (version included), the architecture of the computer, and the build type (debug, release, ...).

D.3.2 Building and linking test executables

CMAKE lists all files in the `tests` directory with pattern `XXX/XXX_YYY.cpp`. Each of these files correspond to a function having the same name. CMAKE uses a template file to build the test main program : `main_XXX_YYY.cpp`. The main function will take an argument, `edit` or `check`, to define the value of the boolean check.

These files are placed in the sub-directory `build` of the directory `tests` are are linked to an executable. Furthermore, one additional file is generated : `main_test.cpp` to launch all tests.