School
Fundamentals and practice of finite elements
Roscoff, april, 2017

A multipurpose FEM-BEM library

E. Lunéville
Unité de Mathématiques Appliquées
POEMS, ENSTA ParisTech
POEMS lab (~15 researchers, ~15 PhD students)
Mathematical and numerical studies about wave propagation

- Develop new mathematical or numerical PDE models, new numerical methods
- Numerical analysis and numerical experiments
- Require a versatile and flexible Finite Element software:
  - **multi-purpose**: various PDE, various methods (FEM, BEM, spectral approximation)
  - **with high coupling potential** (multiple unknowns, various geometrical domains, ...)
  - **easy to use for a mathematician** (not a geek)
  - **rather fast** but not as much as specialized software

A long story of FE softwares at POEMS and IRMAR

<table>
<thead>
<tr>
<th>Year</th>
<th>Version</th>
</tr>
</thead>
<tbody>
<tr>
<td>1980</td>
<td>LENA</td>
</tr>
<tr>
<td>1989</td>
<td>MELINA</td>
</tr>
<tr>
<td>2002</td>
<td>MONTJOIE</td>
</tr>
<tr>
<td>2004</td>
<td>MELINA++</td>
</tr>
<tr>
<td>2008</td>
<td>COFFEE</td>
</tr>
<tr>
<td>2010</td>
<td>XLiFE++</td>
</tr>
<tr>
<td>2014</td>
<td>XLiFE++ V1.0</td>
</tr>
<tr>
<td>2018</td>
<td>XLiFE++ V2.0</td>
</tr>
</tbody>
</table>
### XLiFE++ objectives

**eXtended LiBrary of Finite Elements in C++**

- Deal with 1D, 2D, 3D scalar/vector transient/stationary/harmonic problems
- High order Lagrange FE, edge FE (Hrot, Hdiv), spectral approximation
- $H_1$, Hrot, Hdiv conform and non conform approximation (DG methods)
- Unassembled FE
- Integral methods (BEM, FEM-BEM)
- Essential condition (standard, periodic, quasi-periodic, moment)
- Absorbing condition, PML, DTN, ...
- Meshing tools and export tool
- Many solvers (direct solvers, iterative solvers, eigen solvers)
- Parallel programming (OPENMP, MPI, CUDA/OPENCL ?)

- Multi platform (linux, mac, windows)
- Online and paper documentation
- Repository and versioning
- Regression testing
- Install and compilation procedures

---

facility to deal with new FE methods, new applications, ...
XliFE++ is based on variational formulation

**Problem to solve**: Helmholtz in a bounded domain

\[
\begin{cases}
\Delta u + k^2 u = -f & \text{in } \Omega \\
u = 0 & \text{on } \Gamma \\
\partial_n u = 0 & \text{on } \Sigma
\end{cases}
\]

Find \( u \) in an approximation space \( V_h \), \( u = 0 \) on \( \Gamma \) such that

\[
\int_{\Omega} \nabla u \nabla v - k^2 \int_{\Omega} u v = \int_{\Omega} f v \quad \forall v \in V_h, \quad v = 0 \text{ on } \Gamma
\]

**Matrix representation**

\[
V_h = \left\{ \sum_{i=1}^{n} u_i w_i(x) \right\}
\]

\[
AU = F \quad \text{with} \quad CU = 0 \quad \text{(constraints)}
\]
XLiFE++ architecture

Geometry lib (1d, 2d, 3d)
Geometry, Mesh, Domain

Space lib
Space, Unknown, TestFunction

finiteElement lib (1d, 2d, 3d)
Interpolation, Integration method

Geometry and Finite Element

essentialConditions lib
EssentialCondition, Constraints

Form lib
BilinearForm, LinearForm

Operator lib
DifferentialOperator, OperatorOnU

Paraview
Geometry

Term lib
TermMatrix, TermVector

IterativeSolver lib
GMRES, CG, BiCG, SSOR, …

EigenSolver lib
EigenSparse, Arpack

MathRessources lib
Special functions, Kernels
Exact solutions

Util s lib
String, Complex, Vector, Matrix
Value, Function, Parameter
Environment, Timer, Message, …

Eigen

Arpack

Umfpack

Amos
What XLIFE++ looks like

\[ \int_{\Omega} \nabla u \cdot \nabla v - k^2 \int_{\Omega} u \, v = \int_{\Omega} f \, v \quad \forall v \in V_h \quad (V_h \text{ a P1 approximation}) \]

A C++ main program

```cpp
Real f(const Point& P, Parameters& pa = defaultParameters) { return P(1)*P(2);}  // define data function
void main{
    init(fr);
    Mesh m(Square(0, 1, 0, 1, 100, "Omega"), _triangle, _structured);  // choose french mesh the unit square
    Domain omega = m.domain("Omega");  // get domain
define P1 space
    Space V(omega, P1);
    Unknown u(V,"u"); TestFunction v(u, "v");  // define symbolic unknown and test function (dual)
    Real k2=2.;
    BilinearForm auv = intg(omega, grad(u) | grad(v)) – k2* intg(omg, u * v);  // define symbolic bilinear and linear forms
    LinearForm fv = intg(omega, f * v);
    TermMatrix A(auv);
    TermVector B(fv);
    TermVector U = directSolve(A, B);  // define algebraic representations A and B
    solve AU=B using direct method
    saveToFile("U.vtk",U,_vtk);}  // save U on vtk file (Paraview)
```

Syntax as close as possible to Mathematics

basic C++ (no pointer, no template) 🤝 // XLiFE++ introduction
XLiFE++ main skills

V 2.0

- Mesh tools (structured, subdivision and gmsh interface)
- Lagrange (P_k, Q_k, 1D/2D/3D) at any order
- Nedelec edge and face on triangle and tetrahedron at any order
- real/complex scalar/vector unknowns
- FEM bilinear/linear form and generalized BEM bilinear/linear form
- Dense, skyline, sparse and hierarchical matrix
- Direct solvers (umfpack, magma), iterative solvers, eigen solvers (arpack)
- OMP parallelization (FE computation and solvers)

 licences GPL3 for academic usage
 about 170000 C++ lines (220000 with comments), 170 C++ classes,
 user doc (240 pages), developer doc (375 pages), online doc (doxygen)
 git repository (INRIA gforge/gitlab)
 source and binary distributions
 cmake stuff (windows, mac os, linux) to configure and create XLiFE++ project
 non regressive tests (INRIA jenkins)

 website: https://uma.ensta-paristech.fr/soft/XLiFE++/
XLiFE++ team

Poems
- **Eric Lunéville** (FE computation)
- **Nicolas Kielbasiewicz** (Development environment and mesh tools)
- **Colin Chambeyron** (Iterative solvers)

Irmar
- **Yvon Lafranche** (Mesh tools, EigenSolvers)
- **Eric Darrigrand** (Fast Multipole methods)
- **Pierre Navaro** (Test environment)

thanks to **Marc Lenoir** (Scientific support)
**Man Ha Nguyen** (Eigen solvers and parallel computing)
**Nicolas Salles** (Singular integrals)

and **students** (often beta-testers)
Course plan

XLiFE++ basics (FEM)

- Manage geometry and mesh
- Define spaces and unknowns
- Define variational forms
- Manage essential conditions
- Compute algebraic representations
- Solve linear systems and eigen problems
- Export solutions to viewer
- Recent realizations

Advanced XLiFE++

- Defining complex geometries
- Dealing with vector problems
- Coupling FEM methods (multiple unknowns)
- Dealing with DTN
- Dealing with BEM
- Coupling FEM-BEM, FEM-IR
- Dealing with transient problem
- Dealing with non linear problem
Geometry in XLiFE++

**XLiFE++ Geometry** : a general class to handle geometrical object
- predefined geometries : Segment, Square, Cube, Disk, Sphere, Ellipse, Cylinder, Cone, ...
- combined geometry : union, intersection, hull, extrusion, crack, ...

Segment $S(\_\text{xmin}=0, \_\text{xmax}=1, \_\text{nnodes}=11)$;
Disk $D(\_\text{center}=\text{Point}(0.,0.), \_\text{radius}=1, \_\text{hsteps}=0.1, \_\text{domain_name}="\text{omega1}")$;
Sphere $Sp(\_\text{center}=\text{Point}(0.,0.,0.), \_\text{radius}=1, \_\text{hsteps}=0.1, \_\text{side_names}="\text{Sigma}")$;

main geometrical keywords :
- $\_\text{xmin}, \_\text{ xmax}, \_\text{ymin}, ..., \_\text{ v1}, \_\text{ v2}, ..., \_\text{ center}, \_\text{ origin}, \_\text{ radius}, ...$
- $\_\text{ nnodes}$ : numbers of nodes on sides or faces, may be a vector (Numbers), default : 2
- $\_\text{ hsteps}$ : steps on sides or faces, may be a vector of real (Reals), default : 0.1
- $\_\text{ domain_name}$ : the name of the geometry, default : "Omega"
- $\_\text{ side_names}$ : the name of the sides, may be a vector (Strings), default : no name

Disk $D(\_\text{center}=\text{Point}(0.,0.), \_\text{radius}=1, \_\text{nnodes}=0.1\_\text{side_names}=\text{Strings}("S1","","S2","S1"))$;

name all sides!
read the documentation for the designing rules
Geometry in XLiFE++

combined geometries

Point O(0.,0.); Real r=1.; Number n=10;

**Disk** D1(_center=O, _radius=r, _nnodes=n, _domain_name="omg1", _side_names="gamma");

**Disk** D2(_center=O, _radius=2*r, _nnodes=2*n, _domain_name="omega", _side_names="sigma");

**Geometry** C = D2 - D1;

more on geometry and complex designing in the "Advanced XLiFE++" session
**Mesh in XLiFE++**

**Mesh** is a class mainly handling
- the list of geometrical elements (segment, triangle, tetrahedron, ... of any order)
- the list of geometrical domains (plain domains, side domains)

and built from
- a XLiFE++ geometry and a mesher (_structured, _subdiv or _gmsh)
- a geometry file (.geo) or a mesh file (.msh, .ply, .mel)

_structured and _subdiv mesher are limited to few geometries!
_gmsh refers to the GMSH mesher that XLiFE++ calls by passing a geo file built from the geometry

Square \( S(_\text{origin}=\text{Point}(0.,0.),_\text{length}=1.,_\text{nnodes}=11); \)

Mesh \( \text{MS}(S,_\text{triangle},1,_{\text{structured}}); \)

Mesh \( \text{MSG}(S,_\text{triangle},1,_{\text{gmsh}}); \)

Disk \( D(_\text{center}=\text{Point}(0.,0.),_\text{radius}=1.,_\text{nnodes}=9); \)

Mesh \( \text{MD}(D,_\text{triangle},1,_{\text{subdiv}}); \)

Mesh \( \text{MDG}(D,_\text{triangle},1,_{\text{gmsh}}); \)

meshing surface with segment or volume with 2D element gives manifold meshes.
Domain in XLiFE++

**Domain** is a fundamental object in FE computations, it handles either
- a list of mesh elements (plain domain, all or part of)
- a list of side of mesh elements (boundary or interface)

```plaintext
Square S(_origin=Point(-1.,-1.), _length=2.,steps=0.1,  
    _domain_name="S", _side_names="Gamma");
Disk D(_center=Point(0.,0.), _radius=0.5, _hsteps=0.1,  
    _domain_name="D", _side_names="Sigma");
Mesh MDS(D+S, _triangle, 1, _gmsh);

Domain omg_D=MDS.domain("D");
Domain omg_S=MDS.domain("S");
Domain sigma = MDS.domain("Sigma");
Domain omega = merge(omg_D, omg_S, "Omega");
```

**few operations on Domain**
- setMaterialId(Number)
- setDomainId(Number)
- setNormalOrientation(OrientationType, [Domain])
  normal orientation: _towardsInfinite, _outwardsInfinite, _towardsDomain, _outwardsDomain
Spaces

**Space** object handles discretized space, say

\[ V = \left\{ \sum_{i=1}^{n} u_i \, w_i(x), u_i \in \mathbb{C} \text{ or } \mathbb{R} \right\} (w_i)_i \text{ basis functions} \]

- **FE space** based on “local” functions defined on the mesh elements of a domain

```plaintext
Square Sq(_origin=Point(0.,0.),_length=1.,_nnodes=11);
Mesh M(Sq,_triangle,1,_gmsh);
Domain omg = M.domain(“Omega”);
Space V(omg, Lagrange, 1);
```

- standard Lagrange on segment, triangle, quadrangle, tetrahedron hexahedron, prism at any order, pyramid (up to order 2)
- Crouzeix-Raviart on triangle and tetrahedron (order 1)
- Gauss-Lobatto Lagrange on segment, triangle, quadrangle, hexahedron, prism at any order
- Nedelec (Hrot) first and second family available on triangle and tetrahedron at any order
- Raviart-Thomas (Hdiv) first family available on triangle and tetrahedron at any order

- **Note that the basis functions may be vectors (Nedelec for instance)**
- **Use symbolic polynomials calculus to build shape functions at any order**
- **FE space mainly handles a list of FE elements and a list of FE d.o.f.**
- **By default, d.o.f. numbering is optimized to reduce matrix storage (can be disable)**
- **Space may be built from a side domain**
**Spectral space** based on “global” functions defined on a mesh domain from an explicit C++ function or some FE computed objects *(advanced)*

---

Real \(\text{cosny} \) (const Point& P, Parameters& pa = defaultParameters)  
{
    Real y = P(2);  
    Int n = pa("basis index") - 1;  
    get the index of function to compute  
    if(n==0) return 1;  
    else return sqrt(2.)*cos(n*pi_*y);  
}

**Square** \(\text{Sq} \)(_origin=Point(0.,0.),_length=1.,_nnodes=11,  
_side_names=Strings("y=0", "x=1", "y=1", "x=0");)

**Mesh** \(\text{M} \)(Sq,_triangle,1,_gmsh);  
**Domain** omn = M.domain("Omega"), gamma = M.domain("x=0");  
**Space** Sp(gamma, Function(cosny), 10, "cos(n*pi*y)");

Functions involved in DtN operator for waveguide

\[
\varphi_0(y) = 1  
\varphi_n(y) = \sqrt{2} \cos n\pi y  
\quad n > 0
\]

---

*When computation of \text{cosny} is required by the code, the basis index is passed to the function using the Parameters object \text{pa}***
**Unknowns**

- **Unknown** object is a **symbolic** element of a Space

```
Square Sq(_origin=Point(0.,0.),_length=1.,_nnodes=11);
Mesh M(Sq,_triangle,1,_gmsh);
Domain omg = M.domain("Omega");
Space V(omg, Lagrange, 1);
Unknown u( V, "u") ;
```

- **Several unknowns can be defined on the same space.**

- **By default an unknown is a scalar unknown, but it may be a vector unknown**

\[ u(x) = \sum_{i=1}^{n} u_i w_i(x), \quad u_i \in \mathbb{C}^d \text{ or } \mathbb{R}^d \quad w_i \text{ can not be a vector function!} \]

```
Unknown u3( V, "u", 3) ;
```

  (will be seen in advanced session)

- **TestFunction** object is also a **symbolic** element of a Space, but related to an unknown:

```
TestFunction v( u, "v") ;
```

- **Will play the role of the dual unknown in bilinear forms**
- **In theory all the computations should be consistent with the couple (u,v) but in practice XLiFE++ is tolerant**
Variational formulation of linear PDE involves integrals that are either bilinear or linear forms that XLiFE++ manages as objects: **BilinearForm** and **LinearForm** relating integrals on **Domain** of differential operators acting on **Unknown** and/or **TestFunction**:

- **OperatorOnUnknown** handles expression of type ([[optional, | or]):

\[
\text{opu}(u) = [\text{tr}([\text{dif}(F,K)|C])] \text{ op } \text{dif}(U) \text{ op } [\text{tr}([\text{dif}(F,K)|C])]
\]

**op** one of the product operator

<table>
<thead>
<tr>
<th>* standard product</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
</tr>
<tr>
<td>^ cross product</td>
</tr>
<tr>
<td>% contracted product</td>
</tr>
</tbody>
</table>

**dif** a **DifferentialOperator**

<table>
<thead>
<tr>
<th>id(u) or u</th>
<th>explicit</th>
<th>symbolic</th>
</tr>
</thead>
<tbody>
<tr>
<td>dx(u), dy(u), dz(u)</td>
<td>nx(u)</td>
<td>_n*u</td>
</tr>
<tr>
<td>div(u)</td>
<td>ndot(u)</td>
<td>_n.u</td>
</tr>
<tr>
<td>curl(u) or rot(u)</td>
<td>ncross(u)</td>
<td>_n^u</td>
</tr>
<tr>
<td>epsilon(u)</td>
<td>ncrossncross(u)</td>
<td>_n^(_n^u)</td>
</tr>
<tr>
<td>gradS(u) or nablaS(u)</td>
<td>ndotgrad(u)</td>
<td>_n.grad(u)</td>
</tr>
<tr>
<td>divS(u)</td>
<td>ncrossgrad(u)</td>
<td>_n^grad(u)</td>
</tr>
<tr>
<td>curlS(u) or rotS(u)</td>
<td>ndiv(u)</td>
<td>_n*div(u)</td>
</tr>
<tr>
<td>...</td>
<td>ncrosscurl(u)</td>
<td>_n^curl(u)</td>
</tr>
</tbody>
</table>

F a XLiFE++ **Function** or a C++ function, K a XLiFE++ **Kernel** or a C++ function, C a real/complex scalar/vector/matrix and U an **Unknown** or a **TestFunction**
Linear form

- **LinearForm**: main construction

  \[
  \int_D opu(v) \\
  \int_{D_x} \int_{D_y} opu(x, v(y))
  \]

  \[
  \int_\Omega v, \int_\Omega 3v \\
  \int_\Omega d \cdot \nabla v \\
  \int_\Omega f v
  \]

  **Space** \( V(\Omega, \text{Lagrange}, 1) \);
  **Unknown** \( u( V, "u") \); **TestFunction** \( v(u,"v") \);
  **LinearForm** \( l_1=\text{intg}(\Omega,v) \);
  **LinearForm** \( l_2=\text{intg}(\Omega, 3*v) \);
  **Vector** \( d(3,1.) \);
  **LinearForm** \( l_3=\text{intg}(\Omega, d \cdot \text{grad}(v)) \);
  **Function** \( f(\text{myfun}, \text{params}) \);
  **LinearForm** \( l_4=\text{intg}(\Omega, f*v) \);

  **LinearForm** may be combined:

  **LinearForm** \( l = 3*\text{intg}(\Omega,v) - \text{intg}(\Omega,f*v) + j*\text{intg}(\Omega, d \cdot \text{grad}(v)) \);

  **LinearForm** \( l = 3*\text{intg}(\sigma, v1) - \text{intg}(\Omega,f*v2) \) *different domains, different unknowns*
Bilinear form

**OperatorOnUnknowns**: $\text{opu}(u) \text{ op } \text{opu}(v)$

($\text{opu}$ OperatorOnUnknown, $\text{op}$ product operator)

- **BilinearForm**: main construction

  \[
  \text{intg}(\text{Domain, OperatorOnUnknowns}) \quad \int_D \text{opu}(u) \otimes \text{opu}(v) \quad \text{unknown at left}
  \]

  - **Space** $V(\omega, \text{Lagrange}, 1)$;
  - **Unknown** $u$ (V, “u”); **TestFunction** $v(u,"v")$;
  - **BilinearForm** $a1=\text{intg}(\omega, u*v)$;
  - **BilinearForm** $a2=\text{intg}(\omega, \text{grad}(u)|\text{grad}(v))$;
  - **Function** $f(\text{myfun}, \text{params});$ // vector function
  - **BilinearForm** $a3=\text{intg}(\omega, dx(u)*(f|\text{grad}(v))$;

  Combine **BilinearForm**: 

  \[
  \text{BilinearForm } a = \text{intg}(\omega, \text{grad}(u)|\text{grad}(v)) - k^2 \text{intg}(\omega, u*v);
  \]

  \[
  \int_D \nabla(u) \cdot \nabla(v) - k^2 \int_D u \, v
  \]

  \[
  \int_\Omega \nabla(u) \cdot \nabla(v) - k^2 \int_\Omega u \, v
  \]

  **intg(Domain, Domain, KernelOperatorOnUnknowns)**

  \[
  \int_{D_x} \int_{D_y} \text{opu}(u(y)) \otimes \text{opk}(x, y) \otimes \text{opu}(v(x)) \quad \text{(advanced session)}
  \]
User functions

LinearForm, BilinearForm and other XLiFE++ objects may involve functions. XLiFE++ can deal with

- standardized C++ functions for functions and kernels:

  \[
  \text{\texttt{type function\_name}}(\text{\texttt{const Point\& p, Parameters\& par=defaultParameters}}) \]

  \[
  \text{\texttt{type function\_name}}(\text{\texttt{const Point\& p, const Point\& q, Parameters\& par=defaultParameters}}) \]

  \textit{type} is one of \texttt{Real}, \texttt{Complex}, \texttt{RealVector}, \texttt{ComplexVector}, \texttt{RealMatrix}, \texttt{ComplexMatrix}

  \textit{input arguments have to follow the previous syntaxes} 🚨

\[
\text{Real } f(\text{\texttt{const Point\& p, Parameters\& par=defaultParameters}}) \{\text{return p(1)*p(2);}\}
\]

\[
\text{Real } \text{lap}(\text{\texttt{const Point\& p, const Point\& q, Parameters\& par=defaultParameters}}) \{\text{return over4pi_*norm2(p-q);}\}
\]

- \textit{Function} objects or \textit{Kernel} objects handling a standardized C++ function and a \textit{Parameters} object (list of Parameter)

\[
\text{Complex } \text{helm}(\text{\texttt{const Point\& p, const Point\& q, Parameters\& par=defaultParameters}}) \{\text{Real } k=\text{par}("k"), r = \text{norm2(p-q)}; // get parameter value return over4pi_*exp(i_*k*r)/r;}\}
\]

\[
\text{Parameters } \text{mypar}(5, "k"); //set parameters
\text{Kernel } H(\text{helm, mypar}); // associate parameters to Kernel object H
\]

\text{to do in main}
Quadrature rule

- Almost all integrals are computed using quadrature methods
- By default, XLiFE++ uses the best quadrature available for a given shape $s$ and a given polynomial order $p$
- the order $p$ is chosen regarding the (bi)linear form:
  $$ p = \text{degree}(u) - \text{diforder}(\text{opu}(u)) + \text{degree}(v) - \text{diforder}(\text{opu}(v)) $$
  $$ + \max(\text{degree}(u), \text{degree}(v)) \text{ if there is a function/kernel in form} $$

<table>
<thead>
<tr>
<th>shape</th>
<th>Gauss-Legendre</th>
<th>Gauss-Lobatto</th>
<th>Grundmann-Moller</th>
<th>symmetrical Gauss</th>
</tr>
</thead>
<tbody>
<tr>
<td>segment</td>
<td>any odd degree</td>
<td>any odd degree</td>
<td></td>
<td></td>
</tr>
<tr>
<td>quadrangle</td>
<td>any odd degree</td>
<td>any odd degree</td>
<td></td>
<td>odd degree up to 21</td>
</tr>
<tr>
<td>triangle</td>
<td>any odd degree</td>
<td></td>
<td>any odd degree</td>
<td>degree up to 10</td>
</tr>
<tr>
<td>hexahedron</td>
<td>any odd degree</td>
<td>any odd degree</td>
<td></td>
<td>degree up to 11</td>
</tr>
<tr>
<td>tetrahedron</td>
<td>any odd degree</td>
<td></td>
<td>any odd degree</td>
<td>degree up to 10</td>
</tr>
<tr>
<td>prism</td>
<td>any odd degree</td>
<td>any odd degree</td>
<td></td>
<td>degree up to 10</td>
</tr>
<tr>
<td>pyramid</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

general quadrature rules

Change the quadrature rule in a **LinearForm** or a **BilinearForm**

```plaintext
BilinearForm a = intg(omega, f*grad(u)|grad(v), Gauss_Legendre, 5);
```

_efaultRule, Gauss_Legendre, Gauss_Lobatto, Grundmann_Moller, symmetrical_Gauss
Essential condition

Essential conditions are **constraints in space**, main types:

- **Dirichlet condition**: \( u = 0 \) on \( \Gamma \)
- **Transmission condition**: \( [u] = g \) on \( \Gamma \)
- **Periodicity condition**: \( u|_{\Sigma^+} = u|_{\Sigma^-} \) for \( \Sigma^- \) and \( \Sigma^+ \)
- **Average condition**: \( \int_{\Omega^+} u = 0 \)

\[
\begin{array}{c|c}
\Omega_- & \Omega_+ \\
\hline
\Gamma & \Sigma_+ \\
\end{array}
\]

\( \text{lcop} = \text{val or fun on } D \) (one domain)

\( \text{lcop1 on } D_1 + \text{lcop2 on } D_2 = \text{val or fun} \) (two domains)

\( \text{lcop}, \text{lcop1}, \text{lcop2} \) are linear combinations of operators on unknown

\( D, D_1, D_2 \) are domains where act essential conditions

Vector<Real> **mapPM** (const Point& P, Parameters& pa = defaultParameters)

\{ Vector<Real> Q = P; Q.y = 1; return Q; \}  

**map Sigma+ -> Sigma-**

**EssentialCondition** ecd = (u|sigmaM = 1);  
**Dirichlet condition**

**EssentialCondition** ect = (uM|gamma) - (uP|gamma) = f;  
**Transmission condition**

**defineMap**(sigmaP, sigmaM, mapPM);

**EssentialCondition** ecp = (u|sigmaP) - (u|sigmaM) = 0;  
**Periodic condition**

**XLiFE++ basics**
Essential conditions

Collect **EssentialCondition** in a list: **EssentialConditions**

- **EssentialConditions** $ecs = (u | \gamma_M = 0) \& (u | \gamma_P = 0) \& ((u | \sigma_P) - (u | \sigma_M) = 0)$;
- **EssentialConditions** $ecs = (uM | \sigma_M = 1) \& (uP | \sigma_P = 1) \& ((uM | \gamma) - (uP | \gamma) = 0)$;

⚠️ Conditions may be not consistent

General process to deal with many conditions (very powerful)

- Collect essential conditions in a constraints system: $CU = G$
- Reduce by QR algorithm to a minimal system: $UE + DU_R = S$
- Reduce problem to solve $AU = F$ \quad $(A_R - A_E D)U_R = F - A_E S$

 $\text{(for Dirichlet condition} \quad u = 0 : D = 0 \rightarrow A_R U_R = F)$

**Detects and eliminates redundant or conflicting constraints** 😊

Average condition

$\int \sum u = 0$

**EssentialCondition** $ec = (\text{intg(}\sigma, u) = 0)$;

may be memory expansive and time expansive when global constraints

XLiFE++ basics

22
Algebraic representation

Space \( V(\omega, \text{Lagrange}, 1); \)
Unknown \( u(\ V, \ "u") \); TestFunction \( v(u, "v") \);
BilinearForm \( a=\text{intg}(\omega, \text{grad}(u)|\text{grad}(v)) \);
LinearForm \( l=\text{intg}(\omega, f*v) \);
EssentialCondition \( ec=(u|\gammaamma = 0) \);

Move to algebraic representation:

**TermVector** : vector representing either

- an element of space related to an unknown:
  \[ (u_i)_{i=1,n} \]
  \[ u = \sum_{i=1,n} u_i w_i \]

- a linear form on space (dual identification):
  \[ (l(w_i))_{i=1,n} \]

- \( u_i \) may be real or complex (managed by XLiFE++)
- **Danger** when vector unknown, \( u_i \) is a vector (vector of vectors)
- **Danger** two vectors linked to different unknowns are considered different even if they are related to the same space!

**TermVector** \( B(l, "B") \);

**TermVector** \( U1(u, \omega, 1.) \);
**TermVector** \( F(u, \omega, f) \);

\[ sYmbolic \ \text{representation} \]
\[ a(u,v) = \int_{\Omega} \nabla(u).\nabla(v) \]
\[ l(v) = \int_{\Omega} f v \]
\[ u = 0 \ \text{on } \Gamma \]

compute
\[ l(w_i) = \int_{\Omega} f w_i \]

\[ (1)_{i=1,n} \rightarrow u = \sum_{i=1,n} w_i \]
\[ (f(M_i))_{i=1,n} \rightarrow \Pi f = \sum_{i=1,n} f(M_i)w_i \ \text{(Lagrange)} \]
**TermMatrix**: matrix representing a bilinear form $a$ (related to Unknown and TestFunction)

\[
A_{ij} = \left( a(w_j, \tau_i) \right)_{i=1,m; j=1,n}
\]

- not necessary the same basis, the same space
- Unknown related to matrix columns, TestFunction to matrix rows
- storage managed by XLiFE++ (sparse storage for FEM, dense for BEM)
- when vector unknown or testfunction, matrix of matrices representation

**BilinearForm** $a=\int g(\omega, \text{grad}(u)\mid\text{grad}(v))$;  
**TermMatrix** $A(a, "A")$;  
**compute** \[ a(w_j, w_i) = \int_\Omega \nabla w_j \cdot \nabla w_i \]

Essential conditions are usually attached to spaces. To avoid too many spaces, XLiFE++ attached essential conditions to **TermMatrix**:

**BilinearForm** $a=\int g(\omega, \text{grad}(u)\mid\text{grad}(v))$;  
**EssentialCondition** $ec=(u\mid\gamma = 0)$;  
**TermMatrix** $A(a, ec, "A")$;  
**compute** \[ a(w_j, w_i) = \int_\Omega \nabla w_j \cdot \nabla w_i \]

\[ \forall i, j \quad w_i|_\Sigma = 0 \quad w_j|_\Sigma = 0 \]

In practice, XLiFE++ uses a virtual reduction of essential conditions:

\[ \forall i, j \quad w_i|_\Sigma = 0 \quad w_j|_\Sigma = 0 \quad A_{ij} = 0, \quad i \neq j, \quad A_{ii} = \alpha \]
When building **TermVector** or **TermMatrix**, some options are available, to disable computation, to force symmetry property or to change the storage:

- `compute`, `notCompute`,
- `nonSymmetricMatrix`, `symmetricMatrix`, `selfAdjointMatrix`, `skewSymmetricMatrix`, `skewAdjointMatrix`,
- `csRowStorage`, `csColStorage`, `csDualStorage`, `csSymStorage`
- `denseRowStorage`, `denseColStorage`, `denseDualStorage`
- `skylineSymStorage`, `skylineDualStorage`

Many operations are available on **TermVector** and **TermMatrix**

- **linear combination**

  ```
  TermVector U1(u, omega, f), U2=..., U3 =...; some TermVector's
  TermVector R = 2*U1 - _i*U2 + U3/5;
  ```

  *Be care when mixing unknowns*

- **non linear computations (on components)**, only **TermVector**

  ```
  TermVector U (u, omega, f);
  TermVector U2 = U*U, U3 = U^3, SU=sin(U);
  ```

  \[ U = (1,2,3,4,....) \]
  \[ U2 = (1,4,9,16,....) \]

- **evaluate a TermVector at a point (in domain!)**

  ```
  Point x(0.,1.);
  Real v = U(x, v);
  ```

  *the coefficient type has to be consistent*
Algebraic representation

- restriction of a **TermVector** on a domain
  - \( \text{TermVector} \ U(u, \omega, f); \)
  - \( \text{TermVector} \ Us = U|\sigma; \)

- other operations on **TermVector** and **TermMatrix**
  - abs, real, imag, conj, dotRC, innerProduct, hermiteianProduct
  - operator | (inner or hermitian product)
  - norm1, norm2, norminfty, norm (\( \text{norm}_2 \))

```
**TermVector** Uex(u, omega, uexact);
**TermVector** E = U-Uex;
**TermMatrix** M(intg(omega, u*v));
Real er_L2 = sqrt( (M*E|E) );
```

Example: compute a L2 error

*see user documentation for an exhaustive list of operations*
Solve linear systems

To solve linear systems $AX=B$ where $A$ is a TermMatrix and $B$ a TermVector, XLiFE++ proposes the following methods:

- **Direct methods**
  - LU, LDLt, LDL* factorization for skyline storage (sparse move to skyline)
  - Interface to Umfpack factorization for sparse storage
  - Gauss reduction with permutation for dense storage
  - Interface to Magma (GPU) for dense storage

General call

```
TermVector U = directSolve(A, B);
```

*By default, the TermMatrix is overwritten, to keep original TermMatrix*

```
TermVector U = directSolve(A, B, _keep);
```

To get the factorized matrix, do

```
TermMatrix Af;
factorize(A, Af);
TermVector U = factSolve(Af, B);
```

*Factorization type may be imposed.*
Solve linear systems

- Iterative methods proposed by XLiFE++
  - Conjugate Gradient methods (CG, CGS, BiCG, BiCGStab)
  - Generalized Minimal RESidual (GMRes)
  - Quasi Minimal Residual (QMR)

To call an iterative method (gmres for instance):

```plaintext
TermVector U = gmresSolve(A,B);
```

or

```plaintext
TermVector U = iterativeSolve(A, B, _solver=_gmres);
```

Optionnaly, an initial guess TermVector and a Preconditioner may be passed to iterative methods and some parameters can be set

- `_tolerance` : tolerance on the residue (default = 1e-6)
- `_maxIt` : maximum number of iterations (default = 10x the number of unknowns)
- `_verbose` : verbose level (default = 0)
- `_name` : the name of the TermVector solution computes by the routines (default = “U”)
- `_krylovDim` : dimension of the Krylov space used in gmresSolve (default = 20)
Compute eigenvalues

XLiFE++ can solve for any matrix

the standard eigenvalue problems \( AX = \lambda X \)
the generalized eigenvalue problems \( AX = \lambda BX \)

To compute eigenelements of TermMatrix's, the simplest way is to use the front ends:

```
EigenElements Es = eigenSolve(A);
EigenElements Egs = eigenSolve(A,B);
```

**EigenElements** object collects the computed eigenvalues and eigenvectors

```
EigenElements Es = eigenSolve(A);
Complex lambda1 = Es.value(1);  // first eigenvalue
TermVector V1 = Es.vector(1);    // first eigenvector
```

eigenSolve(...) uses Arpack functions if available else internal eigensolvers are used
It is possible to tune the eigensolver by using key values:

- _nev : number of eigen elements to be computed (default = 10)
- _which : part of the spectrum to scan (default = "LM")
- _sigma : shift value used to scan a portion of the spectrum around (default = 0)
- _mode : method used (default = _krylovSchur)
- _tolerance : precision of the computation (default = 1e-6)
- _maxIt : maximum number of iterations (default = 10000)
- _sort : sort criterion (default = _incr_module)
Print and export results

- all user object may be printed either using a C++ ostream or a print() member function

```cpp
TermVector U ...;
theCout<<U<<eol;
U.print(theCout);
```

- save a `Mesh` to a file

```cpp
Mesh M ...;
M.saveToFile("mymesh", _msh);
```

- save a `TermVector` to a file

```cpp
TermVector U ...; V ...;
saveToFile("myU", U, _msh);
saveToFile("myU", U, V, _vtu);
```

- save a `TermMatrix` to a file

```cpp
TermMatrix A ...;
saveToFile("myU", U, _msh);
saveToFile("myU", U, V, _vtu);
```

- _msh : GMSH format
- _vtk : PARAVIEW format
- _mel : MELINA format

- _msh : GMSH format
- _vtu : PARAVIEW format
- _xyzv : coordinates and values
- _matlab : MATLAB script

- _dense : dense raw format
- _coo : coordinates ijv (Matlab sparse)
Main execution controls

- language of messages (French, English, German, Spanish) at initialization:
  ```
  int main() {
      init(_lang=fr); ...
  }
  ```

- control the verbosity (0:nothing, 1:a little, when increasing more and more)
  ```
  verboseLevel(0);
  ```

- control the number of threads (omp)
  ```
  Number n = numberOfThreads(); // give the number of threads
  numberOfThreads(4); // set the number of threads
  ```

- output to the terminal and to the file print.txt, use theCout stream
  ```
  TermVector X ...
  theCout << X << eol;
  output only to the file print.txt, use thePrintStream stream
  ```

- measure the execution time
  ```
  cpuTime("..."); // cpu time from last time call
  elapsedTime("..."); // elapsed time from last time call
  ```
Some recent realizations at POEMS

**Half space matching method**
Y. TJANDRAWIDJAJA – A.S BONNET-BENDHIA - S. FLISS

**Coupling BEM and Physical optics**
E. LUNEVILLE – N. SALLES

**Transparent obstacle in waveguide**
A.S BONNET BENDHIA – A. BERA

**Identifying obstacles from boundary data using quasi-reversibility and level set**
L. BOURGEOIS- D. PONOMAREV

XLiFE++ basics