XLiFE++: an eXtended Library of Finite Elements in C++

Course 1: XLiFE++' basics

Nicolas KIELBASIEWICZ, Yvon LAFRANCHE

Unité de Mathématiques Appliquées,
ENSTA - Paristech

June 14, 2016

From a file

1. GMSH (.geo or .msh)
2. MELINA (.mel)
3. PLY: Polygon File Format (.ply)
4. PARAVIEW (.vtk, .vtu) (soon)
5. MEDIT (.mesh) (soon)

```
Mesh m("mesh.msh","My mesh from mesh.msh", msh);
```

From a file

1. GMSH (.geo or .msh)
2. MELINA (.mel)
3. PLY: Polygon File Format (.ply)
4. PARAVIEW (.vtk, .vtu) (soon)
5. MEDIT (.mesh) (soon)

```
Mesh m("mesh.msh","My mesh from mesh.msh", msh);
```

From a geometry

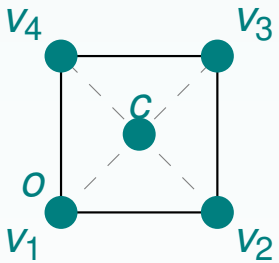
1. Internal structured mesh generator
2. Internal unstructured mesh generator using subdivision algorithm
3. External unstructured mesh generator using GMSH

```
Mesh m(Cuboid(_origin=Point(0.,0.,0.), _xlength=2., _ylength=3., _zlength=1.,
       _nnodes=Numbers(21, 31, 11), _tetrahedron, 2, _gmsh, "My mesh from a cuboid");
```

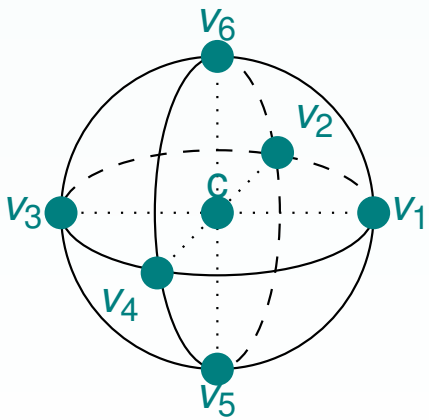
geometrical parameters: for instance, vertices of a segment, center and length of a square, center and radius of a ball, ...

```
// segment [0,1]
Segment s1(_v1=Point(0.,0.,0.), _v2=Point(1.,0.,0.), ...);
Segment s2(_v1=Point(0.,0.), _v2=Point(1.,0.), ...);
Segment s3(_xmin=0., _xmax=1., ...);
```


```
// square [0,1]^2
Square sq1(_v1=Point(0.,0.),
      _v2=Point(1.,0.),_v4=Point(0.,1.), ...);
Square sq2(_center=Point(0.5,0.5), _length=1., ...);
Square sq3(_origin=Point(0.,0.), _length=1., ...);
```



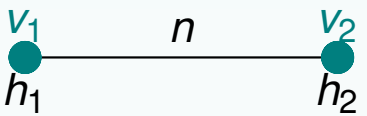

```
// unit sphere
Ball b1(_center=Point(0.,0.,0.), _radius=1., ...);
Ball b2(_center=Point(0.,0.,0.), _v1=Point(1.,0.,0.),
      _v2=Point(0.,1.,0.), _v6=Point(0.,0.,1.), ...);
```



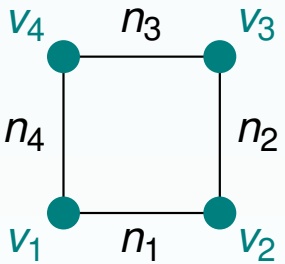
geometrical parameters: for instance, vertices of a segment, center and length of a square, center and radius of a ball, ...

mesh parameters: number of nodes on edges, local step on vertices

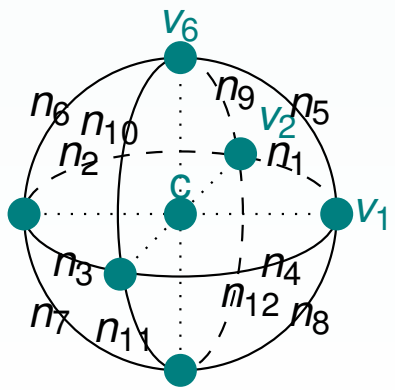
```
// segment [0,1]
Segment s1(..., _hsteps=0.1, ...);
Segment s2(..., _hsteps=Reals(0.1, 0.2), ...);
Segment s3(..., _nnodes=11, ...);
```




```
// square [0,1]^2
Square sq1(..., _hsteps=Reals(0.1, 0.2, 0.3, 0.4), ...);
Square sq2(..., _nnodes=Numbers(1, 2, 3, 4, ...);
```




```
// unit sphere
Ball b1(..., _hsteps=Reals(0.1, 0.2, 0.3, 0.4, 0.5, 0.6), ...);
Ball b2(..., _nnodes=Numbers(10, 20, 30), ...);
Ball b3(..., _nnodes=Numbers(1, 2, 3, 4, 5, 6, 7, 8, 9, 10,
    11, 12), ...);
```

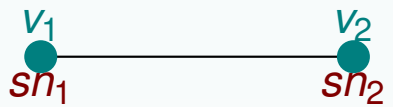


geometrical parameters: for instance, vertices of a segment, center and length of a square, center and radius of a ball, . . .

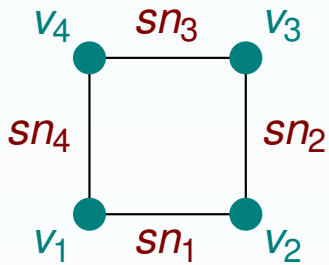
mesh parameters: number of nodes on edges, local step on vertices

domain parameters: name of the main geometry and/or its sides

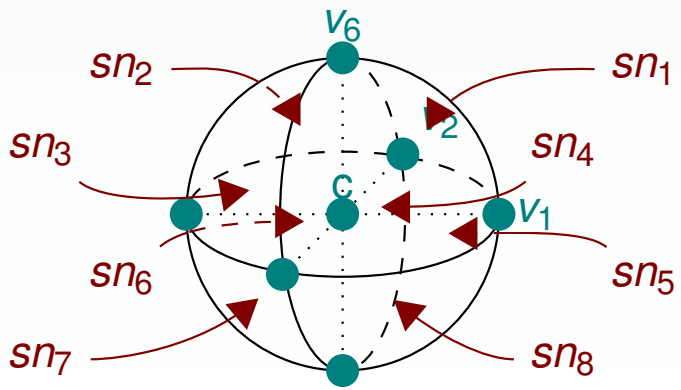
```
// segment [0,1]
Segment s1(..., _domain_name="Omega", _side_names="Gamma");
Segment s2(..., _domain_name="Omega", _side_names=Strings("sn1", "sn2"));
```




```
// square [0,1]^2
Square sq1(..., _domain_name="Omega",
      _side_names=Strings("sn1", "sn2", "sn3", "sn4"));
```

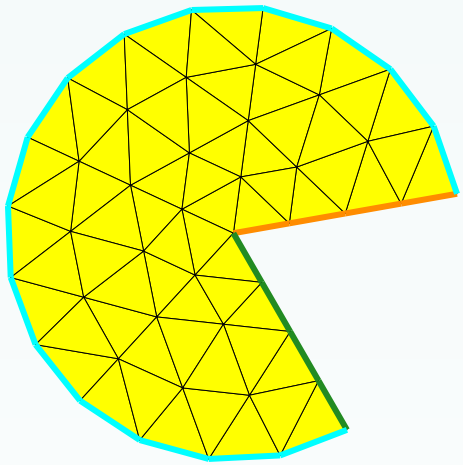



```
// unit sphere
Ball b1(..., _domain_name="Omega",
      _side_names=Strings("sn1", "sn2", "sn3", "sn4",
      "sn5", "sn6", "sn7", "sn8"));
```



How to define advanced geometrical configurations ? with subdivision mesh gen.

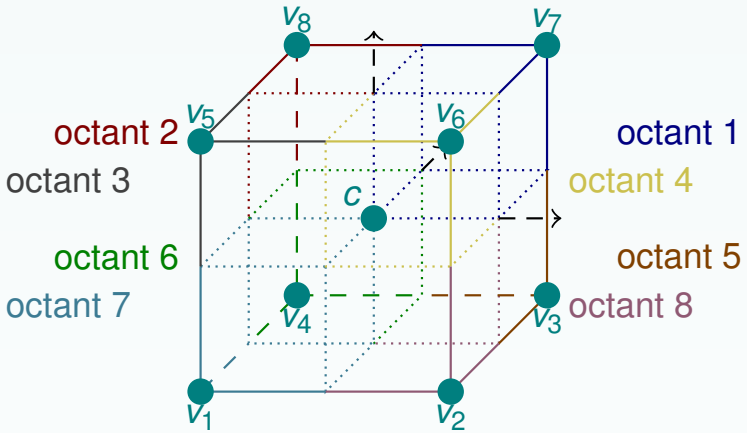
Disk **_angle1**, **_angle2** to set angular sector in degrees



How to define advanced geometrical configurations ? with subdivision mesh gen.

Disk **_angle1**, **_angle2** to set angular sector in degrees

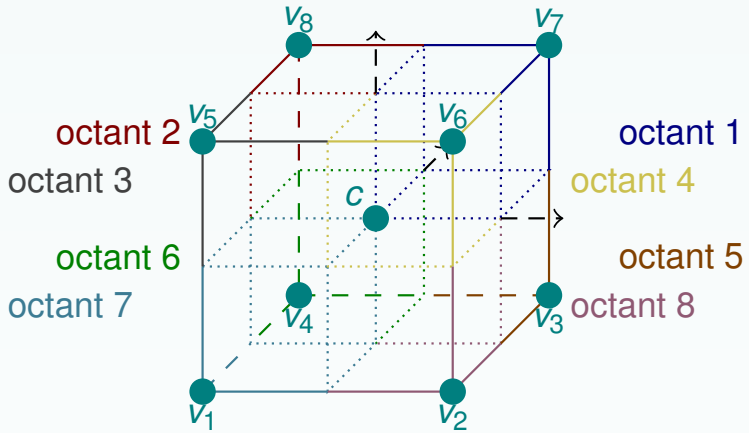
Cube, Ball **_nboctants** to set the number of octants to draw

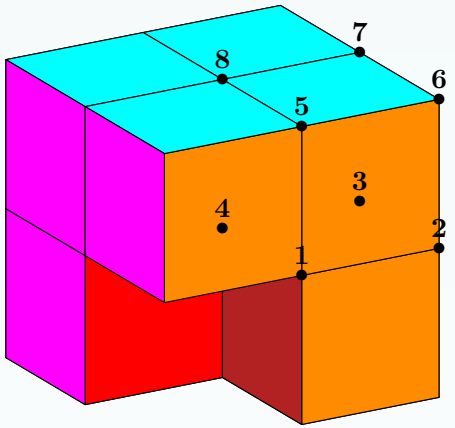


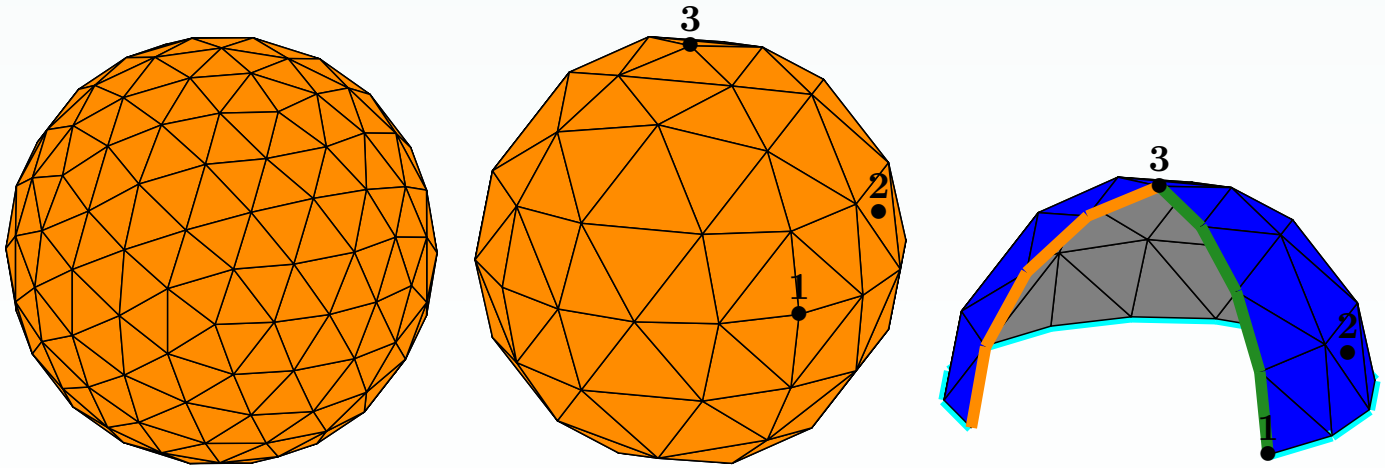
How to define advanced geometrical configurations ? with subdivision mesh gen.

Disk **_angle1**, **_angle2** to set angular sector in degrees

Cube, Ball **_nboctants** to set the number of octants to draw







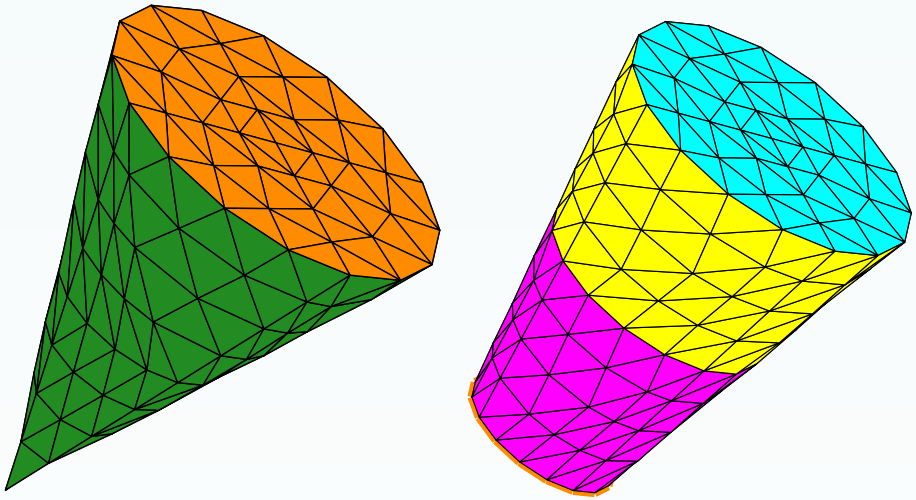
How to define advanced geometrical configurations ? with subdivision mesh gen.

Disk **_angle1**, **_angle2** to set angular sector in degrees

Cube, Ball **_nboctants** to set the number of octants to draw

RevTrunk, . . . ,

1. **_nbsubdomains** to set the number of slices of the geometry you want



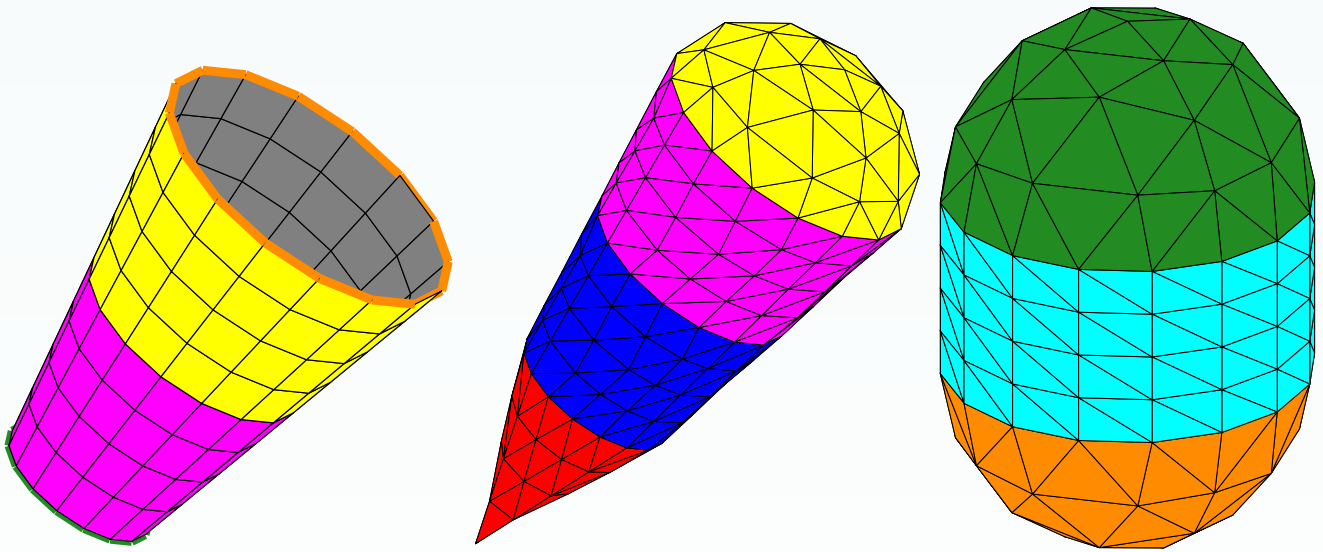
How to define advanced geometrical configurations ? with subdivision mesh gen.

Disk **_angle1**, **_angle2** to set angular sector in degrees

Cube, Ball **_nboctants** to set the number of octants to draw

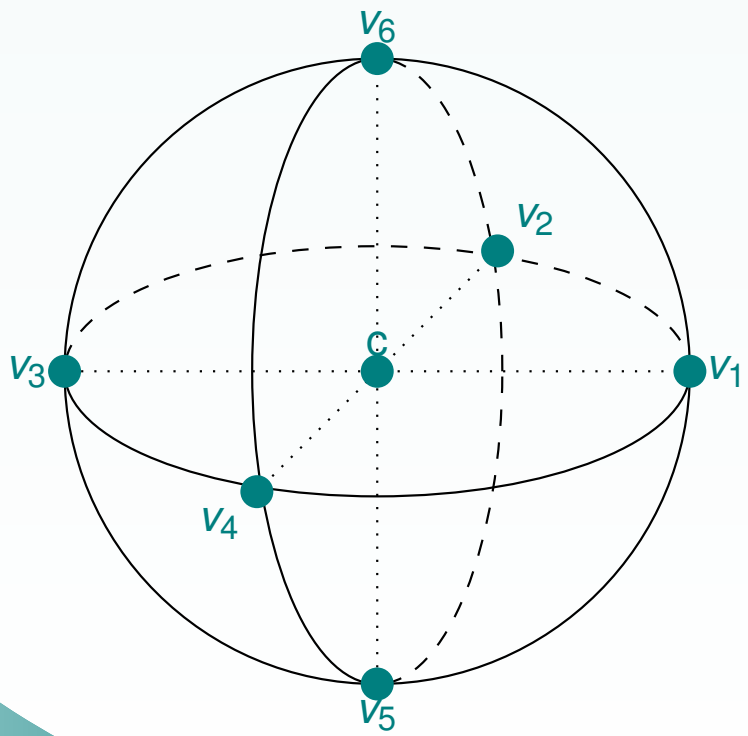
RevTrunk, . . . ,

1. **_nbsubdomains** to set the number of slices of the geometry you want
2. **_end1_shape**, **_end2_shape**, **_end_shape** to set the extensions type you want, and **_end1_distance**, **_end2_distance**, **_end_distance** to set the heights of extensions



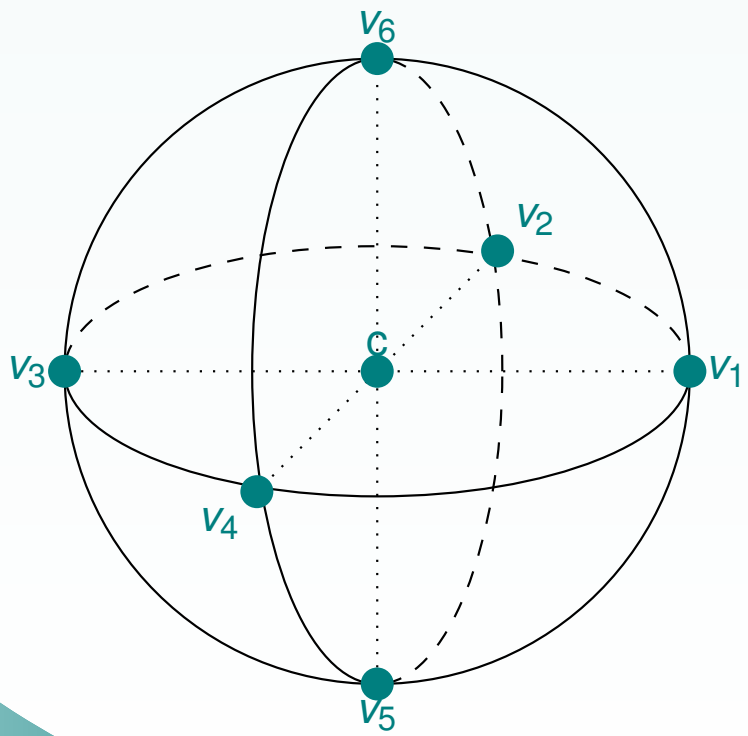
GMSH (http://gmsh.info)

- a free 1D-2D-3D finite element mesh software with build-in pre and post-processing
- works with a GUI or by scripting (.geo files)



GMSH (http://gmsh.info)

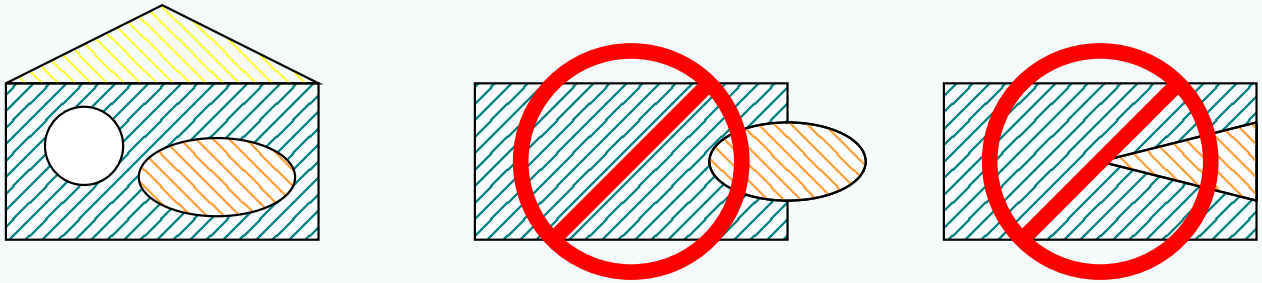
- a free 1D-2D-3D finite element mesh software with build-in pre and post-processing
- works with a GUI or by scripting (.geo files)



< 5 instructions in XLiFE++

> 37 instructions in GMSH

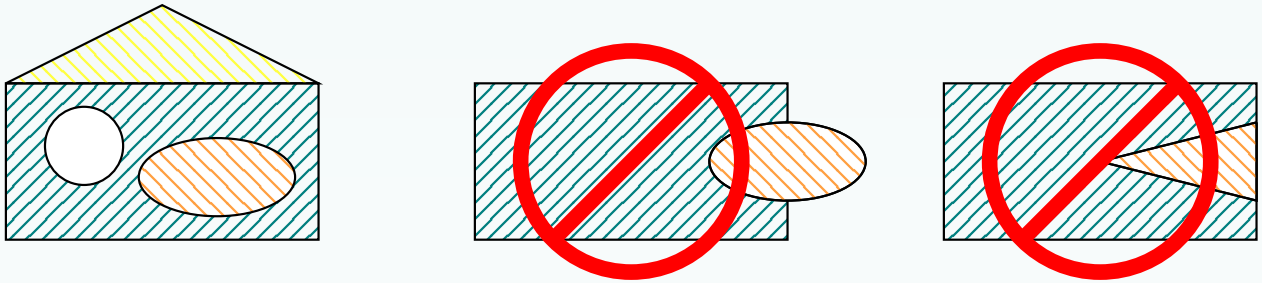
1. How to manage inclusion of geometries ? "composite geometries"



```
Rectangle r(..., _domain_name="Omega1");
Disk d(...);
Ellipse e(..., _domain_name="Omega2");
Triangle t(..., _domain_name="Omega3");
Geometry g=r-d+e+t;
```

The + operator (or +=): inclusions are meshed only if a domain name is given
Common borders are detected
The - operator (or -=): inclusions are not meshed.

1. How to manage inclusion of geometries ? "composite geometries"

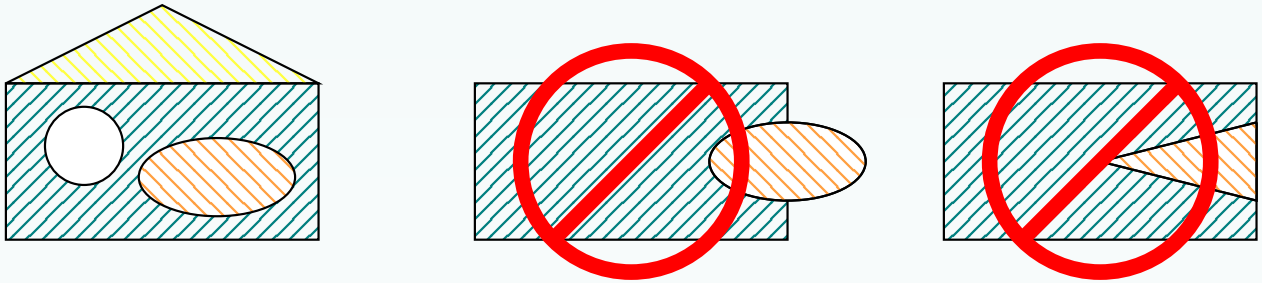


```
Rectangle r(..., _domain_name="Omega1");
Disk d(...);
Ellipse e(..., _domain_name="Omega2");
Triangle t(..., _domain_name="Omega3");
Geometry g=r-d+e+t;
```

The + operator (or +=): inclusions are meshed only if a domain name is given
Common borders are detected
The - operator (or -=): inclusions are not meshed.

- Inclusion between geometries is automatically detected, but can fail in some cases. It can be forced

1. How to manage inclusion of geometries ? "composite geometries"

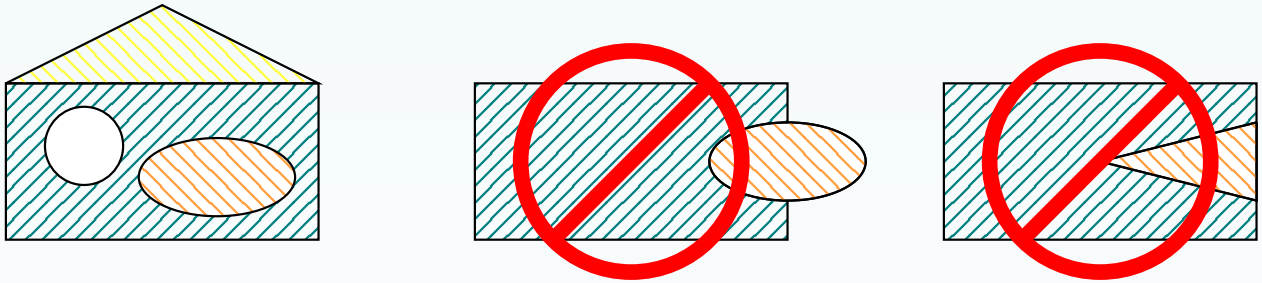


```
Rectangle r(..., _domain_name="Omega1");
Disk d(...);
Ellipse e(..., _domain_name="Omega2");
Triangle t(..., _domain_name="Omega3");
Geometry g=r-d+e+t;
```

The + operator (or +=): inclusions are meshed only if a domain name is given
Common borders are detected
The - operator (or -=): inclusions are not meshed.

- Inclusion between geometries is automatically detected, but can fail in some cases. It can be forced
- Inclusion is not checked when using - operator. It is automatically forced

1. How to manage inclusion of geometries ? "composite geometries"

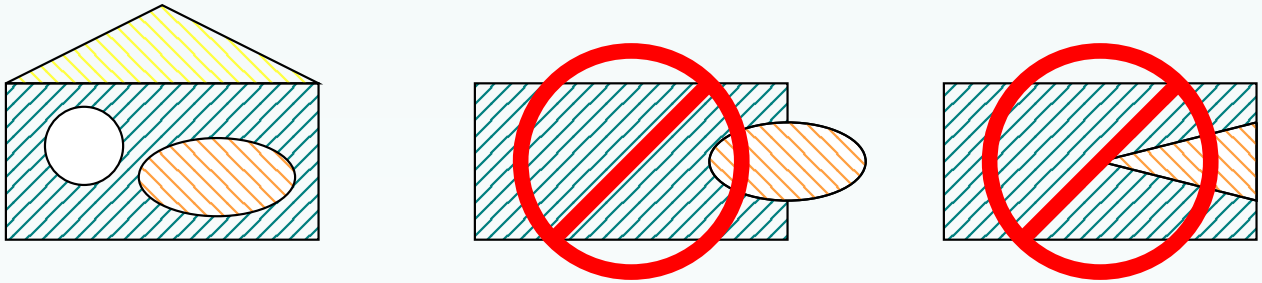


```
Rectangle r(..., _domain_name="Omega1");
Disk d(...);
Ellipse e(..., _domain_name="Omega2");
Triangle t(..., _domain_name="Omega3");
Geometry g=r-d+e+t;
```

The + operator (or +=): inclusions are meshed only if a domain name is given
Common borders are detected
The - operator (or -=): inclusions are not meshed.

- Inclusion between geometries is automatically detected, but can fail in some cases. It can be forced
- Inclusion is not checked when using - operator. It is automatically forced
- Definition of g can be written in any order, on condition that containers of holes are written before their holes (meaning - operator)

1. How to manage inclusion of geometries ? "composite geometries"



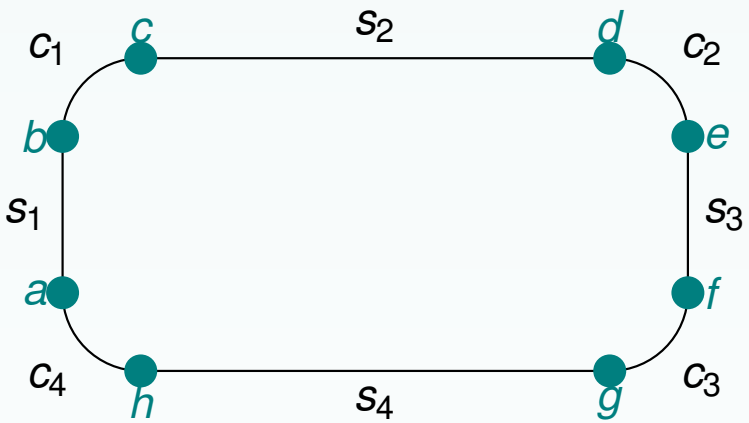
```
Rectangle r(..., _domain_name="Omega1");
Disk d(...);
Ellipse e(..., _domain_name="Omega2");
Triangle t(..., _domain_name="Omega3");
Geometry g=r-d+e+t;
```

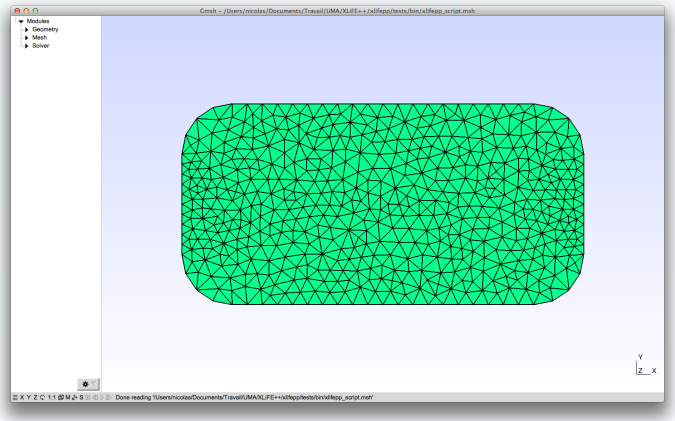
The + operator (or +=): inclusions are meshed only if a domain name is given
Common borders are detected
The - operator (or -=): inclusions are not meshed.

- Inclusion between geometries is automatically detected, but can fail in some cases. It can be forced
- Inclusion is not checked when using - operator. It is automatically forced
- Definition of g can be written in any order, on condition that containers of holes are written before their holes (meaning - operator)
- Using parenthesis may accelerate management of inclusions

How to define advanced geometrical configurations ? with gmsh mesh gen.

1. How to manage inclusion of geometries ? "composite geometries"
2. How to define geometries from its borders ?

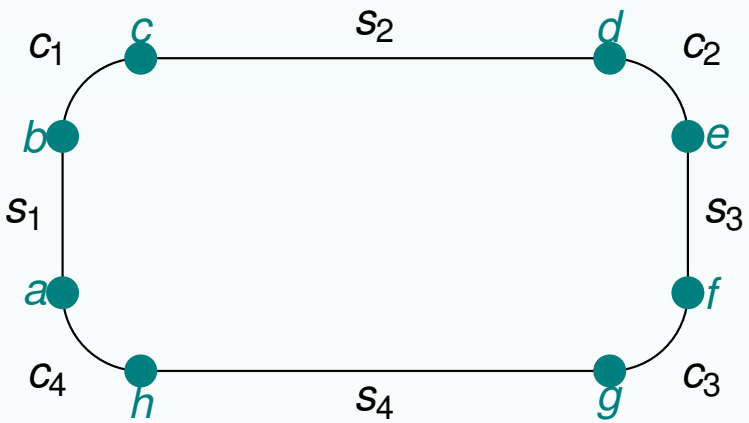


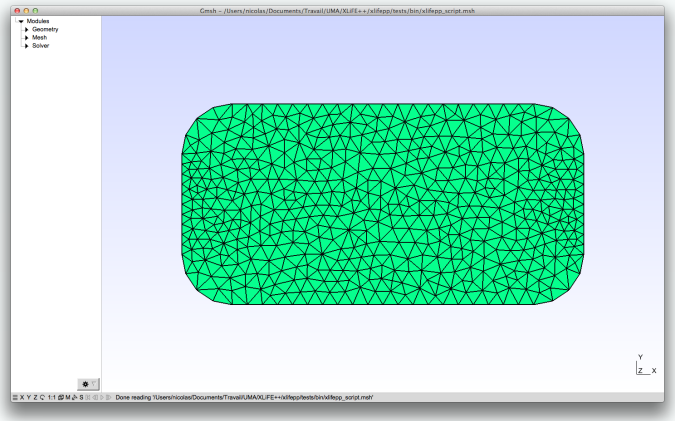


```
Point a(...), b(...), c(...), d(...), e(...), f(...), g(...), h(...);
Segment s1(...), s2(...), s3(...), s4(...);
CircArc c1(...), c2(...), c3(...), c4(...);
Mesh mesh2dP1Loop(planeSurfaceFrom(s1+c1+s2+c2+s3+c3+s4+c4), _triangle, 1, _gmsh);
```

How to define advanced geometrical configurations ? with gmsh mesh gen.

1. How to manage inclusion of geometries ? "composite geometries"
2. How to define geometries from its borders ?



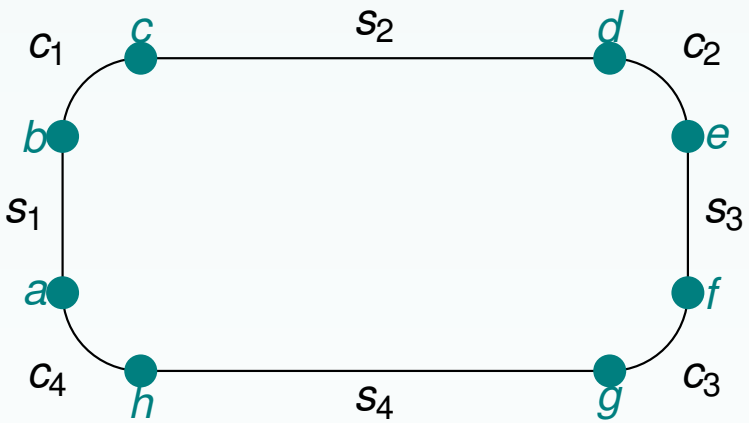


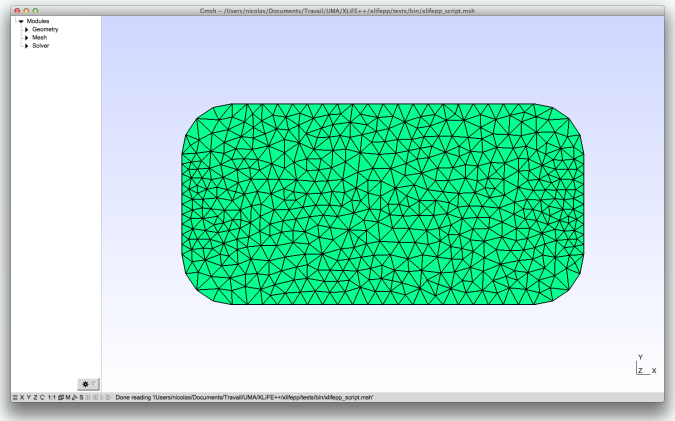
```
Point a(...), b(...), c(...), d(...), e(...), f(...), g(...), h(...);
Segment s1(...), s2(...), s3(...), s4(...);
CircArc c1(...), c2(...), c3(...), c4(...);
Mesh mesh2dP1Loop(planeSurfaceFrom(s1+c1+s2+c2+s3+c3+s4+c4), _triangle, 1, _gmsh);
```

- Geometries must share the same orientation. When needed, you can reverse it by using the $\sim$ operator

How to define advanced geometrical configurations ? with gmsh mesh gen.

1. How to manage inclusion of geometries ? "composite geometries"
2. How to define geometries from its borders ?



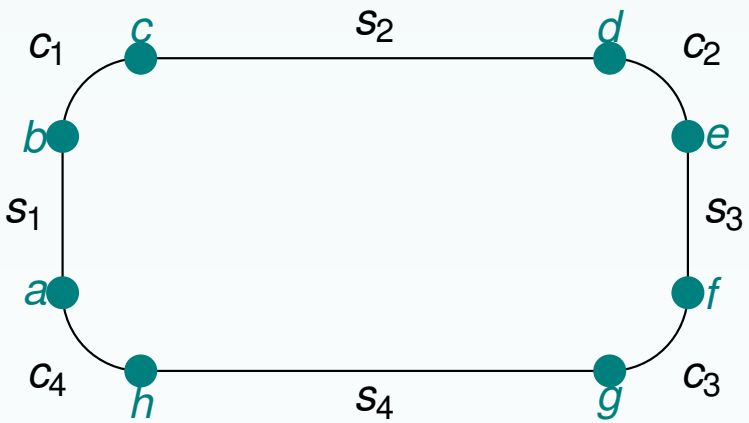


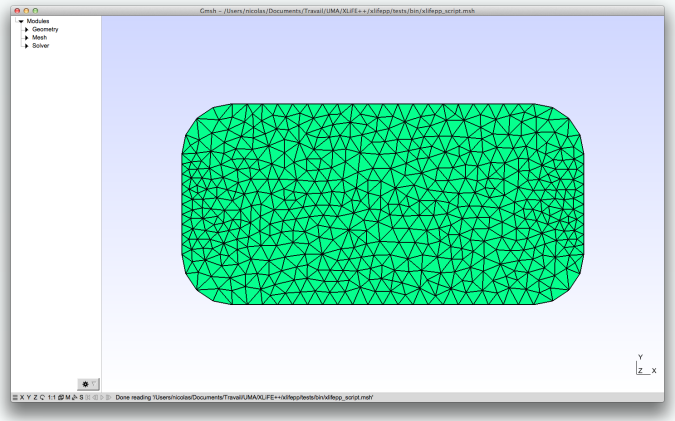
```
Point a(...), b(...), c(...), d(...), e(...), f(...), g(...), h(...);
Segment s1(...), s2(...), s3(...), s4(...);
CircArc c1(...), c2(...), c3(...), c4(...);
Mesh mesh2dP1Loop(planeSurfaceFrom(s1+c1+s2+c2+s3+c3+s4+c4), _triangle, 1, _gmsh);
```

- Geometries must share the same orientation. When needed, you can reverse it by using the $\sim$ operator
- Sum s1+c1+s2+c2+s3+c3+s4+c4 can be written in any order

How to define advanced geometrical configurations ? with gmsh mesh gen.

1. How to manage inclusion of geometries ? "composite geometries"
2. How to define geometries from its borders ?

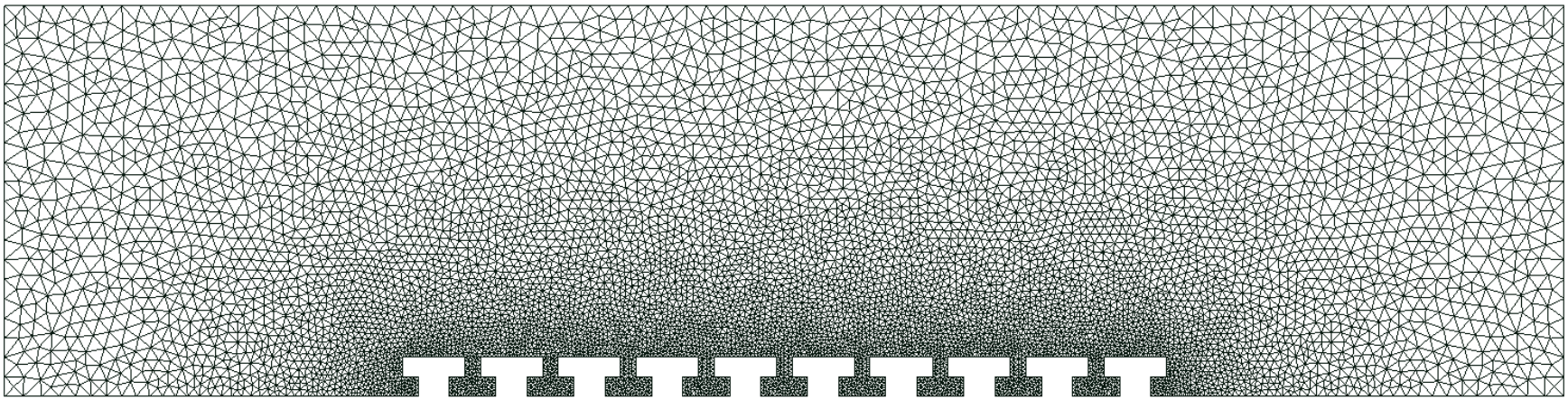




```
Point a(...), b(...), c(...), d(...), e(...), f(...), g(...), h(...);
Segment s1(...), s2(...), s3(...), s4(...);
CircArc c1(...), c2(...), c3(...), c4(...);
Mesh mesh2dP1Loop(planeSurfaceFrom(s1+c1+s2+c2+s3+c3+s4+c4), _triangle, 1, _gmsh);
```

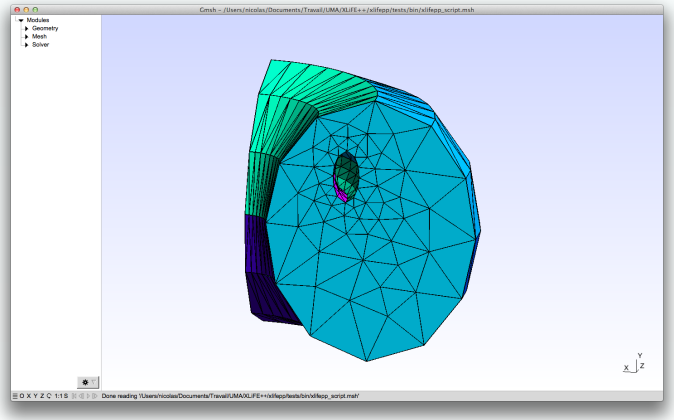
- Geometries must share the same orientation. When needed, you can reverse it by using the $\sim$ operator
- Sum s1+c1+s2+c2+s3+c3+s4+c4 can be written in any order
- You can define ruled surfaces and volumes by the same way

- You can merge meshes
- You can apply geometrical transformations on points, geometries and meshes: translation, rotation, homothety, reflection, ...



```
...
// Definition of the cavities
Real cL=2.*l1+l3; // cavity length
Number nbcav=10; // number of cavities
Geometry cavities=cavity;
for (Number n=1;n<nbcav; n++) { cavities+=translate(cavity,n*cL,0.); }
...
```

- You can merge meshes
- You can apply geometrical transformations on points, geometries and meshes: translation, rotation, homothety, reflection, ...
- You can define geometries by extrusion by translation, rotation or composition of translation and rotation



```
Ellipse ell1(...);
Ellipse ell2(...);
Geometry extr3=extrude(ell1-ell2, Rotation3d(Point(5.,0.,0.), 0., 5., 0., pi_/2), 10,
      "ExtrOmega3");
```

- When no domain names are given in geometries, a main domain named "Omega" is created with elements of highest dimension.
- You can use a domain name several times to name several parts of your geometry. In this case, only one domain will be created
- You have to define domains in your mesh files if you want to load them.
- You can merge domains from a same mesh
- How to get domains from a mesh ?

- When no domain names are given in geometries, a main domain named "Omega" is created with elements of highest dimension.
- You can use a domain name several times to name several parts of your geometry. In this case, only one domain will be created
- You have to define domains in your mesh files if you want to load them.
- You can merge domains from a same mesh
- How to get domains from a mesh ?

- When no domain names are given in geometries, a main domain named "Omega" is created with elements of highest dimension.
- You can use a domain name several times to name several parts of your geometry. In this case, only one domain will be created
- You have to define domains in your mesh files if you want to load them.
- You can merge domains from a same mesh
- How to get domains from a mesh ?

- When no domain names are given in geometries, a main domain named "Omega" is created with elements of highest dimension.
- You can use a domain name several times to name several parts of your geometry. In this case, only one domain will be created
- You have to define domains in your mesh files if you want to load them.
- You can merge domains from a same mesh
- How to get domains from a mesh ?

- When no domain names are given in geometries, a main domain named "Omega" is created with elements of highest dimension.
- You can use a domain name several times to name several parts of your geometry. In this case, only one domain will be created
- You have to define domains in your mesh files if you want to load them.
- You can merge domains from a same mesh
- How to get domains from a mesh ?

```
Parallelepiped p(_v1=Point(0.,0.,0.), _v2=Point(5.,0.,0.), _v4=Point(3.,0.,0.),
      _v5=Point(1.,0.,2), _hsteps=0.1, _domain_name="Omega", _side_names="Gamma");
Mesh m(p,_tetrahedron,2,_gmsh);
Domain omega=m.domain("Omega"); // or m.domain(0);
Domain gamma=m.domain("Gamma"); // or m.domain(1);
```

To define a space, we need to give a geometrical definition of the domain where the problem is to be solved and a finite element interpolation

To define a space, we need to give a geometrical definition of the domain where the problem is to be solved and a finite element interpolation

Different ways to define Lagrange Finite Element spaces:

```
Space vh(omega, P1, "Vh");
Space vh2(omega, Q6, "Vh");
Interpolation& interp(Lagrange, standard, 1, H1);
Space vh3(omega, interp, "Vh");
```

You can also build spaces for conform Hdiv / Hrot elements and for spectral elements

To define a space, we need to give a geometrical definition of the domain where the problem is to be solved and a finite element interpolation

Different ways to define Lagrange Finite Element spaces:

```
Space vh(omega, P1, "Vh");
Space vh2(omega, Q6, "Vh");
Interpolation& interp(Lagrange, standard, 1, H1);
Space vh3(omega, interp, "Vh");
```

You can also build spaces for conform Hdiv / Hrot elements and for spectral elements

- Dofs can be scalar or vector and of various supports (points, edges, . . .)
- You can define interpolation up to order 2 when elements are on prisms or pyramids
- You can define interpolation at any order when elements are on segments, triangles, quadrangles, tetrahedra or hexahedra.

A unknown is an abstract object defined on a space.

```
Space sp(...);
Unknown u(sp,"u");
```

A unknown can be scalar (default) or vector. In this case, you can give the number of components:

```
Unknown u2(sp,"u2",2);
```

A unknown is an abstract object defined on a space.

```
Space sp(...);
Unknown u(sp,"u");
```

A unknown can be scalar (default) or vector. In this case, you can give the number of components:

```
Unknown u2(sp,"u2",2);
```

A test function can be defined as an unknown or from the unknown of which it is the dual

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

XLiFE++ authorizes to use unknown as test function

Integrals are bilinear forms or linear forms

mathematical expression	XLiFE++ translation	comment
$u * v$	u * v	u unknown, v test function
$f * v$	f * v	f function, v test function
$\nabla(u) \cdot \nabla(v)$	**grad**(u) | **grad**(v)	u unknown, v test function
$(A * \nabla(u)) \cdot \nabla(v)$	(A***grad**(u)) | **grad**(v)	u unknown, v test function, A a matrix
$(F(x) * \nabla(u)) \cdot \nabla(\overline{v})$	(F***grad**(u)) | **grad**(**conj**(v))	u unknown, v test function, F a function
$u * \mathrm{div} q$	u * **div**(q)	u unknown, q test function
$\mathrm{curl}(u) \vert \mathrm{curl}(v)$	**curl**(u) | **curl**(v)	u unknown, v test function
$\mathrm{div}(u) * \mathrm{div}(v)$	**div**(u) * **div**(v)	u unknown, v test function
$\mathcal{E}(u) : \mathcal{E}(v)$	**epsilon**(u) % **epsilon**(v)	u unknown, v test function
u(x)*G(x,y)*v(y)	u*G*v	u unknown, v test function, G a kernel
$n \wedge u$	_n $\wedge$ u or **ncross**(u)	_n normal, u unknown

Integrals are bilinear forms or linear forms

mathematical expression	XLiFE++ translation	comment
$u * v$	u * v	u unknown, v test function
$f * v$	f * v	f function, v test function
$\nabla(u) \cdot \nabla(v)$	**grad**(u) | **grad**(v)	u unknown, v test function
$(A * \nabla(u)) \cdot \nabla(v)$	(A***grad**(u)) | **grad**(v)	u unknown, v test function, A a matrix
$(F(x) * \nabla(u)) \cdot \nabla(\overline{v})$	(F***grad**(u)) | **grad**(**conj**(v))	u unknown, v test function, F a function
$u * \mathrm{div} q$	u * **div**(q)	u unknown, q test function
$\mathrm{curl}(u) \| \mathrm{curl}(v)$	**curl**(u) | **curl**(v)	u unknown, v test function
$\mathrm{div}(u) * \mathrm{div}(v)$	**div**(u) * **div**(v)	u unknown, v test function
$\mathcal{E}(u) : \mathcal{E}(v)$	**epsilon**(u) % **epsilon**(v)	u unknown, v test function
u(x)*G(x,y)*v(y)	u*G*v	u unknown, v test function, G a kernel
$n \wedge u$	_n $\wedge$ u or **ncross**(u)	_n normal, u unknown

```
BilinearForm a1 = intg(omega, grad(u)|grad(v)); // single integral for FEM
BilinearForm a2 = intg(gamma,gamma, u*G*v); // double integral for BEM
BilinearForm a3=2.*a1-i*a2;
LinearForm l = intg(omega, f*v);
```

Integrals are bilinear forms or linear forms

mathematical expression	XLiFE++ translation	comment
$u * v$	u * v	u unknown, v test function
$f * v$	f * v	f function, v test function
$\nabla(u) \cdot \nabla(v)$	**grad**(u) | **grad**(v)	u unknown, v test function
$(A * \nabla(u)) \cdot \nabla(v)$	(A***grad**(u)) | **grad**(v)	u unknown, v test function, A a matrix
$(F(x) * \nabla(u)) \cdot \nabla(\overline{v})$	(F***grad**(u)) | **grad**(**conj**(v))	u unknown, v test function, F a function
$u * \mathrm{div} q$	u * **div**(q)	u unknown, q test function
$\mathrm{curl}(u) \| \mathrm{curl}(v)$	**curl**(u) | **curl**(v)	u unknown, v test function
$\mathrm{div}(u) * \mathrm{div}(v)$	**div**(u) * **div**(v)	u unknown, v test function
$\mathcal{E}(u) : \mathcal{E}(v)$	**epsilon**(u) % **epsilon**(v)	u unknown, v test function
u(x)*G(x,y)*v(y)	u*G*v	u unknown, v test function, G a kernel
$n \wedge u$	_n $\wedge$ u or **ncross**(u)	_n normal, u unknown

```
BilinearForm a1 = intg(omega, grad(u)|grad(v)); // single integral for FEM
BilinearForm a2 = intg(gamma,gamma, u*G*v); // double integral for BEM
BilinearForm a3=2.*a1-i*a2;
LinearForm l = intg(omega, f*v);
```

How can I give an integration method ?

Available quadrature 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

	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

Available quadrature 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

	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

XLiFE++ chooses automatically what is supposed to be the most appropriate and accurate quadrature formula, but you can select it yourself

1. You declare a `QuadratureIM` object:

```
QuadratureIM quadIM1(Gauss_Legendre,2);
QuadratureIM quadIM2(symmetrical_Gauss,18);
```

1. You declare a `QuadratureIM` object:

```
QuadratureIM quadIM1(Gauss_Legendre,2);
QuadratureIM quadIM2(symmetrical_Gauss,18);
```

2. You give it to the form:

```
BilinearForm blf1=intg(omega,u*v,quadIM);
```

1. You declare a `QuadratureIM` object:

```
QuadratureIM quadIM1(Gauss_Legendre,2);
QuadratureIM quadIM2(symmetrical_Gauss,18);
```

2. You give it to the form:

```
BilinearForm blf1=intg(omega,u*v,quadIM);
```

You can write it more quickly:

```
BilinearForm blf1=intg(omega,u*v,Gauss_Legendre,2);
BilinearForm blf2=intg(omega,u*v,symmetrical_Gauss,18);
```

How to define functions ?

```
LinearForm l = intg(omega, f*v);
```

```
LinearForm l = intg(omega, f*v);
```

How to define functions (non constant coefficients) ?

```
LinearForm l = intg(omega, f*v);
```

How to define functions (non constant coefficients) ?

Signature of such functions is always the same:

- first argument is the point on which function will be evaluated (it can be a `vector<Point>` for vector case);
- second argument, optional, will contains additional parameters:

```
Real f(const Point& p, Parameters& pa=theDefaultParameters) {
  Real x=p(1), y=p(2), a=1., b=1;
  return x*(a-x)*y*(b-y);
}

int main() {
...
}
```

```
LinearForm l = intg(omega, f*v);
```

How to define functions (non constant coefficients) ?

Signature of such functions is always the same:

- first argument is the point on which function will be evaluated (it can be a `vector<Point>` for vector case);
- second argument, optional, will contains additional parameters:

```
Real f(const Point& p, Parameters& pa=theDefaultParameters) {
  Real x=p(1), y=p(2), a=1., b=1;
  return x*(a-x)*y*(b-y);
}

int main() {
...
}
```

Return values can be Real/Complex, Reals/Complexes, RealMatrices/ComplexMatrices

```
LinearForm l = intg(omega, f*v);
```

How to define functions (non constant coefficients) ?

Signature of such functions is always the same:

- first argument is the point on which function will be evaluated (it can be a `vector<Point>` for vector case);
- second argument, optional, will contains additional parameters:

```
Real f(const Point& p, Parameters& pa=theDefaultParameters) {
  Real x=p(1), y=p(2), a=1., b=1;
  return x*(a-x)*y*(b-y);
}

int main() {
...
}
```

Return values can be Real/Complex, Reals/Complexes, RealMatrices/ComplexMatrices

Suppose that you want to give values of parameters *a* and *b* outside definition of f.
How do you do it ?

1. You consider both parameters are given by *pa*:

```
Real f(const Point& p, Parameters& pa=theDefaultParameters) {
  Real x=p(1), y=p(2), a=pa("a"), b=pa("b");
  return x*(a-x)*y*(b-y);
}
```

1. You consider both parameters are given by *pa*:

```
Real f(const Point& p, Parameters& pa=theDefaultParameters) {
  Real x=p(1), y=p(2), a=pa("a"), b=pa("b");
  return x*(a-x)*y*(b-y);
}
```

2. You define a `Parameters` object in your main program containing both values:

```
int main() {
  ...
  Parameters params("a",2.);
  params << Parameter("b",3.);
```

1. You consider both parameters are given by *pa*:

```
Real f(const Point& p, Parameters& pa=theDefaultParameters) {
  Real x=p(1), y=p(2), a=pa("a"), b=pa("b");
  return x*(a-x)*y*(b-y);
}
```

2. You define a `Parameters` object in your main program containing both values:

```
int main() {
  ...
  Parameters params("a",2.);
  params << Parameter("b",3.);
```

3. You associate your parameters to the function in a `Function` object:

```
  Function myf(f,params);
```

1. You consider both parameters are given by *pa*:

```
Real f(const Point& p, Parameters& pa=theDefaultParameters) {
  Real x=p(1), y=p(2), a=pa("a"), b=pa("b");
  return x*(a-x)*y*(b-y);
}
```

2. You define a `Parameters` object in your main program containing both values:

```
int main() {
  ...
  Parameters params("a",2.);
  params << Parameter("b",3.);
```

3. You associate your parameters to the function in a `Function` object:

```
  Function myf(f,params);
```

4. You can now use *myf* in the linear form definition:

```
  LinearForm l=intg(omega,myf*v); // instead of intg(omega,f*v)
```

1. You consider both parameters are given by *pa*:

```
Real f(const Point& p, Parameters& pa=theDefaultParameters) {
  Real x=p(1), y=p(2), a=pa("a"), b=pa("b");
  return x*(a-x)*y*(b-y);
}
```

2. You define a `Parameters` object in your main program containing both values:

```
int main() {
  ...
  Parameters params("a",2.);
  params << Parameter("b",3.);
```

3. You associate your parameters to the function in a `Function` object:

```
  Function myf(f,params);
```

4. You can now use *myf* in the linear form definition:

```
  LinearForm l=intg(omega,myf*v); // instead of intg(omega,f*v)
```

You can set more or less any type of variable in a `Parameter`

How to get normal vector inside a function ?

1. You call the routine **getNormalVectorFrom** inside the function:

```
Real f(const Point& P, Parameters& pa = defaultParameters)
{
  Reals vn = getNormalVectorFrom(pa);
  return vn(1);
}
```

How to get normal vector inside a function ?

1. You call the routine **getNormalVectorFrom** inside the function:

```
Real f(const Point& P, Parameters& pa = defaultParameters)
{
  Reals vn = getNormalVectorFrom(pa);
  return vn(1);
}
```

2. You define a `Parameter` whose name is "_n"

```
Vector<Real> nv(2);
Parameters pars(nv,"_n");
Function myf(f,pars);
```

How to get normal vector inside a function ?

1. You call the routine **getNormalVectorFrom** inside the function:

```
Real f(const Point& P, Parameters& pa = defaultParameters)
{
  Reals vn = getNormalVectorFrom(pa);
  return vn(1);
}
```

2. You define a `Parameter` whose name is "_n"

```
Vector<Real> nv(2);
Parameters pars(nv,"_n");
Function myf(f,pars);
```

The `Parameters` will contain the normal vector on condition that the computation engine detects there is one `Parameter` named "_n" given to function. In this case only, the true normal vector is computed and stored in `Parameter` "_n".

Every set of linear essential conditions can be managed as a constraint linear system

Every set of linear essential conditions can be managed as a constraint linear system

Homogeneous Dirichlet:

```
EssentialConditions ecs = (u|sigmaM = 0);
EssentialConditions ecs2 = (ncross(e)|gamma = 0); // or (_n^e | gamma = 0)
```

Every set of linear essential conditions can be managed as a constraint linear system

Homogeneous Dirichlet:

```
EssentialConditions ecs = (u|sigmaM = 0);
EssentialConditions ecs2 = (ncross(e)|gamma = 0); // or (_n^e | gamma = 0)
```

Non homogeneous constant Dirichlet:

```
EssentialConditions ecs = (u|sigmaM = 1.);
```

Every set of linear essential conditions can be managed as a constraint linear system

Homogeneous Dirichlet:

```
EssentialConditions ecs = (u|sigmaM = 0);
EssentialConditions ecs2 = (ncross(e)|gamma = 0); // or (_n^e | gamma = 0)
```

Non homogeneous constant Dirichlet:

```
EssentialConditions ecs = (u|sigmaM = 1.);
```

Non-homogeneous non-constant Dirichlet:

```
Real un(const Point& P, Parameters& pa = defaultParameters) {
  return P(1);
}

EssentialConditions ecs = (u|sigmaM = un) & (u|sigmaP = 0.);
```

Every set of linear essential conditions can be managed as a constraint linear system

Homogeneous Dirichlet:

```
EssentialConditions ecs = (u|sigmaM = 0);
EssentialConditions ecs2 = (ncross(e)|gamma = 0); // or (_n^e | gamma = 0)
```

Non homogeneous constant Dirichlet:

```
EssentialConditions ecs = (u|sigmaM = 1.);
```

Non-homogeneous non-constant Dirichlet:

```
Real un(const Point& P, Parameters& pa = defaultParameters) {
  return P(1);
}

EssentialConditions ecs = (u|sigmaM = un) & (u|sigmaP = 0.);
```

You can define all sorts of essential conditions: transmission, periodic, average, ...

The linear and bilinear forms are used to build the terms of the equation, which in turn are used to solve the problem with the help of linear algebra methods.

The linear and bilinear forms are used to build the terms of the equation, which in turn are used to solve the problem with the help of linear algebra methods.

`TermVector` is the object associated to a linear form:

```
LinearForm fv = intg(omega, f*v);
TermVector F(fv, "F");
```

The linear and bilinear forms are used to build the terms of the equation, which in turn are used to solve the problem with the help of linear algebra methods.

`TermVector` is the object associated to a linear form:

```
LinearForm fv = intg(omega, f*v);
TermVector F(fv, "F");
```

Usual algebraic operations on vectors (+, -, *, /) can be performed on `TermVector` objects to create new objects:

```
LinearForm f1 = ..., f2 = ...;
TermVector B1(f1, "B1"), B2(f2, "B2");

TermVector B = 2*B1 + 3.67*B2; // new object B, linear combination of B1 and B2
```

The linear and bilinear forms are used to build the terms of the equation, which in turn are used to solve the problem with the help of linear algebra methods.

`TermVector` is the object associated to a linear form:

```
LinearForm fv = intg(omega, f*v);
TermVector F(fv, "F");
```

Usual algebraic operations on vectors (+, -, *, /) can be performed on `TermVector` objects to create new objects:

```
LinearForm f1 = ..., f2 = ...;
TermVector B1(f1, "B1"), B2(f2, "B2");

TermVector B = 2*B1 + 3.67*B2; // new object B, linear combination of B1 and B2
```

or, using one the operators +=, -=, *= or /=, to update an existing one:

```
B *= pi_; // B is modified
```

A `TermVector` can also be converted:

```
A.toAbs(), B.toReal(), C.toImag(); // A, B and C are modified
```

Standard tools are available:

```
Complex pr = U | V, qr = innerProduct(U, V);          // pr == qr if U and V real
Complex pc = U | V, qc = hermitianProduct(U, V);      // pc == qc
Real r1 = norm1(U), r2 = norm2(U), rinf = norminfty(U);
```

Standard tools are available:

```
Complex pr = U | V, qr = innerProduct(U, V);          // pr == qr if U and V real
Complex pc = U | V, qc = hermitianProduct(U, V);      // pc == qc
Real r1 = norm1(U), r2 = norm2(U), rinf = norminfty(U);
```

We can also make copies of `TermVector` objects:

```
TermVector U(...);           // U is the original vector
TermVector V(U, "V");        // V is a new vector equal to U
TermVector W(U, 10.5, "W"); // W is a copy of U with its components replaced by 10.5
```

Standard tools are available:

```
Complex pr = U | V, qr = innerProduct(U, V);          // pr == qr if U and V real
Complex pc = U | V, qc = hermitianProduct(U, V);      // pc == qc
Real r1 = norm1(U), r2 = norm2(U), rinf = norminfty(U);
```

We can also make copies of `TermVector` objects:

```
TermVector U(...);          // U is the original vector
TermVector V(U, "V");       // V is a new vector equal to U
TermVector W(U, 10.5, "W"); // W is a copy of U with its components replaced by 10.5
```

In the case of Lagrange FE, a `TermVector` may also be defined from its components computed at every interpolation node of the FE space ("nodal value" term):

```
Reals f(const Point& P, Parameters& pa = defaultParameters) {...}

Space Vh(omega,P1,"V");
Unknown u(Vh,"u");
TermVector B(u, omega, f, "B");
```

$\longrightarrow$ The argument f above may be a function (standard or `Function` object) or a constant value

By default, a `TermVector` object is computed when defined. It is possible to delay the computation:

```
TermVector B(fv, _notCompute, "B");   // do not compute B
...
compute(B); // now compute B
```

By default, a `TermVector` object is computed when defined. It is possible to delay the computation:

```
TermVector B(fv, _notCompute, "B");   // do not compute B
...
compute(B); // now compute B
```

More advanced operations are available, namely:

- restriction to a sub-domain (`V = U.onDomain(Sigma)`),
- selection based on unknowns in the case of a problem with multiple unknowns.

By default, a `TermVector` object is computed when defined. It is possible to delay the computation:

```
TermVector B(fv, _notCompute, "B");   // do not compute B
...
compute(B); // now compute B
```

More advanced operations are available, namely:

- restriction to a sub-domain (`V = U.onDomain(Sigma)`),
- selection based on unknowns in the case of a problem with multiple unknowns.

Output: shown in Postprocessing section

`TermMatrix` is the object associated to a bilinear form:

```
BilinearForm auv = intg(omega, grad(u)|grad(v));
TermMatrix A(auv, "A");
```

`TermMatrix` is the object associated to a bilinear form:

```
BilinearForm auv = intg(omega, grad(u)|grad(v));
TermMatrix A(auv, "A");
```

If there exist essential conditions, they should be specified here:

```
BilinearForm auv=intg(omega,grad(u)|grad(v));
EssentialConditions ecs= (u|sigmaM = 1) & (u|sigmaP = 1);
TermMatrix A(auv, ecs, "A");
```

`TermMatrix` is the object associated to a bilinear form:

```
BilinearForm auv = intg(omega, grad(u)|grad(v));
TermMatrix A(auv, "A");
```

If there exist essential conditions, they should be specified here:

```
BilinearForm auv=intg(omega,grad(u)|grad(v));
EssentialConditions ecs= (u|sigmaM = 1) & (u|sigmaP = 1);
TermMatrix A(auv, ecs, "A");
```

$\longrightarrow$ Essential conditions are never attached to a `TermVector` (automatic).

`TermMatrix` is the object associated to a bilinear form:

```
BilinearForm auv = intg(omega, grad(u)|grad(v));
TermMatrix A(auv, "A");
```

If there exist essential conditions, they should be specified here:

```
BilinearForm auv=intg(omega,grad(u)|grad(v));
EssentialConditions ecs= (u|sigmaM = 1) & (u|sigmaP = 1);
TermMatrix A(auv, ecs, "A");
```

$\longrightarrow$ Essential conditions are never attached to a `TermVector` (automatic).

Algebraic operations on `TermMatrix` (analogous to `TermVector`):

```
            /*  with scalars  */
BilinearForm auv = ...;
TermMatrix A(auv);
Complex s =...;
TermMatrix B = s*A, C = A*s, D = A/s; // new objects B, C, D
A *= s, B /=s;                        // A and B modified
```

```
            /*  with TermMatrix  */
TermMatrix E = 2*A + 3.67*B - 4.1*C;  // allow linear combinations
TermMatrix F = A * B * C;             // matrix product
F += E;                               // F modified
```

`TermMatrix` is the object associated to a bilinear form:

```
BilinearForm auv = intg(omega, grad(u)|grad(v));
TermMatrix A(auv, "A");
```

If there exist essential conditions, they should be specified here:

```
BilinearForm auv=intg(omega,grad(u)|grad(v));
EssentialConditions ecs= (u|sigmaM = 1) & (u|sigmaP = 1);
TermMatrix A(auv, ecs, "A");
```

$\longrightarrow$ Essential conditions are never attached to a `TermVector` (automatic).

Algebraic operations on `TermMatrix` (analogous to `TermVector`):

```
            /*  with scalars  */
BilinearForm auv = ...;
TermMatrix A(auv);
Complex s =...;
TermMatrix B = s*A, C = A*s, D = A/s; // new objects B, C, D
A *= s, B /=s;                        // A and B modified
```

```
            /*  with TermMatrix  */
TermMatrix E = 2*A + 3.67*B - 4.1*C;  // allow linear combinations
TermMatrix F = A * B * C;             // matrix product
F += E;                               // F modified
```

$\longrightarrow$ linear combinations are best done at the bilinear form level

```
                /*   with TermVector   */
TermMatrix A(...);
TermVector U(...);
TermVector B = A * U;                  // matrix-vector product
```

```
                /*   with TermVector   */
TermMatrix A(...);
TermVector U(...);
TermVector B = A * U;                  // matrix-vector product
```

These are sophisticated operations. There are underlying checks and compatibility rules to ensure the coherence of the requested operations.

Example: sum of A, related to domain Ω, and B related to $\partial\Omega$.

```
                /*   with TermVector   */
TermMatrix A(...);
TermVector U(...);
TermVector B = A * U;                  // matrix-vector product
```

These are sophisticated operations. There are underlying checks and compatibility rules to ensure the coherence of the requested operations.

Example: sum of A, related to domain Ω, and B related to $\partial\Omega$.

XLiFE++ makes default choices, but additional options may be useful. They are related to:

- computation: keyword _notCompute (idem `TermVector`),
- assembling: keyword _unassembled (blocs are not merged),
- kind of symmetry: e.g. skew symmetric, self adjoint,... through an appropriate keyword,
  $\longrightarrow$ useful to impose kind of symmetry not obvious to deduce from the bilinear form
- kind of storage: compressed sparse, skyline or dense,
  $\longrightarrow$ useful to avoid storage conversion
  $\longrightarrow$ after computation, the storage may be changed with the **setStorage** function
- reduction method: to take into account the constrainsts due to essential conditions.

```
TermMatrix A(auv, _notCompute, _nonSymmetricMatrix, "A");
```

A `TermMatrix` can be converted:

```
A.toComplex(), B.toReal(), C.toImag();                    // A, B and C are modified
TermMatrix D = toComplex(A), E = real(A), F = imag(A);    // D, E and F are created
```

A `TermMatrix` can be converted:

```
A.toComplex(), B.toReal(), C.toImag();                    // A, B and C are modified
TermMatrix D = toComplex(A), E = real(A), F = imag(A);    // D, E and F are created
```

There exists some other ways to create a `TermMatrix` object in particular contexts (advance usage).

A `TermMatrix` can be converted:

```
A.toComplex(), B.toReal(), C.toImag();                    // A, B and C are modified
TermMatrix D = toComplex(A), E = real(A), F = imag(A); // D, E and F are created
```

There exists some other ways to create a `TermMatrix` object in particular contexts (advance usage).

A `TermMatrix` can be printed:

```
TermMatrix A(auv, "A");

verboseLevel(30);           // to limit the information amount
A.print(thePrintStream);    // prints into print.txt file
thePrintStream << A;        // idem previous

A.printSummary(cout);       // prints characteritics of the matrix to the screen
A.viewStorage(cout);        // prints information on storage to the screen
```

A `TermMatrix` can be converted:

```
A.toComplex(), B.toReal(), C.toImag();                    // A, B and C are modified
TermMatrix D = toComplex(A), E = real(A), F = imag(A); // D, E and F are created
```

There exists some other ways to create a `TermMatrix` object in particular contexts (advance usage).

A `TermMatrix` can be printed:

```
TermMatrix A(auv, "A");

verboseLevel(30);          // to limit the information amount
A.print(thePrintStream);   // prints into print.txt file
thePrintStream << A;       // idem previous

A.printSummary(cout);      // prints characteritics of the matrix to the screen
A.viewStorage(cout);       // prints information on storage to the screen
```

It can be saved into a file in dense or coordinate format:

```
A.saveToFile("matAdense.dat",_dense); // every Aij
A.saveToFile("matAcoor.dat",_coo);    // (i, j, val) format compatible with Matlab sparse

saveToFile(A,"matAdense.dat",_dense); // alternative syntax
saveToFile(A,"matAcoor.dat",_coo);    // idem
```

Direct solvers: Gauss with or without pivoting, LU, LDL^T, LDL*, UMFPACK

Automatic mode:

```
TermMatrix A(auv);
TermVector b(fv);
TermVector x = directSolve(A, b);    // compute "directly" the solution to  A x = b

TermMatrix factA;                    // create a new TermMatrix to store the factorization
factorize(A, factA);                 // factorize A
TermVector y = factSolve(factA, b); // solve the linear system using the factorization
TermVector z = factSolve(factA, c); // re-use the factorization
```

Direct solvers: Gauss with or without pivoting, LU, LDL^T, LDL*, UMFPACK

Automatic mode:

```
TermMatrix A(auv);
TermVector b(fv);
TermVector x = directSolve(A, b);    // compute "directly" the solution to  A x = b

TermMatrix factA;                    // create a new TermMatrix to store the factorization
factorize(A, factA);                 // factorize A
TermVector y = factSolve(factA, b); // solve the linear system using the factorization
TermVector z = factSolve(factA, c); // re-use the factorization
```

Specifying the method:

```
TermVector x = gaussSolve(A,b);      // for dense matrices

TermMatrix factA;
ldltFactorize    (A, factA); TermVector x = factSolve(factA, b); // A symmetric
ldlstarFactorize(A, factA); TermVector x = factSolve(factA, b); // A hermitian
luFactorize      (A, factA); TermVector x = factSolve(factA, b);

// Alternatively:
TermVector x = ldltSolve   (A, b, factA); // solve and output factorized matrix...
TermVector y = factSolve(factA, c);       // ... which can be reused later

TermVector x = ldlstarSolve(A, b, factA); // idem for LDL*
TermVector x = luSolve     (A, b, factA); // idem for LU
```

$\longrightarrow$ UMFPACK is used if the installation of XLIFE++ was settled with this choice

Iterative solvers: BiCG, BiCGStab, CG, CGS, GMRES, QMR, SOR, SSOR, with or without preconditioner

Functional syntax (example with cg):

```
TermVector x = iterativeSolve(A, b, x0, _solver=_cg); // compute the solution to  A x = b
TermVector y = cgSolve(A, b, x0);                     // idem (x0 = initial guess)

PreconditionerTerm P(A,_luPrec);                      // with preconditioner and additional
TermVector y = cgSolve(A, b, x0, P, _tolerance=1e-8, _maxIt=1000); //              parameters
```

$\longrightarrow$ Change cg for cgs, gmres, etc. to select another method

Iterative solvers: BiCG, BiCGStab, CG, CGS, GMRES, QMR, SOR, SSOR, with or without preconditioner

Functional syntax (example with cg):

```
TermVector x = iterativeSolve(A, b, x0, _solver=_cg); // compute the solution to  A x = b
TermVector y = cgSolve(A, b, x0);                     // idem (x0 = initial guess)

PreconditionerTerm P(A,_luPrec);                      // with preconditioner and additional
TermVector y = cgSolve(A, b, x0, P, _tolerance=1e-8, _maxIt=1000); //            parameters
```

$\longrightarrow$ Change cg for cgs, gmres, etc. to select another method

Object syntax:

```
CgSolver myCG(1e-8, 1000);      // Define an iterative solver object, specifying
                                // the tolerance and the maximum number of iterations

TermVector x = myCG(A, b, x0); // Solve the system with initial guess x0
```

Eigen solvers: internal eigen solver (with KrylovSchur or Davidson method), ARPACK

```
// Build the generalized eigenvalue problem (R+M) u = lambda M u
BilinearForm muv = intg(omega, u * v),
             auv = intg(omega, grad(u) | grad(v)) + muv;
TermMatrix A(auv, "R+M"), M(muv, "M");

// Compute the 10 first smallest in magnitude
EigenElements eigsI = eigenInternSolve(A, M, _nev=10, _mode=_krylovSchur, _which="SM");

EigenElements eigsA = arpackSolve(A, M, _nev=10, _which="SM"); // Arpack "regular mode"
```

$\longrightarrow$ Other parameters are available, e.g. **_sigma**, **_tolerance**, and still other ones for Arpack to tune the computation.

Output for postprocessing:

```
saveToFile("eigsI", eigsI.vectors, vtu); // Save eigenvectors into files in vtu format
saveToFile("eigsA", eigsA.vectors, msh); // Idem for gmsh use

saveToFile("eigs", eigsA, msh); // Save eigenvalues into file eigs_eigenvalues
                                // and eigenvectors into separate files eigs_N.msh
```

Output: a `TermVector` (and a `TermMatrix`) may be printed or saved into a file:

```
LinearForm fv=intg(omega, f*u);
TermVector B(fv, "B");

cout << "vector B " << endl;
cout << B; // or B.print(cout);    // B printed on the screen
thePrintStream << B;               // B printed to the file print.txt
saveToFile("file.dat", B, _vtk);   // B saved into a file in paraview default format
```

$\longrightarrow$ Other available formats are `_vtu` (paraview xml format), `_matlab` (matlab-octave), `_msh` (gmsh) and `_raw` (only values). Except the last one, they all include geometrical information.

Output: a `TermVector` (and a `TermMatrix`) may be printed or saved into a file:

```
LinearForm fv=intg(omega, f*u);
TermVector B(fv, "B");

cout << "vector B " << endl;
cout << B; // or B.print(cout);    // B printed on the screen
thePrintStream << B;               // B printed to the file print.txt
saveToFile("file.dat", B, _vtk);   // B saved into a file in paraview default format
```

$\longrightarrow$ Other available formats are `_vtu` (paraview xml format), `_matlab` (matlab-octave), `_msh` (gmsh) and `_raw` (only values). Except the last one, they all include geometrical information.

Several `TermVector` objects can be exported:

```
TermVector B1(...), B2(...);
saveToFile("file.dat", B1, B2, _vtk);   // B1 and B2 saved into the same file

TermVectors Bs;                         // group of TermVector objects
saveToFile("file.dat", Bs, _vtk);   // saved into the same file
```

Output: a `TermVector` (and a `TermMatrix`) may be printed or saved into a file:

```
LinearForm fv=intg(omega, f*u);
TermVector B(fv, "B");

cout << "vector B " << endl;
cout << B; // or B.print(cout);    // B printed on the screen
thePrintStream << B;               // B printed to the file print.txt
saveToFile("file.dat", B, _vtk);   // B saved into a file in paraview default format
```

$\longrightarrow$ Other available formats are `_vtu` (paraview xml format), `_matlab` (matlab-octave), `_msh` (gmsh) and `_raw` (only values). Except the last one, they all include geometrical information.

Several `TermVector` objects can be exported:

```
TermVector B1(...), B2(...);
saveToFile("file.dat", B1, B2, _vtk);  // B1 and B2 saved into the same file

TermVectors Bs;                        // group of TermVector objects
saveToFile("file.dat", Bs, _vtk);      // saved into the same file
```

A viewer may be directly called to visualize a `TermVector`:

```
TermVector B(...);
plot(B, _vtk); // or _vtu (paraview is launched), or _msh (gmsh is launched)
```