XLiFE++
Days 2016

june, 14-15, 2016

Unité de Mathématiques Appliquées
ENSTA ParisTech

Mardi 14 juin 2016
09h30 Accueil des participants
10h00 Introduction à XLiFE++ *(E. Lunéville)*
10h30 C++ pour les nuls *(E. Lunéville)*
10h45 Cours 1 : Concepts XLiFE++ de base *(N. Kielbasiewicz, Y. Lafranche)*
11h30 Pause
11h45 Cours 2 : Concepts XLiFE++ avancés *(E. Lunéville)*
12h15 Cours 3 : BEM, DTN, couplage *(N. Salles)*
13h00 Déjeuner
14h00 Installer et utiliser XLiFE++ *(N. Kielbasiewicz, N. Salles)*
15h00 TP1 : Helmhotz2D, méthode FEM
16h30 Pause
17h00 TP2 : Hemholtz3D, méthode BEM
19h30 : Barbecue au « 10 ter »

Mercredi 15 juin 2016
Présentation d'applications
09h00 Obstacle invisible dans un guide d'onde *(AS. Bonnet, A. Bera)*
09h30 Couplage FEM – représentation demi-espace *(A.S Bonnet, S. Fliss, N. Gmati, Y. Tjandrawidjadja)*
10h00 Aéroacoustique via le modèle de Goldstein *(JF. Mercier, E. Lunéville)*
10h30 Pause
11h00 Linear Sampling Method en régime transitoire *(L. Bourgeois, A. Recoquillay)*
11h30 Couplage FEM - méthode de rayon *(M. Lenoir)*
12h00 Perspectives XLiFE++
12h30 Déjeuner
13h30 TP3 : Problème de Laplace par méthode mixte
15h00 Pause
15h15 TP4 : Libre
17h00 Fin des journées XLiFE++

XLiFE++
from yesterday to today

XLiFE++ origins

A long story of FE softwares at POEMS

- **In 1980s : Lena in Fortran77 *(D. Martin)***
- **In 1990s : Melina (Finite Element library in Fortran 77) *(D. Martin)***
  variational approach, multi unknowns, Lagrange FE
- **In 2002 : Montjoie (C++) *(M. Duruflé)***
  problem approach, Lagrange FE, DG, specific methods
- **In 2004 : move Melina to Melina++ (C++) *(D. Martin, E. Lunéville)***
  Melina has been used successfully but new developments are more and more hard -> new library in C++
- **In 2010 : go further with a richest C++ library : XLiFE++ *(E. Lunéville)***
  supported by Simposium european contract

SIMPOSIUM
Interoperable NDE simulation tools

Today : version 1.4

XLiFE++ origins

eXtended Library of Finite Elements in C++

- Deal with 1D, 2D, 3D scalar/vector transient/stationnary/harmonic pbs
- High order Lagrange FE, edge FE (Hrot, Hdiv), spectral FE
- H1 Conform and non conform approximation (DG methods)
- Unassembling 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 programmation (OPENMP, 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++ team

Poems

- **Eric Lunéville** (FE computation)

- **Nicolas Kielbasiewicz** (Development environment and mesh tools)

- **Colin Chambeyron** (Iterative solvers)

- **Nicolas Salles** (Integral equations)

Irmar

- **Yvon Lafranche** (Mesh tools)
- **Eric Darrigrand** (Fast Multipole methods)

- **Pierre Navaro** (Test environment)

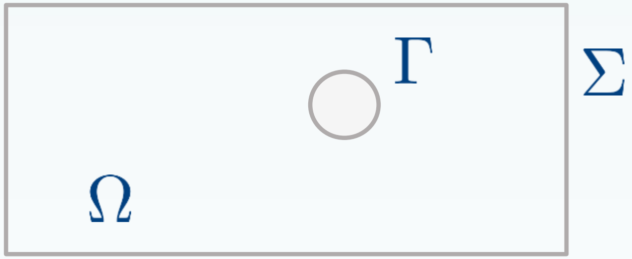
thanks to **Marc Lenoir** (moral support)
Man Ha Nguyen (Eigen solvers and parallel computing)

and **students** (beta-testers)

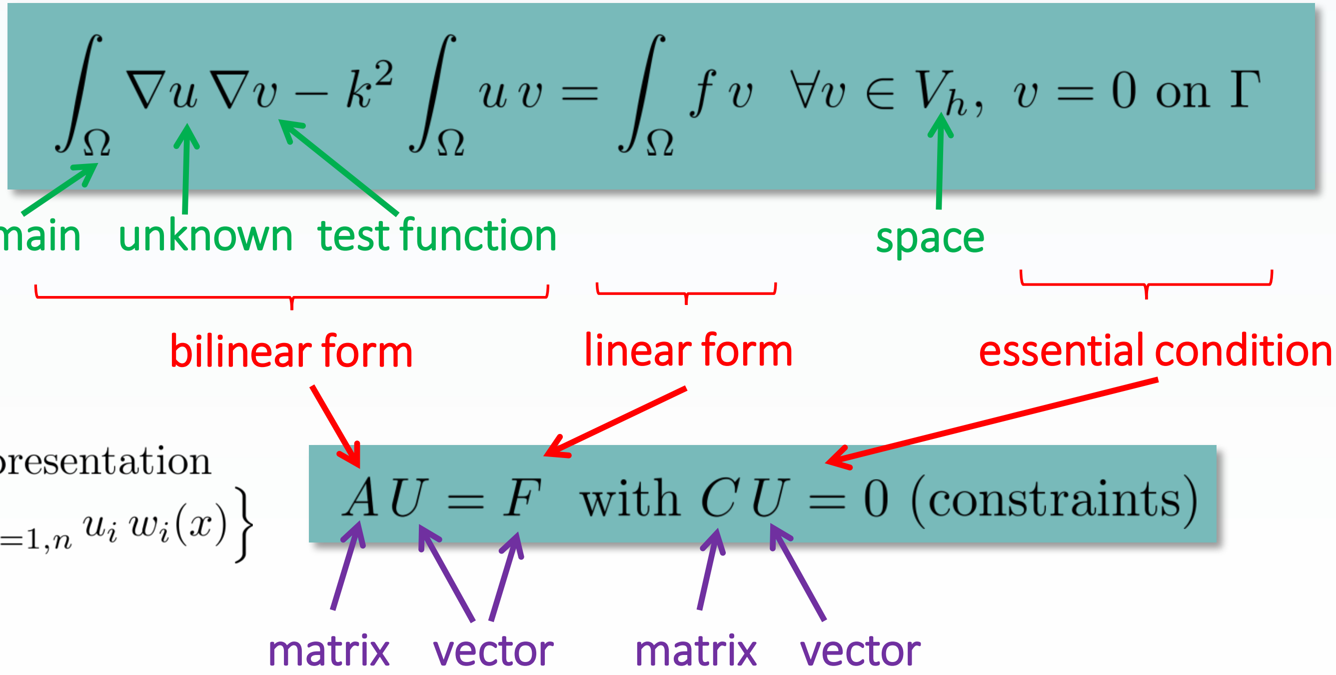
XliFE++ philosophy

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 Γ such that



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

domain unknown test function space

bilinear form linear form essential condition

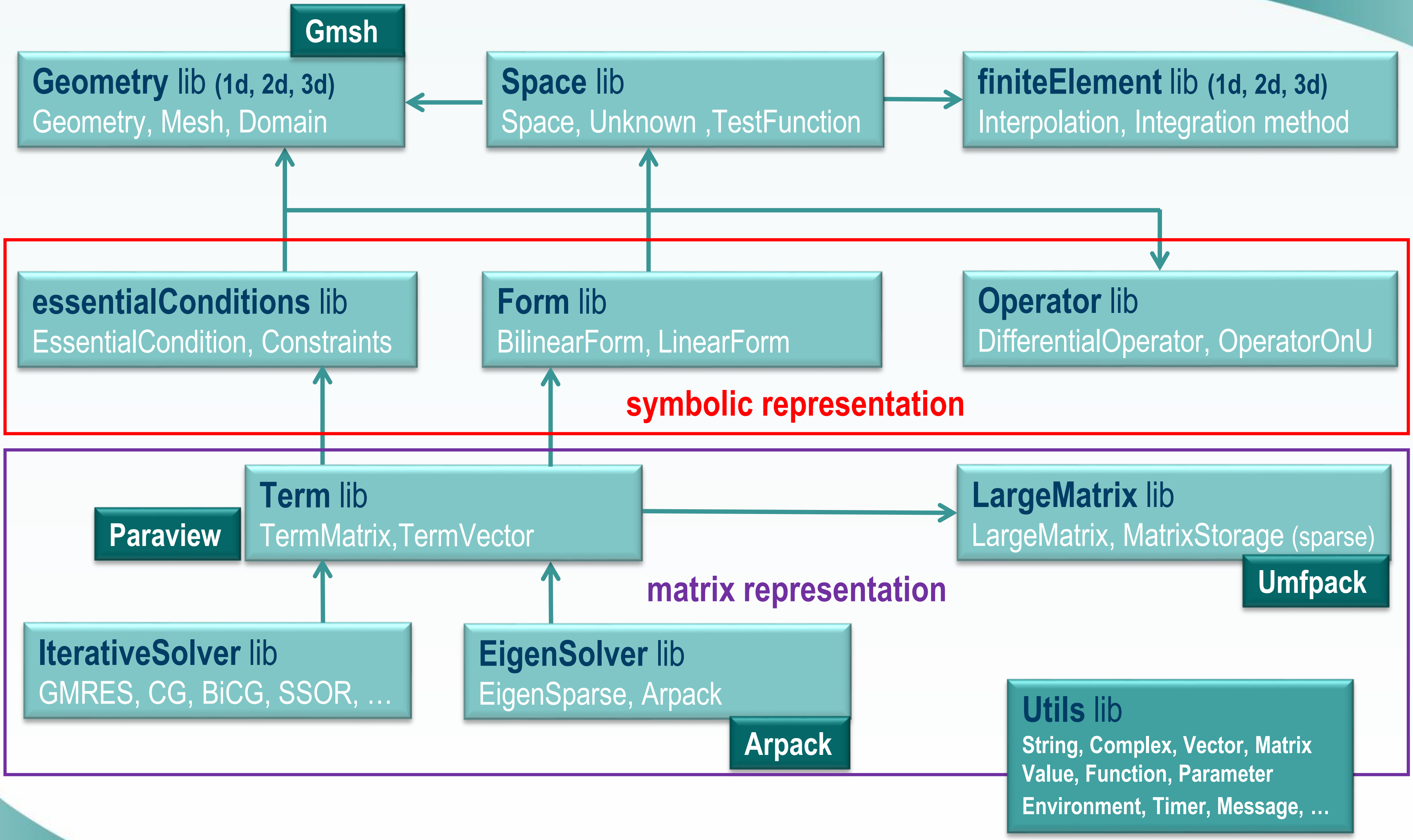
matrix representation

$V_h = \left\{ \sum_{i=1,n} u_i\, w_i(x) \right\}$

$$A\,U = F \quad \text{with } C\,U = 0 \text{ (constraints)}$$

matrix vector matrix vector

XLiFE++ architecture



Gmsh

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

Space lib
Space, Unknown ,TestFunction

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

essentialConditions lib
EssentialCondition, Constraints

Form lib
BilinearForm, LinearForm

Operator lib
DifferentialOperator, OperatorOnU

symbolic representation

Paraview

Term lib
TermMatrix,TermVector

LargeMatrix lib
LargeMatrix, MatrixStorage (sparse)

Umfpack

matrix representation

IterativeSolver lib
GMRES, CG, BiCG, SSOR, …

EigenSolver lib
EigenSparse, Arpack

Arpack

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

XLiFE++ main user classes

Number, Int, Real, Complex, String : basic variable types
Reals, Complexes, RealMatrix, ComplexMatrix to deal with real/complex vector/matrix
Point to deal with in 1D, 2D, 3D point
Parameter : named parameter of type Real, Complex, Integer, String
Parameters : list of parameters
Function : generalized function handling a c++ function and a list of parameters
Kernel : generalized kernel managing a Function (the kernel) and some additional data
TensorKernel : special form of kernel useful to DtN map
Geometry : geometric objects (segment, rectangle, circle, sphere, ...)
Mesh : structure containing nodes, geometric elements, ...
Domain : alias of geometric domains describing part of the mesh
Space : discrete spaces (FE space or spectral space)
Unknown : symbolic element of space (**TestFunction** is an alias of Unknown)
LinearForm, BilinearForm : symbolic representation of a linear/bilinear form
EssentialConditions : symbolic representation of an essential conditions
TermVector : algebraic representation of a linear form or element of space as vector
TermMatrix : algebraic representation of a bilinear form
TermVectors : list of TermVector's,
EigenElements : list of eigen elements

How XLiFE++ works

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

Write a c++ main program

```
Real f(const Point& P, Parameters& pa = defaultParameters) { return P(1)*P(2);}
void main(){
  init(fr);
  Mesh m(Square(_origin=Point(0.,0.),_length=1., _nnodes= 100, "Omega"),
                      _triangle, _structured);
  Domain omega = m.domain("Omega");
  Space V(omega, P1);
  Unknown u("u", V); TestFunction v(u, "v");
  Real k2=2.;
  BilinearForm auv = intg(omega, grad(u) | grad(v)) – k2* intg(omg, u * v);
  LinearForm fv = intg(omega, f * v);
  TermMatrix A(auv);
  TermVector B(fv);
  TermVector U = directSolve(A, B);
  saveToFile("U.vtk",U,_vtk);}
```

define data function

choose french

mesh the unit square

get domain

define P1 space

define symbolic unknown and test function (dual)

define symbolic bilinear and linear forms

define algebraic representations A and B

solve AU=B using direct method

save U on vtk file (Paraview)

Syntax as close as possible to Mathematics
basic C++ (no pointer, no template)

XLiFE++ still in progress

V 1.4: Lagrange for FE, IE, SP method, edge elt for FE, IE

- licence GPL3 for open software, other licence for closed software
- about 130000 C++ lines (190000 with comments), 160 C++ classes,
- user doc (185 pages), developer doc (370 pages)
- doxygen
- forge repository (INRIA gforge)
- cmake stuff (windows, mac os, linux)
- non regressive automatic tests

XLiFE++
C++ for dummies

Instruction block

C++ is a sequential language, a sequence of instructions, to be compiled

Simple instruction ends by a semicolon, instruction sequence delimited by braces

```
{
  instruction;              ended by ; no limit of the length
  instruction;              use at least one space to separate name of variable
  …
}
```

Nested instruction blocks

```
{
  instruction;
  {
    instruction;
  }
  …
}
```

Comment in C++

```
/*
  multiple lines comment
*/
// single line comment
```

Primary variable types

As other languages, C++ proposes few primary variable types (integer, real, char)

Any variable has to be declared with its type

```
{
  int i;                   integer (32 bits), range :  [-2 147 483,648 , 2 147 483 647]
  unsigned int j;          unsigned integer (32 bits), range : [0, 4 294 967 295]
  float x;                 real single precision (32 bits), max ± 3.4 · 10^38, min ± 1.17 · 10^-38
  double y;                real double precision (64 bits), max  ± 1.7 · 10^± 308, min  ± 2.22 · 10^-308
  char c;                  character  (8bits) range [0,255] -> ascii code …, ‘0’, ‘1’, …, ‘A’, ’B’,…, ‘a’, ’b’, …
  bool flag;               logical (true or false)
}
```

Name variable can use almost any character, <u>but never begins by a number</u>

```
{
 int my_integer = 0;                assign 0 when declare int variable
 double _pi = 3.1415926;            assign  pi value when declare double variable
 float  gamma_ =  5.772156e-1;      assign in scientific format
 bool  flag = true ;                assign true to a boolean variable
}
```

Scope of variables

Variables may be declared everywhere in an instruction block, only once
They exist only in the block where they have been declared (variable scope)

```
{
   float x=0.;
   {
     float y=x;      OK
      …
   }
  float z = y;       NO, y no longer exists
}
```

XLiFE++ aliases some primary variables types to shadow compiler dependences

XLiFE++ type	C++ type	comment
Int	**int or long int**	**depends on config**
Number	**unsigned int or long int**	**depends on config**
Real	**float or double**	**depends on config**
Complex	***not a primary C++ type***	***std::complex<Real>***

Basic operations

Main C++ operators

operator	meaning	usage
=	assignment	int i = 0;
+, -, *, /	standard operations	x=(a+b)/2*c;
+=, -=, *=, /=	operations applied on left variable	x+=(a-b); ⇔ x=x+a-b;
++, --	to increment or decrement by 1	i--; ⇔ i=i-1;
!	negation operator	bool b = !true; *(=false)*
==, !=	is equal or is not equal	bool b= (x==1);
<, >, <=, >=	comparison operators	bool b = (x<=1);
&&, ||	logical operator ‘and ‘, ‘or’	b = (x<-1) || (x>1); *(⇔|x|>1)*
<<, >>	insert or extract from stream	read and write operations

be careful with priority rules in operations

Define and use a function

Define a function

```
type_out   function_name ( type_in_1 v1, type_in_2 v2,  ….)
{
   …
  return type_out_var;
}
```

only one return argument

Use the function

```
type_out res = function_name (type_in_1_var, type_in_2_var, …)
```

define f(x,y)

```
float f ( float x, float y)
{
   return x+y;
}
```

use f(x,y)

```
{
  float pi=3.1415926, a=2.;
  float y = f( pi, a ) ;
}
```

If function return nothing, use void as type_out and return nothing!

“Last” input arguments may have some default values : :

```
type_out function_name (type_in_1 v1, type_in_2 v2 = def_value)
```

The main function

A main program is a function with the name “main”

```
int main( int argc, char** argv)
{
  …
 return 0;           if ok
}
```

argc : number of arguments in the command line

argv : list of arguments as an array of strings

standard XLiFE++ main

```
#include "xlife++.h"
using namespace xlifepp;            to use anything of XLiFE++

int main(int argc, char** argv)
{
  init(_lang=en);                   mandatory initialization of xlifepp
  …
  return 0;
}
```

Understanding behaviour of function

Arguments (input and output) of a function are ALWAYS copied

Means you work in function with copy of arguments, original arguments being safe

```
float f( float x)
{
   x=x+1;
  return x;
}
```

```
float a =1.;
float b = f(a);
// value of a is still 1. and value of b is 2.
```

Advantage : original variable is safe

Inconvenient : original variable cannot be modified and copy may be time expansive (large matrix for instance)

C++ answer : use reference (something related to the address of the variable)

```
float f( float& x)
{
   x=x+1;
  return x;
}
```

```
float a =1.;
float b = f(a);
// value of a is now 2. and value of b is 2.
```

Advantage : can be changed and no copy of x (copy of the reference !)

To protect argument and have no copy of x use **const float&** *instead of* **float&**

Flow process

if … else

```
if ( boolean expression)
{
…
}
```

```
if ( boolean expression)
{  … }
else
{ … }
```

switch … case …

```
switch ( enumeration variable)
{
case value1 :  { … } break;
case value2 :  { … } break;
…
default : { … };
}
```

enumeration type : integer, char or explicit enum

```
enum factorisationType
{  undefFactorization = 0,   LUFactorization,
   LDLtFactorization,    LLtFactorization}
 …
 factorisationType ft = LUFactorization;
switch(ft)
{
    case LDLtFactorisation : {…} break;
    case LUFactorisation : {…} break;
    default : {…}
     …
}
```

Flow process

loop

```
for( init_instruction; stop_criteria; increment instructions)
{
…
}
```

simple loop

```
for( int i=0; i<10; i++)
{
…
}
```

combine instructions in loop

```
float x= 50.;
for( int i=0; i<10 && x>0; i++, x-=2.)
{
…
}
```

while

```
init_instruction;
while (stop_criteria)
{
…
increment instructions
}
```

simple while

```
int i =0;
while (i<10)
{
  …
  i++;
}
```

Printing

C++ provides an easy way to print : the operator << and stream

```
float pi = 3.1415926;
cout << "pi =   "<< pi << endl;
```

terminal

```
pi = 3.1415926
>
```

cout : standard stream to print on terminal, endl : end of line character

write to a text file

```
ofstream out("outfile.txt");
float pi = 3.1415926;
out << "pi =   "<< pi << endl;
out.close();
```

outfile.txt

```
pi = 3.1415926
```

ofstream : out file stream defined in the stl (standard library)

read from a text file

```
ifstream in("infile.txt");
float pi;
in >> pi;
in.close();
```

infile.txt

```
3.1415926
```

ifstream : input file stream defined in the stl (standard library)

Classes

C++ allows to define new types of variable, say classes

A class may have

- *some data structures (members), any type of variables*
- *some related functions (member functions) , ordinary C++ functions except constructor*

An instance of a class is called an object

To access to members of an object use operator **.**

example : Complex class

```
class Complex
{
 public :
   float x , y ;
   Complex( float a=0., float b=0.) : x ( a ) , y (b)  {}       // constructor from a real pair
   float abs ( ) {return sqrt(x*x+y*y ) ;}                       // complex norm
   Complex& operator+=(const Complex& c )                        // z+=c
     {x+=c.x ; y+=c.y ; return  *this ; }
   ...
};

Complex z ( 0., 1. );        // Complex z = Complex (0.,1.); do the same
cout<<" real part of z = "<<z.x<<" |z| = "<< z.abs();
z+=z;
```

Classes

Design a new class may be a hard job, use it is quite simple

Main XLiFE++ classes

Int, Number, Real, Complex to deal with integers, reals and complexes
String to deal with character string
Numbers, Reals, Complexes, Strings to deal with vector of integer, real; complex and string
Point to deal with point in 1D, 2D, 3D
Parameter, Parameters to deal with named parameter of type integer, real, complex, string
Function handling a c++ function and a **Parameters** object
Kernel managing a Function (the kernel) and some additional data
TensorKernel a special form of kernel
Geometry, **Segment, Rectangle, Ellipse, Ball, Cylinder, . . .** to handle simple geometries
Mesh, Domain describing mesh (nodes, elements, ...) and geometrical domain
Space, Unknown, TestFunction handles discrete spaces and symbolic element of space
LinearForm, BiLinearForm : symbolic representation of a bilinear and linear forms
EssentialCondition, EssentialConditions: symbolic representation of essential
TermVector, TermMatrix, TermVectors, EigenElements : algebraic representation of a linear, bilinear form or algebraic representation of element of space

Using XLiFE++ objects

```
#include "xlife++.h"
using namespace xlifepp;

Real f(const Point& P, Parameters& pa = defaultParameters)
 { return P(1)*P(2);}


int main()
{
  init(fr);
  Square Sq(_origin=Point(0.,0.), _length=1., _nbnodes=100, "Omega");
  Mesh m (Sq, _triangle, _structured);
  Domain omega = m.domain("Omega");
  Space V(omega, P1);
  Unknown u(V, "u"); TestFunction v(u, "v");
  Real k2=2.;
  BilinearForm auv = intg(omega, grad(u) | grad(v)) – k2* intg(omg, u * v);
  LinearForm fv = intg(omega, f * v);
  TermMatrix A(auv);
  TermVector B(fv);
  TermVector U = directSolve(A, B);
  saveToFile("U.vtk",U,_vtk);
}
```