XLIFE++ examples

Nicolas KIELBASIEWICZ, Eric LUNÉVILLE

April 8, 2022

Contents

1 1D problems

Solving 1D problems is sometimes regarded to be out of interest. Anyway, most of existing FE softwares do not handle this case. But in fact, 1D problems are of interest, often as a part of more complex problems. Thus, XLIFE++ deals with 1D problems.

1.1 Dirichlet condition

As a first example, we show how to solve the very simple problem, involving Dirichlet conditions:

$$\begin{cases} -u'' = f & \text{in } \Omega =]0,1[\\ u(0) = u(1) = 0 \end{cases}$$

Its variational formulation is

$$\left|\begin{array}{l} \text{Find } u \in V = \left\{v \in L^2(\Omega),\ v' \in L^2(\Omega),\ u(0) = u(1) = 0\right\} \text{ such that} \\ \displaystyle\int_0^1 u'(x)\, v'(x) dx = \int_0^1 f(x)\, v(x) dx \quad \forall v \in V. \end{array}\right.$$

The following main program corresponds to solving this problem with $f(x) = 1$ using P1 Lagrange element (100 elements):

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

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

int main(int argc, char** argv)
{
  init(argc, argv, _lang=en); // mandatory initialization of xlifepp

  // mesh and domains
  Strings sn("x=0", "x=1");
  Segment seg(_xmin=0, _xmax=1, _nnodes=101, _domain_name="Omega", _side_names=sn);
  Mesh mesh1d(seg, 1, structured, "P1-mesh");
  Domain omega = mesh1d.domain("Omega");
  Domain sigmaL = mesh1d.domain("x=0"), sigmaR = mesh1d.domain("x=1");

  // space, unknows, and test functions
  Space Vh(_domain=omega, _interpolation=P1, _name="Vh");
  Unknown u(Vh, "u");
  TestFunction v(u, "v");

  // define problem
  BilinearForm a = intg(omega, grad(u)|grad(v));
  LinearForm lf = intg(omega, f*v);
  EssentialConditions ecs = (u|sigmaL = 0) & (u|sigmaR = 0);

  // compute matrix and rhs
  TermMatrix A(a, ecs, "A");
  TermVector F(lf, "F");
  // solve linear system and save solution
  TermVector U=directSolve(A, F);
  saveToFile("U_1d", U, vtu);
  return 0;
}
```

The following figure shows a graphical representation of the solution using PARAVIEW:

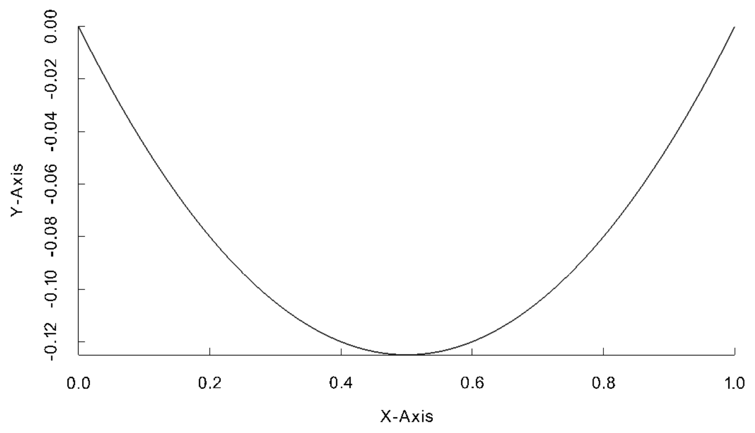


Figure 1.1: Solution of the Laplace 1D problem on the unit segment $[0,1]$

1.2 Robin condition

The second example shows that XLIFE++ can also handle non homogeneous Neumann conditions or Robin-Fourier conditions. This problem also involve a Dirichlet condition. Given three real functions f_Ω, α and f_N, the problem is:

$$\begin{cases} -u'' + u = f_\Omega & \text{in } \Omega =]a,b[\\ u(a) = 0 & \\ u'(b) + \alpha(b)\, u(b) = f_N(b) & \end{cases}$$

Its variational formulation is:

$$\left| \begin{array}{c} \text{Find } u \in V = \left\{ v \in L^2(\Omega),\ v' \in L^2(\Omega),\ u(a) = 0 \right\} \text{ such that} \\ \displaystyle\int_a^b u'(x)\, v'(x)\, dx + \int_a^b u(x)\, v(x)\, dx + \alpha(b)\, u(b) = \int_a^b f(x)\, v(x)\, dx + f_N(b), \quad \forall v \in V. \end{array} \right.$$

$\alpha(b)u(b)$ can be interpreted as $\displaystyle\int_{\{b\}} \alpha(\gamma)u(\gamma)\, v(\gamma)d\gamma$ and $f_N(b)$ can be interpreted as $\displaystyle\int_{\{b\}} f_N(\gamma)\, v(\gamma)d\gamma$ where γ is the variable over the side domain here reduced to a point. This allows to handle these conditions in a uniform syntactic way by defining linear forms as shown in the previous examples.

The following main program corresponds to solving this problem with $\alpha(x) = \dfrac{7}{2}x^2 - 8x$ using the P10 Lagrange element over the interval $]a,b[= \left]0, \dfrac{13}{4}\pi\right[$ using 4 elements ; the functions $f_\Omega(x) = 2\, sin(x)$ and $f_N(x) = cos(x) + \alpha(x)\, sin(x)$ are chosen so that the solution is $sin(x)$:

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

/*
  Test problem:
    -u" + u = fOm    on the domain Om = [a, b]
      u(a) = 0
      u'(b) + alpha(b) u(b) = fN(b)
*/

Real fctEx (const Point& P, Parameters& pa = defaultParameters)
{ return sin(P[0]); }

Real fctOm (const Point& P, Parameters& pa = defaultParameters)
{ return 2 * sin(P[0]); }

Real alpha (const Point& P, Parameters& pa = defaultParameters)
{ return 3.5*P[0]*P[0] - 8*P[0]; }

Real fctfN (const Point& P, Parameters& pa = defaultParameters)
{ return cos(P[0]) + (3.5*P[0]*P[0] - 8*P[0]) * sin(P[0]); }

int main(int argc, char** argv) {
  init(argc, argv, _lang=en); // mandatory initialization of xlifepp

  // Mesh and domains
  Strings sidenames("x=a", "x=b");
  Segment seg(_xmin=0., _xmax=3.25*pi_, _nnodes=5, _domain_name="Omega", _side_names=sidenames);
  Mesh mesh1d(seg, 1, structured);
  mesh1d.printInfo();
  Domain Omega = mesh1d.domain("Omega");
  Domain xA = mesh1d.domain("x=a");
  Domain xB = mesh1d.domain("x=b");

  // Space and unknowns
  Space Vh(_domain=Omega, _FE_type=Lagrange, _order=10, _name="Vh");
  Unknown u(Vh, "u");
  TestFunction v(u, "v");

  // Bilinear forms
  BilinearForm gugv = intg(Omega, grad(u)|grad(v)), uv = intg(Omega, u*v);
  BilinearForm aluv = intg(xB, alpha*u*v);
  LinearForm fOm = intg(Omega, fctOm*v), fN = intg(xB, fctfN*v);

  // Terms with essential conditions
  EssentialConditions ecs = (u|xA = 0);
  TermMatrix A(gugv + uv + aluv, ecs, "A");
  TermVector F(fOm + fN, "F");

  // Solve linear system and save solution
  TermVector U = directSolve(A, F);
  saveToFile("U", U, matlab);

  // Compare with exact solution
  TermVector Uex(u, Omega, fctEx, "Uex");
  theCout << "||U-Uex||inf = " << norminfty(U-Uex) << eol;
  return 0;
}
```

The following figure shows a graphical representation of the solution using OCTAVE (see user documentation of XLIFE++):

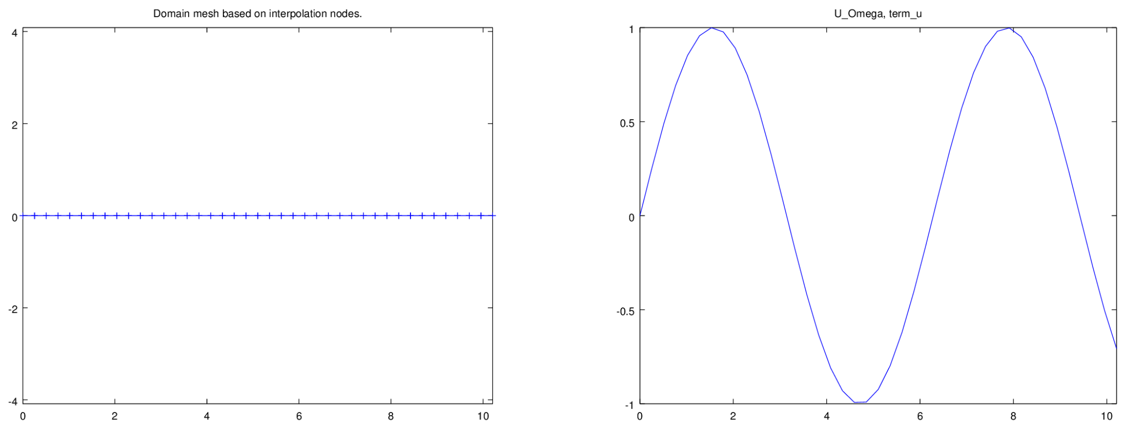


Figure 1.2: Solution of the Laplace 1D problem with Dirichlet and Robin conditions

The left figure shows the interpolation nodes which form a uniform distribution of points. This is the default behavior.

Comparing the exact solution U_{ex} with the computed one U, at the interpolation abscissae, leads to $||U - U_{ex}||_\infty = 4.44278 \times 10^{-10}$, value which is currently printed by the program.
By adding **_FE_subtype**=GaussLobatto to the `Space` constructor, one can toggle to the Gauss-Lobatto abscissae which are more suitable with higher interpolation degrees. With this example, choosing these abscissae leads to a better approximation: we then get $||U - U_{ex}||_\infty = 2.55367 \times 10^{-11}$.

2 Laplace Problems

We investigate here problems involving laplacian operator in a 2D bounded domain, say Ω :

$$-\Delta u + a\,u = f \quad \text{in}\ \Omega \quad (a = -k^2 \text{ for Helmholtz equation})$$

and various essential conditions (Dirichlet, transmission, quasi periodic, average condition).

2.1 Neumann condition

First, let us consider the case of the homogeneous Neumann condition on $\partial\Omega$, the boundary of Ω:

$$\frac{\partial u}{\partial n} = 0 \quad \text{on}\ \partial\Omega.$$

The variational formulation we deal with is

$$\left|\begin{array}{l} \text{find } u \in V = \left\{v \in L^2(\Omega),\ \nabla v \in (L^2(\Omega))^2\right\} \text{ such that} \\ \displaystyle\int_\Omega \nabla u.\nabla v + a\int_\Omega u\,v = \int_\Omega f\,v \quad \forall v \in V. \end{array}\right.$$

The following main program corresponds to solving this problem on unity square $\Omega =]0,1[\times]0,1[$ with $f(x) = \cos\pi x \cos\pi y$ using P1 Lagrange element (20x20 elements):

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

Real cosx2(const Point& P, Parameters& pa = defaultParameters)
{
  Real x=P(1), y=P(2);
  return cos(pi_ * x) * cos(pi_ * y);
}

int main(int argc, char** argv)
{
  init(argc, argv, _lang=en);

  // mesh square
  SquareGeo sq(_origin=Point(0., 0.), _length=1, _nnodes=21);
  Mesh mesh2d(sq, triangle, 1, structured);
  Domain omega = mesh2d.domain("Omega");

  // build space and unknown
  Space Vk(_domain=omega, _interpolation=P1, _name="Vk");
  Unknown u(Vk, "u");
  TestFunction v(u, "v");

  // define variational formulation
  BilinearForm auv = intg(omega, grad(u) | grad(v)) + intg(omega, u * v);
  LinearForm fv=intg(omega, cosx2 * v);

  // compute matrix and right hand side
  TermMatrix A(auv, "a(u, v)");
  TermVector B(fv, "f(v)");
  // LLt factorize and solve
  TermMatrix LD;
  ldltFactorize(A, LD);
  TermVector U = factSolve(LD, B);

  saveToFile("U_LN", U, vtu);
  return 0;
}
```

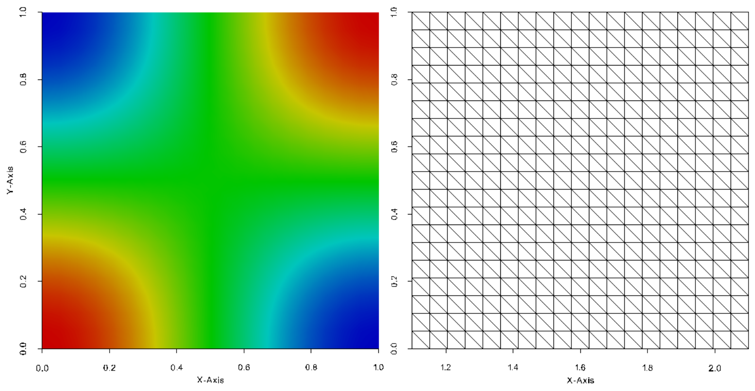


Figure 2.1: Solution of the Laplace 2D problem with Neumann condition on the square $[0,1]^2$

Solving this problem with P2 Lagrange interpolation should be the same except the line defining the space:

```
Space Vh(_domain=omega, _interpolation=P2, _name="Vh", _optimizeNumbering);
```

Solving this problem in a 3D domain should be the same except the line defining the mesh and the right hand side function. For instance, on the unity cube, the mesh construction command using GMSH tool is:

```
Real f(const Point& P, Parameters& pa = defaultParameters)
{
   Real x=P(1), y=P(2), z=P(3);
   return cos(pi*x) * cos(pi*y) * cos(pi*z);
}
...
Mesh mesh(Cube(_origin=Point(0.,0.,0.), _length=1, _nnodes=10), tetrahedron, 1, _gmsh,"P1 mesh");
...
```

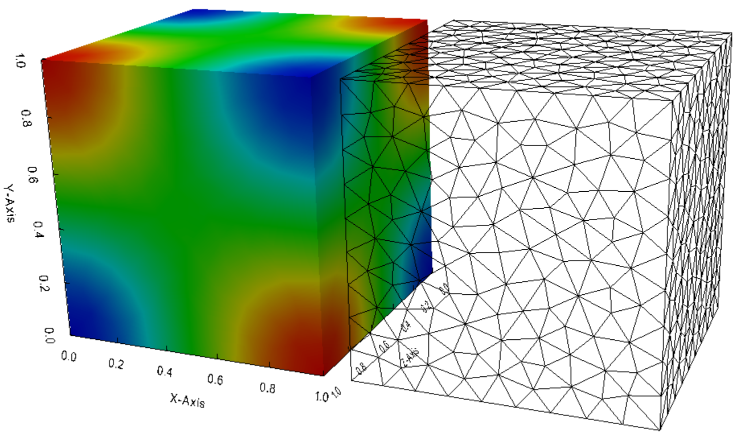


Figure 2.2: Solution of the Laplace 3D problem with Neumann condition on the unit cube $[0,1]^3$

2.2 Dirichlet condition

Let us consider now the case of non homogeneous Dirichlet condition on the boundaries $x = 0$ (Σ^-) and $x = 1$ (Σ^+):

$$u = 1 \text{ on } \Sigma^- \cup \Sigma^+.$$

The variational formulation is now ($a = 0$)

$$\left|\begin{array}{ll} \text{find } u \in V = \left\{v \in L^2(\Omega), \nabla v \in (L^2(\Omega))^2\right\} \text{ such that} & \\ \displaystyle\int_\Omega \nabla u.\nabla v = \int_\Omega f\, v & \forall v \in V,\ v = 0 \text{ on } \Sigma^- \cup \Sigma^+ \\ u = 1 & \text{on } \Sigma^- \cup \Sigma^+ \end{array}\right.$$

Its approximation by P1 Lagrange finite element is implemented in XLiFE++ as follows:

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

Real f(const Point& P, Parameters& pa = defaultParameters)
{return -8.;}

int main(int argc, char** argv)
{
  init(argc, argv, _lang=en); // mandatory initialization of xlifepp

  // create mesh of square
  Strings sn("y=0", "x=1", "y=1", "x=0");
  SquareGeo sq(_origin=Point(0., 0.), _length=1, _nnodes=20, _domain_name="Omega",
       _side_names=sn);
  Mesh mesh2d(sq, triangle, 1, structured);
  Domain omega=mesh2d.domain("Omega");
  Domain sigmaM=mesh2d.domain("x=0"), sigmaP=mesh2d.domain("x=1");

  // create interpolation
  Space V(_domain=omega, _interpolation=P1, _name="V");
  Unknown u(V, "u");
  TestFunction v(u, "v");

  // create bilinear form, linear form and their algebraic representation
  BilinearForm auv=intg(omega, grad(u)|grad(v));
  LinearForm fv=intg(omega, f*v);
  EssentialConditions ecs= (u|sigmaM = 1) & (u|sigmaP = 1);
  TermMatrix A(auv, ecs, "A");
  TermVector B(fv, "B");

  // solve linear system AX=B
  TermVector U=directSolve(A, B);
  saveToFile("U_LD", U, vtu);
  return 0;
}
```

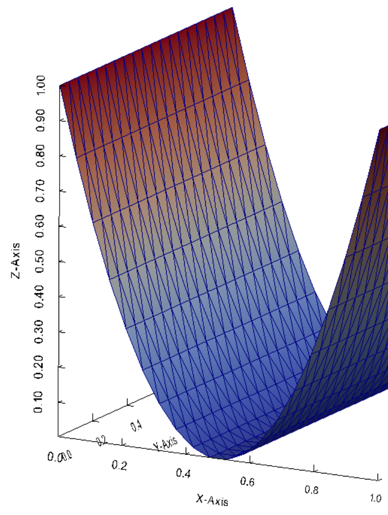


Figure 2.3: Solution of the Laplace 2D problem with Dirichlet condition on the unit square $[0,1]^2$

Note how easy is to take into account essential conditions. Only two lines has to be modified!

2.3 Periodic condition

Now we consider the Laplace problem on the unit square $\Omega =]0,1[\times]0,1[$ equipped with Dirichlet condition on and periodic condition:

$$\begin{cases} -\Delta u = f & \text{in } \Omega \\ u_{|\Gamma^-} = 0 \text{ and } u_{|\Gamma^+} = 0 & \\ u_{|\Sigma^-} = u_{|\Sigma^+} \text{ and } \partial_x u_{|\Sigma^-} = \partial_x u_{|\Sigma^+} & \end{cases}$$

and its variational formulation in $V = \{v \in H^1(\Omega),\ u_{|\Gamma} = 0 \text{ and } u_{|\Sigma^-} = u_{|\Sigma^+}\}$:

$$\left| \begin{array}{l} \text{find } u \in V \text{ such that} \\ \displaystyle\int_\Omega \nabla u.\nabla v = \int_\Omega f\, v \quad \forall v \in V. \end{array} \right.$$

Its approximation by P^2 Lagrange finite element is implemented in XLiFE++ as follows:

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

Real f(const Point& P, Parameters& pa = defaultParameters)
{
  Real x=P(1), y=P(2);
  return (4*pi_*pi_*y*(y-1)-2)*sin(2*pi_*x);
}

Vector<Real> mapPM(const Point& P, Parameters& pa = defaultParameters)
{
    Vector<Real> Q(P);
    Q(1)-=1;
    return Q;
}

int main(int argc, char** argv)
{
  init(argc, argv, _lang=en); // mandatory initialization of xlifepp

  // mesh square
  Strings sn("y=0", "x=1", "y=1", "x=0");
  SquareGeo sq(_origin=Point(0., 0.), _length=1, _nnodes=20, _domain_name="Omega",
       _side_names=sn);
  Mesh mesh2d(sq, triangle, 1, structured);
  Domain omega=mesh2d.domain("Omega");
  Domain sigmaM=mesh2d.domain("x=0"), sigmaP=mesh2d.domain("x=1");
  Domain gammaM=mesh2d.domain("y=0"), gammaP=mesh2d.domain("y=1");
  defineMap(sigmaP, sigmaM, mapPM);  // useful to periodic condition

  // create P2 Lagrange interpolation
  Space V(_domain=omega, _interpolation=P2, _name="V");
  Unknown u(V, "u");
  TestFunction v(u, "v");

  // create bilinear form and linear form
  BilinearForm auv=intg(omega, grad(u)|grad(v));
  LinearForm fv=intg(omega, f*v);
  EssentialConditions ecs = (u|gammaM = 0) & (u|gammaP = 0)
                          & ((u|sigmaP) - (u|sigmaM) = 0); // EssentialConditions ecs
  TermMatrix A(auv, ecs, "A");
  TermVector B(fv, "B");

  // solve linear system AX=F using factorization
  TermVector U=directSolve(A, B);
  saveToFile("U_LP", U, vtu);

  // Solving eigen problem intg(omega, grad(u)|grad(v))=lambda intg(omega, u*v) with ecs
  BilinearForm uv=intg(omega, u*v);
  TermMatrix Ae(auv, ecs, ReductionMethod(_pseudoReduction, 0., 1000.), "Ae");
  TermMatrix Me(uv, ecs, ReductionMethod(_pseudoReduction, 0., 1.), "Me");
  EigenElements eigs=eigenSolve(Ae, Me, _nev=10, _which="SM");
  Complex l1=eigs.value(1);          // first eigen value
  TermVector e1=eigs.vector(1);      // first eigen vector, "eliminated" components are not up to
      date
  e1.applyEssentialConditions(ecs);  // update "eliminated" components of e1
  eigs.applyEssentialConditions(ecs);// update "eliminated" components of all eigen vectors

  return 0;
}
```

Note that at corners, periodic condition and Dirichlet condition are redundant. When executing, the following warning message is thrown

```
Constraints::reduceConstraints() : in essential conditions
    Dirichlet condition u = 0 on y=0
    Dirichlet condition u = 0 on y=1
    periodic condition u|x=1 - u|x=0 = 0
2 redundant constraint row(s) detected and eliminated
```

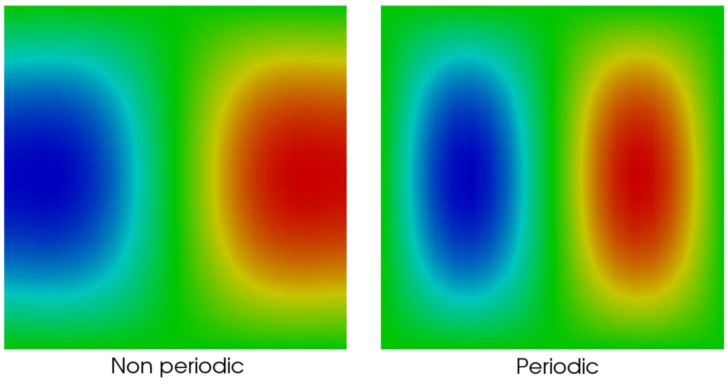
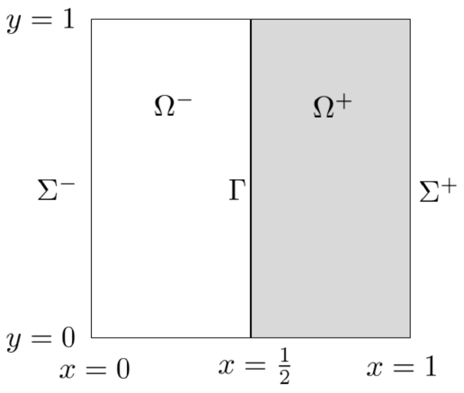


Figure 2.4: Solution of the Laplace 2D problem with periodic condition on the unit square $[0,1]^2$

2.4 Transmission condition



We turn to the Laplace problem with transmission condition:

$$\begin{cases} -\Delta u^- = f & \text{in } \Omega^- \\ -\Delta u^+ = f & \text{in } \Omega^+ \\ u_{|\Sigma^-} = 1 \text{ and } u_{|\Sigma^+} = 1 & \\ u^-_{|\Gamma} = u^+_{|\Gamma} \text{ and } \partial_x u^-_{|\Gamma} = \partial_x u^+_{|\Gamma} & \end{cases}$$

Its variational formulation in

$$V = \{(v^-, v^+) \in H^1(\Omega^-) \times H^1(\Omega^+),\ v^-_{|\Sigma^-} = 0, v^+_{|\Sigma^-} = 0,\ v^-_{|\Gamma} = v^+_{|\Gamma}\}$$

is

$$\left|\begin{array}{l} \text{find } (u^-, u^+) \in H^1(\Omega^-) \times H^1(\Omega^+), u^-_{|\Sigma^-} = 1, u^+_{|\Sigma^-} = 1,\ u^-_{|\Gamma} = u^+_{|\Gamma} \text{ such that} \\ \displaystyle\int_{\Omega^-} \nabla u^-.\nabla v^- + \int_{\Omega^+} \nabla u^+.\nabla v^+ = \int_{\Omega^-} f\, v^- + \int_{\Omega^+} f\, v^+ \quad \forall v \in V. \end{array}\right.$$

Note that derivatives matching is taken into account in a weak sense. The implementation in XLIFE++, using P^2 Lagrange finite element, looks like:

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

Real f(const Point& P, Parameters& pa = defaultParameters)
{
  return -8.;
}

int main(int argc, char** argv)
{
  init(argc, argv, _lang=en); // mandatory initialization of xlifepp

  // mesh domain
  Strings sn(4);
  sn[1] = "x=1/2-"; sn[3] = "x=0";
  Rectangle r1(_xmin=0, _xmax=0.5, _ymin=0, _ymax=1, _nnodes=Numbers(20, 40),
       _domain_name="Omega-", _side_names=sn);
  Mesh mesh2d(r1, triangle, 1, _structured);
  sn[1] = "x=1"; sn[3] = "x=1/2+";
  Rectangle r2(_xmin=0.5, _xmax=1, _ymin=0, _ymax=1, _nnodes=Numbers(20, 40),
       _domain_name="Omega+", _side_names=sn);
  Mesh mesh2d_p(r2, triangle, 1, structured);
  mesh2d.merge(mesh2d_p);
  Domain omegaM=mesh2d.domain("Omega-"), omegaP=mesh2d.domain("Omega+");
  Domain sigmaM=mesh2d.domain("x=0"), sigmaP=mesh2d.domain("x=1");
  Domain gamma=mesh2d.domain("x=1/2- or x=1/2+");

  // create P2 interpolation
  Space VM(_domain=omegaM, _interpolation=P2, _name="VM");
  Unknown uM(VM, "u-");
  TestFunction vM(uM, "v-");
  Space VP(_domain=omegaP, _interpolation=P2, _name="VP");
  Unknown uP(VP, "u+");
  TestFunction vP(uP, "v+");

  // create bilinear form and linear form
  BilinearForm auv=intg(omegaM, grad(uM)|grad(vM))+intg(omegaP, grad(uP)|grad(vP));
  LinearForm fv=intg(omegaM, f*vM)+intg(omegaP, f*vP);
  EssentialConditions ecs= (uM|sigmaM = 1) & (uP|sigmaP = 1) & ((uM|gamma) - (uP|gamma) = 0);
  TermMatrix A(auv, ecs, "A");
  TermVector B(fv, "B");

  // solve linear system AX=B using LU factorization
  TermVector U=directSolve(A, B);
  saveToFile("U_LT", U, vtu);
  return 0;
}
```

Here, a tool merging mesh is used to create a two domains mesh. GMSH should be used also. The picture below shows that the solution is continuous across the boundary Γ.

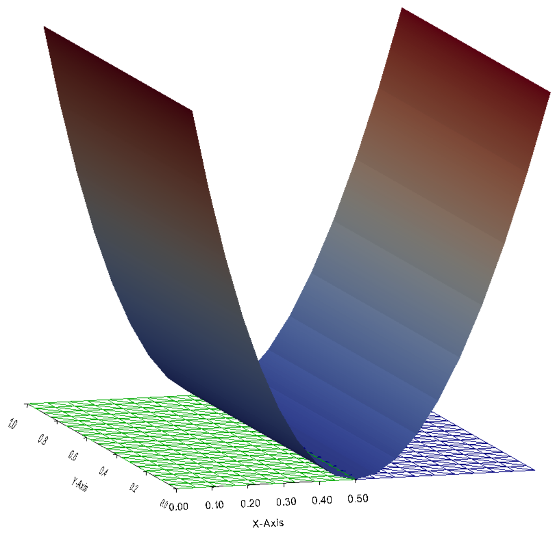


Figure 2.5: Solution of the Laplace 2D problem with transmission condition on the unit square $[0,1]^2$

2.5 Average condition

As a last example of essential condition, we consider average condition, for instance:

$$\int_\Sigma u = 0.$$

Such condition is tricky to take into account in FE softwares. Generally, they do not! Because XLIFE++ uses a powerful process to deal with essential conditions, such condition can be easily adressed:

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

Real f(const Point& P, Parameters& pa = defaultParameters)
{return -8.;}

int main(int argc, char** argv)
{
  init(argc, argv, _lang=en); // mandatory initialization of xlifepp

  // create a mesh and Domains
  SquareGeo sq(_origin=Point(0., 0.), _length=1, _nnodes=10, _domain_name="Omega",
      _side_names=Strings("y=0", "x=1", "y=1", "x=0"));
  Mesh mesh2d(sq, triangle, 1, structured);
  Domain omega=mesh2d.domain("Omega");
  Domain sigmaM=mesh2d.domain("x=0"), sigmaP=mesh2d.domain("x=1");

  // create interpolation
  Space V(_domain=omega, _interpolation=P2, _name"V");
  Unknown u(V, "u");
  TestFunction v(u, "v");

  // create bilinear form and linear form
  BilinearForm auv=intg(omega, grad(u)|grad(v));
  LinearForm fv=intg(omega, f*v);
  EssentialConditions ecs= (intg(sigmaM, u) = 0);
  TermMatrix A(auv, ecs, "A");
  TermVector F(fv , "B");

  // solve linear system AX=F using LU factorization
  TermVector U=directSolve(A, F);
  saveToFile("U_LA", U, vtu);
  return 0;
}
```

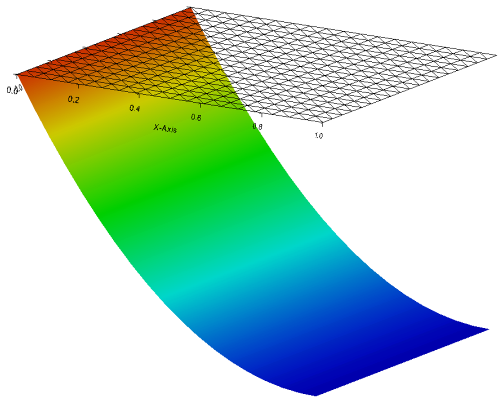


Figure 2.6: Solution of the Laplace 2D problem with average condition on the unit square $[0,1]^2$

Beware of some average conditions. For instance, when adding the "full" average condition

$$\int_\Omega u = 0$$

the resulting reduced matrix is a full matrix. So, the problem is bigger and slower to solve!

3 Discontinuous Galerkin method for 2D Dirichlet problem

Consider the Laplace problem with homogeneous Dirichlet condition:

$$\begin{cases} -\Delta u = f & \text{in } \Omega \\ u = 0 & \text{on } \partial\Omega \end{cases}$$

Consider a discontinuous Galerkin space V_h, e.g. discontinuous P^1-Lagrange space and let introduce the IP (Interior Penalty) formulation:

$$\left| \begin{array}{l} \text{find } u_h \in V_h \text{ such that} \\ \displaystyle\int_\Omega \nabla_h u_h.\nabla_h v - \int_\Gamma \{\nabla_h u_h.n\}\,[v] - \int_\Gamma [u_h]\,\{\nabla_h v.n\} + \int_\Gamma \mu [u_h]\,[v] = \int_\Omega f\,v \quad \forall v \in V_h. \end{array} \right.$$

where $\Omega = \bigcup T_\ell$, $\Gamma = \bigcup \partial T_\ell$ (the set of all sides of mesh elements) and, $S_{k,\ell}$ denoting the side shared by the elements T_k and T_ℓ:

$$\{v\}_{|S_{k\ell}} = \frac{1}{2}\left((v_{|T_k})_{|S_{k\ell}} + (v_{|T_\ell})_{|S_{k\ell}}\right) \quad [v]_{|S_{k\ell}} = \left((v_{|T_k})_{|S_{k\ell}} - (v_{|T_\ell})_{|S_{k\ell}}\right).$$

For a non shared side S_k, we set

$$\{v\}_{|S_k} = [v]_{|S_k} = (v_{|T_k})_{|S_k}.$$

The operators $\{.\}$ and $[.]$ are implemented in XLiFE++ as `mean(.)` and `jump(.)` operators. The normal vector n is chosen as the outward normal from T_k on side $S_{k\ell}$; in practice outward from the "first" parent element of the side. μ is a penalty function usually chosen to be proportional to the inverse of the measure of the side $S_{k\ell}$. Note that the Dirichlet boundary condition is a natural condition in this formulation.

To deal with a such formulation, the following objects have to be constructed from a geometrical domain, say `omega` :

- a FE space specifying $L2$ conformity (discontinuous space) defined on `omega`
- the geometrical domain `Gamma` of sides of `omega` using the function `sides(omega)`
- the bilinear form related to IP formulation and involving `mean(.)` and `jump(.)` operators

The XLiFE++ implementation of this problem using discontinuous P1-Lagrange is the following:

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

Real uex(const Point& p, Parameters& pars=defaultParameters)
{return sin(pi_*p(1))*sin(pi_*p(2));}
Real f(const Point& p, Parameters& pars=defaultParameters)
{return 2*pi_*pi_*sin(pi_*p(1))*sin(pi_*p(2));}
Real fmu(const Point& p, Parameters& pars=defaultParameters)
{GeomElement* gelt=getElementP();
 if(gelt!=0) return 1/gelt->measure();
 return 0.;
}

int main(int argc, char** argv)
```

```
{
  init(argc, argv, _lang=en); // mandatory initialization of xlifepp
  // create mesh and geometrical domains
  Rectangle R(_origin=Point(0.,0.),_xlength=1., _ylength=1.,_nnodes=31,_domain_name="Omega");
  Mesh mR(R,_triangle,1,_structured,_alternateSplit);
  Domain omega=mR.domain("Omega");
  Domain gamma=sides(omega);  // create domain of sides
  // create discontinuous P1-Lagrange space
  Interpolation in(Lagrange,standard,1,L2);
  Space V(omega,in,"V");
  Unknown u(V,"u");TestFunction v(u,"v");
  // create bilinear form and TermMatrix
  Function mu(fmu); mu.require("element");
  BilinearForm a=intg(omega,grad(u)|grad(v))-intg(gamma,mean(grad(u)|_n)*jump(v))
                 -intg(gamma,jump(u)*mean(grad(v)|_n))+intg(gamma,mu*jump(u)*jump(v));
  TermMatrix A(a,"A");
  // create the right hand side an solve the linear system
  TermVector F(intg(omega,f*v),"F");
  U=directSolve(A,F);
  saveToFile("U",U,_vtu);
  // compute L2 error
  TermMatrix M(intg(omega,u*v),"M");
  TermVector Uex(u,omega,uex);
  TermVector E=U-Uex;
  theCout<<"  IP : |U-uex|L2 = "<<sqrt(M*E|E)<<eol;
  return 0;
}
```

The L^2 error is about 0.00317 for 5400 dofs. Using Paraview with the *Warp by scalar* filter that produces elevation, the approximated field u looks like:

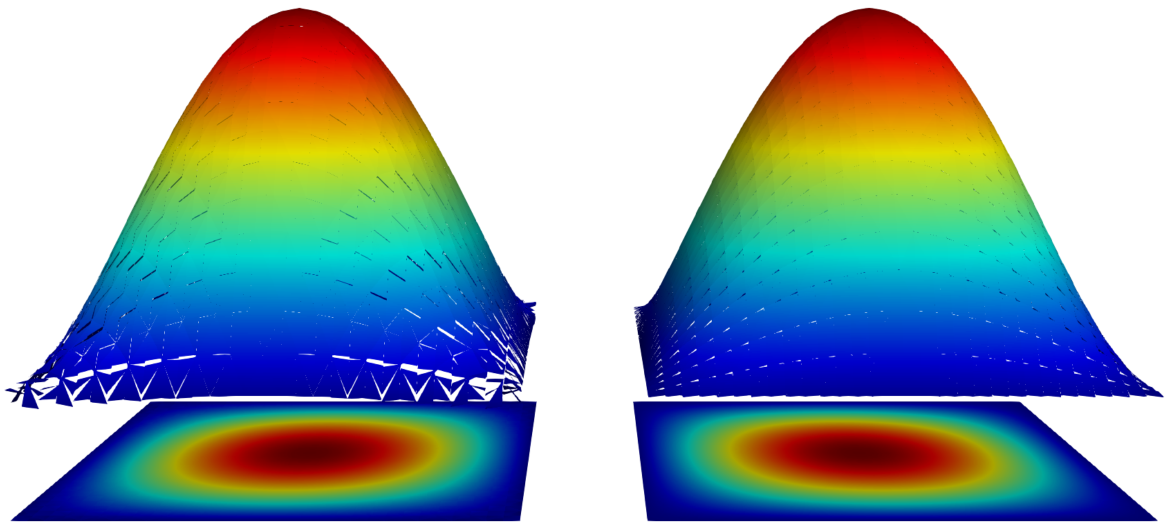


Figure 3.1: Solution of the 2D Dirichlet problem on the unit square $[0,1]^2$ with discontinuous P1-Lagrange elements, IP formulation (left) and NIPG formulation (right)

It can be noticed that the field is discontinuous and major errors is located on the corners. NIPG formulation is an other penalty method corresponding to the bilinear form:

$$\int_\Omega \nabla_h u_h.\nabla_h v - \int_\Gamma \{\nabla_h u_h.n\}\,[v] + \int_\Gamma [u_h]\,\{\nabla_h v.n\} + \int_\Gamma \mu[u_h]\,[v]$$

that is close to the IP formulation but non symmetric. For NIPG, the L^2 error is weaker (about 0.00141) and the solution is less polluted at the corners.

4 Mixed formulation using P0 and Raviart-Thomas elements

Consider the Laplace problem with homogeneous Dirichlet condition:

$$\begin{cases} -\Delta u = f & \text{in } \Omega \\ u = 0 & \text{on } \partial\Omega \end{cases}$$

Introducing $p = \nabla u$, it is rewritten as a mixed problem in (u, p):

$$\begin{cases} -\operatorname{div} p = f & \text{in } \Omega \\ p = \nabla u & \text{in } \Omega \\ u = 0 & \text{on } \partial\Omega \end{cases}$$

with the following variational formulation:

$$\left|\begin{array}{ll} \text{find } (u,p) \in L^2(\Omega) \times H(div,\Omega) \text{ such that} & \\ -\displaystyle\int_\Omega \operatorname{div} p\, v = \int_\Omega f\, v & \forall v \in L^2(\Omega) \\ \displaystyle\int_\Omega u \operatorname{div} q + \int_\Omega p.q = 0 & \forall q \in H(div,\Omega). \end{array}\right.$$

Note that the Dirichlet boundary condition is a natural condition in this formulation.

The XLIFE++ implementation of this problem using P0 approximation for $L^2(\Omega)$ and an approximation of $H(div,\Omega)$ using Raviart-Thomas elements of order 1 is the following:

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

Real f(const Point& P, Parameters& pa = defaultParameters)
{ Real x=P(1), y=P(2);
  return 32*(x*(1-x)+y*(1-y));}

int main(int argc, char** argv)
{
  init(argc, argv, _lang=en);
  // mesh square
  SquareGeo sq(_origin=Point(0., 0.), _length=1, _nnodes=21);
  Mesh mesh2d(sq, triangle, 1, structured);
  Domain omega=mesh2d.domain("Omega");
  // create approximation P0 and RT1
  Space H(_domain=omega, _interpolation=P0, _name="H", _notOptimizeNumbering);
  Space V(_domain=omega, _FE_type=RaviartThomas, _order=1, _name="V");
  Unknown p(V, "p");
  TestFunction q(p, "q");  // p=grad(u)
  Unknown u(H, "u");
  TestFunction v(u, "v");
  // create problem (Poisson problem)
  TermMatrix A(intg(omega, p|q) + intg(omega, u*div(q)) - intg(omega, div(p)*v));
  TermVector b(intg(omega, f*v));
  // solve and save solution
  TermVector X=directSolve(A, b);
  saveToFile("u", X(u), vtu);
  return 0;
}
```

Using Paraview with the *Cell data to point data* filter that moves P0 data to P1 data and the *Warp by scalar* filter that produces elevation, the approximated field u looks like:

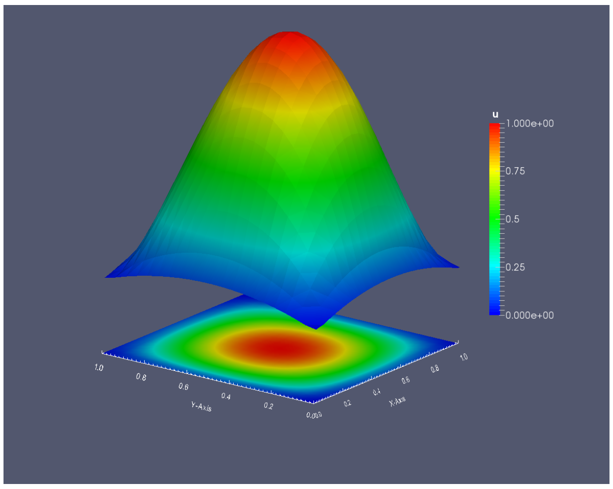
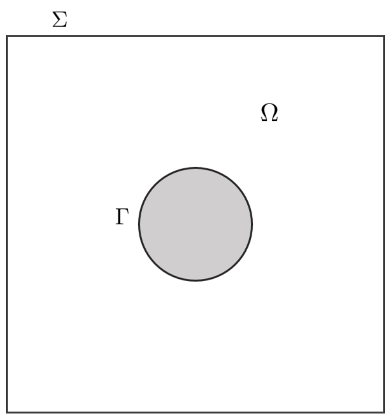


Figure 4.1: Solution of the Laplace 2D problem with mixed formulation P0-RT1 on the unit square $[0,1]^2$

5 Fictitious domain method

A well-known method to avoid some complex meshes consists in using the ficticious domain method where the boundary condition on an inside obstacle is taken into account by a Lagrange multiplier. To illustrate it, we consider the following 2D Laplace problem in the domain $\Omega = C\backslash B(O,R)$with $C=]0,1[\times]0,1[$ the unit square, $B(O,R)$ the ball with center C and radius R such that $B\subset C$:

$$\begin{cases} -\Delta u = f & \text{in } \Omega \\ u = 0 & \text{on } \partial C = \Sigma \\ u = g & \text{on } \partial B = \Gamma \end{cases}$$



Let us introduce the minimization problem:

$$\min_{\tilde v\in\tilde V(g)} \frac{1}{2}\int_C |\nabla \tilde V|^2 - \int_C \tilde f\,\tilde v$$

where

$$\tilde V(g) = \{\tilde v \in H^1(C),\ \tilde v_{|\Sigma} = 0 \text{ and } \tilde v_{|\Gamma} = g\} \text{ and } \tilde f = \begin{cases} f & \text{in } \Omega \\ 0 & \text{in } B \end{cases}.$$

It has a unique solution $\tilde u \in \tilde V(g)$ such that $\tilde u = u$ on Ω and there exists $p\in H^{-\frac{1}{2}}(\Gamma)$ such that

$$\begin{cases} \displaystyle\int_C \nabla\tilde u.\nabla\tilde v - \int_\Gamma p\,\tilde v = \int_C \tilde f\,\tilde v & \forall \tilde v \in H^1_0(C) \\ \displaystyle\int_\Gamma \tilde u\, q = \int_\Gamma g\, q & \forall q\in H^{-\frac{1}{2}}(\Gamma), \end{cases}$$

where the integrals on Γ are to be understood as the duality product on $H^{-\frac{1}{2}}(\Gamma)\times H^{\frac{1}{2}}(\Gamma)$.

The key idea of ficticious domain method is to use two different meshes for C and Γ. As a consequence, the computation of integrals on Γ involves shape functions that lie on two different meshes, inducing interpolation operations from one mesh to the other one. XLIFE++ manages this interpolation operations in an hidden way for the user:

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

Real R;

Real f(const Point& P, Parameters& pa = defaultParameters)
{if(norm(P)<=R) return 0.;
```

```
 return 2*pi_*pi_*sin(pi_*P(1))*sin(pi_*P(2));}
Real g(const Point& P, Parameters& pa = defaultParameters)
{return sin(pi_*P(1))*sin(pi_*P(2));}

int main(int argc, char** argv)
{
  init(argc, argv, _lang=en); // mandatory initialization of xlifepp
<<<<<<< HEAD
  //create meshes (rectangle and circle)
=======

  // create meshes (rectangle and circle)
>>>>>>> space_keyvalues
  R=0.2;
  SquareGeo sq(_xmin =0, _xmax = 1, _ymin = 0, _ymax = 1,
               _hsteps=0.05,
               _domain_name = "Omega", _side_names="Sigma");
  Disk circle(_center=Point(0.5, 0.5), _radius =R, _hsteps=0.05,
              _domain_name = "Gamma");
<<<<<<< HEAD
  Mesh meshR(rectangle,_triangle);
  Mesh meshG(circle,_segment, 1, gmsh);
  Domain rect =   meshR.domain("Omega"),
         sigmat = meshR.domain("Sigma"),
         gammat = meshG.domain("Gamma");
  //create spaces and unknowns
  Space Vt(rect,P1,"Vt"); Unknown ut(Vt,"ut"); TestFunction vt(ut,"vt");
  Space W(gammat,P0,"W"); Unknown p(W,"p"); TestFunction q(p,"q");
  //create bilinear forms and Terms
  BilinearForm at = intg(rect,grad(ut)|grad(vt))-intg(gammat,p*vt)
                  - intg(gammat,ut*q);
  BoundaryConditions bcst = (ut|sigmat =0);
  TermMatrix At(at,bcst,"At");
  TermVector Ft(intg(rect,f*vt)-intg(gammat,g*q),"Ft");
  //solve linear system and save the solution to a vtu file
  TermVector Ut = directSolve(At,Ft);
  saveToFile("Ut",Ut,_vtu);
=======
  Mesh meshS(sq, triangle);
  Mesh meshG(circle, segment, 1, gmsh);
  Domain omega  = meshS.domain("Omega"),
         sigmat = meshS.domain("Sigma"),
         gammat = meshG.domain("Gamma");

  // create spaces and unknowns
  Space Vt(_domain=omega, _interpolation=P1, _name="Vt");
  Unknown ut(Vt, "ut"); TestFunction vt(ut, "vt");
  Space W(_domain=gammat, _interpolation=P0, _name="W");
  Unknown p(W, "p"); TestFunction q(p, "q");

  // create bilinear forms and Terms
  BilinearForm at = intg(omega, grad(ut)|grad(vt))-intg(gammat, p*vt)
                  - intg(gammat, ut*q);
  BoundaryConditions bcst = (ut|sigmat =0);
  TermMatrix At(at, bcst, "At");
  TermVector Ft(intg(omega, f*vt)-intg(gammat, g*q), "Ft");

  // solve linear system and save the solution to a vtu file
  TermVector Ut = directSolve(At, Ft);
  saveToFile("Ut", Ut, vtu);

>>>>>>> space_keyvalues
  return 0;
```

```
}
```

To extract the solution on the real domain Ω, a L2 projection is used:

```
// create a mesh for Omega
Disk disk(_center=Point(0.5,0.5),_radius =R, _hsteps=0.05,
          _domain_name = "Obstacle", _side_names="Gamma");
Mesh mesh(rectangle-disk,_triangle);
Domain omega = mesh.domain("Omega");
Space V(omega,P1,"V"); Unknown u(V,"u"); TestFunction v(u,"v");

//compute L2 projection of ut on omega
TermMatrix M(intg(omega,u*v)), M2(intg(omega,ut*v));
TermVector PUt = directSolve(M,M2*Ut);
saveToFile("PUt",PUt,_vtu);
```

Note that the computation of the matrix `M2` also requires a computation of an integrals involving two meshes!

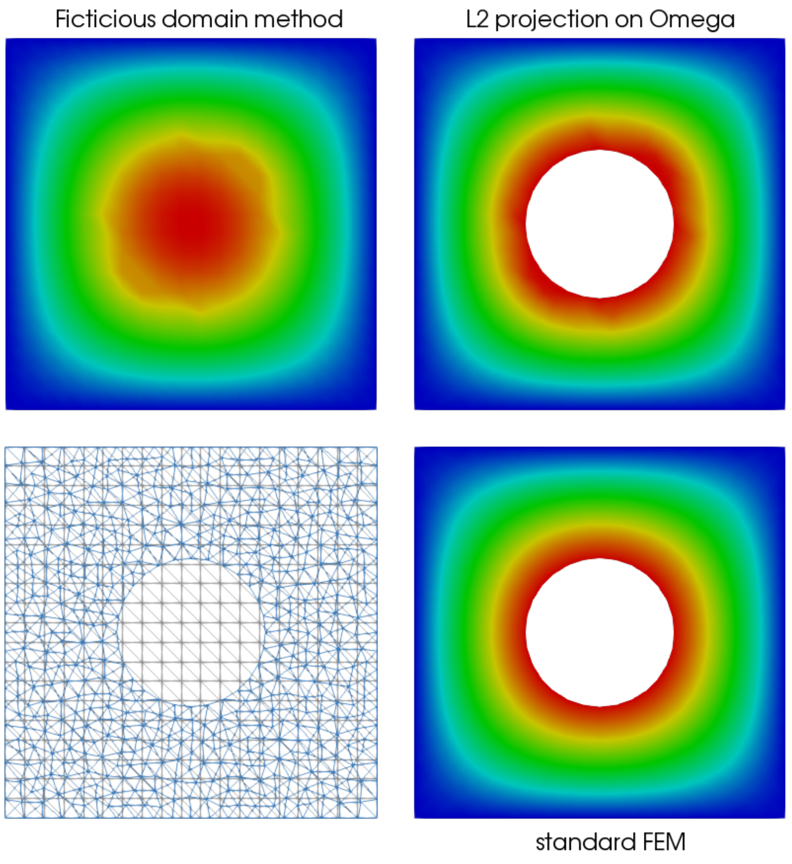


Figure 5.1: Solution of the Laplace 2D problem with the ficticious domain method

6 2D Maxwell equations using Nedelec elements

XLIFE++ provides Nedelec elements (first and second family) that are H(curl) comforming. Consider the following academic Maxwell problem:

$$\begin{cases} \operatorname{curl}\operatorname{curl}\mathbf{E} - \omega^2\mu\varepsilon\mathbf{E} = \mathbf{f} & \text{in } \Omega \\ \mathbf{E}\times n = 0 & \text{on } \partial\Omega \end{cases}$$

with the following weak form:

$$\left|\begin{array}{l} \text{find } \mathbf{E}\in V = \{\mathbf{v}\in H(curl,\Omega),\ \mathbf{v}\times n = 0 \text{ on } \partial\Omega\} \text{ such that} \\ \displaystyle\int_\Omega \operatorname{curl}\mathbf{E}\operatorname{curl}\mathbf{v} = \int_\Omega \mathbf{E}\mathbf{v} \quad \forall \mathbf{v}\in V. \end{array}\right.$$

Using first family Nedelec's element, the XLIFE++ program looks like:

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

Real omg=1, eps=1, mu=1, a=pi_, ome=omg*omg*mu*eps;

Vector<Real> f(const Point& P, Parameters& pa = defaultParameters)
{
  Real x=P(1), y=P(2);
  Vector<Real> res(2);
  Real c=2*a*a-ome;
  res(1)=-c*cos(a*x)*sin(a*y);
  res(2)= c*sin(a*x)*cos(a*y);
  return res;
}

Vector<Real> solex(const Point& P, Parameters& pa = defaultParameters)
{
  Real x=P(1), y=P(2);
  Vector<Real> res(2);
  res(1)=-cos(a*x)*sin(a*y);
  res(2)= sin(a*x)*cos(a*y);
  return res;
}

int main(int argc, char** argv)
{
  init(argc, argv, _lang=en);
  // mesh square using gmsh
  SquareGeo sq(_xmin=0, _xmax=1, _ymin=0, _ymax=1, _nnodes=50, _side_names="Gamma");
  Mesh mesh2d(sq, triangle, 1, gmsh);
  Domain omega=mesh2d.domain("Omega");
  Domain gamma=mesh2d.domain("Gamma");
  // define space and unknown
  Space V(_domain=omega, _FE_type=Nedelec, _order=1, _name="V");
  Unknown e(V, "E");
  TestFunction q(e, "q");
  // define forms, matrices and vectors
  BilinearForm aev=intg(omega, curl(e)|curl(q)) - ome*intg(omega, e|q);
  LinearForm l=intg(omega, f|q);
  EssentialConditions ecs = (ncross(e)|gamma=0);
  // compute
  TermMatrix A(aev, ecs, "A");
  TermVector b(l, "B");
  // solve
  TermVector E=directSolve(A, b);
  // P1 interpolation, L2 projection on H1
  Space W(_domain=omega, _interpolation=P1, _name="W");
  TermVector EP1=projection(E, W, 2);
  EP1.name("E");
  saveToFile("E", EP1, vtu);
  return 0;
}
```

As Nedelec finite elements approximation are not conforming in H1, the solution is not continuous across elements (only tangent component is continuous). So to represent the solution, it is projected on H1 as follows:

$$\left|\begin{array}{l} \text{find } \mathbf{E}_1 \in L^2(\Omega) \text{ such that} \\ \displaystyle\int_\Omega \mathbf{E}_1\,\mathbf{w} = \int_\Omega \mathbf{E}\,\mathbf{w} \quad \forall \mathbf{w} \in L^2(\Omega). \end{array}\right.$$

Using an H1 conforming approximation for $\mathbf{E}_1$ leads to a continuous representation of the projection. We show on the next figure the E_x component field provided by this example:

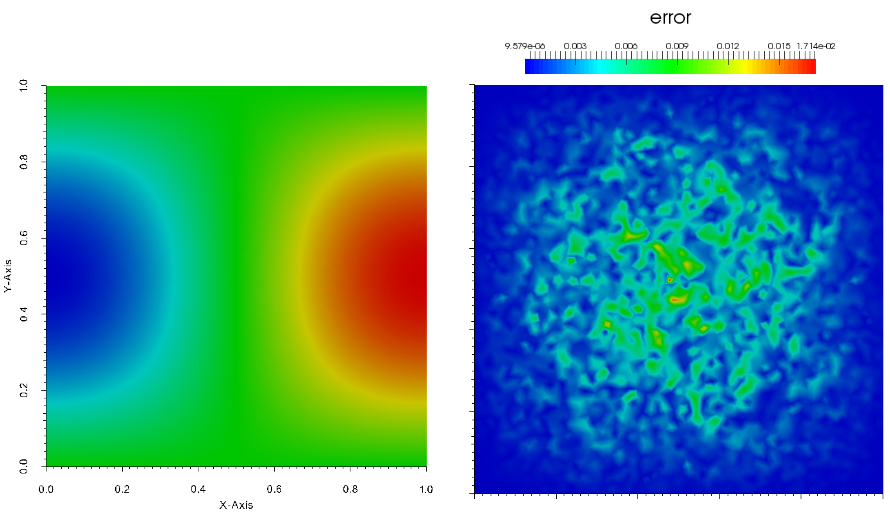


Figure 6.1: First component of the solution of the Maxwell 2D problem using Nedelec first family elements, and nodal error

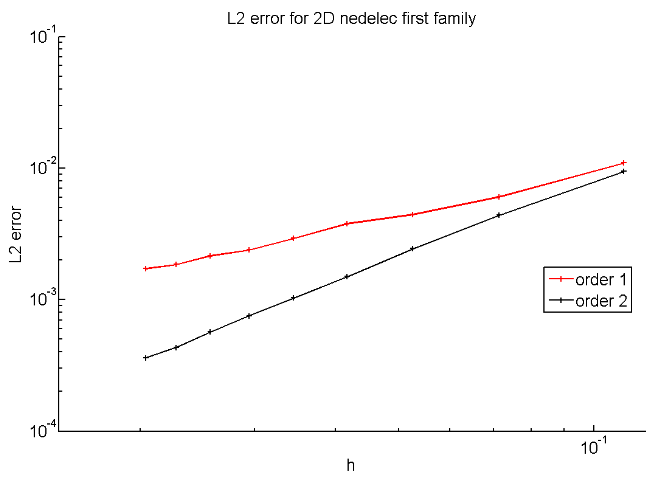


Figure 6.2: L^2 errors versus the step h for 1 and 2 order Nedelec first family approximation

7 Eigenvalues and eigenvectors of Laplace operator

This exemple shows how to get eigen functions of Laplace operator equipped with homogeneous Neumann condition:

$$\begin{cases} -\Delta u + u = \lambda u & \text{in } \Omega \\ \partial_n u = 0 & \text{on } \partial\Omega \end{cases}$$

and its variational formulation in $V = H^1(\Omega)$:

$$\left|\begin{array}{l} \text{find } (u,\lambda) \in V \times \mathbb{R} \text{ such that} \\ \displaystyle\int_\Omega \nabla u.\nabla v + \int_\Omega u\,v = \lambda \int_\Omega u\,v \quad \forall v \in V. \end{array}\right.$$

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

int main(int argc, char** argv)
{
  init(argc, argv, _lang=en); // mandatory initialization of xlifepp

  // mesh square
  SquareGeo sq(_origin=Point(0., 0.), _length=1, _nnodes=20);
  Mesh mesh2d(sq, triangle, 1, gmsh);
  Domain omega = mesh2d.domain("Omega");

  // build  P2 interpolation
  Space Vk(_domain=omega, _interpolation=P2, _name="Vk");
  Unknown u(Vk, "u");
  TestFunction v(u, "v");

  // build eigen system
  BilinearForm auv = intg(omega, grad(u) | grad(v)) + intg(omega, u * v) ,
               muv = intg(omega, u * v);
  TermMatrix A(auv, "auv"), M(muv, "muv");

  // compute the 10 first smallest in magnitude
  EigenElements eigs = eigenInternSolve(A, M, _nev=10, _mode=krylovSchur, _which="SM");    //
      internal solver
  theCout << eigs.values;
  saveToFile("eigs", eigs.vectors, vtu);
  return 0;
}
```

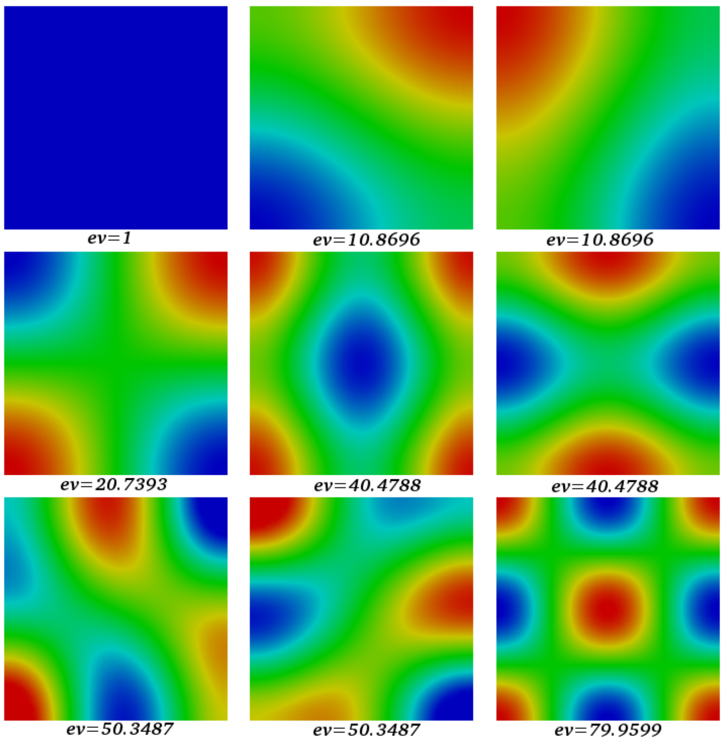


Figure 7.1: 9 first eigen vectors of the Laplace 2D problem with P2 elements

8 3D Helmholtz problem using single layer potential integral equation

XLIFE++ is also able to deal with integral equation. This example illustrates the computation of the acoustic scattering by a sphere:

$$\begin{cases} \Delta u + k^2 u = 0 & \text{in } \Omega = \mathbb{R}^3/B(0,R) \\ u = -u_{inc} & \text{on } S \end{cases}$$

Using single layer potential leads to the integral equation, :

$$\int_S G(x,y)p(x)dx = -u_{inc} \quad \text{on } S$$

where G is the Green function of the Helmhotz equation:

$$G(x,y) = \frac{e^{ik|x-y|}}{4\pi|x-y|}.$$

We deal with the variational formulation in $V = H^{\frac{1}{2}}(S)$:

$$\left| \begin{array}{c} \text{find } p \in V \text{ such that} \\ \displaystyle\int_S\int_S p(x)\,G(x,y)\,\bar{q}(y)\,dx\,dy = -\int_S u_{inc}\bar{q} \quad \forall q \in V. \end{array} \right.$$

The solution u is get from potential p from the integral representation:

$$u(x) = \int_S G(x,y)p(y)dy.$$

This example has been implemented in XLIFE++ using a P^0 Lagrange interpolation:

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

// incident plane wave
Complex uinc(const Point& p, Parameters& pa = defaultParameters)
{
  Real kx=pa("kx"), ky=pa("ky"), kz=pa("kz");
  Real kp=kx*p(1)+ky*p(2);
  return exp(i_*kp);
}

int main(int argc, char** argv)
{
  init(argc, argv, _lang=en); // mandatory initialization of xlifepp
  numberOfThreads(2);
  // define parameters and functions
  Parameters pars;
  pars << Parameter(1., "k");                  // wave number k
  pars << Parameter(1., "kx") << Parameter(0., "ky") << Parameter(0., "kz"); // kx, ky, kz
  pars << Parameter(1., "radius");             // disk radius
  Kernel G = Helmholtz2dKernel(pars);       // load Helmholtz2D kernel
  Function finc(uinc, pars);                // define right hand side function
  Function scatSol(scatteredFieldDiskDirichlet, pars);  // exact solution
  // meshing the unit disk
  Number npa=16;       // nb of points by diameter of disk
  Disk sp(_center=Point(0., 0.), _radius=1, _nnodes=npa, _domain_name="disk");
  Mesh mS(sp, segment, 1 , gmsh);
  Domain disk = mS.domain("disk");

  // Lagrange P0 space and unknown
  Space V1(_domain=disk, _interpolation=P1, _name="V1", _notOptimizeNumbering);
  Unknown u1(V1, "u1");  TestFunction v1(u1, "v1");

  // form definitions
  IntegrationMethods ims(Duffy, 5, 0., Gauss_Legendre, 5, 1., Gauss_Legendre, 4, 2.,
      Gauss_Legendre, 3 );
  BilinearForm blf0=intg(disk, disk, u1*G*v1, ims);
  LinearForm fv0 = -intg(disk, finc*v1);

 // compute matrix and right hand side and solve system
  TermMatrix A0(blf0, denseDualStorage, "A0");
  TermVector B0(fv0, "B0");
  TermVector U0 = directSolve(A0, B0);

  // integral representation on x plane (far from disk), using P1 nodes
  Number npp=20, npc=8*npp/10;
  Real xm=4., eps=0.0001;
  Point C1(0., -xm), C2(0., xm), C3(0., -xm);
  SquareGeo sqx(_center=Point(0., 0.), _length=4., _nnodes=npp, _domain_name="Omega");
  Disk dx(_center=Point(0., 0.), _radius=1.25, _nnodes=npc);
  Mesh mx0(sqx-dx, triangle, 1 , gmsh);
  Domain planx0 = mx0.domain("Omega");
  Space Wx(_domain=planx0, _interpolation=P1, _name="Wx", _notOptimizeNumbering);
  Unknown wx(Wx, "wx");
  TermVector U0x0=integralRepresentation(wx, planx0, intg(disk, G*u1), U0);
  TermMatrix Mx0(intg(planx0, wx*wx), "Mx0");

  // compare to exact solution
  TermVector solx0(wx, planx0, scatSol);
  TermVector ec0x0=U0x0 - solx0;
  theCout << "L2 error on x=0 plane: " << sqrt(abs((Mx0*ec0x0)|ec0x0)) << eol;

  // export solution to file
  saveToFile("U0", U0, vtu);
  saveToFile("U0x0", U0x0, vtu);
  return 0;
}
```

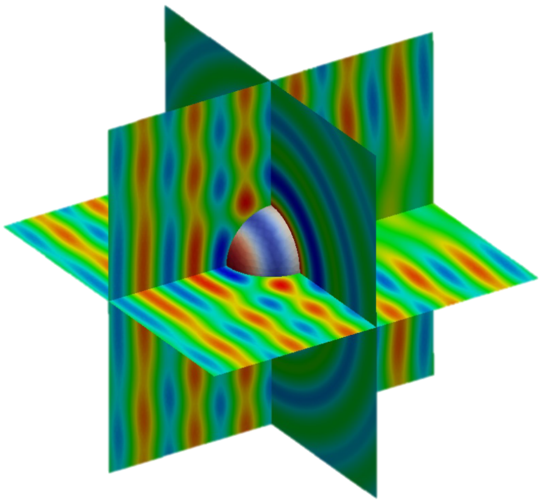


Figure 8.1: Solution of the 3D Helmholtz problem using single layer BEM on the unit sphere

9 2D Helmholtz problem coupling FEM and integral representation

We want to solve the acoustic diffraction of a plane wave on the disk of radius 1, with the boundary Γ:

$$\begin{cases} \Delta u + k^2 u = 0 & \text{in } \mathbb{R}^2/D \\ \partial_n u = g & \text{on } \Gamma \ (n \text{ the outward normal}) \end{cases}$$

where $g = \partial_n \left(e^{ikx}\right)$.

Let Ω be a domain that strictly surrounding the disk D and Σ its boundary. We have to point out that in this case, we use normals going outside the domain of computation Ω but then the normal on the obstacle (defined on Γ) is going inside the obstacle, that is opposite to usual case (see Figure 9.1). Then, because of the normal inverted, the solution u may be represented by the integral representation formula (G is the Green function related to the 2D Helmholtz equation in free space):

$$\forall x \in \Sigma, \quad u(x) = -\int_\Gamma \partial_{n_y} G(x,y)\, u(y)\, dy + \int_\Gamma G(x,y)\, \partial_{n_y} u(y)\, dy \tag{9.1}$$

say, because the boundary condition:

$$u(x) = -\int_\Gamma \partial_{n_y} G(x,y)\, u(y)\, dy + \int_\Gamma G(x,y)\, g(y)\, dy.$$

n_y is the outward normal (to Ω not the obstacle) on Γ and n_x will denote the outward normal on Σ. Now matching values and normal derivative on Σ, we introduce the boundary condition:

$$(\partial_{n_x} + \lambda) u(x) = -(\partial_{n_x} + \lambda) \int_\Gamma \partial_{n_y} G(x,y)\, u(y)\, dy + (\partial_{n_x} + \lambda) \int_\Gamma G(x,y)\, g(y)\, dy$$

that reads, because G is not singular on $\Gamma \times \Sigma$:

$$\begin{aligned} (\partial_{n_x} + \lambda) u(x) = & -\int_\Gamma \partial_{n_x} \partial_{n_y} G(x,y)\, u(y)\, dy - \lambda \int_\Gamma \partial_{n_y} G(x,y)\, u(y)\, dy \\ + & \int_\Gamma \partial_{n_x} G(x,y)\, g(y)\, dy + \lambda \int_\Gamma G(x,y)\, g(y)\, dy = \mathscr{R}_\lambda(u)(x) \end{aligned}$$

Using this exact boundary condition, if $Im(\lambda) \neq 0$) the initial problem is equivalent to :

$$\begin{cases} \Delta u + k^2 u = 0 & \text{in } \Omega \\ \partial_n u = g & \text{on } \Gamma \\ (\partial_{n_x} + \lambda) u = \mathscr{R}_\lambda(u) & \text{on } \Sigma \end{cases}$$

Its variational formulation in $V = H^1(\Omega)$ is:

$$\left| \begin{array}{l} \qquad\qquad \text{find } u \in V \text{ such that } \forall v \in V \\ \displaystyle\int_\Omega \nabla u . \nabla \bar{v} - k^2 \int_\Omega u\, \bar{v} + \lambda \int_\Sigma u\, \bar{v} + \int_\Sigma \int_\Gamma u(y) \partial_{n_x} \partial_{n_y} G(x,y)\, \bar{v}(x) + \lambda \int_\Sigma \int_\Gamma u(y) \partial_{n_y} G(x,y)\, \bar{v}(x) \\ \displaystyle\qquad = \int_\Gamma g\, \bar{v} + \int_\Sigma \int_\Gamma g(y) \partial_{n_x} G(x,y)\, \bar{v}(x) + \lambda \int_\Sigma \int_\Gamma g(y) G(x,y)\, \bar{v}(x). \end{array} \right.$$

Considering the geometrical configuration:

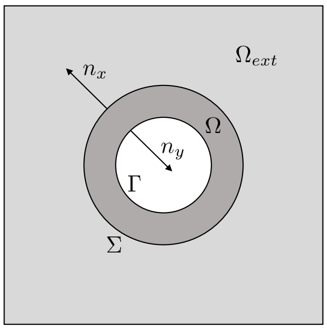


Figure 9.1: Geometrical configuration for the FEM-Integral Representation problem. The normal on Γ is going inside the obstacle (to point outside Ω).

the variational formulation is implemented as follows:

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

Complex data_g(const Point& P, Parameters& pa = defaultParameters)
{
  Real x=P(1), k=pa("k");
  Vector<Complex> g(2, 0.);
  g(1) = i_*k*exp(i_*k*x);
  return dot(g, P/norm2(P));  //dr(e^{ikx}
}

Complex u_inc(const Point& P, Parameters& pa = defaultParameters)
{
  Real x=P(1), k=pa("k");
  return exp(i_*k*x);
}

int main(int argc, char** argv)
{
  init(argc, argv, _lang=en); // mandatory initialization of xlifepp
  //parameters
  Number nh = 10;               // number of elements on Gamma
  Real h=2*pi_/nh;              // size of mesh
  Real re=1.+2*h;               // exterior radius
  Number ne=Number(2*pi_*re/h); // number of elements on Sigma
  Real l = 4*re;                // length of exterior square
  Number nr=Number(4*l/h);      // number of elements on exterior square
  Real k= 4, k2=k*k;            // wavenumber
  Parameters pars;
  pars << Parameter(k, "k") << Parameter(k2, "k2");
  Kernel H=Helmholtz2dKernel(pars);
  Function g(data_g, pars);
  Function ui(u_inc, pars);
  //Mesh and domains definition
  Disk d1(_center=Point(0., 0.), _radius=1, _nnodes=nh, _side_names=Strings(4, "Gamma"));
  Disk d2(_center=Point(0., 0.), _radius=re, _nnodes=ne, _domain_name="Omega",
      _side_names=Strings(4, "Sigma"));
  SquareGeo sq(_center=Point(0., 0.), _length=l, _nnodes=nr, _domain_name="Omega_ext");
  Mesh mesh(sq+(d2-d1), triangle, 1, gmsh);
  Domain omega=mesh.domain("Omega");
  Domain sigma=mesh.domain("Sigma");
  Domain gamma=mesh.domain("Gamma");
  Domain omega_ext=mesh.domain("Omega_ext");  //for integral representation
  sigma.setNormalOrientation(_outwardsDomain, omega); //outwards normals
  gamma.setNormalOrientation(_outwardsDomain, omega);
  //create P2 Lagrange interpolation
  Space V(_domain=omega, _interpolation=P2, _name="V");
  Unknown u(V, "u");  TestFunction v(u, "v");
  // create bilinear form and linear form
  Complex lambda=-i_*k;
  BilinearForm auv =
    intg(omega, grad(u)|grad(v))-k2*intg(omega, u*v)+lambda*intg(sigma, u*v)
    +intg(sigma, gamma, u*(grad_y(grad_x(H)|_nx)|_ny)*v)
    +lambda*intg(sigma, gamma, u*(grad_y(H)|_ny)*v);
  BilinearForm alv =
    intg(sigma, gamma, u*(grad_x(H)|_nx)*v)+lambda*intg(sigma, gamma, u*H*v);
  TermMatrix A(auv), ALV(alv);
  TermVector B(intg(gamma, g*v));
  TermVector G(u, gamma, g);
  B+=ALV*G;
  //solve linear system AU=F
  TermVector U=directSolve(A, B);
  saveToFile("U.vtk", U, vtk);
  //integral representation on omega_ext
  Space Vext(_domain=omega_ext, _interpolation=P2, _name="Vext", _notOptimizeNumbering);
  Unknown uext(Vext, "uext");
  TermVector Uext =
    -integralRepresentation(uext, omega_ext, intg(gamma, (grad_y(H)|_ny)*U))
    +integralRepresentation(uext, omega_ext, intg(gamma, H*G));
  saveToFile("Uext.vtk", Uext, vtk);
  //total field
  TermVector Ui(u, omega, ui), Utot=Ui+U;
  TermVector Uiext(uext, omega_ext, ui), Utotext=Uiext+Uext;
  saveToFile("Utot.vtk", Utot, vtk);
  saveToFile("Utotext.vtk", Utotext, vtk);
  return 0;
}
```

In the beginning, some geometric parameters used to design crown surrounded by a square, are given. Next the mesh is generated using gmsh mode and the geometrical domains are get from the mesh. The normal orientations are chosen in order to have outwards normals to the crown `omega`.

Then a P2 Lagrange space over the elements of the crown `omega` is constructed and all bilinear and linear forms involved in variational form are defined. Then the TermMatrix and TermVector are computed and the problem is solved using a direct method (Umfpack if it is installed, LU factorization else), that leads to the solution `U` in the crown `omega`.

Finally, using integral representation formula 9.1, the solution is computed in the exterior domain `omega_ext`. The vectors `U` and `Uext` are diffracted fields. To get total field, the incident field has to be added to the diffracted filed. This is the final job that it is done.

The real part of the total field computed is presented on the figure 9.2.

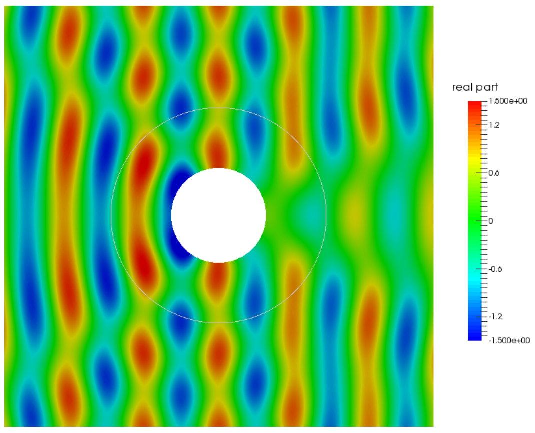


Figure 9.2: 2D Helmholtz diffraction problem using FE-IR method: real part of the total field

10 2D Helmholtz problem coupling FEM and BEM

We want to solve the acoustic propagation of a plane wave in a heterogeneous medium. In order to do that, we distinguish a domain Ω that is heterogeneous, its boundary Γ and the exterior domain Ω_{ext} that is homogeneous (see Figure 10.1).

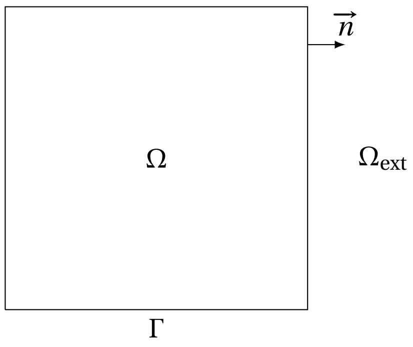


Figure 10.1: Domains for computation: Ω the heterogeneous medium, Ω_{ext} the homogeneous exterior domain and $\Gamma = \partial\Omega$.

We solve:

$$\begin{cases} \Delta u(x) + k^2\eta^2(x)u(x) = 0 & \text{in } \mathbb{R}^2 \\ u(x) = -u_i(x) & \text{on } \Gamma \end{cases}$$

with $\eta(x) = 1$ in Ω_{ext}, and $\eta(x)$ that can vary in Ω, and finally with $u_i = e^{ikx}$.
We will use: $\Omega = [-0.5, 0.5]^2$ and

$$\eta(x) = \begin{cases} \exp\left(-(x_1^2 - 0.25) * (x_2^2 - 0.25)/(2. * 0.05)\right), \text{ when } \max(x_1, x_2) < 0.5. \\ 1 \text{ otherwise.} \end{cases}$$

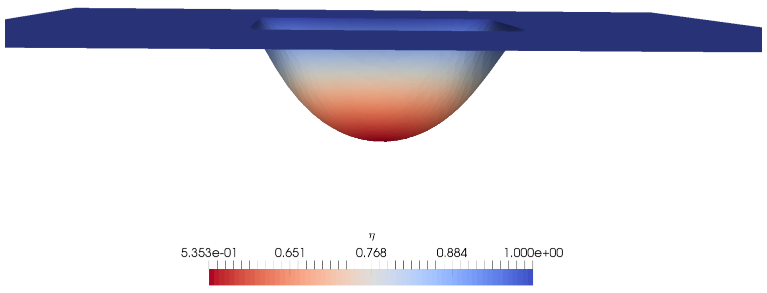


Figure 10.2: $\eta(x)$ in $\Omega \cup \Omega_{\text{ext}}$.

We decompose the problem in a coupled system of two equations:

- in the FEM part, the solution solves the following equation:

$$\Delta u + k^2 \eta^2 u = 0$$

which gives the variational formulation:

$$\left| \begin{array}{c} \text{Find } u \in H^1(\Omega) \text{ such that :} \\ \int_\Omega \nabla u(x) \cdot \nabla \bar{v}(x) dx - k^2 \int_\Omega \eta^2(x) u(x) \bar{v}(x) dx - \int_\Gamma \lambda(x) \bar{v}(x) dx = 0, \quad \forall v \in H^1(\Omega) \end{array} \right., \tag{10.1}$$

with $\lambda = \frac{\partial u}{\partial n}$ is the normal trace of u on Γ.

- in the BEM part, we solve:

$$\begin{cases} \Delta u + k^2 u = 0 & \text{in } \Omega_{\text{ext}} \\ u = -u_i & \text{on } \Gamma. \end{cases} \tag{10.2}$$

The scattered field verifies:

$$u_s(x) = -S_\Gamma \lambda(x) + K_\Gamma u(x), x \in \Omega_{\text{ext}}, \tag{10.3}$$

with u the total field solution of the equation and λ the normal trace of u on Γ, S_Γ and K_Γ are the single and double layer boundary potentials:

$$S_\Gamma \phi(x) = \int_\Gamma G(x,y)\phi(y) dy,$$
$$K_\Gamma \phi(x) = \int_\Gamma \frac{\partial G(x,y)}{\partial n_y} \phi(y) dy,$$

and

$$G(x,y) = \frac{e^{ik\|x-y\|}}{4\pi \|x-y\|}$$

Since $u_s = u - u_i$, and taking the limit when x goes to Γ, we obtain the integral equation:

$$\left(\frac{I}{2} - K_\Gamma \right) u(x) + S_\Gamma \lambda(x) = u_i(x), x \in \Gamma. \tag{10.4}$$

The resulting variational formulation for the BEM part is then:

$$\left| \begin{array}{l} \text{Find } u \in H^{1/2}(\Gamma) \text{ and } \lambda \in H^{-1/2}(\Gamma) \text{ such that :} \\ \frac{1}{2} \int_\Gamma u(x) \bar{\tau}(x) dx - \int_{\Gamma \times \Gamma} u(y) \frac{\partial G(x,y)}{\partial n_y} \bar{\tau}(x) dy dx + \int_{\Gamma \times \Gamma} \lambda(y) G(x,y) \bar{\tau}(x) dy dx \\ \qquad = \int_\Gamma u_i(x) \bar{\tau}(x) dx, \forall \tau \in H^{1/2}(\Gamma). \end{array} \right. \tag{10.5}$$

By adding the variational formulations relatives to the two linked problems, we obtain the final variational formulation.
Finally, the solution is obtained directly from u for the FEM part and we need to compute the integral representation to obtain u_s, the scattered field, and then to add the incident field to obtain the total field for this problem.

The last step is to merge the FEM solution in Ω and the BEM solution in Ω_{ext} to obtain a solution on the whole domain $\Omega \cup \Omega_{\text{ext}}$ to simplify the visualisation.

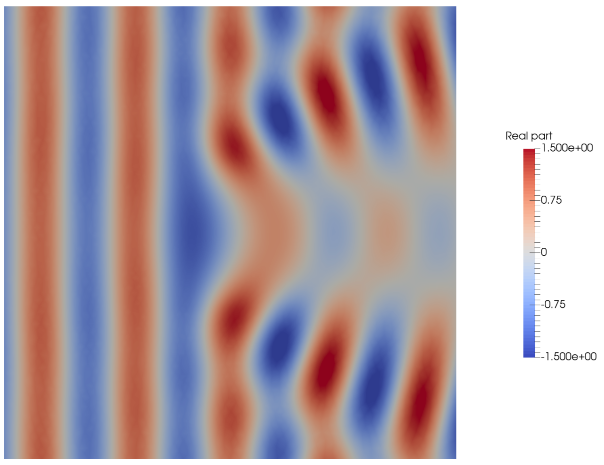


Figure 10.3: Solution of the FEM-BEM problem.

The code of this example follows:

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

// find = eta(x)
Real find(const Point & M, Parameters & pa = defaultParameters)
{
  Real res=1.;
  if (std::max(std::abs(M[0]), std::abs(M[1])) < 0.5)
    res=std::exp(-((M[0]*M[0]-0.25)*(M[1]*M[1]-0.25))/(2.*0.05));
  return res;
}
Real eta2(const Point & M, Parameters & pa = defaultParameters)
{
  Real tmp=find(M);
  return tmp*tmp;
}
Complex g1(const Point& M, Parameters& pa = defaultParameters)
{
  Real k=real(pa("k"));
  Point d(1., 0.);
  return exp(i_*(k*dot(M, d)));
}

int main(int argc, char** argv)
{
  init(argc, argv, _lang=en);    // mandatory initialization of xlifepp
  verboseLevel(10);
  Real k=10.;
  // meshing
  Real hsize=(2*pi_/k)/15.;
  SquareGeo sp(_center=Point(0., 0.), _length=1., _hsteps=hsize, _domain_name="Omega",
      _side_names="Gamma");
  Mesh m1=Mesh(sp, triangle, 1, gmsh);
  Domain omega = m1.domain("Omega");
  Domain gamma = m1.domain("Gamma");
  theCout << "Mesh size = " << hsize << eol;
  theCout << "Number of Triangles = " << m1.nbOfElements() << eol;
  // defining parameter and kernel
  Parameters pars;
  pars << Parameter(k, "k");
  Vector<Real> nv(2);
  pars << Parameter(&nv, "_n");
  Kernel G=Helmholtz2dKernel(pars);
  Function finc(g1, pars);
  // defining space, unknown and test function
  Space V1(_domain=omega, _interpolation=P1, _name="V1", _notOptimizeNumbering);
  Space V0(_domain=gamma, _interpolation=P1, _name="V0", _notOptimizeNumbering);
  Unknown u1(V1, "u1"); TestFunction v1(u1, "v1");
  Unknown l0(V0, "l0"); TestFunction lt0(l0, "lt0");
  theCout << "Nb dofs BEM= " << V0.nbDofs() << " Nb dofs FEM= " << V1.nbDofs() << eol;
  // defining bilinear and linear form
  IntegrationMethods ims(Duffy, 15, 0., _defaultRule, 12, 1., _defaultRule, 10, 2., _defaultRule,
      8);
  BilinearForm blf=intg(omega, grad(u1)|grad(v1))-k*k*intg(omega, eta2*u1*v1)
                   - intg(gamma, l0*v1) + 0.5*intg(gamma, u1*lt0)
                   - intg(gamma, gamma, u1*ndotgrad_y(G)*lt0, ims)
                   + intg(gamma, gamma, l0*G*lt0, ims);
  LinearForm lf=intg(gamma, finc*lt0);
  // computing FEM/BEM matrix and right hand side vector
  TermMatrix lhs(blf, "lhs");
  TermVector rhs(lf);
  // solving linear system using direct method
  TermVector sol=directSolve(lhs, rhs);

  // Representing the solution FEM and BEM
  SquareGeo Sint(_center=Point(0., 0.), _length=1, _hsteps=hsize, _domain_name="S_int");
  SquareGeo Sext(_center=Point(0., 0.), _length=3, _hsteps=1.5*hsize, _domain_name="S_ext");
  Mesh mrep(Sext+Sint, _triangle, 1, _gmsh);
  Domain S_ext=mrep.domain("S_ext"), S_int=mrep.domain("S_int");
  Domain S=merge(S_ext, S_int, "S");
  Space Vrep(_domain=S, _interpolation=P1, _name="Vrep", _notOptimizeNumbering);
  Unknown ur(Vrep, "ur");
  Function Find(find, pars);
  TermVector findex(ur, S, Find);
  saveToFile("findex", findex, _vtu); // Representing eta
  TermVector Uint=interpolate(ur, S_int, sol(u1)); // FEM solution (total field)
  saveToFile("Uint", Uint, _vtu);

  // Representing of the BEM part
  IntegrationMethods imr(_GaussLegendreRule, 20, 1., _GaussLegendreRule, 10, 2.,
      _GaussLegendreRule, 5);
  TermVector Uext =
      - integralRepresentation(ur, S_ext, intg(gamma, G*sol(l0), imr))
      + integralRepresentation(ur, S_ext, intg(gamma, ndotgrad_y(G)*sol(u1), imr));

  TermVector Uinc(ur, S_ext, finc);
  saveToFile("Uinc", Uinc, vtu); // Incident field
  saveToFile("Uext", Uext, vtu); // scattered field in exterior domain
  TermVector Uext_t = Uext + Uinc;
  saveToFile("Uext_t", Uext_t, vtu); // Total field in exterior domain
  TermVector U=merge(Uint, Uext_t); // Merged FEM and BEM solutions
  saveToFile("U", U, vtu);
  theCout << "Program finished" << eol;
  return 0;
}
```

11 3D Maxwell problem using EFIE

Solving diffraction of an electromagnetic plane wave on a obstacle using BEM is more intricate. Indeed, it is a vector problem and it involves Raviart-Thomas elements. We show how XLIFE++ can deal easily with.

Let Γ be the boundary of a bounded domain Ω of $\mathbb{R}^3$, we want to solve the Maxwell problem on the exterior domain Ω_e:

$$\left\{\begin{array}{ll}\operatorname{curl}\mathbf{E}-ik\mathbf{H}=0 & \text{in } \Omega_e\\ \operatorname{curl}\mathbf{H}+ik\mathbf{E}=0 & \text{in } \Omega_e\\ \mathbf{E}\times\mathbf{n}=0 & \text{on } \Gamma\\ \displaystyle\lim_{|x|\to\infty}\left((\mathbf{H}-\mathbf{H}_{inc})\times\frac{x}{|x|}-(\mathbf{E}-\mathbf{E}_{inc})\right)=0 & \text{(Silver-Muller condition)}\end{array}\right.$$

where $(\mathbf{E}_{inc},\mathbf{H}_{inc})$ is an incident field (a solution of Maxwell equation in free space), for instance a plane wave.

The EFIE (Electric Field Integral Equation) consists in finding the potential $\mathbf{J}$ in the space

$$H_{\text{div}}(\Gamma)=\left\{\mathbf{V}\in L^2(\Gamma)^3,\mathbf{V}.\mathbf{n}=0,\operatorname{div}_\Gamma\mathbf{V}\in L^2(\Gamma)\right\}$$

such that, $\forall\mathbf{V}\in H_{\text{div}}(\Gamma)$

$$k\int_\Gamma\int_\Gamma\mathbf{J}(y)\,G(x,y).\mathbf{V}(x)-\frac{1}{k}\int_\Gamma\int_\Gamma\operatorname{div}_\Gamma\mathbf{J}(y)\,G(x,y)\operatorname{div}_\Gamma\mathbf{V}(x)=-\int_\Gamma\mathbf{E}_{inc}.\mathbf{V}$$

where G is the Green function related to the Helmholtz 3D problem in free space.

This equation has a unique solution, except for a discrete set of wavenumbers corresponding to the resonance frequencies of the cavity Ω.

Using the Stratton-Chu representation formula, the scattered electric field may be reconstructed in Ω_e:

$$\mathbf{E}(x)=\mathbf{E}_{inc}(x)+\frac{1}{k}\int_\Gamma\nabla_x G(x,y)\operatorname{div}_\Gamma\mathbf{J}(y)+k\int_\Gamma G(x,y)\,\mathbf{J}(y).$$

This problem is implemented in XLIFE++ as follows:

```
#include "xlife++.h"
using namespace xlifepp;
Vector<complex_t> data_incField(const Point& P, Parameters& pars)
{
  Vector<real_t> incPol(3, 0.); incPol(1)=1.; Point incDir(0., 0., 1.) ;
  Real k = pars("k");
  return  incPol * exp(i_*k * dot(P, incDir));
}

Vector<complex_t> uinc(const Point& P, Parameters& pars)
{
    Vector<real_t> incPol(3, 0.); incPol(1)=1.; Point incDir(0., 0., 1.) ;
    Real k = pars("k");
    return  incPol*exp(i_*k * dot(P, incDir));
}

int main(int argc, char** argv)
```

```
{
  init(argc, argv, _lang=en);
  // define parameters and functions
  Real k= 1, R=1.; Parameters pars;
  pars << Parameter(k, "k") << Parameter(R, "radius");
  Kernel H = Helmholtz3dKernel(pars);           // load Helmholtz3D kernel
  Function Einc(data_incField, pars);           // define right hand side
  Function Uex(scatteredFieldMaxwellExn, pars);
  // meshing the unit sphere
  Number npa=15; Point O(0, 0, 0);
  Sphere sphere(_center=O, _radius=R, _nnodes=npa, _domain_name="Gamma");
  Mesh meshSh(sphere, triangle, 1, gmsh);
  Domain Gamma = meshSh.domain("Gamma");
  // define FE-RT1 space and unknown
  Space V_h(_domain=Gamma, _interpolation=RT_1, _name="Vh");
  Unknown U(V_h, "U"); TestFunction V(U, "V");
  // compute BEM system and solve it
  IntegrationMethods ims(_SauterSchwabIM, 4, 0., _defaultRule, 5, 2., _defaultRule, 3);
  BilinearForm auv = k*intg(Gamma, Gamma, U*H|V, ims)
                  -(1./k)*intg(Gamma, Gamma, div(U)*H*div(V), ims);
  TermMatrix A(auv, "A");
  TermVector B(-intg(Gamma, Einc|V));
  TermVector J = directSolve(A, B);
  // get P1 representation of solution and export it to vtu file
  Space L_h(_domain=Gamma, _interpolation=P1, _name="Lh");
  Unknown U3(L_h, "U3", 3); TestFunction V3(U3, "V3");
  TermVector JP1=projection(J, L_h, 3, _L2Projector);
  saveToFile("JP1", JP1(U3[1]), vtu);
  // integral representation on y=0 plane (excluding sphere), using P1 nodes
  Number npp=30, npc=5;
  Square sqx(_center=O, _length=20., _nnodes=npp, _domain_name="Omega");
  Disk dx(_center=O, _radius=1.2*R, _nnodes=npc);
  Mesh mx0(sqx-dx, triangle, 1, gmsh);
  mx0.rotate3d(1., 0., 0., pi_/2);
  Domain py0 = mx0.domain("Omega");
  Space Vy0(_domain=py0, _interpolation=P1, _name="Vy0", _notOptimizeNumbering);
  Unknown W(Vy0, "W", 3);
  IntegrationMethods im(_defaultRule, 10, 1., _defaultRule, 5);
  TermVector Uext=
   (1./k)*integralRepresentation(W, py0, intg(Gamma, grad_x(H)*div(U), im), J)
      + k*integralRepresentation(W, py0, intg(Gamma, H*U, im), J);
  saveToFile("Uext", Uext, vtu);
  // build exact solution, export to vtu file and compute error
  TermVector Uexa(W, py0, Uex);
  saveToFile("Uexa", Uexa, vtu);
  TermMatrix M(intg(py0, W|W));
  TermVector E=Uext-Uexa;
  theCout << "L2 error = " << sqrt(real(M*E|E)) << eol;
  return 0;
}
```

In order to build an approximated space of $H_{\text{div}}(\Gamma)$ we use the Raviart-Thomas element of order 1.
As the integrals involved in bilinear form are singular, we use here the Sauter-Schwab method to compute them when two triangles are adjacent, a quadrature method of order 5 if the two triangles are close ($0 < d(T1,T2) < 2h$) and a quadrature method of order 3 when the two triangles are far ($d(T1,T2) >= 2h$).
Note that the unknowns in RT approximation are the normal fluxes on the edge of the triangulation. In order to plot the potential $\mathbf{J}$, we have to move to a P1 representation, say $\widetilde{\mathbf{J}}$. This can be done using a L2 projection from $H_{\text{div}}(\Gamma)$ to $L^2(\Gamma)$:

$$\int_\Gamma \widetilde{\mathbf{J}}|\mathbf{V} = \int_\Gamma \mathbf{J}|\mathbf{V} \quad \forall V \in L^2(\Gamma)$$

This is what is done by the XLIFE++ function projection.
We obtain the following potential:

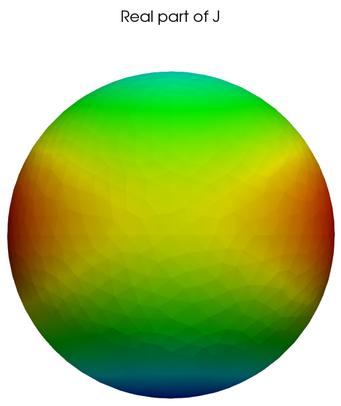


Figure 11.1: 3D Maxwell problem on the unit sphere, using EFIE, potential

On the following figures, we show the approximated electric field and the exact electric field. The component E_y is not shown because it is zero.

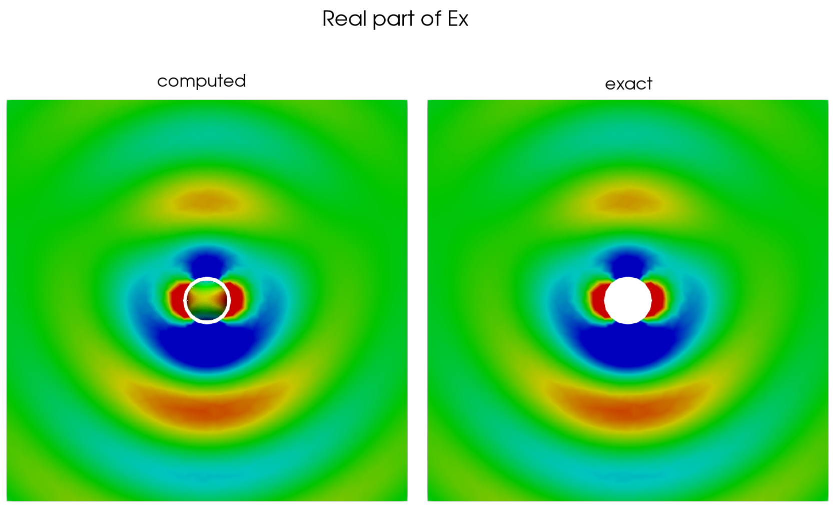


Figure 11.2: 3D Maxwell problem on the unit sphere, using EFIE, x component

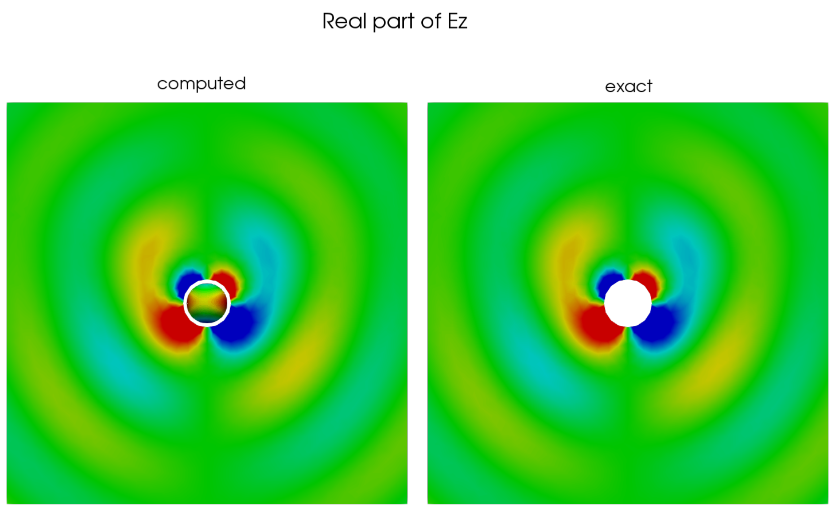


Figure 11.3: 3D Maxwell problem on the unit sphere, using EFIE, y component

12 2D Elasticity problem

The elasticity problem illustrates how to use vector unknown in XLIFE++:

$$\begin{cases} -\text{div}(\sigma(\mathbf{u}) - \omega^2\mathbf{u} = \mathbf{f} & \text{in } \Omega \\ \sigma(\mathbf{u})\mathbf{n} = 0 & \text{on } \partial\Omega \end{cases}$$

For homogeneous isotropic material:

$$\sigma(\mathbf{u}) = \lambda \text{div}(\mathbf{u})\mathbb{I} + 2\mu\varepsilon(\mathbf{u})\ \varepsilon_{ij}(\mathbf{u}) = \partial_i u_j.$$

The variational formulation in $V = (H^1(\Omega))^3$ is:

$$\left|\begin{array}{c} \text{find } \mathbf{u} \in V \text{ such that} \\ \lambda \int_\Omega \varepsilon(\mathbf{u}) : \varepsilon(\bar{\mathbf{v}}) + 2\mu \int_\Omega \text{div}(\mathbf{u})\,\text{div}(\bar{\mathbf{v}}) - \omega^2 \int_\Omega u\,\bar{v} = \int_\Omega \bar{\mathbf{f}}.\bar{\mathbf{v}} \quad \forall \mathbf{v} \in V. \end{array}\right.$$

This is implemented as follows:

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

// data function
Vector<Real> f(const Point& P, Parameters& pa = defaultParameters)
{  Vector<Real> F(2, 0.); F(2)=-0.005; return F;}

int main(int argc, char** argv)
{
  init(argc, argv, _lang=en); // mandatory initialization of xlifepp

  // mesh rectangle
  Rectangle rect(_center=Point(0., 0.), _xlength=20, _ylength=2, _nnodes=Numbers(50, 5),
      _domain_name="Omega", _side_names=Strings("", "", "", "Gamma"));
  Mesh mesh2d(rect, triangle, 1, gmsh);
  Domain omega=mesh2d.domain("Omega"), Gamma=mesh2d.domain("Gamma");
  // create P1 Lagrange interpolation
  Space V(_domain=omega, _interpolation=P1, _name="V");
  Unknown u(V, "u", 2);  TestFunction v(u, "v");
  // create bilinear form and linear form
  Real lambda=112.134, mu=83.53, omg2=0, rho=7.86;
  BilinearForm auv = lambda*intg(omega, epsilon(u) % epsilon(v))
                   + 2*mu*intg(omega, div(u)*div(v)) - omg2*intg(omega, u|v);
  LinearForm fv=intg(omega, f|v);
  EssentialConditions ecs= (u|Gamma=0);
  TermMatrix A(auv, ecs, "A");
  TermVector B(fv, "B");
  //solve linear system AX=B using direct method
  TermVector U=directSolve(A, B);
  thePrintStream<<U;
  saveToFile("U", U, vtu);

  // create the deformation of the mesh
  for (number_t i=0;i<mesh2d.nbOfNodes();i++)
    mesh2d.nodes[i] += U.evaluate(mesh2d.nodes[i]).value<std::vector<Real> >();
  mesh2d.saveToFile("Ud", msh);
  return 0;
}
```

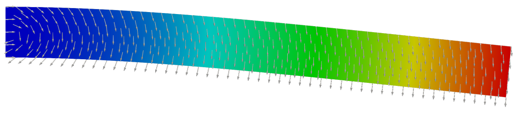


Figure 12.1: Displacement and modulus of the solution of the elasticity 2D problem

13 2D Bilaplacian problem

The 2d bilaplacian problem illustrates how to use Morley or Argyris element in XLIFE++:

$$\begin{cases} \Delta^2 u = \mathbf{f} & \text{in } \Omega \\ u = 0 \text{ and } \nabla u.n = 0 & \text{on } \partial\Omega \end{cases}$$

The variational formulation in $V = \{v \in H^2(\Omega),\ u = 0 \text{ and } \nabla u.n = 0 \text{ on } \partial\Omega\}$ is:

$$\left| \begin{array}{l} \text{find } \mathbf{u} \in V \text{ such that} \\ \displaystyle\int_\Omega \left(\partial_{xx} u \partial_{xx} v + \partial_{yy} u \partial_{yy} v + 2\partial_{xy} u \partial_{xy} v\right) = \int_\Omega f\, v \quad \forall \mathbf{v} \in V. \end{array} \right.$$

The implementation in XLIFE++ using Morley elements is the following :

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

// data function
Real uex(const Point& P, Parameters& pa = defaultParameters)
{
  Real x=P(1), y=P(2), kp=pi_;
  Real r=sin(kp*x)*sin(kp*y);
  return r*r;
}

Real f(const Point& P, Parameters& pa = defaultParameters)
{
  Real x=P(1), y=P(2);
  Real dkp=2*k*pi_;
  Real cx=cos(dkp*x), cy=cos(dkp*y);
  return 0.25*dkp*dkp*dkp*dkp*(4*cx*cy-cx-cy);
}

int main(int argc, char** argv)
{
  init(argc, argv, _lang=en); // mandatory initialization of xlifepp
  // mesh rectangle
  SquareGeo sq(_xmin=0, _xmax=1, _ymin=0, _ymax=1., _hsteps=0.05, _domain_name="Omega",
      _side_names="Gamma");
  Mesh mesh(sq, triangle, 1, gmsh);
  Domain omega=mesh.domain("Omega"), gamma=mesh.domain("Gamma");
  // create space
  Space V(_domain=omega, _FE_type=Morley, _order=1, _name="V");
  Unknown u(V, "u"); TestFunction v(u, "v");
  // create problem
  EssentialConditions ecs= (u|gamma = 0) & (ndotgrad(u)|gamma = 0);
  TermMatrix A(intg(omega, dxx(u)*dxx(v))+intg(omega, dyy(u)*dyy(v))+2*intg(omega,
      dxy(u)*dxy(v)), ecs);
  TermVector B(intg(omega, f*v), "B");
  // solve problem
  TermVector U=directSolve(A, B);
  // interpolate on P1 and export to vtu file
  Space W(_domain=omega, _interpolation=P1, _name="W", _notOptimizeNumbering);
  Unknown w(W, "w");
  TermVector Up1=interpolate(w, omega, U, "Up1");
  saveToFile("U", Up1, vtu);
  TermVector Uel(w, omega, uex, "Uel");
  TermVector Er=Up1-Uel;
  saveToFile("Er", Er, vtu);
  return 0;
}
```

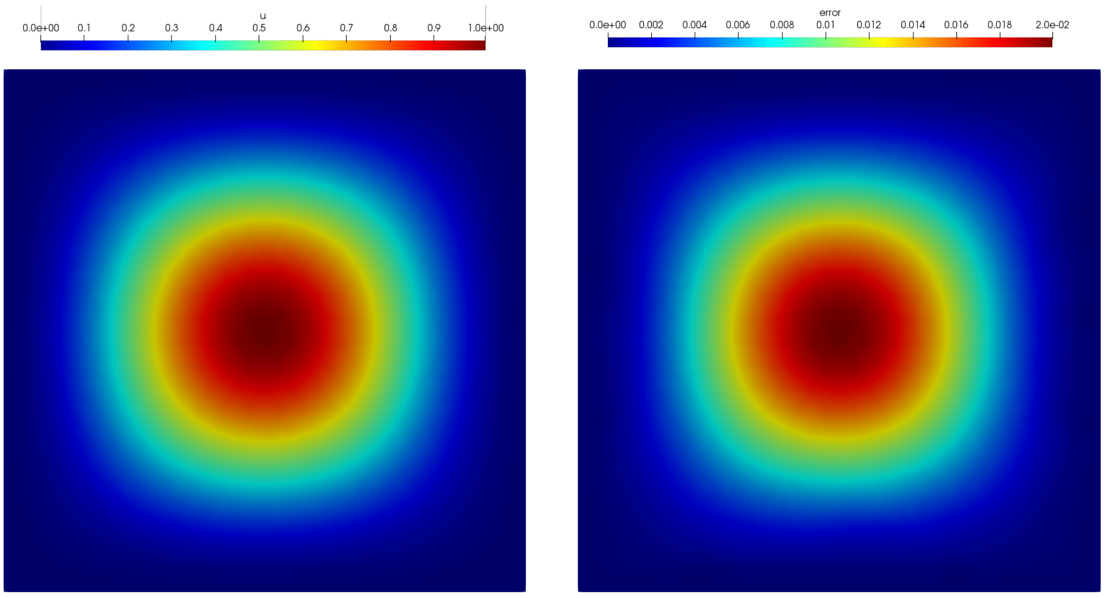


Figure 13.1: Approximate solution and difference with exact solution of the bilaplacian 2D problem

In order to use Argiris element in place of Morley element, replace the space definition:

```
Space V(omega,Argyris,1,"V");
```

The next figure shows the L^2 convergence rate of the Morley approximation when solving the 2d bilaplacian problem; in agreement with the order 2 predicting by the theory.

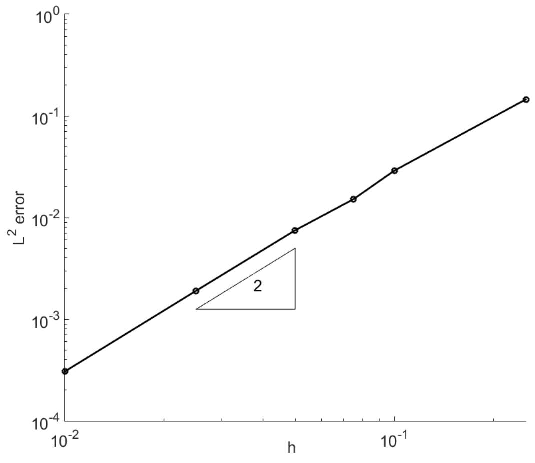


Figure 13.2: L^2 convergence rate of the Morley approximation

14 Solving wave equation

So far, only the harmonic problems were considered. Time problem may also be solved using XLiFE++. But there is no specific tools dedicated to. Users have to implement the time loop related to the finite difference time scheme they choose.

As an example, consider the wave equation:

$$\begin{cases} \dfrac{\partial^2 u}{\partial t^2} - c^2 \Delta u = f & \text{in } \Omega \times]0,T[\\ \dfrac{\partial u}{\partial n} = 0 & \text{in } \partial\Omega \times]0,T[\\ u(x,0) = \dfrac{\partial u}{\partial t}(x,0) = 0 & \text{in } \Omega \end{cases}$$

Using classical leap-frog scheme with time discretization $t^n = n\Delta t$, leads to (u^n approximates $u(x,t^n)$):

$$\begin{cases} u^{n+1} = 2u^n - u^{n-1} + (c\Delta t)^2 \Delta u^n + (\Delta t)^2 f^n & \text{in } \Omega,\ \forall n > 1 \\ \dfrac{\partial u^n}{\partial n} = 0 & \text{in } \partial\Omega,\ \forall n > 1 \\ u^0 = u^1 = 0 & \text{in } \Omega \end{cases}$$

or, in variational form, $\forall v \in V = H^1(\Omega)$:

$$\begin{cases} \displaystyle\int_\Omega u^{n+1} v = 2\int_\Omega u^n v - \int_\Omega u^{n-1} v - (c\Delta t)^2 \int_\Omega \nabla u^n . \nabla v + (\Delta t)^2 \int_\Omega f^n\, v \ \ \forall n > 1 \\ u^0 = u^1 = 0 \text{ in } \Omega \end{cases}$$

When approximating space V by a finite dimension space V_h with basis $(w_i)_{i=1,p}$, the variational formulation is reinterpreted in terms of matrices and vectors as follows:

$$\begin{cases} U^{n+1} = 2U^n - U^{n-1} - \mathbb{M}^{-1}\left((c\Delta t)^2 \mathbb{K} U^n - (\Delta t)^2 F^n\right) \ \ \forall n > 1 \\ U^0 = U^1 = 0 \text{ in } \Omega \end{cases}$$

where

$$\mathbb{M}_{ij} = \int_\Omega w_i w_j,\ \mathbb{K}_{ij} = \int_\Omega \nabla w_i . \nabla w_j,\ (F^n)_i = \int_\Omega f^n\, w_i.$$

The XLiFE++ implementation of this scheme on the unity square when using P1 Lagrange interpolation looks like ($f(x,t) = h(t)g(x)$):

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

Real g(const Point& P, Parameters& pa = defaultParameters)
{
  Real d=P.distance(Point(0.5, 0.5));
  Real R= 0.02;          // source radius
  Real amp= 1./(pi_*R*R); // source amplitude (constant power)
  if (d<=0.02) return amp; else return 0.;
}
```

```
Real h(const Real& t)
{
  Real a=10000, t0=0.04 ; //gaussian slope and center
  return exp(-a*(t-t0)*(t-t0));
}

int main(int argc, char** argv)
{
  init(argc, argv, _lang=en);
  // create a mesh and domain omega
  SquareGeo sq(_origin=Point(0., 0.), _length=1, _nnodes=70);
  Mesh mesh2d(sq, triangle, 1, structured);
  Domain omega=mesh2d.domain("Omega");
  // create interpolation
  Space V(_domain=omega, _interpolation=P1, _name="V");
  Unknown u(V, "u");
  TestFunction v(u, "v");
  // define FE terms
  TermMatrix A(intg(omega, grad(u)|grad(v)), "A"), M(intg(omega, u*v), "M");
  TermVector G(intg(omega, g*v), "G");
  TermMatrix L; ldltFactorize(M, L);
  // leap-frog scheme
  Real c=1, dt=0.004, dt2=dt*dt, cdt2=c*c*dt2;
  Number nbt=200;
  TermVectors U(nbt);   // to store solution at t=ndt
  TermVector zeros(u, omega, 0.);  U(1)=zeros;  U(2)=zeros;
  Real t=dt;
  for (Number n=2; n<nbt; n++, t+=dt)
  {
    U(n+1)=2.*U(n)-U(n-1)-factSolve(L, cdt2*(A*U(n))-dt2*h(t)*G);
  }
  saveToFile("U", U, vtu);
  return 0;
}
```

Note the very simple syntax taken into account the leap-frog scheme. The Figure 14.1 represents the solution at different instants for a constant source localized in disk with center $(0.5, 0.5)$, radius $R = 0.02$ and time excitation that is a Gaussian function. For chosen parameter $dt = 0.04$, the leap-frog scheme is stable (it satisfies the CFL condition) but dispersion effects obviously appear.

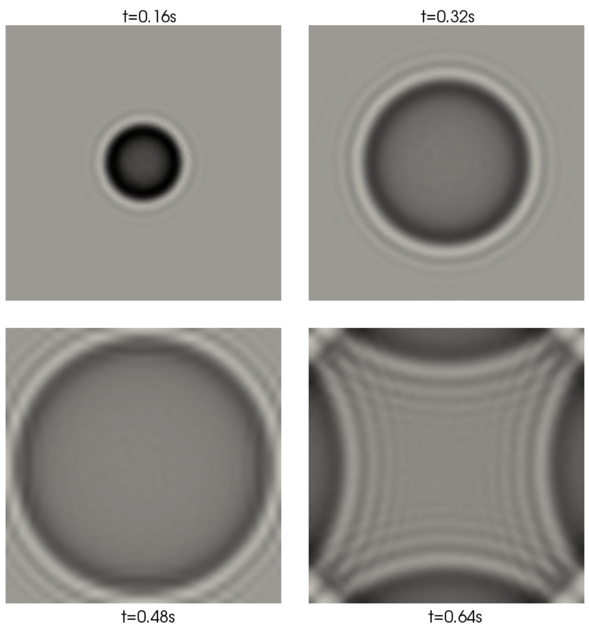


Figure 14.1: Solution of the wave equation at different instants for a constant source localized in disk with center $(0.5, 0.5)$