Domain Specific Embedded Languages

A Galerkin framework for finite element and spectral element

C. Prud'homme

Université de Strasbourg
Université de Grenoble

Strasbourg, Jan 25, 2012

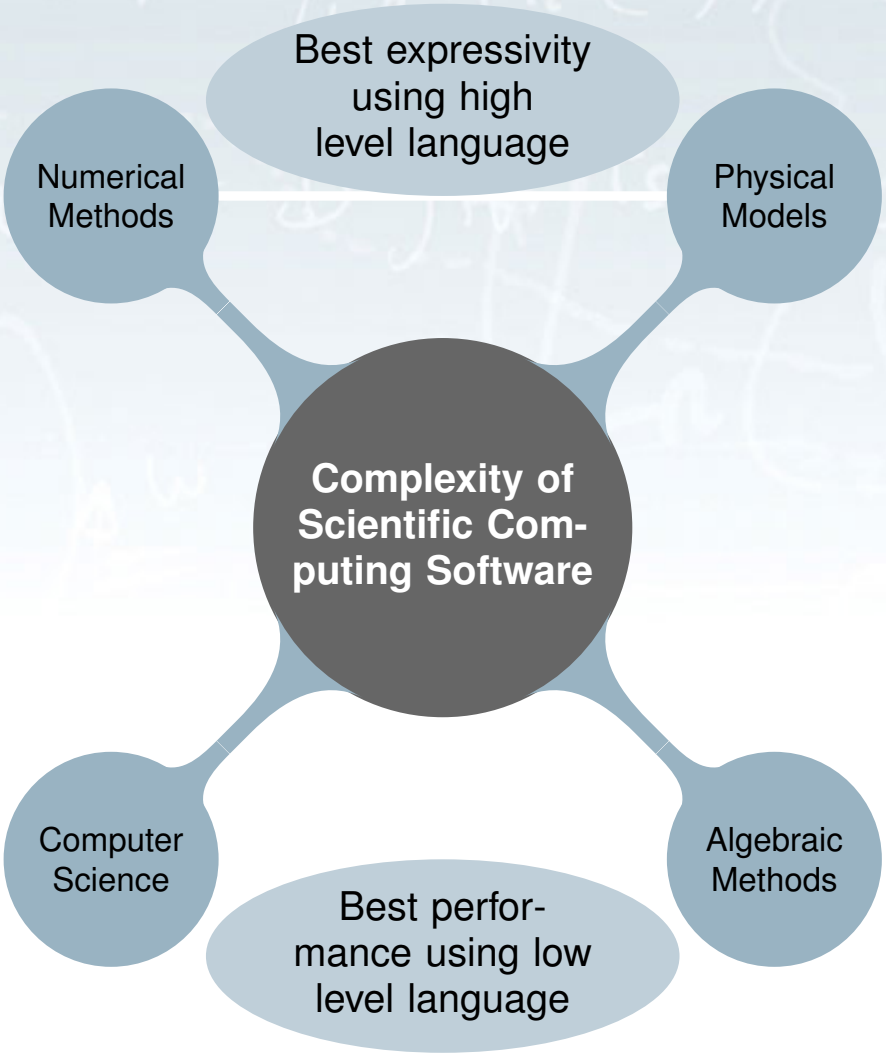
Collaborators

S. Bertoluzza (IMATI/CNR/Pavia)
V. Chabannes (UJF/LJK)
C. Daversin (CNRS/LNCMI)
V. Doyeux (UJF/LIPHY)
J.M. Gratien (IFPEN)
M. Ismail (UJF/LIPHY)
P. Jolivet (UPMC/LJLL)
F. Nataf (UPMC/LJLL)
G. Pena (UC/CMUC)
D. Di Pietro (IFPEN)
A. Samake (UJF/LJK)
C. Trophime (CNRS/LNCMI)
S. Veys (UJF/LJK)
+ Master students

Outline

1. Motivations
2. Mathematical Framework
3. Seamless Interpolation Tool
4. FSI solver
5. Conclusions, Some on-going and future work
6. Benchmarking

Generative Programming and DS(E)L

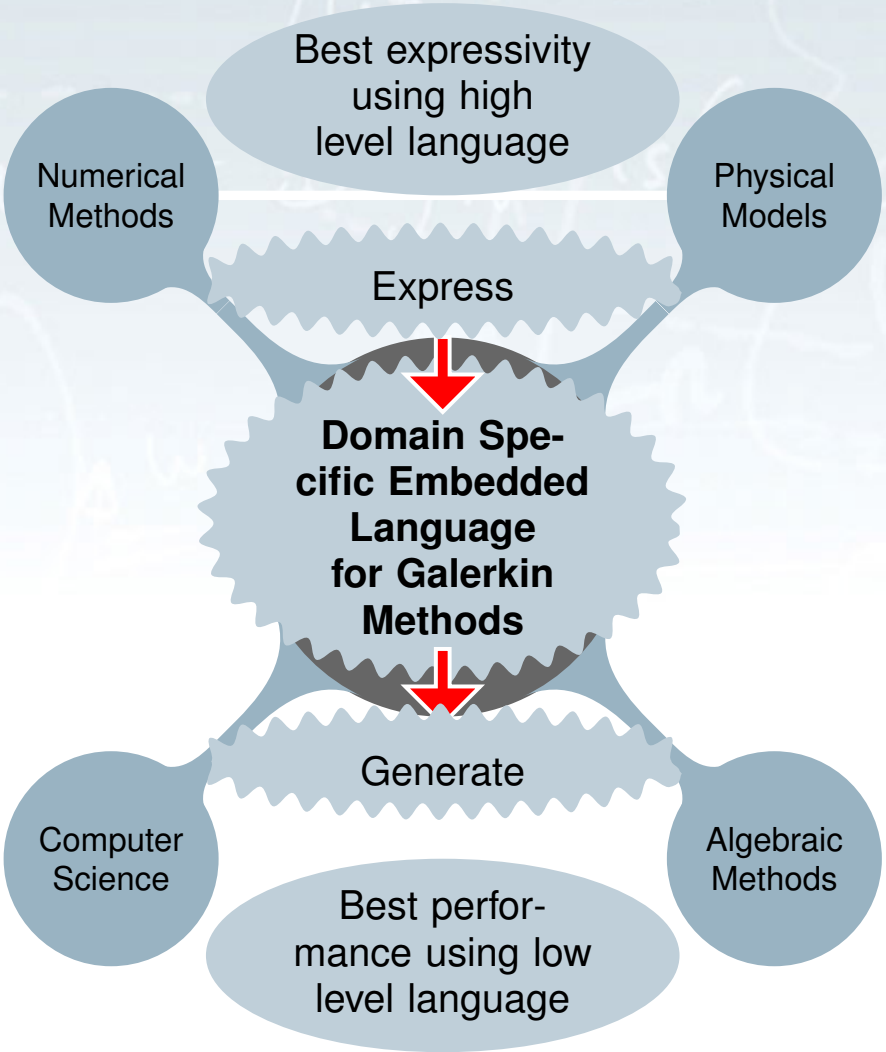


Complexity Types

- Algebraic
- Numerical
- Models
- Computer science

- Numerical and model complexity are better treated by a high level language
- Algebraic and computer science complexity perform often better with low level languages

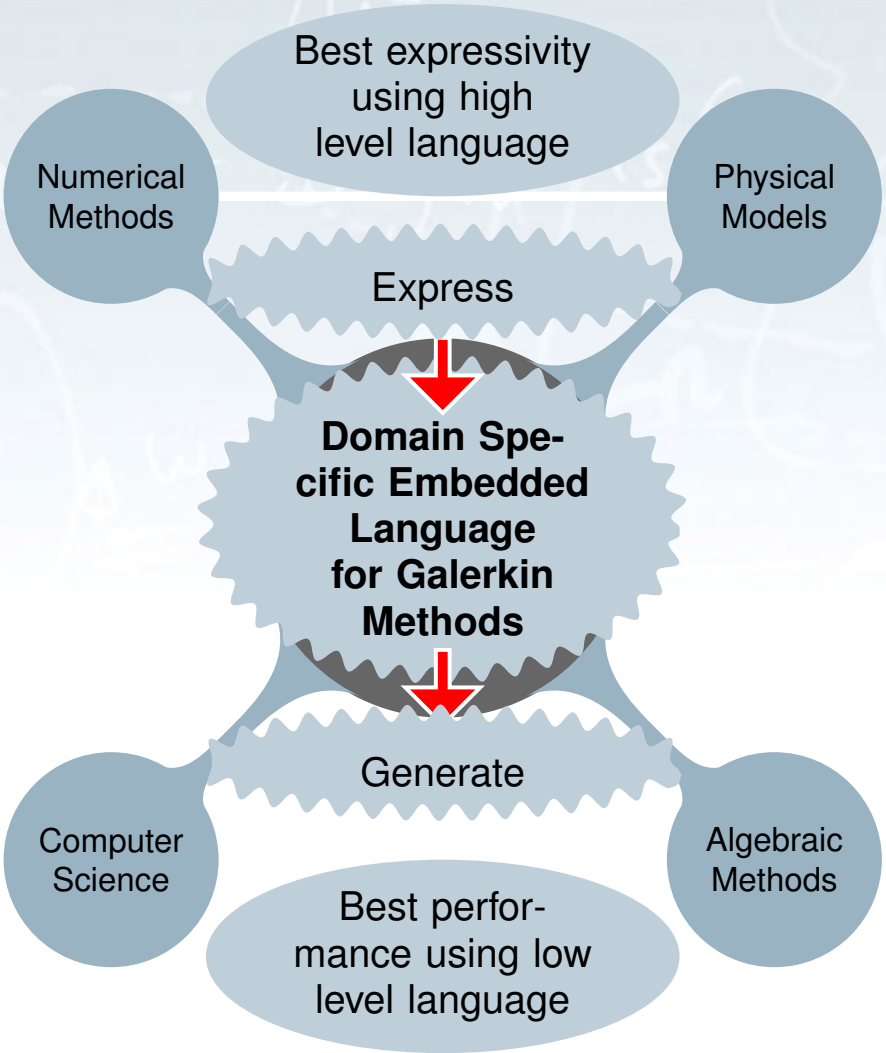
Generative Programming and DS(E)L



Generative paradigm

- distribute/partition complexity
- developer: The computer science and algebraic complexity
- user(s): The numerical and model complexity

Generative Programming and DS(E)L



Definitions

- A Domain Specific Language (DSL) is a programming or specification language dedicated to a particular domain, problem and/or a solution technique
- A Domain Specific Embedded Language (DSEL) is a DSL integrated into another programming language (e.g. C++)

Example of DS(E)L – $-\Delta u = f$

Applied Mathematicians favorite equation: find u such that

$$-\Delta u = f, \quad u = g \text{ on } \partial\Omega$$

which, using a Galerkin method, reads find $u \in X_h$ such that $\forall v \in X_h$

$$\int_\Omega \nabla u \cdot \nabla v = \int_\Omega f v$$

which leads to solving $Au = b$ where

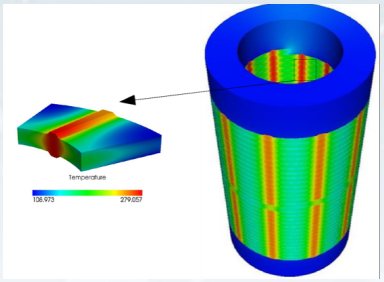
$$A = \left(\int_\Omega \nabla\phi_i \cdot \nabla\phi_j \right)_{i,j}, \; f = \left(\int_\Omega f\phi_i \right)_i$$

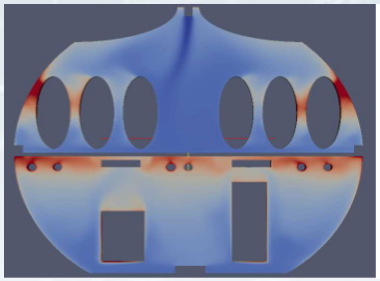
```
auto mesh = Mesh<Simplex<2>>::New();
loadGMSHMesh( _mesh=mesh, _filename="mesh.msh" );
// P3 finite element space on
// triangular elements
auto Xh = FunctionSpace<Mesh<Simplex<2>>,
          bases<Lagrange<3>>>::New(mesh);
auto u = Xh->element();
auto v = Xh->element();
// f a function e.g. f = 2π^2 sin(πx) * cos(πy)
auto f = 2*π^2*sin(πPx())*cos(πPy());
form1(_test=Xh,_vector=b) =
    // ∫_Ω f φ_i
    integrate( elements(mesh),
               f*id(v) );
// \int_\Omega \nabla u \cdot \nabla v
form1(_test=Xh,_trial=Xh,_matrix=A) =
    // ∫_Ω ∇φ_i · ∇φ_j
    integrate( elements(mesh),
               gradt(u)*trans(grad(v)) );
auto g =sin(πPx())*cos(πPy());
form1(_test=Xh,_trial=Xh,_matrix=A) +=
    on( boundaryfaces(mesh), u, F, g );

// solve M u = b
backend->solve( _matrix=A, _solution=u, _rhs=b );

std::cout << "‖u − g‖_L2 ="
 << integrate(elements(mesh),
              (idv(u)-g)*(idv(u)-g) ).evaluate()
 << "\n";
```

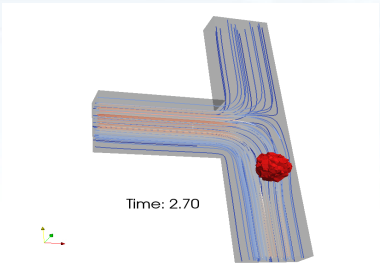
Languages for PDEs

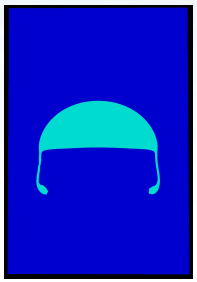
- Generate automatically operators, matrices, vectors with variational formulations(FEM,SEM) for an easy integration with linear algebra software (Petsc, Trilinos, ...), it is called “Automatic Computational Mathematical Modeling” (ACMM)
- High level language (DS(E)L) close to mathematical formulation
- See UFL,FFC (FeniCs project), Sundance, Freefem++/3D, FEEL++, and possibly others

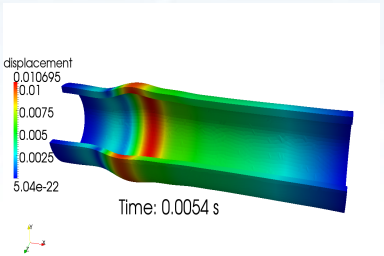
Gallery

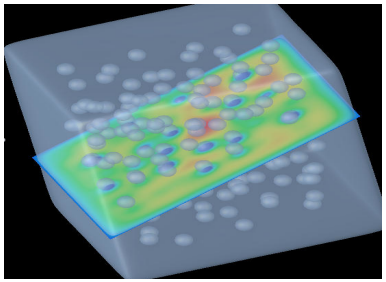

Temperature
138.973
279.057

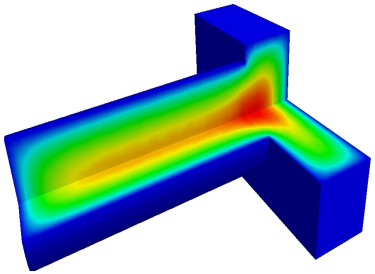


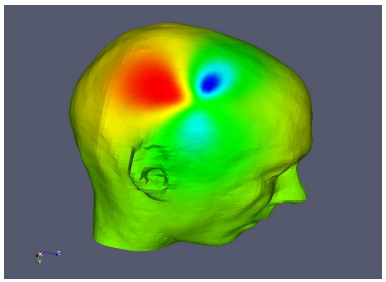

Time: 2.70




displacement
0.010695
0.01
0.0075
0.005
0.0025
5.04e-22
Time: 0.0054 s







Outline

1 Motivations
2 Mathematical Framework
3 Seamless Interpolation Tool
4 FSI solver
5 Conclusions, Some on-going and future work
6 Benchmarking

FEEL++

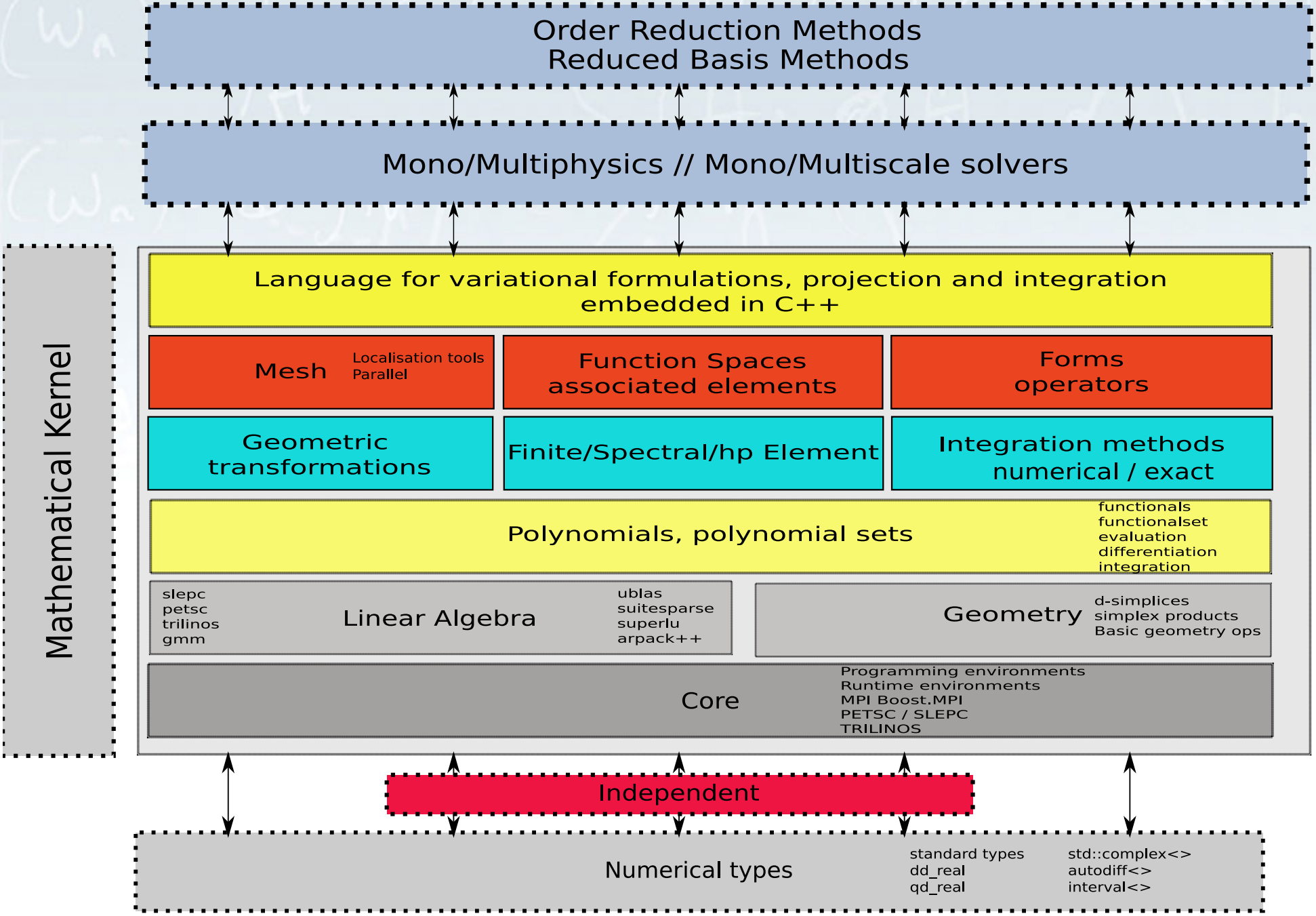
http://www.feelpp.org, http://forge.imag.fr/projects/feelpp

- UdG: LIPHY and LJK
- Dept. of Mathematics, U. Coimbra
- CNRS: LNCMI
- IFPEN
- CNR: IMATI

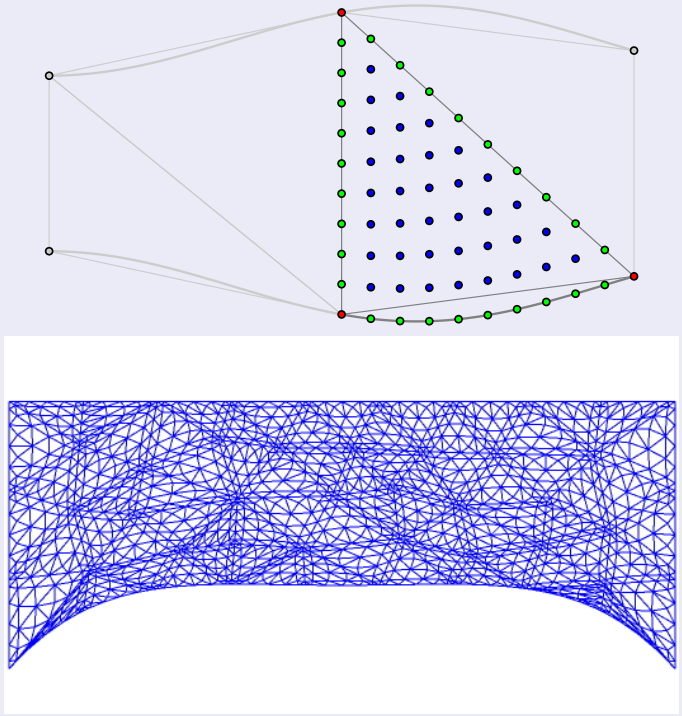
FEEL++ Principles

- The mathematical language is considered the language between the applications and the computing ressources to break complexity
- A computational framework that maps closely the mathematical one
- Delay as much as possible the manipulation of the algebraic representations, work as much as possible with the functional framework
- Use state of the art C++ techniques and develop new ones
- Use Boost as much as possible
- Use C++11 as much as possible :
  - reduce considerably code complexity
  - introduce very powerful tools
  - allows to write C++ code (strongly typed) that ressembles scripts (weakly typed) using the keyword `auto` (type inference)

Architecture



Mesh



- Convexes and associated geometric transformation ($\mathbb{P}_N, N = 1, 2, 3, 4, 5...$)
- Support for high order ALE maps [Pena et al., 2010]
- Geometric entitites are stored using Boost.MultiIndex
- Element-wise partitioning using Scotch/Metis, sorting over process id key

Example

```
elements(mesh [, processid]);
markedfaces(mesh, marker [, processid]);
```

Function Spaces

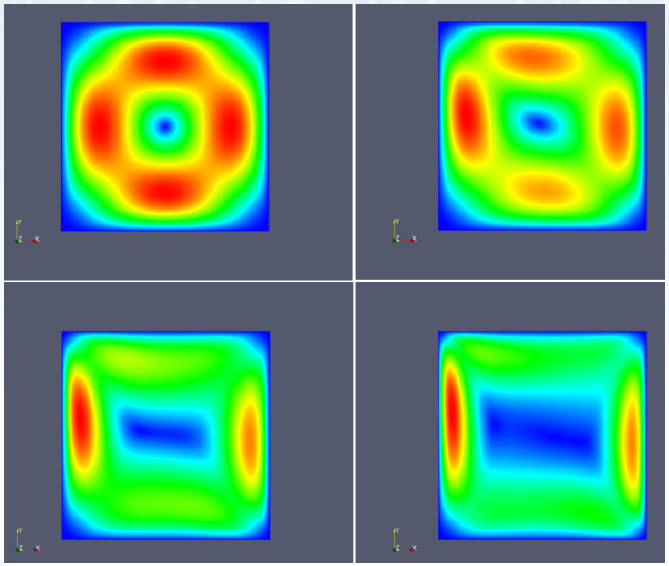
- Product of N-function spaces (a mix of scalar,vectorial, matricial and different basis types)
- Get each function space and associated "component" spaces
- Associated elements/functions of N products and associated components, can use different backend (gmm,petsc/slepc,trilinos)

Example

```
typedef FunctionSpace<Mesh,bases<Lagrange<2,Vectorial>,
                                 Lagrange<1,Scalar> > > space_t;
space_t Xh( mesh );
auto Uh = Xh.functionSpace<0>();
auto x = Xh.element();
auto p = x.element<1>(); // view
```

Natural convection

Coupling heat transfer and fluid flow in a closed square domain



Natural convection: flow field for different values of the Grashof number.

Example

```
// velocity space
typedef Lagrange<Order_s, Vectorial> basis_u_type;
// pressure space
typedef Lagrange<Order_p, Scalar> basis_p_type;
// temperature space
typedef Lagrange<Order_t, Scalar> basis_t_type;
// multipliers for pressure space
typedef Lagrange<0, Scalar> basis_l_type;

typedef bases< basis_u_type , basis_p_type ,
               basis_t_type ,basis_l_type> basis_type;

typedef FunctionSpace<mesh_type,
                      basis_type> space_type;
```

Operators and Forms

- Linear Operators/ Bilinear Forms represented by full, blockwise matrices/vectors
  - Full matrix $\begin{pmatrix} A & B^T \\ B & C \end{pmatrix}$, Matrix Blocks A, B^T, B, C
- The link between the variational expression and the algebraic representation

Example

```
X1 Xh; X2 Vh;
auto u = Xh.element(); auto v = Vh.element();
// operator T : X1 → X2
auto T = LO( Xh, Vh [, backend] );
T = integrate(elements(mesh), id(u)*idt(v) );
// linear functional f : X2 → ℝ
auto f = LF( Vh [, backend] );
T.apply( u, f ); f.apply( v );
```

A Language for variational formulations

Enablers and Features

- Meta/Functional - programming (Boost.MPL...): high order functions, recursion, ...
- Crossing Compile-time to Run-time (Boost.fusion...)
- Lazy evaluations (multiple evaluation engines) use `Expr<...>` (expression) and `Context<...>` (evaluation) (e.g. Boost.Proto)

Features: Use the *C++* compiler/language optimizations

- Optimize away redundant calculations (*C++*)
- Optimize away expressions known at compile time(*C++*)

Example

$a : X_1 \times X_2 \to \mathbb{R} \quad a = \int_\Omega \nabla u \cdot \nabla v$

```
// a : X1 × X2 → ℝ   a = ∫Ω ∇u · ∇v
form (_test=X1,_trial=X2,_matrix=M) =
  integrate( elements(mesh), gradt(u)*trans(grad(v)));
```

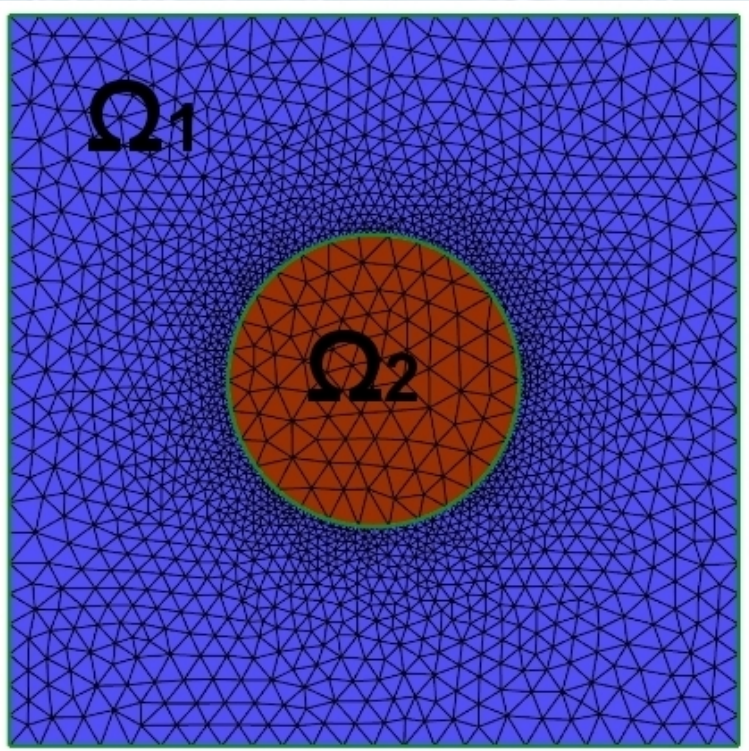
Outline

1 Motivations
2 Mathematical Framework
3 Seamless Interpolation Tool
4 FSI solver
5 Conclusions, Some on-going and future work
6 Benchmarking

Context

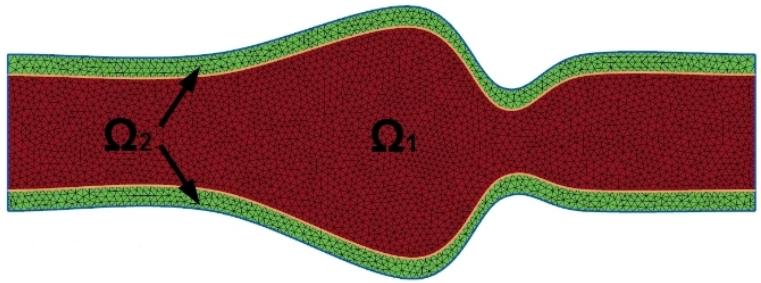
Motivations

- Interpolation between different meshes (h) or function spaces (N)
- $\forall \mathrm{d} = 1, 2, 3, \forall \mathrm{N}, \forall \mathrm{Ngeo}$ at dof or quadrature nodes
- Computation of different operations (id, ∇, $\nabla\cdot$, $\nabla\times$,...)
- $\mathrm{I}_\mathrm{h}^{\mathrm{LAG}}, \mathrm{I}_\mathrm{h}^{\mathrm{CR}}, \mathrm{I}_\mathrm{h}^{\mathrm{RT}}, \mathrm{I}_\mathrm{h}^{\mathrm{Her}}, \ldots$



Some Applications

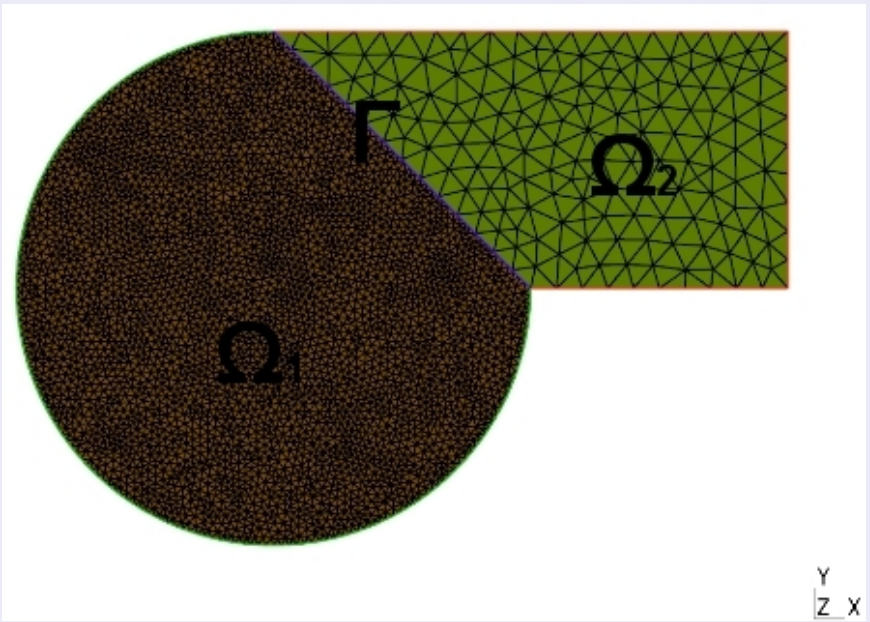
- Multiphysics coupling (FSI)
- Fictitious domain meth. (FBM)
- Domain decomposition meth.



DD-Schwartz

Global problem

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



Schwarz non Overlap

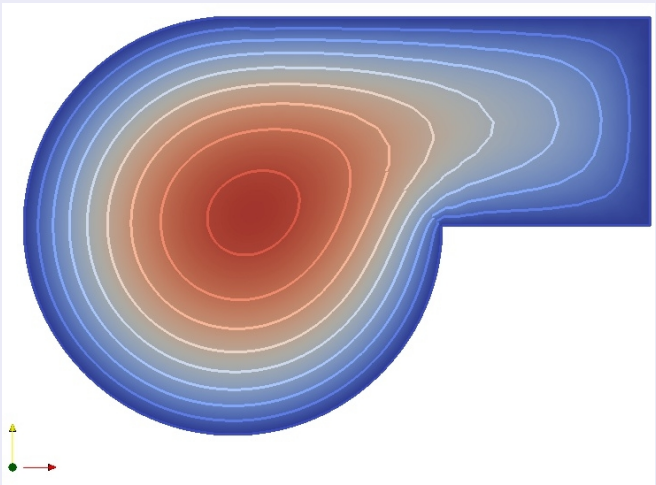
- Problem_i

$$\begin{cases} -\Delta u_i &= f & \text{in } \Omega_i \\ u_i &= 0 & \text{in } \partial\Omega_i \setminus \Gamma \\ u_i &= \pm\lambda & \text{in } \Gamma \end{cases}$$

- While not convergence

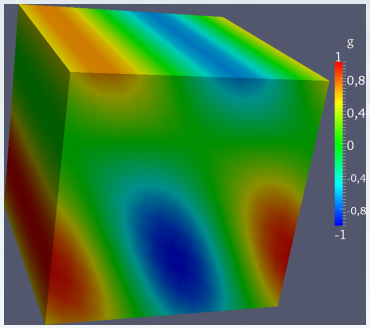
$$\begin{Vmatrix} \text{solveProblem}_1 \\ \text{solveProblem}_2 \\ \lambda = \lambda - \frac{1}{2}(u_1 - u_2) \end{Vmatrix}$$

```
// interpolation operator
auto opI = IO( Xh2, Xh1 );
while ((err1+err2)>err_tol) {
    u1old=u1; u2old=u2;

    localProblem(u1,-idv(lambda));
    localProblem(u2,idv(lambda));
    err1=computeConv(u1,u1old);
    err2=computeConv(u2,u2old);

    u2bis = opI(u2);
    lambda = 0.5*(u1-u2bis);
 }
```

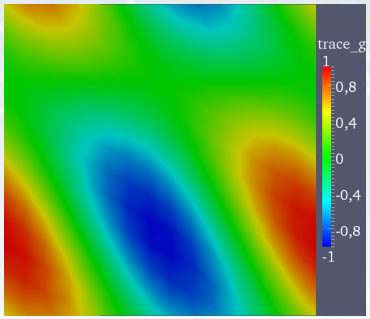


```
template<typename RhsExprType>
localProblem( element_type& u, RhsExprType RhsExp )
{
  auto Xh=u.functionSpace();
  auto mesh=Xh->mesh();
  auto v = Xh->element("v");
  auto f=cst(1.);

  auto A = backend->newMatrix(Xh,Xh);
  form2( Xh, Xh, A, _init=true ) =
    integrate( elements(mesh),
               gradt(u)*trans(grad(v)) );

  auto B = backend->newVector( Xh );
  form1( Xh, B, _init=true ) =
    integrate( elements(mesh), f*id(v) );
  form1( Xh, B ) +=
    integrate( markedfaces(mesh,"Interface"),
               RhsExp*id(v) );
  form2( Xh, Xh, A ) +=
    on( markedfaces(mesh,"Boundary") ,
        u1, B, cst(0.) );

  backend->solve( _matrix=A, _solution=u, _rhs=B );
}
```

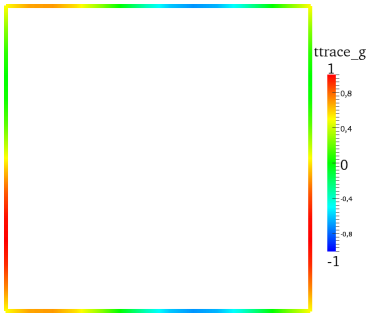
Trace of Trace and Lift of lift



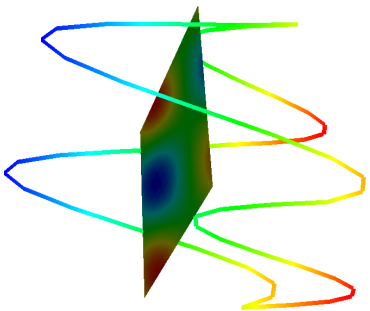
(l) volume and the function g



(m) trace mesh and trace of g



(n) wirebasket and trace trace of g

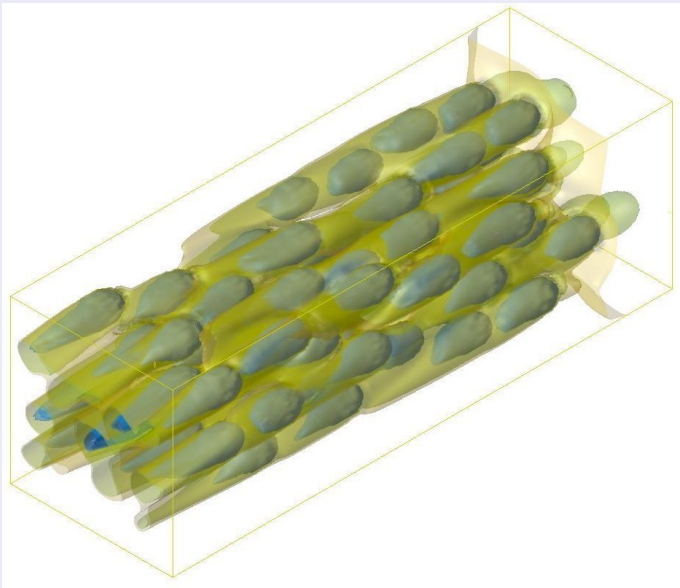


(o) warp with respect the function

```
auto Xh = space_type::New( mesh );
// trace function space associated to trace(mesh)
auto TXh = trace_space_type::New(
       mesh->trace(markedfaces(mesh,marker)) );
// trace function space associated to trace(trace(mesh))
auto TTXh = trace_trace_space_type::New(
    TXh->mesh()->trace(boundaryfaces(TXh->mesh())));
// Let g be an function given on 3D mesh
auto g = sin(pi*(2*Px()+Py()+1./4))*cos(pi*(Py()-1./4));
/* trace and trace of trace of g  */
// trace of g on the 2D trace_mesh
auto trace_g = vf::project( TXh,
                        elements(TXh->mesh()), g );
// trace of g on the 1D trace_trace_mesh
auto trace_trace_g = vf::project( TTXh,
                         elements(TTXh->mesh()), g );
/* lift and lift of lift of trace_trace_g  */
// extension of trace_trace_g by zero on 2D trace_mesh
auto zero_extension = vf::project( TXh,
         boundaryfaces(TXh->mesh()),idv(trace_trace_g) );
// extension of trace_trace_g by the mean of
// trace_trace_g  on trace_mesh
auto const_extension = vf::project( TXh,
                     boundaryfaces(TTXh->mesh()),
                         idv(trace_trace_g)-mean );
const_extension += vf::project( TXh,
                     elements(TXh->mesh()), cst(mean) );
// harmonic extension of const_extension on 3D mesh
auto op_lift = operatorLift(Xh);
auto glift = op_lift->lift(_range=markedfaces(mesh,marker
```

Fat Boundary Method

Flow of particules



Strategy

Navier-Stokes Equations

↓

Characteristic method

+

Projection Scheme

⇓

elliptic problems in the perforated domaines

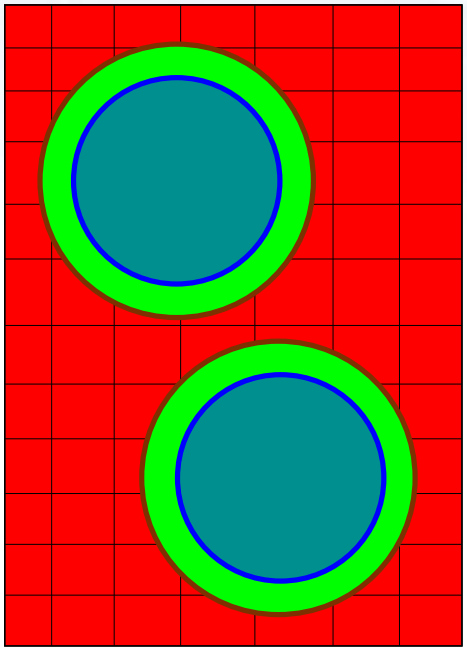
Requires the solution of two types of problem

$$\begin{cases} \alpha \mathbf{v} - \Delta \mathbf{v} = f_1 & \text{in} \quad \Omega_f \\ \mathbf{v} = g & \text{on} \quad \partial\Omega_f \end{cases} \tag{1}$$

$$\begin{cases} -\Delta p = f_2 & \text{in} \quad \Omega_f \\ \frac{\partial p}{\partial n} = 0 & \text{on} \quad \partial\Omega_f \end{cases} \tag{2}$$

Fat Boundary Method

How to use rapid solvers without accuracy loss in the neighborhood of particules?



Original Problem :

$$-\Delta u = f \text{ In } \square \setminus \overline{B}$$
$$u = 0 \text{ On } \Gamma \cup \gamma$$

FBM: [Bertoluzza et al., 2010]

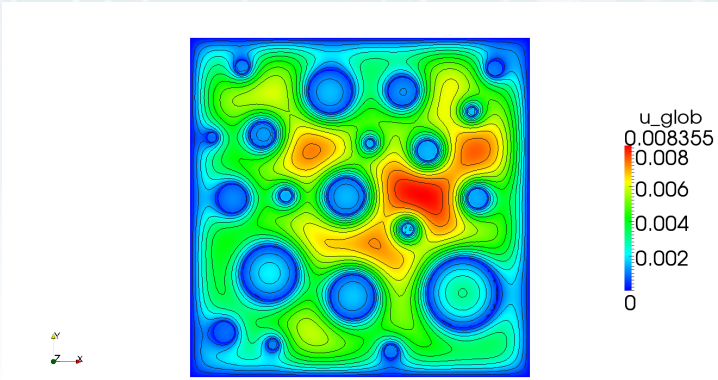
- one Global :

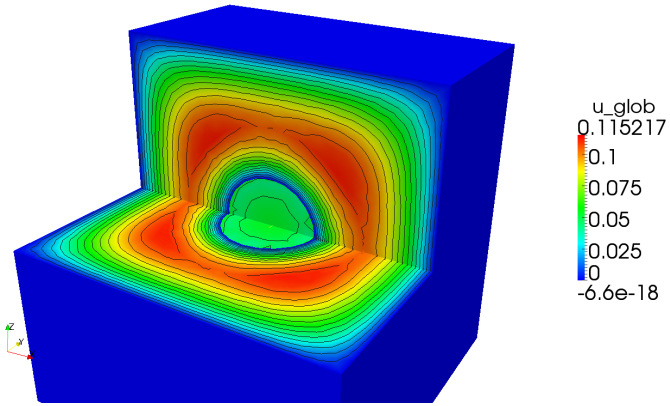
$$-\Delta \widehat{u} = \bar{f} + \partial_n v \delta_\gamma$$

- one Local :

$$-\Delta v = f \qquad v|_{\gamma'} = \widehat{u}|_{\gamma'}$$

Fat Boundary Method





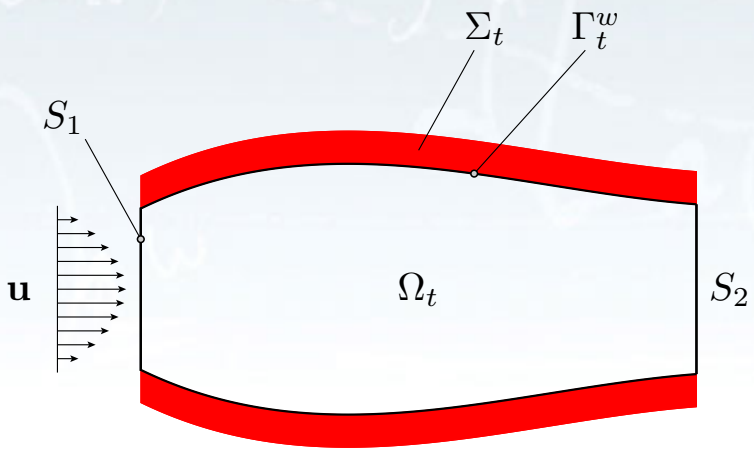
Fix point alg. for FBM

```
// global problem
form2( Xh, Xh, Mg ) =
 integrate(elements(globalmesh),gradt(u)*trans(grad(w)));
form2( Vh, Vh, Ml ) =
 integrate(elements(localmesh),gradt(ul)*trans(grad(wl)));

// fix point iteration with relaxation, u^0,...,u^n
do {
// iteration i
u^i_relax = θ u^{i-1}_relax + (1-θ) u^{i-1}
form1( Vh, Fl, _init=true ) =
  integrate(elements(localmesh), f*id(wl) );
form1( Vh, Fl ) +=
  on( markedfaces(localmesh, γ'), u_l, u^i_relax );

// solve for local problem u_l
// χ_p : characteristic function of the particle
form1( Xh, Fg, _init=true ) =
  integrate(elements(globalmesh), f*(1-χ_p)*id(w) );
form1( Xh, Fg ) +=
  // interpolation tool: integrate on local mesh
  // against global test functions
  integrate( markedfaces(localmesh,γ),
             (gradv(u_l)*N())*id(w));

// solve for global problem on u^i
} // until convergence
```

Outline

1. Motivations
2. Mathematical Framework
3. Seamless Interpolation Tool
4. FSI solver
5. Conclusions, Some on-going and future work
6. Benchmarking

Implementation of an FSI solver using Feel++ – Navier-Stokes Solver

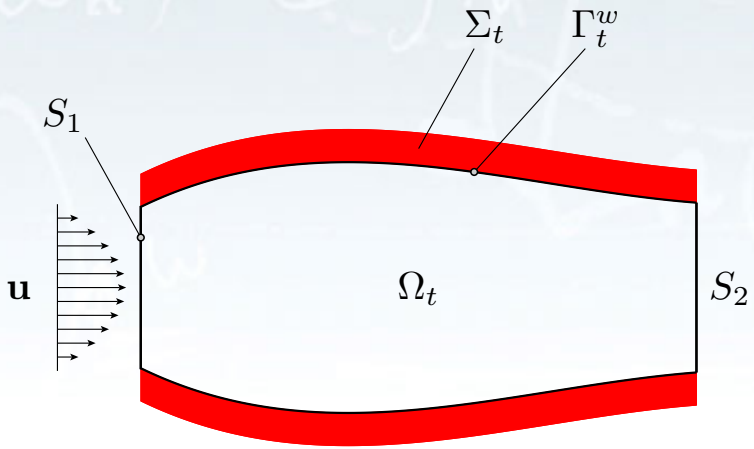



```
// automatic type (need to specify -std=c++0x)
// test strain tensor
auto def = 0.5*(grad(v)  + trans(grad(v)));
// trial strain tensor
auto deft = 0.5*(gradt(u) + trans(gradt(u)));
// oseen
form2( _test=Xh, _trial=Xh, _matrix=M) =
  // automatic quadature
  integrate( elements(Xh->mesh()),
             alpha*trans(idt(u))*id(v)
             + 2.0*nu*trace(trans(deft)*def)
             + trans(gradt(u)*idv(beta))*id(v)
             - div(v)*idt(p) + divt(u)*id(q) );
```

$$\left.\frac{\partial \mathbf{u}}{\partial t}\right|_{\mathbf{Y}} + [(\mathbf{u}-\mathbf{w})\cdot\boldsymbol{\nabla}]\mathbf{u} + \nabla p - 2\nu\mathbf{D}(\mathbf{u}) = \mathbf{f}, \text{ in } \Omega_t$$

$$\boldsymbol{\nabla}\cdot\mathbf{u} = 0, \text{ in } \Omega_t$$

Implementation of an FSI solver using Feel++ – Structure Solver



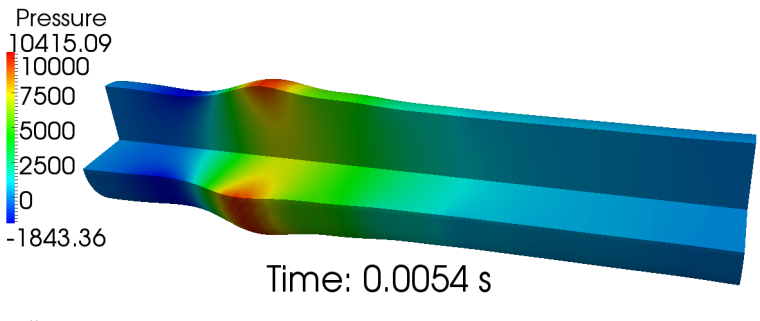

```
FunctionSpace<mesh_type,bases<Lagrange<1>>> space_
auto Xh = space_type::New(mesh);
auto u = Xh->element(), v = Xh->element();
// strain tensor  .5*(∇u + ∇u^T)
auto deft = 0.5*( gradt(u)+trans(gradt(u)) );
auto def = 0.5*( grad(v)+trans(grad(v)) );
form( _test=Xh, _trial=Xh, _matrix=D ) =
 integrate( elements(mesh),
      lambda*divt(u)*div(v)  +
      2*mu*trace(trans(deft)*def) ) +
 on( markedfaces(mesh,mesh->markerName("clamped"))
      u, F, constant(0)*one() );
MeshMover<mesh_type> meshmove;
meshmove.apply( mesh, u );
```

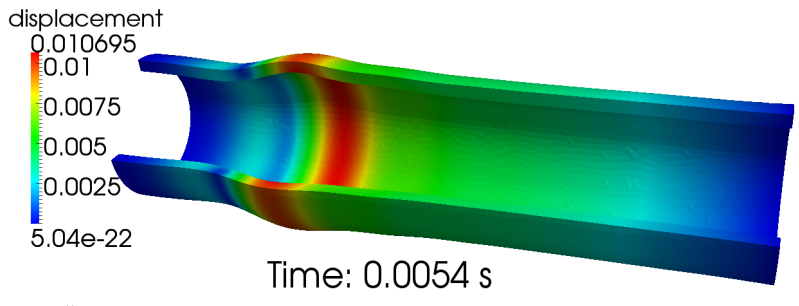
$$
\begin{cases}
\rho_s \dfrac{\partial^2 \boldsymbol{d}}{\partial t^2} + \operatorname{div}\hat{\hat{\boldsymbol{T}}} = \boldsymbol{f} & \text{in } \hat{\Omega}_s \\
\hat{\hat{\boldsymbol{T}}} \cdot \hat{\boldsymbol{n}}_s = \det\left(\dfrac{\partial \mathcal{A}_t}{\partial \hat{x}}\right) \boldsymbol{\sigma}_f \dfrac{\partial \mathcal{A}_t}{\partial \hat{x}} \hat{\boldsymbol{n}}_s & \text{on } \hat{\Sigma} \\
\boldsymbol{d} = \mathbf{o} \text{ or } \hat{\hat{\boldsymbol{T}}} \cdot \hat{\boldsymbol{n}}_s = \mathbf{o} & \text{on } \partial\hat{\Omega}_s / \hat{\Sigma}
\end{cases}
\tag{3}
$$

Implementation of an FSI solver using Feel++

- Solve structure model
- Compute and apply ALE map on fluid domain
- Solve fluid model
- Implicit and semi implicit FSI schemes using High order methods in space, time and geometry

```
// solve structure
elasticity.solve();
auto Ustruct=elasticity.displacement();
// generate the ale map A_t using structure
// displacement
ale->generateMap( Ustruct );
// get mesh displacement and generate a new mesh
// using A_t applied on the reference domain
meshmove.apply( mesh,
                ale->displacement() );
// compute mesh velocity
// current position after displacement
auto x = vf::project( Xh, elements(mesh), P() );
// w = (x^{n+1} - x^n)/dt
// with x^{n+1} = A_t(Y, t_{n+1})
// and x^n = A_t(Y, t_n)
w = (x - bdf_ale->last()).scale(1/dt);
// update and solve fluid
navierstokes.update(w);
```





Outline

1 Motivations
2 Mathematical Framework
3 Seamless Interpolation Tool
4 FSI solver
5 Conclusions, Some on-going and future work
6 Benchmarking

Conclusion

- Generative programming for PDE works thanks to C++ (GCC and C++11) and compilation time improves (not there yet but better)
- Feedback: fast prototyping(at least for methodology), domain specific language, devil lurks in the details (interpolation, ...), used by physicist in micro-fluidic
- Wide range of applications

Some on-going/future work

- Mathematical framework (EF, FV, MDF...), more abstractions
- Full parallelisation, domain decomposition framework
- Exploit hybrid architectures (CPU-GPGPU)

Dissemination

- `http://www.feelpp.org`, `http://forge.imag.fr/projects/feelpp`

References I

Bertoluzza, S., Ismail, M., and Maury, B. (2010).

Analysis of the fully discrete fat boundary method.

Numerische Mathematik.

accepted.

Burman, E. and Fernández, M. (2007).

Continuous interior penalty finite element method for the time-dependent navier-stokes equations: space discretization and convergence.

Numer. Math., 107(1):38–77.

Pena, G., Prud'homme, C., and Quarteroni, A. (2010).

High order methods for the approximation of the incompressible navier-stokes equations in a moving domain.

Accepted.

Prud'homme, C. and Pena., G. (2009).

Construction of a high order fluid-structure interaction solver.

J. Comput. Appl. Math.

To appear.

Outline

1. Motivations
2. Mathematical Framework
3. Seamless Interpolation Tool
4. FSI solver
5. Conclusions, Some on-going and future work
6. Benchmarking

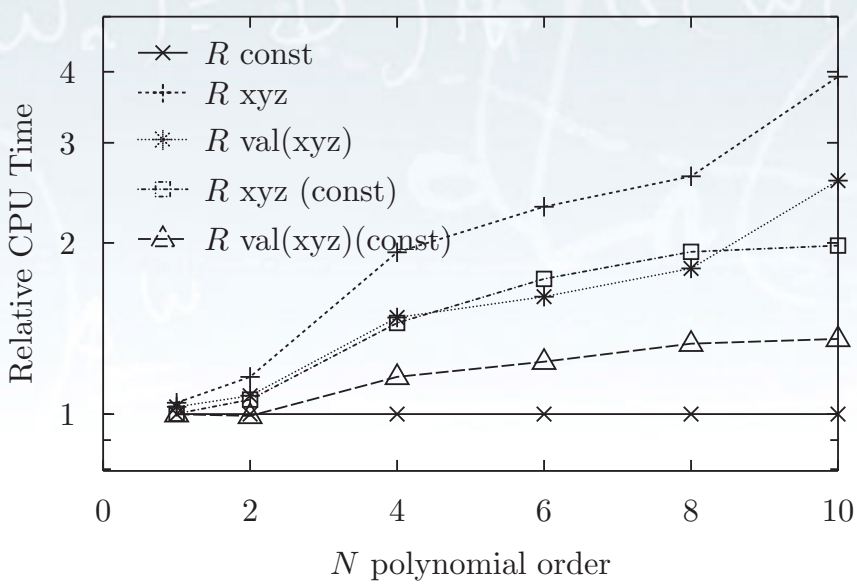
Benchmark — Equations

$$\begin{aligned} \mu\Delta u &= 0 \quad D, \\ \mu\Delta u + \sigma u &= 0 \quad DR, \\ \mu\Delta u + \beta \cdot \nabla u + \sigma u &= 0 \quad DAR \end{aligned}$$

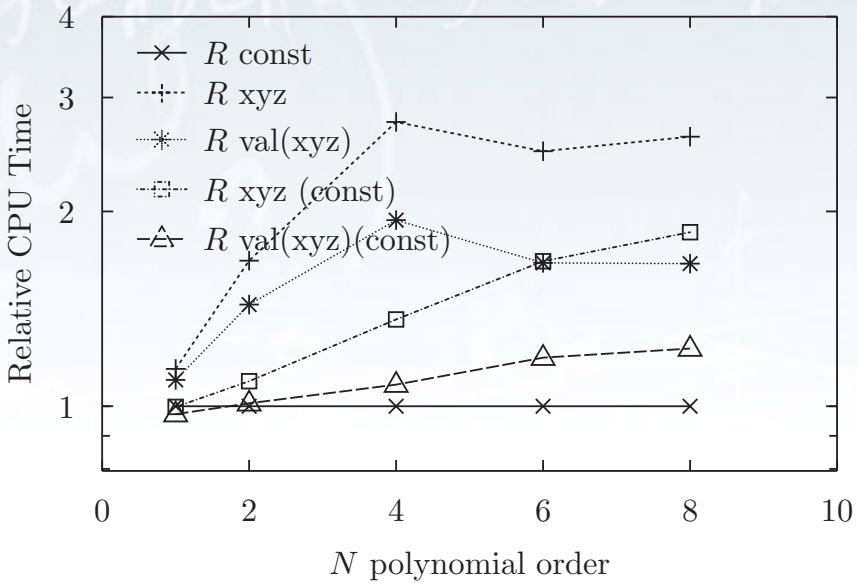
with μ, σ and β constant or space-dependant:

	nD	**2D**	**3D**
$\mu(x,y,z) =$	1	$x^3 + y^2$	$x^3 + y^2 z$
$\beta(x,y,z) =$	$(1)_{i=1,n}$	$(x^3 + y^2, x^3 + y^2)$	$(x^3 + y^2 z, x^3 + y^2, x^3)$
$\sigma(x,y,z) =$	1	$x^3 + y^2$	$x^3 + y^2 z$

Performances I

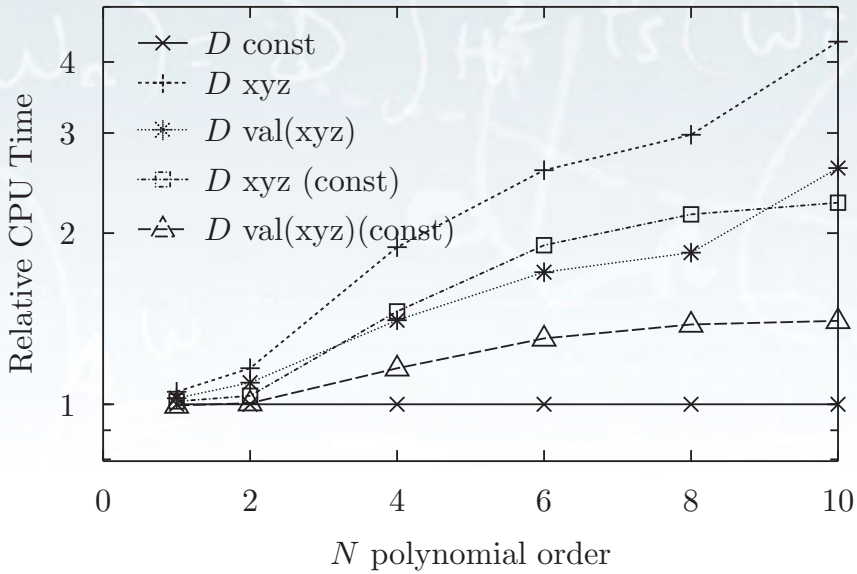


(c) 2D

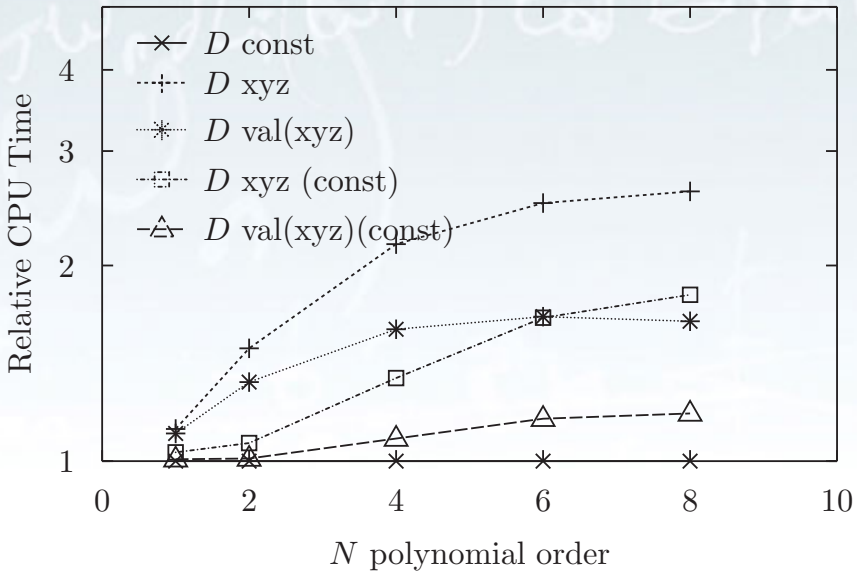


(d) 3D

Performances II

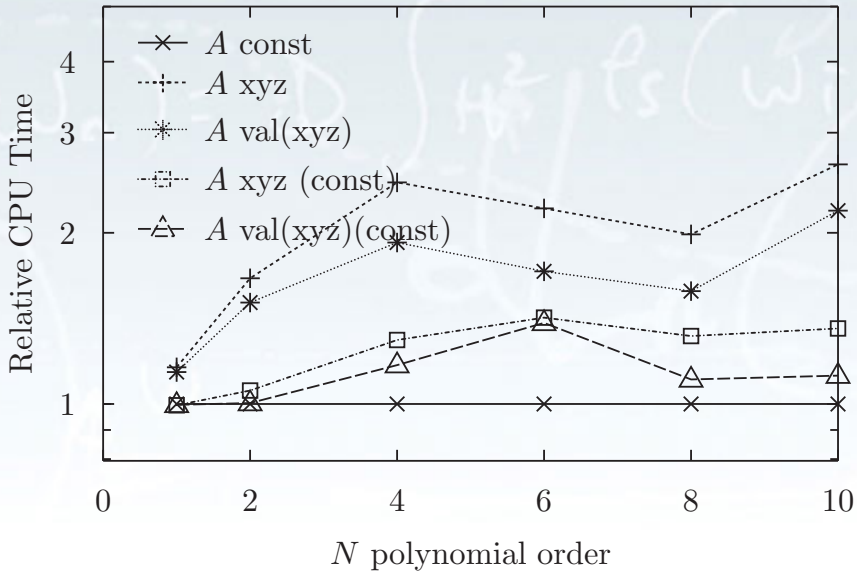


(e) 2D

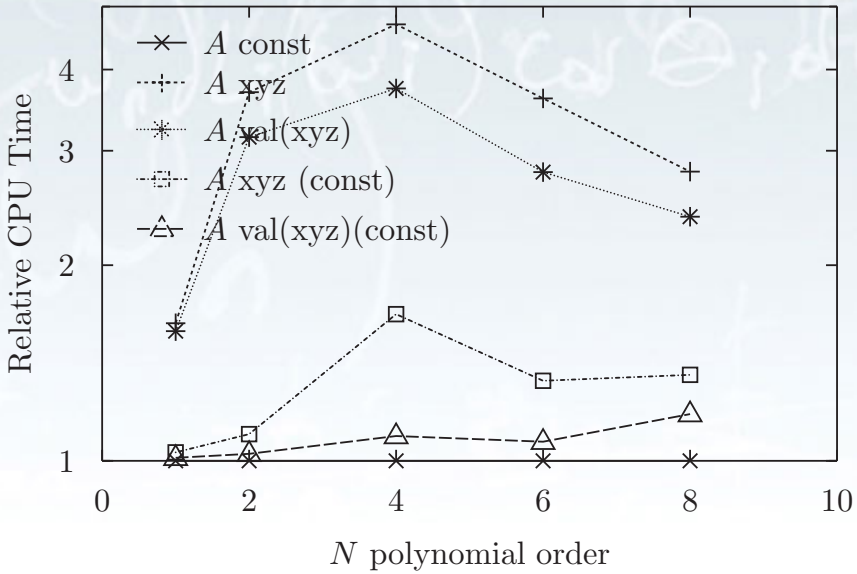


(f) 3D

Performances III



(g) 2D



(h) 3D

Relative CPU Time for R with respect to polynomial order N per matrix entry and per quadrature point.

Testing '0' Terms

Influence of terms that should be 0

```
// 2D -> dz(.)-> 0
vf_D = integrate( elements(*mesh),
IM, dxt(u)*dx(v)+dyt(u)*dy(v)+dzt(u)*dz(v));
vf_D = integrate( elements(*mesh),
IM, dxt(u)*dx(v)+dyt(u)*dy(v));
```

Order	Elt	Dof	Without '0' Terms	With '0' Terms
1/4	20786	10592	0.11	0.11
2/9	20786	41969	0.48	0.5
1/4	20786	41618	0.47	0.45
2/9	20786	165673	1.92	1.94