Implementing a Domain Specific Embedded Language in C++ for lowest-order variational methods with Boost.Proto

Jean-Marc Gratien

Department of Applied math and computer science
IFP New Energy, Rueil-Malmaison France
jean-marc.gratien@ifpen.fr

CppNow2012, Aspen Colorado, May 15th 2012

Introduction

- IFP New Energy
  - Technology for energy and environnement
  - Reservoir and Bassin modeling
  - CO2 storage
  - Combustion, engine modeling
  - ...
- A joined work with Christophe Prud'Homme of the LJK, Universtité de Grenoble, France

Outline

Outline

Context : Increasing complexity

Example : CO2 sequestration

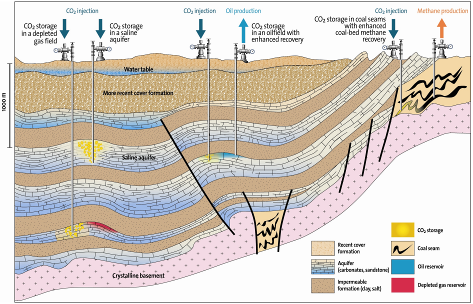


Figure: CO2 storage simulation

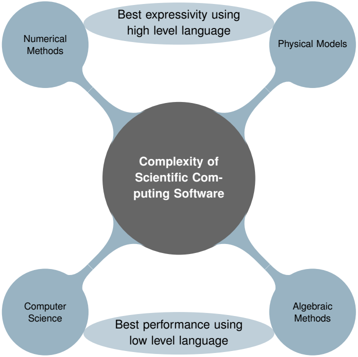
Various physical models :

- Bassin modeling ;
- Reservoir modeling ;
- Well modeling ;
- Reactive transport models ;
- Chemistry, Geo-mecanics

Various numerical methods :

- FV/FE methods ;
- Non linear solvers ;
- Coupling/Splitting methods ;
- Space/Time stepping. . .

Context : complexity in computer science

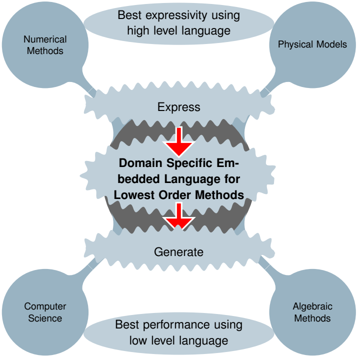


Complexity Types

- 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

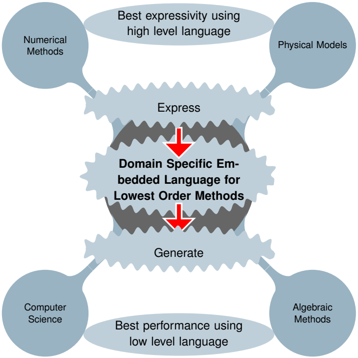
Context : complexity in computer science



Generative paradigm

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

Context : complexity in computer science



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++)

Outline

State of art :

- Frameworks to manage parallelism, mesh and linear solver: Arcane, Dune, Trilinos, Petsc. . .
- Frameworks for Finite Element or Galerkin methods :
  - based on an existing unified formalism
  - DSL solution: FreeFem++, GetDP, GetFem++, Fenics
  - DSEL solution: Feel++, Sundance

Motivation of our reseach work :

- No framework for lowest order methods :
  - Finite Volume, Mimetic Finite Difference, Mixed/Hybrid Finite methods ;
- An unified perspective to describe these methods is emerging ;
- What about extending DSEL solutions for FE/DG methods to lowest order methods?

Outline

Variational formulation

A unified perspective for FE/DG/FV methods :

Based on : **Variational Formulations** :

1. Functional spaces : Trial space U_h, Test space V_h
2. trial and test functions $(u_h, v_h) \in U_h \times V_h$
3. Bilinear form $a_h(u_h, v_h)$, linear form $b_h(v_h)$
4. Find $u_h \in U_h$ so that $\forall v_h \in V_h$: $a_h(u_h, v_h) = b_h(v_h)$

Key ingredients to design Functional Spaces for FV methods :

- **Mesh** ;
- **Space of Degree Of Freedoms** (DOFs) ;
- **Gradient Reconstruction Operator**.

Example : the Poisson problem

The continuous settings :

$$\begin{cases} -\triangle u = f & \text{in } \Omega, \\ u = g & \text{on } \partial\Omega. \end{cases}$$

A variational formulation :
U_h and V_h some Hybrid spaces,
Find $u_h \in U_h$, so that
$\forall v_h \in V_h, a_h(u_h, v_h) = b(v_h)$,
where

$$a_h(u_h, v_h) \stackrel{\text{def}}{=} \int_\Omega \nabla_h u_h \cdot \nabla_h v_h$$

$$b_h(v_h) \stackrel{\text{def}}{=} \int_\Omega f \; v_h$$

Outline

Mesh

Mesh : Ω domain of $\mathscr{R}^d$, $\mathscr{T}_h = \{\tau\}$ and $\mathscr{F}_h = \{\sigma\}$ mesh representation of Ω
SubMesh : $\mathscr{S}_h$ submesh of $\mathscr{T}_h$, 3 kinds : (*i*) $\mathscr{S}_h = \mathscr{T}_h$, (*ii*) pyramidal or (*iii*) node center.

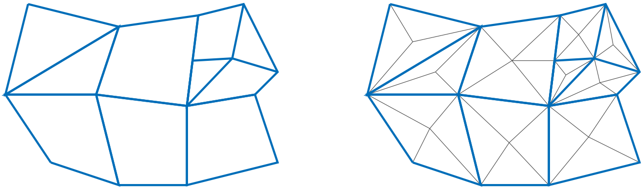


Figure: *Left.* Mesh $\mathscr{T}_h$ *Right.* Pyramidal submesh $\mathscr{P}_h$

Outline

Space of DOFs

$$\mathbb{T}_h \stackrel{\text{def}}{=} \mathbb{R}^{\mathscr{T}_h}, \qquad \mathbb{F}_h \stackrel{\text{def}}{=} \mathbb{R}^{\mathscr{F}_h},$$

$\mathbb{V}_h$: the space of degree of freedoms

- **Cell centered Space of DOFs** :

$$\mathbb{V}_h \stackrel{\text{def}}{=} \mathbb{T}_h = \mathbb{R}^{\mathscr{T}_h}$$

  DOFs indexed by elements of $\mathscr{T}_h$.

- **Hybrid Space of DOFs**:

$$\mathbb{V}_h \stackrel{\text{def}}{=} \mathbb{T}_h \times \mathbb{F}_h = \mathbb{R}^{\mathscr{T}_h} \times \mathbb{R}^{\mathscr{F}_h}$$

  DOFs indexed by elements of $\mathscr{T}_h \cup \mathscr{F}_h$.

Outline

Functional space :
A mapping of every vector of DOFs onto a piecewise affine function

$$\mathfrak{R}_h : \mathbb{V}_h \rightarrow \mathbb{P}^1_d(\mathscr{S}_h)$$

Recover different families of lowest order methods:

- $\mathbb{V}_h = \mathbb{T}_h$: cell centered finite volume (CCFV) and cell centered Galerkin (CCG) methods ;
- $\mathbb{V}_h = \mathbb{T}_h \times \mathbb{F}_h$: mimetic finite difference (MFD) and mixed/hybrid finite volume (MHFV) methods.

Key ingredient :
A piecewise constant linear **Gradient Reconstruction Operator** $\mathfrak{G}_h : \mathbb{V}_h \rightarrow [\mathbb{P}^0_d(\mathscr{S}_h)]^d$.
Define $\mathfrak{R}_h$ such that for all $\mathbf{v}_h \in \mathbb{V}_h$,
$\forall S \in \mathscr{S}_h, S \subset T_S \in \mathscr{T}_h, \forall \mathbf{x} \in S, \qquad \mathfrak{R}_h(\mathbf{v}_h)|_S(\mathbf{x}) = v_{T_S} + \mathfrak{G}_h(\mathbf{v}_h)|_S \cdot (\mathbf{x} - \mathbf{x}_{T_S}).$

Outline

Examples of **Gradient Reconstruction Operator** $\mathfrak{G}_h$

- **The G-method**
  - $\mathscr{S}_h$: pyramidal submesh, $\mathbb{V}_h = \mathbb{R}^{\mathscr{T}_h}$
  - $\mathfrak{G}_h$ based on the L-method
- **The ccG-method**
  - $\mathscr{S}_h = \mathscr{T}_h$, $\mathbb{V}_h = \mathbb{R}^{\mathscr{T}_h}$
  - A trace operator $\mathbf{T}_h^{\mathrm{g}} : \mathbb{T}_h \rightarrow \mathbb{F}_h$: $\mathbf{T}_h^{\mathrm{g}}(\mathbf{v}_h^{\mathscr{T}}) = (v_F)_{F \in \mathscr{F}_h} \in \mathbb{F}_h$.

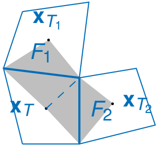


Figure: L-construction

- $\mathfrak{G}_h^{\mathrm{green}} : \mathbb{T}_h \rightarrow [\mathbb{P}_d^0(\mathscr{T}_h)]^d$ based on the Green's formula:
$\mathfrak{G}_h^{\mathrm{green}}(\mathbf{v}^{\mathscr{T}})|_T = \frac{1}{|T|_d} \sum_{F \in \mathscr{F}_T} |F|_{d-1} (\mathbf{T}_h^{\mathrm{g}}(F) - v_T) \mathrm{n}_{T,F}$

Examples of gradient reconstruction operator

- **The Hybrid-method** (SUSHI scheme)
  - $\mathscr{S}_h$: pyramidal submesh, $\mathbb{V}_h = \mathbb{R}^{\mathscr{T}_h} \times \mathbb{R}^{\mathscr{F}_h}$
  - $\mathfrak{G}_h^{\text{green}} : \mathbb{T}_h \times \mathbb{F}_h \to [\mathbb{P}^0_d(\mathscr{T}_h)]^d$ based on the Green's formula :
  $\mathfrak{G}_h^{\text{green}}(\mathbf{v}^{\mathscr{T}}, \mathbf{v}^{\mathscr{F}})|_T = \frac{1}{|T|_d} \sum_{F \in \mathscr{F}_T} |F|_{d-1}(v_F - v_T)\mathrm{n}_{T,F}$. -
  $\mathfrak{R}_{hT,F} = \frac{\sqrt{d}}{d_{T,F}}(u_F - u_T - \mathfrak{G}_h u \cdot (x_F - x_T))$
- This space allows a **Flux Reconstruction Operator**:
  $\mathfrak{F}_h(\mathbf{u})|_{F,T} = \sum_{F' \in \mathscr{F}_T} A_T^{FF'}(u_T - u_{F'})$,
  where :
  $A_T^{FF'} = \sum_{F" \in \mathscr{F}_T} y^{F"F} . v_{T,F''} y^{F"F}$
  $y^{F,F} = \frac{|F|}{|T|\mathrm{n}_{T,F}} + \frac{\sqrt{d}}{d_{T,F}}(1 - \frac{|F|}{|T|}\mathrm{n}_{T,F} \cdot (x_F - x_T))\mathrm{n}T, F$
  $y^{F,F'} = \frac{|F'|}{|T|\mathrm{n}_{T,F'}} - \frac{\sqrt{d}}{d_{T,F}|T|}|F'|\mathrm{n}_{T,F'} \cdot (x_F - x_T)\mathrm{n}T, F)$

Outline

DSEL design for FV methods : Key ingredients

Key ingredients to design a DSEL :

1. Meta-programming :
   - programs that transform types at compile time
2. Generic programming :
   - design generic components composed of abstract programs with generic types
3. Generative programming :
   - generate concrete programs, transforming types to create concrete types to use with abstract programs of generic components
4. Expression template :
   - expression tree representation of a problem
   - tools to describe, parse and evaluate a tree

DSEL design for FV methods : Principles

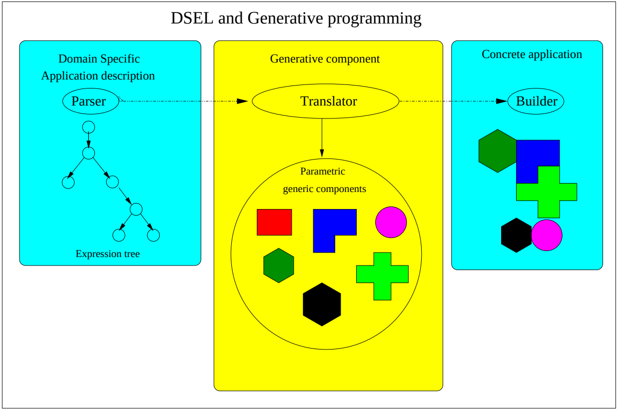


Figure: Generative programming

Outline

DSEL design for FV methods : Principles

Language Front ends and Back ends :

- **Front ends** : function space, test, trial functions, discrete variables ;
- **Back ends** : space of dofs, linear combination, matrix and vectors ;
- **DSEL** : linear and bilinear forms, bilinear operator (+,-,*,/) predefined keywords (**integrate**(.,.), **grad**(.), **flux**(.), **div**(.), **jump**(.), **avg**(.), **dot**(.,.))
- **Purpose** :
  - define linear and bilinear forms representing the discrete formulation of the PDE problem,
  - solve the problem evaluating the expressions of the forms.

Standard tools :

- Boost::Proto library to design the DSEL
- Boost::MPL, Fusion,. . . : MetaProgramming
- standard C++ : Generic Programming
- Arcane : C++ parallel framework providing mesh structures, network, IO services, post traitment tools,. . .
- External C++ libraries (Mesh, linear solvers,. . .)

DSEL design for FV methods : Principles

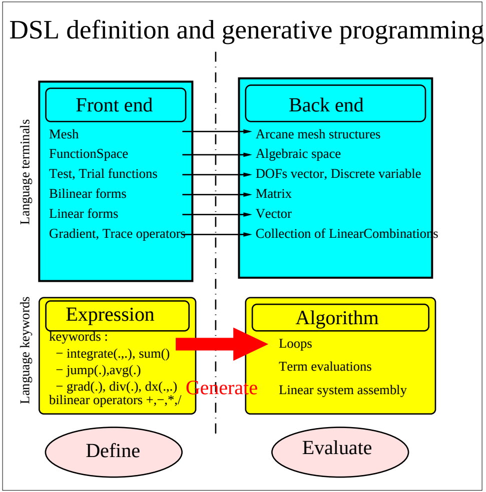
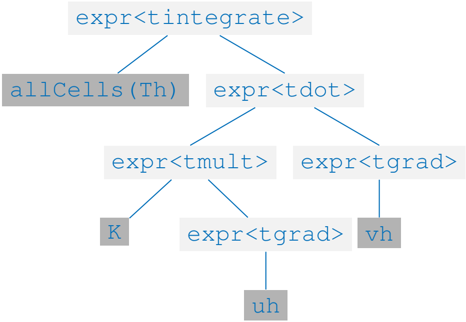


Figure: DSEL and generative programming

DSEL design for FV methods : Principles

```
Example of Bilinear Expression :
integrate(allCells(Th), dot(K*grad(u),grad(v)) ;
```



Tree expression representation

```
auto Th = mesh(itemGroupExpr) ;
auto Uh = trialSpace(trialExpr);
auto Vh = testSpace(testExpr);
LinearAlgebra::Matrix matrix(Uh,Vh);
std::for_each(itemGroupExpr.begin(),
              itemGroupExpr.end(),
              [& Th,& matrix](Cell&
{
  auto meas = measure(Th,cell);
  auto KGuGv =
   LCAlg::dot(eval(trialExpr,cell),
              eval(testExpr,cell));
  matrix.assemble(meas,KGuGv);
}
```

Generic algorithm for linear evaluation

Language Domain definition

```
template<typename Expr> struct FVDSLExpr;

struct FVDSLGrammar : proto::or_<
        proto::terminal<boost::proto::_>,
        proto::and_<proto::nary_expr<boost::proto::_,
        proto::vararg<FVDSLGrammar> > > {};

// Expressions in the FVDSL domain will be wrapped in FVDSLExpr<>
// and must conform to the FVDSLGrammar
struct FVDSLDomain
  : proto::domain<proto::generator<FVDSLExpr>, FVDSLGrammar>
{};

template<typename Expr>
struct FVDSLExpr
  : proto::extends<Expr, FVDSLExpr<Expr>, FVDSLDomain>
{
    explicit FVDSLExpr(Expr const &expr)
      : proto::extends<Expr, FVDSLExpr<Expr>, FVDSLDomain>(expr)
    {}

    BOOST_PROTO_EXTENDS_USING_ASSIGN(FVDSLExpr)
};
```

Boost proto implementation details : Domain definition

Language Domain definition :
Detecting user terminals
Defines all the overloads to make expressions involving terminals

```
// Define a trait type for terminal.
template<typename T>
struct IsFVDSLTerminal
  : mpl::or_< fvdsel::is_function_type<T>,
              fvdsel::is_base_type<T>,
              fvdsel::is_mesh_var<T>,
              fvdsel::is_mesh_group<T>,
            > {};

namespace PDEOps
{
    BOOST_PROTO_DEFINE_OPERATORS(IsFVDSLTerminal, FVDSLDomain)
}
```

Boost proto implementation details : Langage definition

Language definition Proto provides usefull tools to :

- Build expressions ;

```
namespace tag { struct tgrad{} ; struct tdot{] ; }
template<typename U>
typename proto::result_of::make_expr<tag::tgrad,
                                     FVDSELDomain, U const &>::type
grad(U const &u) {
  return proto::make_expr<tag::tgrad,FVDSELDomain>(boost::ref(u));
}
template<typename L, typename R>
typename proto::result_of::make_expr<tag::tdot,PDEDomain,
                                     L const &,R const &>::type
dot(L const& l, R const& r) {
  return proto::make_expr<tag::tdot,FVDSELDomain>(boost::ref(l),
                                                   boost::ref(r));
}
```

- to parse and introspect them ;

```
proto::display_expr( grad(u) ) ;
proto::right(dot(grad(u),grad(v))) ;
proto::left(dot(grad(u),grad(v)))) ;
proto::result_of::right<Expr>::type
proto::result_of::left<Expr>::type
```

Boost proto implementation details : Grammar definition

How to define a specifc grammar

EBNF grammar definition

```
LinearOperator = "grad"|"jump"|"avg";
TrialExpr      = TrialFunction | CoefExpr * TrialExpr |
                 "dot("CoefExpr, TrialExpr")" | LinearOperator"("TrialExpr")";
TestExpr       = TestFunction | CoefExpr*TestExpr | "dot("CoefExpr,TestExpr")"
                 LinearOperator"("TestExpr")";
BilinearTerm   = TrialExpr * TestExpr |"dot("TrialExpr, TestExpr")" |
                 CoefExpr * BilinearTerm | BilinearTerm + BilinearTerm;
```

Boost proto declaration

```
struct PlusBilinear
: proto::plus< BilinearGrammar, BilinearGrammar >{};
struct MultBilinear
: proto::multiplies< CoefExprGrammar, BilinearGrammar >{};
struct BilinearGrammar
: proto::or_<proto::multiplies<TrialExprGrammar,TestExprGrammar>,
             fvdsel::dotop<TrialExprGrammar,TestExprGrammar>,
             PlusBilinear,MultBilinear >{} ;
```

Boost proto implementation details : Grammar definition

How to define a specifc grammar

Proto provides standard meta-functions

Table: Proto standard tags and meta-functions

operator	narity	tag	meta-function
+	2	`proto::tag::`**plus**	`proto::`**plus**`<.,.>`
−	2	`proto::tag::`**minus**	`proto::`**minus**`<.,.>`
*	2	`proto::tag::`**mult**	`proto::`**mult**`<.,.>`
/	2	`proto::tag::`**div**	`proto::`**div**`<.,.>`

User can define specific meta-functions

Table: DSEL keywords

function	narity	tag	meta-function
integrate `(.,.)`	2	`fvdsel::tag::`**integrate**	**integrateop**`<.,.>`
grad `(.)`	1	`fvdsel::tag::`**grad**	**gradop**`<.>`
jump `(.)`	1	`fvdsel::tag::`**jump**	**jumpop**`<.>`
avg `(.)`	1	`fvdsel::tag::`**avg**	**avgop**`<.>`
dot `(.,.)`	2	`fvdsel::tag::`**dot**	**dotop**`<.,.>`

Boost proto implementation details : Grammar definition

How to design user specific meta-functions

User specific meta-function

```
//! grad metafunction
template<typename T>
struct gradop : proto::transform< gradop<T> > {
  // types
  typedef proto::expr< fvdsel::tag::grad,
                       proto::list1< T > >          type;
  typedef proto::basic_expr< fvdsel::tag::grad,
                             proto::list1< T > >    proto_grammar;

  template<typename Expr,typename State,typename Data>
  struct impl :
  proto::pass_through<gradop>::template impl<Expr, State, Data>{};
};
```

How to design user specific grammar

User specific Grammar structures

```
namespace fvdsel {
  template<typename ExprT>
  struct is_grad_expr :
    boost::is_same< typename boost::proto::tag_of<ExprT>::type,
                    fvdsel::tag::tgrad> {} ;

  struct GradGrammar;

  struct GradGrammarCases {
      // The primary template matches nothing:
      template<typename Tag>
      struct case_
        : proto::not_<proto::_> {};
  };

  template<>
  struct GradGrammarCases::case_<fvdsel::tag::tgrad>
    : proto::_ {};

  struct GradGrammar
    : proto::switch_<GradGrammarCases> {};
};
```

Boost proto implementation details : Grammar definition

How to design user specific grammar structures

DSL FACTORY MACROS to declare useful structures for unary or binary functions

```
//Macro to declare a function <myfunc> associated to a tag mytag
// unary
FVDSL_DEFINE_FUNC1(mytag, myfunc)
// binary
FVDSL_DEFINE_FUNC2(mytag, myfunc)      //reference,reference
FVDSL_DEFINE_FUNC2VR(mytag, myfunc)    //value,reference
FVDSL_DEFINE_FUNC2RV(mytag, myfunc)    //reference,value
FVDSL_DEFINE_FUNC2VV(mytag, myfunc)    //value,value

//Macro to declare a meta function <myfuncop> associated to a tag mytag
FVDSL_DEFINE_METAFUNC1(mytag, myfuncop)
FVDSL_DEFINE_METAFUNC2(mytag, myfuncop)

//Macro to declare grammar structures <mygram> associated to a tag mytag
FVDSL_DEFINE_GRAMMAR1(mytag, myfuncop)
FVDSL_DEFINE_GRAMMAR2(mytag, myfuncop)
```

Boost proto implementation details : Useful algorithms

How to implement algorithms

Two concepts to implement algorithms :

- Context objects for expression evaluation

```
EvalContextT<Cell> ctx(cell);
auto lcomb = proto::eval(grad(u),ctx) ;
auto rcomb = proto::eval(grad(v),ctx) ;
```

- Transform objects
  - kind of grammar objects to match expressions ;
  - transform expression and call specific algorithms.

Boost proto implementation details : Useful algorithm

How to implement algorithms

callable Transform object :

```
struct MultIntegrator
: proto::callable
{
    typedef Real result_type;
    template<typename LExprT,
             typename RExprT,
             typename StateT,
             typename DataT>
    result_type
    operator()(LExprT const& lexpr,
               RExprT const& rexpr,
               StateT& state,
               DataT const& data) const
    {
      // IMPLEMENT ALGORITHM
      return ... ;
    }
  };
```

Transform object :

```
struct BilinearIntegrator :
proto::or_
<
  proto::when< proto::multiplies<TrialFunctionGrammar,
                                 TestFunctionGrammar>,
               MultIntegrator(proto::_left,  //! lexpr
                              proto::_right, //! rexpr
                              proto::_state, //! state
                              proto::_data   //! context
                              )>,
  proto::when<
       fvdsel::dotprod<TrialFunctionGrammar,
                       TestFunctionGrammar>,
      DotIntegrator(proto::_child_c<0>, //! left
                    proto::_child_c<1>, //! trial
                    proto::_state,      //! state
                    proto::_data        //! context
                   )
              >
 > {} ;
```

```
IntegrateContextT<Cell> ctx(allCells());
Real value = 0. ;
Real result = BilinearIntegrator()(dot(grad(u),grad(v)),value,ctx) ;
```

Outline

Definition

Diffusion problem

$$\begin{cases} \nabla\cdot(-\nu\nabla u) = f & \text{in } \Omega, \\ u = g & \text{on } \partial\Omega, \end{cases}$$

We consider the following discrete variational formulations :

- G-method ;
- ccG-method ;
- Hybrid method.

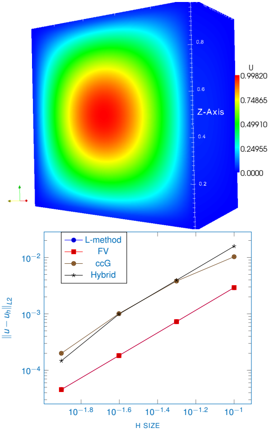


Figure: 3D view and convergence curves

G method

Variational formulation :

$u_h \in U_h^g$ and $v_h \in \mathbb{P}_d^0(\mathscr{T}_h)$

$$a_h^g(u_h, v_h) \stackrel{\text{def}}{=} \sum_{\sigma \in \mathscr{F}_h} \int_\sigma (\{\nabla u_h\} \cdot \mathrm{n}_\sigma)[\![v_h]\!]$$

$$b_h(v_h) \stackrel{\text{def}}{=} \int_\Omega f\, v_h$$

C++ PDE definition

```
MeshType Th ;
auto Uh = newGSpace(Th);
auto Vh = new P0Space(Th);
auto u = Uh->trial() ;
auto v = Vh->test() ;
BilinearForm ah_g =
  integrate(allFaces(Th),
            dot(N(Th),avg(grad(u)))*jump(v));
LinearForm bh =
   integrate(allCells(Th),f*v );
```

Applications : Diffusion problem

ccG method

ccG method
Variational formulation :

$(u_h, v_h) \in U_h^{ccg} \times U_h^{ccg}$,

$$a_h^{ccg}(u_h, v_h) \stackrel{\text{def}}{=} \int_\Omega \nu \nabla_h u_h \cdot \nabla_h v_h - \sum_{\sigma \in \Omega_h} \int_\sigma ([\![u_h]\!](\{\nu \nabla_h u_h\} \cdot \mathrm{n}_\sigma) (\{\nu \nabla_h u_h\} \cdot \mathrm{n}_\sigma)[\![v_h]\!]) \quad . \qquad (1)$$

C++ PDE definition

```
MeshType Th;
auto Uh = newCCGSpace(Th) ;
auto u = Uh->trial() ;
auto v = Uh->test() ;
BilinearForm ah_ccg =
 integrate( allCells(Th),
           dot(nu*grad(u),grad(v)) ) +
 integrate( allFaces(Th),
        -nu*jump(u)*dot(N(Th),avr(grad(v)))
        -nu*dot(N(Th),avr(grad(u)))*jump(v));
```

Hybrid method

Variational formulation :
$(u_h, v_h) \in U_h^{hyb} \times U_h^{hyb}$,

$$a_h^{hyb}(u_h, v_h) \stackrel{\text{def}}{=} \int_\Omega \nu \nabla_h u_h \cdot \nabla_h v_h \qquad (2)$$

C++ PDE definition

```
MeshType Th ;
auto Uh = newHybridSpace(Th) ;
auto u = Uh->trial() ;
auto v = Uh->test() ;
BilinearForm ah_hyb =
   integrate( allCells(Th),
              dot(nu*grad(u),grad(v)) ) ;
```

Applications : Diffusion problem

Performance analysis

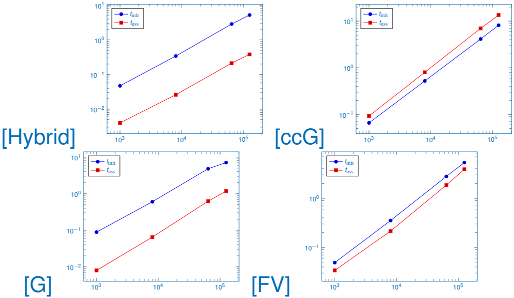


Figure: time vs. N_{DOF}

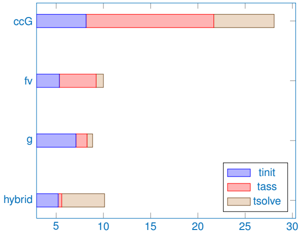


Figure: time vs. $N_{\mathrm{DOF}}, h = 0.02$

Outline

Continuous settings :
$\nu > 0$, $\beta \in \mathbb{R}^d$ and $\mu \geq 0$

$$\begin{cases} \nabla\cdot(-\nu\nabla u) = f & \text{in } \Omega, \\ u = g & \text{on } \partial\Omega_d, \\ \partial_n u = h & \text{on } \partial\Omega_n \end{cases}$$

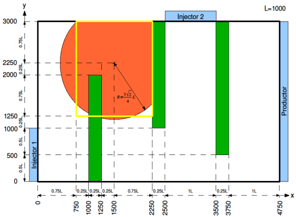
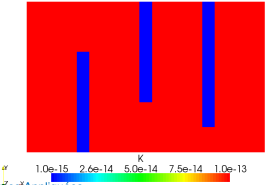


Figure: SHPCO2 problem

Variational formulation :
U_h a SUSHI function space,$(u_h, v_h) \in U_h \times U_h$,

$$a_h(u_h, v_h) \stackrel{\text{def}}{=} \int_\Omega \nu \nabla_h u_h \cdot \nabla v_h$$

$$b_h(v_h) \stackrel{\text{def}}{=} \int_\Omega f * v_h \tag{3}$$



Applications : Darcy problem

Constraint DSL extension

We need constraint DSL extensions:

- constraint expression ;
- new keywords :
  - **on** (<group_expr>,<constraint_expr>)
  - **trace** (<expr>)

```
Example of Constraint expressions :
on(boundaryFaces(Th),trace(u)=g) ;
```

Applications : Darcy problem

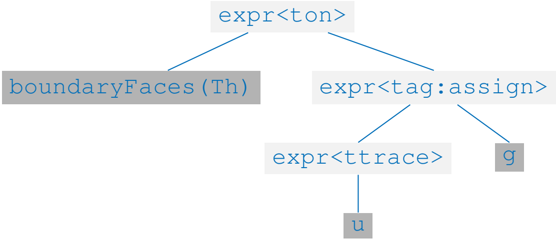
Constraint extension

```
//Define new user tags
namespace tag { struct ton{} ; struct ttrace{] ; }
//Define function, metafunction and grammar
FVDSL_DEFINE_FUNC2(ton,on)
FVDSL_DEFINE_METAFUNC2(ton,onop)
FVDSL_DEFINE_GRAMMAR2(ton,OnGrammar)

FVDSL_DEFINE_FUNC1(ttrace,trace)
FVDSL_DEFINE_METAFUNC1(ttrace,traceop)
FVDSL_DEFINE_GRAMMAR1(ttrace,TraceGrammar)
```

Stokes problem

Vectorial extension

```
Example of Constraint Expression :
on(boundaryFaces(Th), trace(u)=g ) ;
```



Darcy problem

C++ PDE definition

```
MeshType Th ;
auto Uh = newHybridSpace(Th) ;
auto u = Uh->trial() ;
auto v = Uh->test() ;
BilinearForm ah_hyb =
   integrate( allCells(Th),
                nu*dot(grad(u),grad(v)) )
LinearForm bh_hyb =
    integrate( allCells(Th),f*v ) ;

//Dirichlet boundary condition
ah_hyb +=
 on(boundaryFaces(Th,"dirichlet"),
    trace(u)=g );

//Neunman boundary condition
bh_hyb +=
 integrate(boundaryFaces(Th,"neumann"),
           h*trace(u));
```

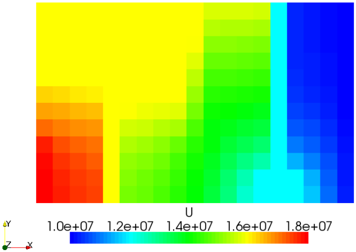


Figure: Darcy problem : solution

Outline

Stokes problem

Continuous settings :
$\Omega \subset \mathbb{R}^d$, $\mathbf{u} : \Omega \to \mathbb{R}^d$ and $p : \Omega \to \mathbb{R}$

$$\begin{cases} -\triangle \mathbf{u} + \nabla p = \mathbf{f} & \text{in } \Omega, \\ \nabla \cdot \mathbf{u} = 0 & \text{in } \Omega, \\ \mathbf{u} = \mathbf{g} & \text{on } \partial\Omega, \\ \int_\Omega p = 0, & \end{cases}$$

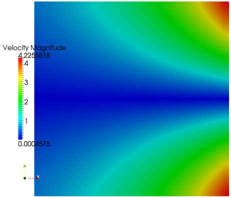


Figure: Stokes problem : norme u

Variational formulation : Find $(\mathbf{u}, p) \in [H_0^1(\Omega)]^d \times L_*(\Omega)$ such that

$$a(\mathbf{u},\mathbf{v}) + b(p,\mathbf{v}) - b(q,\mathbf{u}) = \int_\Omega f \cdot \mathbf{v} \quad \forall (\mathbf{v}, q) \in [H_0^1(\Omega)]^d \times L_*(\Omega),$$

$$a(\mathbf{u},\mathbf{v}) \stackrel{\text{def}}{=} \int_\Omega \nabla \mathbf{u} : \nabla \mathbf{v}, \quad b(q,\mathbf{v}) \stackrel{\text{def}}{=} -\int_\Omega \nabla q \cdot \mathbf{v} = \int_\Omega q \nabla \cdot v.$$

Set $c((\mathbf{u},p),(\mathbf{v},q)) \stackrel{\text{def}}{=} a(\mathbf{u},\mathbf{v}) + b(p,\mathbf{v}) - b(q,\mathbf{u})$.

Vectorial extension

We need vectorial extensions:

- vectorial terminals ;
- Range and index terminals
- new keywords **sum** `(<range>)[scalar_view<vectorial_expr>]`

```
Example of Vectorial expressions :
 IndexType _i(dim), _j(dim) ;
//∇u:∇v
sum(_i)[ dot(grad(u(_i),grad(v(_i)) ] ;

//∑_{i,j} ∂_j u_i ∂_j v_i
sum(_i,_j)[ dx(_j,u(_i))*dx(_j,v(_i)) ] ;

//∇·u
div(u) ;
sum(_i)[ dx(_i)[u] ] ;
```

Stokes problem

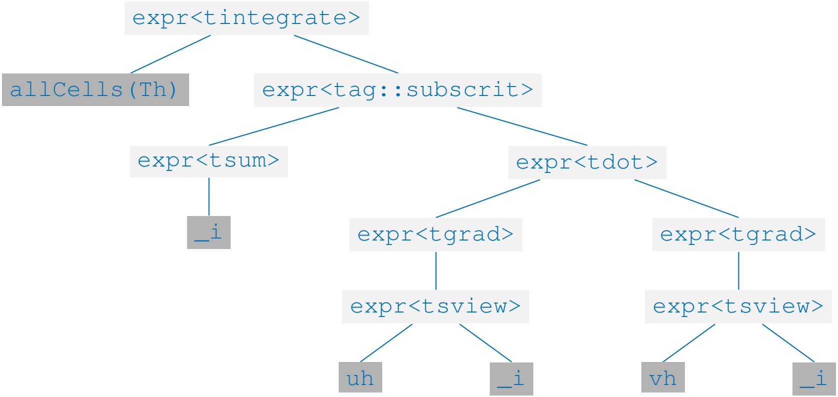
Vectorial extension

```
//Define new user tags
namespace tag {
 struct tsum{} ;       //to manage sum(.) expression
 struct tsview{} ;     //to manage scalar view of vectorial expression
 struct tdx{} ;
}
//Define function, metafunction and grammar
FVDSL_DEFINE_FUNC1(tsum,sum)
FVDSL_DEFINE_METAFUNC1(tsum,sumop)
FVDSL_DEFINE_GRAMMAR1(tsum,SumGrammar)

FVDSL_DEFINE_FUNC2(tsum,sum)
FVDSL_DEFINE_METAFUNC2(tsum,sum2op)
FVDSL_DEFINE_GRAMMAR2(tsum,Sum2Grammar)

FVDSL_DEFINE_FUNC2(tdx,dx)
FVDSL_DEFINE_METAFUNC2(tdx,dxop)
FVDSL_DEFINE_GRAMMAR2(tdx,DxGrammar)
```

Stokes problem

Vectorial extension

```
Example of Bilinear Expression :
integrate(allCells(Th), sum(_i)[ dot(grad(u(_i),grad(v(_i)) ] ;
```



C++ PDE definition

```
MeshType Th ;
auto Uh = newCCGSpace(Th) ;
auto Ph = newP0Space(Th) ;
auto u = Uh->trialArray(Th::dim) ;
auto v = Uh->testArray(Th::dim) ;
auto p = Ph->trial() ;
auto q = Ph->test() ;
FVDomain::algo::Range<1> _i(dim) ;
BilinearForm ah = integrate( allCells(Th),
    sum(_i)[    dot(grad(u(_i)),grad(v(_i))]  ))
                + integrate( Internal<Face>::items(Th),
    sum(_i)[  -dot(N(Th),avg(grad(u(_i))))*jump(v(_i))
              -jump(u(_i))*dot(N(),avg(grad(v(_i))))
              + eta/H(Th)*jump(u(_i))*jump(v(_i))  ]  ) ;
BilinearForm bh =  integrate( allCells(Th), -id(p)*div(v)  )
        + integrate( allFaces(Th), avg(p)*dot(fn,jump(v)  )  ) ;
BilinearForm bth = integrate( allCells(Th),div(u)*id(q)  )
  + integrate( allFaces(Th), -dot(N(Th),jump(u)) * avg(q)  ) ;
BilinearForm sh = integrate(internalFaces(Th),
                            H(Th)*jump(p)*jump(q));
LinearForm fh = integrate( allCells(Th),
                           sum(_i)[ f(_i)*v(_i) ]) ;
```

Outline

Multiscale pressure solver

Continuous settings

Two level mesh :
- $\mathscr{T}_h^f$ and $\mathscr{T}_h^c$

Fine problem on $\mathscr{T}_h^f$:

$$\begin{cases} v = -\nu\nabla u & \text{in } \Omega, \\ \nabla\cdot(-\nu\nabla u) = f & \text{in } \Omega, \\ u = g & \text{on } \partial\Omega, \\ \partial_n u = f & \text{on } \partial\Omega, \end{cases}$$

Basis function definition Φ_{F_c} :

$$\begin{cases} \nabla\cdot(-\nu\nabla\Phi_{F_c}) = w & \text{in } \Omega_{F_c}, \\ w = \frac{trace(\nu)}{\int_{\Omega_{F_c}} \nu} & \text{on } \Omega_{F_c}^{front} \\ w = -\frac{trace(\nu)}{\int_{\Omega_{F_c}} \nu} & \text{on } \Omega_{F_c}^{back} \\ \partial_n u = 0 & \text{on } \partial\Omega_{F_c}, \end{cases}$$

Coarse problem definition on $\mathscr{T}_h^c$:

$U^{hms} = \mathbb{P}_d^0(\mathscr{T}_h) + span < \Phi_{F_c} >_{F_c \in \mathscr{F}_{h\Omega_c}}$

Find $\mathbf{u} \in U^{hms}$,

$\mathbf{u} = \sum_{K_c} \mathbf{u}_{K_c} \chi_{K_c} + \sum_{F_c} \mathbf{v}_{F_c} \Phi_{F_c}$

Multiscale method : basis function support

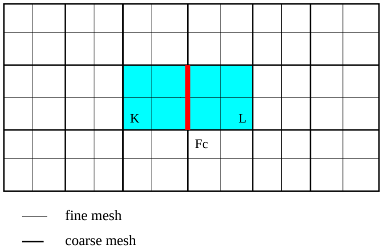


Figure: Basis function

Multiscale pressure solver

Variational formulation

Basis function problem Φ_{F_c}:
$(u_h, v_h) \in U_h^{hyb} \times U_h^{hyb}$,

$$\begin{cases} a_h^{hyb}(u_h, v_h) & \stackrel{\text{def}}{=} \int_{\Omega_b} \nu \nabla_h u_h \cdot \nabla_h v_h \\ b_h(v_h) & \stackrel{\text{def}}{=} \int_{\Omega_b} w * v_h \end{cases}$$

Coarse problem :
$\mathbf{P_c} = \sum_{K_c} \mathbf{p}_{K_c} \chi_{K_c} + \sum_{F_c} \mathbf{v}_{F_c} \Phi_{F_c}$
$(u_h, v_h) \in U_h^{hms} \times U_h^{hms}$,

$$\begin{cases} a_h^{hms}(u_h, v_h) & \stackrel{\text{def}}{=} \int_{\Omega} \nabla_h \nu u_h \cdot \nabla_h v_h \\ & - \sum_{\sigma \in \Omega_h} \int_\sigma ([\![u_h]\!](\{\nu \nabla_h u_h\} \cdot \mathrm{n}_\sigma) + (\{\nu \nabla_h u_h\} \cdot \mathrm{n}_\sigma)[\![v_h]\!]) \\ & + \sum_{\sigma \in \Omega_h} \int_\sigma \frac{\eta}{h} [\![u_h]\!][\![u_h]\!] \end{cases}$$

Fine solution :
$\mathbf{v_f} = \mathbf{flux}(\mathbf{P_c}) = \sum_{F_c} \mathbf{v}_{F_c} \mathbf{flux}(\Phi_{F_c})$

C++ PDE definition

```
MultiscaleMeshType Th(/* ... */) ;
//COARSE PROBLEM DEFINITION
auto Uh = HMSSpaceType::create(Th) ;
/* ... */
auto u = Uh->trial() ;
auto v = Uh->test() ;
BilinearForm ah =
   integrate( allCells(Th),dot(grad(u),grad(v)) ) +
   integrate( allFaces(Th),-jump(u)*dot(N(Th),avg(grad(v)))
                           -dot(N(Th),avg(grad(u)))*jump(v)
                           +eta/H(Th)*jump(u)*jump(v));
ah += on(boundaryFaces(Th), u=ud ) ; //! dirichlet condition
LinearComputeContext lctx(solver) ;
fvdsel::eval(ah,lctx) ;
solver.solve() ;

//FINE PROBLEM SOLUTION
FaceRealVariable& fine_velocity = /* ... */ ;
DownScaleEvalContext dctx(fine_velocity) ;
fvdsel::eval(downscale(allCells(Th),flux(u)),dctx) ;
```

Many computations are independant :

- Basis computations;
- assembling computations;
- downscaling computations.

New hybrid architectures provide different ways of optimization :

- GP-GPU;
- multi-core parallelism;
- multi-node parallelism.

The DSEL separates the numerical level from the back end level. Optimisations are easily handled at the low level.

Test 1D

Results

Test 1D : fine size 100, coarse size 10

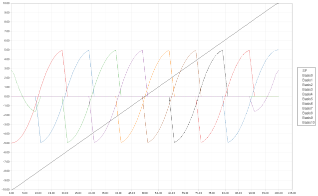


Figure: Basis functions, K=1

Test 1D : fine size 2048, coarse size 16

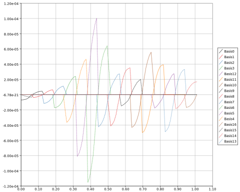


Figure: Basis functions

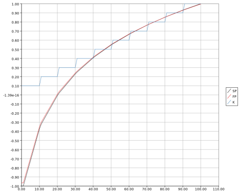


Figure: Multi scale vs Fine solutionK=0.1 to 1

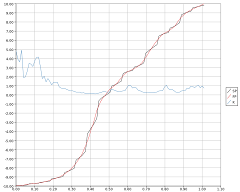


Figure: Multi scale vs Fine solution

Results of the SPE10 study case:

Fine mesh : 65x220x1
Coarse mesh : 10x10x1
Boundary conditions :
-$P_{xmin} = -10$
-$P_{xmax} = 10$

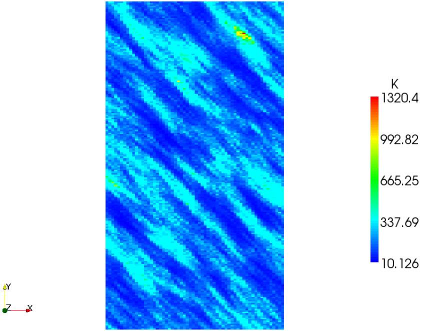


Figure: Fine permeability

SPE10

Results

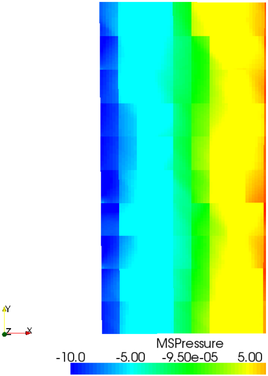


Figure: MS solution

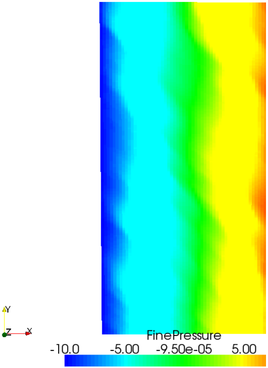


Figure: Fine solution

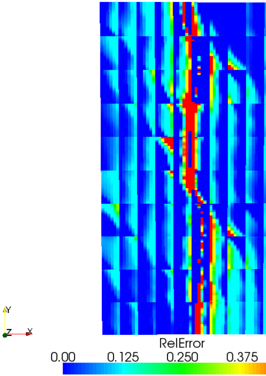


Figure: Relative error

Conclusion

- A new DSEL for lowest order methods ;
- Recover various methods (L-scheme, ccG, Hybrid method) ;
- Implementation of non trivial academic test cases ;
- Performance issues (language overhead, benchmarcks with hand written codes).

Conclusions

Benefits of Boost.Proto framework :

- Productivity for the developper :
  - DSEL to design DSEL ;
  - a lot of of useful generic tools ;
  - enable to design easily complex DSEL ;
  - DSEL can be easily extended ;
  - DSEL Factory to easily extend Proto standard tools.
- Productivity for the end user :
  - Language to design complexe numerical methods ;
  - Language that seperates concerns :
    - mathematics, numerics ;
    - computer science, high performance computing;

Perspectives

- Extend the DSL for :
  - various types of boundary conditions ;
  - non linearity with Frechets derivatives.
- HAMM ANR projects : Multi-scale models and hybrid architecture
  - extend the DSEL for multi scale methods ;
  - use GPU back ends for linear solvers ;
  - take into account hardware specifications :
    - multi nodes ;
    - multi cores ;
    - general purpose accelerators.
- New business applications :
  - Linear elasticity ;
  - poro-mecanic ;
  - dual medium model.

Links

Some links :

- Boost : www.boost.org
- Boost.Proto : www.boost.org/libs/proto
- HAMM projects : www.hamm-project.org
- Arcane framework : POOSC '09 Proceedings of the 8th workshop on Parallel/High-Performance Object-Oriented Scientific Computing
- Dune framework : www.dune-project.org
- Fenics : fenicsproject.org
- Feel++ : www.feelpp.org

- Thank you for attention
- Questions?