Implementing a Domain Specific Embedded Language for lowest-order variational methods with Boost Proto

Jean-Marc Gratien

Implementing a Domain Specific Embedded Language for lowest-order variational methods with Boost Proto

Jean-Marc GRATIEN
IFPEN
1 et 4 av Bois Préau
92588 Rueil-Malmaison, FRANCE
jean-marc.gratien@ifpen.fr

ABSTRACT

In this paper we propose an original implementation for a large family of lowest-order methods to solve diffusive problems with a `FreeFEM`-like domain specific language targeted at defining discrete linear and bilinear forms. We discuss how by using the Boost Proto framework we have developped the back-end and the front-end of the language.We validate the proposed DSEL design by the implementation of several academic problems. The overhead of the language is evaluated by comparing with a more traditional implementation.

General Terms

Concepts and Generic Programming

Keywords

DSEL, Generative programming, Framework, Boost Proto

1. INTRODUCTION

Industrial simulation software have to manage: *(i)* the complexity of the underlying physical models, usually expressed in terms of a PDE system completed with algebraic closure laws, *(ii)* the complexity of numerical methods used to solve the PDE systems, and finally *(iii)* the complexity of the low level computer science services required to have efficient software on modern hardware. Robust and effective finite volume (FV) methods as well as advanced programming techniques need to be combined in order to fully benefit from massively parallel architectures (implementation of parallelism, memory handling, design of connections). Moreover, the above methodologies and technologies become more and more sophisticated and too complex to be handled by physicists alone. Nowadays, this complexity management becomes a key issue for the development of scientific software. Some frameworks already offer a number of advanced tools to deal with the complexity related to parallelism in a transparent way. Hardware complexity is hidden and low level algorithms which need to deal directly with hardware specificity, for performance reasons, are provided. They often offer services to manage mesh data services and linear algebra services which are key elements to have efficient parallel software. However, all these frameworks often provide only partial answers to the problem as they only deal with hardware complexity and low level numerical complexity like linear algebra. The complexity related to discretization methods and physical models lacks tools to help physicists to develop complex applications. New paradigms for scientific software must be developed to help them to seamlessly handle the different levels of complexity so that they can focus on their specific domain. Generative programming, component engineering and domain-specific languages (either DSL or DSEL) are key technologies to make the development of complex applications easier to physicists, hiding the complexity of numerical methods and low level computer science services. These paradigms allow to write code with a high level expressive language and take advantage of the efficiency of generated code for low level services close to hardware specificities. Their application to Scientific Computing has been up to now limited to Finite Element (FE) methods, for which a unified mathematical framework has been existing for a long time. Such kind of DSL have been developped for finite element or Galerkin methods in projects like `Freefem`, `Getdp`, `Getfem++`, `Sundance`, `Feel++`, `Fenics` project. We try to extend this kind of approach to lowest order methods to solve the PDE systems of geo modeling applications. A recent consistent unified mathematical frame allows a unified description of a large family of these methods, and enable then, as for FE methods, the design of a high level language inspired from the mathematical notation, that could helps physicist to implement their application writing the mathematical formulation at a high level, hiding the complexity of numerical methods and low level computer science services guaranty of high performance. We have developped such language, that we have embedded in the C++ language, on top of Arcane plateform [15], with the Boost Proto library [16], a powerful framework providing tools to design DSEL. We focus on the main ingredients of the language and detail how using the Boost Proto framework we have implemented in a user friendly declarative way our new language. We check the capability of our DSEL to allow the description and the resolution of various and complex problems with different lowest-order methods. We validate the design of the DSEL on the implementation of several academic problems. We present some numerical results and compare the performance of their implementation with the DSEL to their

C++Now Aspen CO, USA, May 13–18, 2012

hand written counterpart, evaluating in that way the overhead of the language in order to illustrate the interest of the C++ language as the host language for our DSL.

The paper is organized as follows : in the first section we present the mathematical domain targetted by the proposed DSEL, in the second session we discuss the implementation of the DSEL with Boost Proto framework then in the last session we validate our approach with numerical results.

2. MATHEMATICAL SETTING

The unified mathematical frame presented in [11, 12] allows a unified description of a large family of lowest-order methods. The key idea is to reformulate the method at hand as a (Petrov)-Galerkine scheme based on a possibly incomplete, broken affine space. This is done by introducing a piecewise constant gradient reconstruction, which is used to recover a piecewise affine function starting from cell (and possibly face) centered unknowns.

For example, considering the following heterogeneous diffusion model problem(2):

$$\begin{aligned} -\nabla\cdot(\kappa\nabla u) &= f \quad \text{in } \Omega, \\ u &= 0 \quad \text{on } \partial\Omega, \end{aligned}$$

with source term $f \in L^2(\Omega)$, κ piecewize constant.

The continuous weak formulation reads: Find $u \in H_0^1(\Omega)$ such that

$$a(u,v) = 0 \quad \forall v \in H_0^1(\Omega),$$

with

$$a(u,v) \stackrel{\text{def}}{=} \int_\Omega \nabla u\cdot\nabla v.$$

In this framework, for a given partition $\mathcal{T}_h$ of Ω, a specific lowest-order method is defined by (i) selecting a trial function space $U_h(\mathcal{T}_h)$ and a test function space $V_h(\mathcal{T}_h)$, (ii) defining for all $(u_h, v_h) \in U_h \times V_h$ a bilinear form $a_h(u_h, v_h)$ and a linear form $b_h(v_h)$, Solving the discrete problem consists then in finding $u_h \in U_h$ such that:

$$a_h(u_h, v_h) = b_h(v_h) \quad \forall v_h \in V_h,$$

The definition of a discrete function space U_h is based on three main ingredients :

- $\mathcal{T}_h$ the mesh representing Ω, $\mathcal{S}_h$ a submesh of $\mathcal{T}_h$ where $\forall S \in \mathcal{S}_h, \exists T_S \in \mathcal{T}_h,\ S \subset T_S$;
- $\mathbb{V}_h$ the space of vector of degree of freedoms with components indexed by the mesh entities (cells, faces or nodes) ;
- $\mathfrak{G}_h$ a linear gradient operator that defines for each vector $v_h \in V_h$ a constant gradient on each element of $\mathcal{S}_h$ and ∇_h the broken gradient operator.

Using the above ingredients, we can define for all $\mathbf{v}_h \in \mathbb{V}_h$ a piecewise affine function $v_h \in U_h \subset \mathbb{P}_d^1(\mathcal{S}_h)$ such that: $\forall S \in \mathcal{S}_h,\ S \subset T_S,\ T_S \in \mathcal{T}_h, \forall \mathbf{x} \in S$,

$$v_h(\mathbf{x})|_S = v_{T_S} + \mathfrak{G}_h(\mathbf{v}_h)|_S\cdot(\mathbf{x} - \mathbf{x}_{T_S}).$$

Usually three kind of submesh $\mathcal{S}_h$ are considered : $\mathcal{T}_h$ the mesh itself, $\mathcal{P}_h$ the submesh with pyramidal subcells based on the face entities of $\mathcal{T}_h$ and $\mathcal{N}_h$ with subcells based on the nodes of $\mathcal{T}_h$.

We denote $\mathbb{T}_h$ the space of degree of freedoms with components indexed by cells and $\mathbb{F}_h$ the space of degree of freedoms with components indexed only by faces. Ussually the following choices are considered:

$$\mathbb{V}_h = \mathbb{T}_h \text{ or } \mathbb{V}_h = \mathbb{T}_h \times \mathbb{F}_h. \tag{1}$$

With this framework, the model problem can be solved with various methods :

- the cell centered Galerkin (ccG) method and the G-method with cell unknowns only ;
- the hybrid finite volume method with both cell and face unknowns that recover the mimetic finite difference (MFD) and mixed/hybrid finite volume (MHFV) family.

The G-method [4].

:

The trial space for is obtained with (i) $\mathcal{S}_h = \mathcal{P}_h$, (ii) $\mathbb{V}_h = \mathbb{T}_h$, (iii) and $\mathfrak{G}_h = \mathfrak{G}_h^{\text{g}}$ a gradient operator, piecewise constant on the elements $S \in \mathcal{P}_h$, base on the L construction, detailed in[4]. The method reads then :

$$\text{Find } u_h \in V_h^{\text{g}} \text{ s.t. } a_h^{\text{g}}(u_h, v_h) = \int_\Omega f v_h \text{ for all } v_h \in \mathbb{P}_d^0(\mathcal{T}_h),$$

where $a_h^{\text{g}}(u_h, v_h) \stackrel{\text{def}}{=} \sum_{F\in\mathcal{F}_h} \int_F \{\kappa\nabla_h u_h\}\cdot\mathrm{n}_F[\![v_h]\!]$ with .

The cell centered Galerkin method [9, 10].

: We introduce the linear gradient operator $\mathfrak{G}_h^{\text{green}} : \mathbb{T}_h \times \mathbb{F}_h \to [\mathbb{P}_d^0(\mathcal{T}_h)]^d$ such that, for all $(\mathbf{v}^{\mathcal{T}}, \mathbf{v}^{\mathcal{F}}) \in \mathbb{T}_h \times \mathbb{F}_h$ and all $T \in \mathcal{T}_h$,

$$\mathfrak{G}_h^{\text{green}}(\mathbf{v}^{\mathcal{T}}, \mathbf{v}^{\mathcal{F}})|_T = \frac{1}{|T|_d} \sum_{F\in\mathcal{F}_T} |F|_{d-1}(v_F - v_T)\mathrm{n}_{T,F}. \tag{2}$$

The discrete space for the ccG method is obtained with: (i) $\mathcal{S}_h = \mathcal{T}_h$, (ii) $\mathbb{V}_h = \mathbb{T}_h$, (iii) $\mathfrak{G}_h = \mathfrak{G}_h^{\text{ccg}}$ with $\mathfrak{G}_h^{\text{ccg}}$ such that

$$\forall \mathbf{v}_h \in \mathbb{V}_h, \qquad \mathfrak{G}_h^{\text{ccg}}(\mathbf{v}_h) = \mathfrak{G}_h^{\text{green}}(\mathbf{v}_h, \mathbf{T}_h^{\text{g}}(\mathbf{v}_h)). \tag{3}$$

where $\mathbf{T}_h^{\text{g}}$ is a linear trace reconstruction operator on the faces of $\mathcal{T}_h$.

Let for all $(u_h, v_h) \in V_h^{\text{ccg}} \times V_h^{\text{ccg}}$,

$$\begin{aligned} a_h^{\text{ccg}}(u_h, v_h) \stackrel{\text{def}}{=} & \int_\Omega \kappa\nabla_h u_h\cdot\nabla_h v_h \\ & - \sum_{F\in\mathcal{F}_h} \int_F [\{\kappa\nabla_h u_h\}_\omega\cdot\mathrm{n}_F[\![v_h]\!] + [\![u_h]\!]\{\kappa\nabla v_h\}_\omega\cdot\mathrm{n}_F] \\ & + \sum_{F\in\mathcal{F}_h} \eta\frac{\gamma_F}{h_F}\int_F [\![u_h]\!][\![v_h]\!], \end{aligned} \tag{4}$$

The method reads :

$$\text{Find } u_h \in V_h^{\text{ccg}} \text{ s.t. } a_h^{\text{ccg}}(u_h, v_h) = \int_\Omega f v_h \text{ for all } v_h \in V_h^{\text{ccg}}. \tag{5}$$

The hybrid finite volume method.

recovers the SUSHI scheme[14, 13, 8, 7]. The discrete space with hybrid unknowns is then obtained with: (i) $\mathcal{S}_h = \mathcal{P}_h$, (ii) $\mathbb{V}_h = \mathbb{T}_h \times \mathbb{F}_h$, (iii) $\mathfrak{G}_h = \mathfrak{G}_h^{\text{hyb}}$ with $\mathfrak{G}_h^{\text{hyb}}$ such that,

for all $(\mathbf{v}_h^{\mathcal{T}}, \mathbf{v}_h^{\mathcal{F}}) \in \mathbb{T}_h \times \mathbb{F}_h$, all $T \in \mathcal{T}_h$ and all $F \in \mathcal{F}_T$,

$$\mathfrak{G}_h^{\rm hyb}(\mathbf{v}_h^{\mathcal{T}}, \mathbf{v}_h^{\mathcal{F}})|_{\mathcal{P}_{T,F}} = \mathfrak{G}_h^{\rm green}(\mathbf{v}_h^{\mathcal{T}}, \mathbf{v}_h^{\mathcal{F}})|_T + \mathfrak{r}_h(\mathbf{v}_h^{\mathcal{T}}, \mathbf{v}_h^{\mathcal{F}})|_{\mathcal{P}_{T,F}} \mathrm{n}_{T,F}. \tag{6}$$

where the linear residual operator $\mathfrak{r}_h : \mathbb{T}_h \times \mathbb{F}_h \to \mathbb{P}_d^0(\mathcal{P}_h)$ is defined as follows: For all $T \in \mathcal{T}_h$ and all $F \in \mathcal{F}_T$,

$$\mathfrak{r}_h(\mathbf{v}_h^{\mathcal{T}}, \mathbf{v}_h^{\mathcal{F}})|_{\mathcal{P}_{T,F}} = \frac{d^{\frac{1}{2}}}{d_{T,F}} \left[v_F - v_T - \mathfrak{G}_h^{\rm green}(\mathbf{v}_h^{\mathcal{T}}, \mathbf{v}_h^{\mathcal{F}})|_T \cdot (\mathbf{x}_F - \mathbf{x}_T) \right].$$

This method with hybrid unknowns reads :

Find $u_h \in V_h^{\rm hyb}$ s.t. $a_h^{\rm sushi}(u_h, v_h) = \displaystyle\int_\Omega f v_h$ for all $v_h \in V_h^{\rm hyb}$,

with

$$a_h^{\rm sushi}(u_h, v_h) \stackrel{\rm def}{=} \int_\Omega \kappa \nabla_h u_h \cdot \nabla_h v_h, \tag{7}$$

and ∇_h broken gradient on $\mathcal{P}_h$.

3. IMPLEMENTATION

The framework described in §2 allows a unified description for a large family of lowest methods and as for FE/DG methods, it enables the design of a high level language inspired from the mathematical notation. Such language enables to express the variational discretisation formulation of PDE problem with various methods defining bilinear and linear forms. Algorithms are then generated to solve the problems, evaluating the forms representing the discrete problem. The language is based on concepts (mesh, function space, test trial functions, differential operators) close to their mathematical counterpart. They are the front end of the language. Their implementations use algebraic objects (vectors, matrices, linear operators) which are the back end of the language. Linear and bilinear forms are represented by expressions built with the terminals of the language linked with unary, binary operators (`+,-,*,/,dot(.,.)`) and with free functions like **`grad(.)`**, **`div(.)`** **`integrate(.,.)`**. The purpose of theses expressions is first to express the variational discretization formulation of the user problem but also to solve and find the solution of the problem by evaluating them with specific context.

In the first part of this section, we present the different C++ concepts defining the front end of our language, their mapping onto their mathematical counterpart and their links with algebraic objects corresponding to the back end of the language. We then introduce the DSEL that enables to manipulate these concepts to build complex expressions close to the mathematical discretisation formulation of continuous PDE problems. We finally explain how, evaluating these expressions, we can generate source codes that solve discrete problems.

For our diffusion model problem (2), such DSEL will for instance achieve to express the variational discretization formulation 5 with the programming counterpart presented in listing 1.

Listing 1: Diffusion problem implementation

```
MeshType Th;
Real K;
auto Vh = newSUSHISpace(Th);
auto u = Vh->trial("U");
auto v = Vh->test("V");
BilinearForm a =
  integrate(allCells(Th),dot(K*grad(u),grad(v)));
LinearForm b =
  integrate(allCells(Th),f*v);
```

3.1 Algebraic back-end

In this section we focus on the elementary ingredients used to build the terms appearing in the linear and bilinear forms of §2, which constitute the back-end of the DSEL presented in §3.3.

Mesh.

The mesh concept is an important ingredient of the mathematical frame. Mesh types and data structures are a very standard issue and different kinds of implementation already exist in various framework. We developped above Arcane mesh data structures a **mesh concept** defining
(i) `MeshType::dim` the space dimension, (ii) the subtypes `Cell`, `Face` and `Node` for mesh element of dimension respectively `MeshType::dim`, `MeshType::dim-1` and 0. Some free functions like `allCells(<mesh>)`, `allFaces(<mesh>)`, `boundaryCells(<mesh>)`, `boundaryFaces(<mesh>)`, `internalCells(<mesh>)`, `internalFaces(<mesh>)` are provided to manipulate the mesh and to extract different parts of the mesh.

Vector spaces, degrees of freedom and discrete variables.

The `class Variable` with template parameters `ItemT` and `ValueT` manages vectors of values of type `ValueT` and provides data accessors to these values with either mesh elements of type `ItemT`, integer ids or iterators identifying these elements. Instances of the class `Variable` are managed by `VariableMng`, a class that associates each variable to its unique string key label corresponding to the variable name.

Linear combination, linear and bilinear contribution.

The point of view presented in §2 naturally leads to a finite element-like assembly of local contributions stemming from integrals over elements or faces. This procedure leads to manipulate local vectors indexed by mesh entities represented by the concept of `class LinearCombination`. Associated to an efficient linear algebra, this concept enable to create `LinearContribution` (local vectors) and `BilinearContribution` (local matrices) used in the assembly procedure of the global matrix and vector of the global linear system.

3.2 Functional front-end

Function spaces.

Incomplete broken polynomial spaces defined by (2) are mapped onto `C++` types according to the `FunctionSpace` concept. The key role of a `FunctionSpace` is to bridge the gap between the algebraic representation of DOFs and the functional representation used in the methods of §2. This is achieved by the functions **grad** and **eval**, which are the `C++` counterparts of respectively the linear operators $\mathfrak{G}_h$ and $\mathfrak{R}_h$. More specifically,

(i) for all $S \in \mathcal{S}_h$, **`grad`**`(`S`)` returns a vector-valued linear combination corresponding to the (constant) restric-

tion $\mathfrak{G}_h|_S$;

(ii) for all $S \in \mathcal{S}_h$ and all $\mathbf{x} \in S$, **eval**$(S, \mathbf{x})$ returns a scalar-valued linear combination corresponding to $\mathfrak{R}_h|_S(\mathbf{x})$ defined according to (2).

The linear combinations returned by **grad** and **eval** can be used to build `LinearContributions` and `BilinearContributions` as described in the previous sections.

Function space types also define the sub types `FunctionType`, `TestFunctionType` and `TrialFunctionType` corresponding to the mathematical notions of discrete functions, test and trial functions in variational formulations. Instances of `TrialFunctionType` and `FunctionType` are associated to a `Variable` object containing a vector of DOFs stored in memory associated to a string key corresponding to the variable name. For functions, the vector of DOFs is used in the evaluation on a point $x \in \Omega$ while for trial functions, this vector is used to receive the solution of the discrete problem. Test functions implicitely representing the space basis then are not associated to any `Variable` objects, neither vector of DOFs. Unlike `FunctionType`, the evaluation of `TrialFunctionType` and `TestFunctionType` is lazy in the sense that it returns a linear combination. This linear combination can be used to build local linear or bilinear contributions to the global system, or enables to postpone the evaluation with the variable data.

The **BilinearForm** and **LinearForm** concept represent the linear and bilinear forms described in 2. They allow to define expressions using test and trial functions, unary and binary operators.

3.3 DSEL implementaion

The main goal of the DSEL is to allow a notation as close as possible to the mathematical notation presented in §2. This section focuses bilinear forms, as the ingredients for linear forms are essentially similar. The exposition is not meant to be exhaustive, but instead to present a few significant examples from which others can be inferred. We first define our DSEL giving some production rules that enable to create trial and test expressions as well as bilinear terms using the Extended Backus–Naur Form (EBNF)[1], then we detail how the DSEL has been implemented using the tools provided by the Boost Proto framework.

3.3.1 Language definition

Terminals and Keywords.

The DSEL *Terminals* are composed of a number predefined types categorized in the following families: (i) the `BaseType` family for the standard `C++` types representing integers and reals; (ii) the `VarType` family for all discrete variable types defined in §3.1; (iii) the `MeshGroupType` family for types representing collection of mesh entities such as the ones listed in §3.1; (iv) the `Function`, `TestFunction` and `TrialFunction` families representing the functions, test and trial functions defined in §3.2.

The DSEL is based on some predefined keywords listed in table 1 semantically closed to their counterpart in the matematical framework.

Trial and test expressions.

Trial (resp. test) expressions are obtained as the product of a coefficient γ_u (resp. γ_v) by a linear operator $\mathcal{L}_u$ (resp. $\mathcal{L}_v$) acting on a trial (resp. test) function. The coefficient can result from the algebraic combination of constant values and `Variable`s evaluated at item *I*. Listing 2 defines the production rules that enable to create coefficient expressions involving, in particular, constant values, `Variable`s over `Cell`s and products thereof.

Listing 2: Examples of production rules for the coefficient γ

```
BaseExpr = BaseType | BaseExpr * BaseExpr;
VarExpr  = VarType | BaseExpr * VarExpr | VarExpr * VarExpr;
CoefExpr = BaseExpr | VarExpr;
```

To obtain trial and test expressions, we introduce linear operators acting on test and trial functions. A few examples are provided in Listing 3, and include (i) **grad**, the gradient of the trial/test function; (ii) trace operators like **jump** and **avg** representing, respectively, the jump and average of a trial/test function across a face. Besides linear operators, the production rules for trial and test expressions in Listing 3 include various products by coefficients resulting from the production rules of Listing 2 (**dot** denote the vector inner product).

Listing 3: Production rules for trial and test expressions

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

Bilinear forms.

Once test and trial expressions are available, bilinear terms can be obtained as contraction products of trial and test expressions or as sums thereof, as described in Listing 4.

Listing 4: Production rules for bilinear terms

```
BilinearTerm = TrialExpr * TestExpr          |
               "dot("TrialExpr, TestExpr")"  |
               CoefExpr * BilinearTerm       |
               BilinearTerm + BilinearTerm;
```

Bilinear forms finally result from the integration of bilinear terms on groups of mesh items (cf. Table 3.1). Production rules for bilinear forms are given in Listing 5.

Listing 5: Production rules for bilinear forms

```
IntegrateBilinearTerm = "integrate("MeshGroup, BilinearTerm")";
BilinearForm = IntegrateBilinearTerm         |
               IntegrateBilinearTerm + BilinearForm;
```

3.3.2 Language implentation with Boost Proto

We have based our implementation on the `boost::proto` library by Niebler [16], a powerful framework to build DSEL in C++. This library provides a collection of generic concepts and metafunctions that help to design a DSL, its grammar and tools to parse and evaluate expressions. It provides tools for constructing, type-checking, transforming and executing expression templates [3, 5, 17], more specifically, it

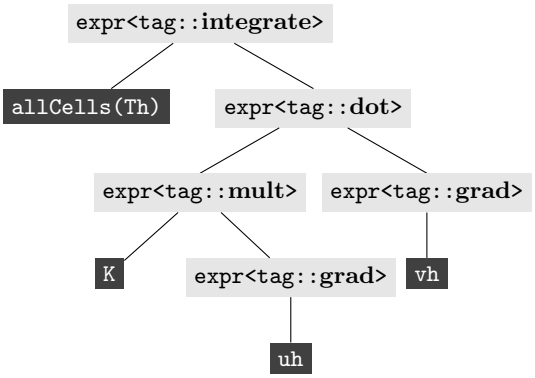


Figure 1: Expression tree for the bilinear form defined at line 7 of Listing 1. Expressions are in *light gray*, language terminals in *dark gray*

provides: (i) an expression tree data structure, (ii) a mechanism for giving expressions additional behaviors and members, (iii) operator overloads for building the tree from an expression, (iv) utilities for defining the grammar to which an expression must conform, (v) an extensible mechanism for immediately executing an expression template, (vi) an extensible set of tree transformations to apply to expression trees. This framework enable to design a DSEL in a declarative way with mechanisms based on concepts like: (i) *tag*, (ii) *meta-function*, (iii) *grammar*, (iv) *context* (v) and *transform*, structures, (see more details in the framework documentation [16]). In this section, we detail how we have translated our language formal definition §3.3.1 in proto objects that enable to define expressions, the language *Grammar*, *Context* and *Transform* structures to evaluate expressions and implement algorithms.

Language front ends.

The language front ends is defined by (i) the terminals, (ii) the keywords listed in 1, (iii) and the grammar based on the production rules of Listings 2, 3, 4, and 5.

Expressions are implemented with proto expression tree structures where each node is an object of type `proto::base_expr` identified by a *Tag* and where the leafs of the tree are occupied by *Terminals* (cf. Listing 2), meshes (cf. Listing 5), test and trial functions (cf. Listing 3).

The bilinear form a_h^{sushi} defined by (7) has the programming counterpart given in Listing 1 and the corresponding expression tree is detailed in Fig. 1.

Tag structures and meta functions.

The implementation of a proto expression tree is based on *Tag* structures and on associated meta-functions that enable to create nodes, implement *Grammar* or *Transform* structures.

The `boost::proto` framework already provides standard *Tags* for standard unary and binary C++ operator (cf tables 2) and meta-functions (like `proto::result_of::tag_of<.>`, `proto::result_of::child_c<.,.>` or `proto::result_of::value<.>`) to easiliy navigate in expression trees.

We have completed them with tags representing : (i) the different types of the DSEL terminals (the leafs of the tree) ; (ii) the DSEL keywords corresponding to the nodes of the tree.

Listing 6: Tags definition

```
namespace fvdsel {
  namespace tag {
    //! DSEL terminal tags
    struct basetype{} ;
    struct meshvartype{} ;
    struct testfunctiontype{} ;
    struct trialfunctiontype{} ;
    struct meshzonetype{} ;
    struct nulltype{} ;

    //! DSEL keyword tags
    struct dot{} ;
    struct grad{} ;
    struct jump{} ;
    struct avg{} ;
    struct integrate{} ;
    }
}
```

FVDSL domain definition.

We have defined the domain `FVDSLDomain` (Listing 7) where all expressions are encapsulated in a `FVDSLExpr` that comform to the grammar `FVDSLGrammar` detailled in §3.3.2. This mechanism enables then to the framework to overload most of C++ operators.

Listing 7: FVDSL expression domain definition

```
template<typename Expr> struct FVDSLExpr;

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

// Expressions in the pde 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)
};
```

DSEL keywords.

The DSEL keywords listed in table 1 are associated to specific tags. For each tag, we have implemented a free function that creates an associated tree node, a meta-function that generates the type of that node, and a grammar element that matches expressions and dispatches to the `proto::pass_through<>` transform, as *PrimitiveTransform* (cf [16], 3.3.3). For instance, Listing 8 illustrates the definition of the unary free function **grad(.)** creating nodes associated `fvdsel::tag::grad` and the definition of `fvdsel::gradop<ExprT>` the meta-function that matches grad expression or dispatches transforms. Listing 9 illustrates the definition of the binary free function **dot(.,.)** creating nodes associated to the tag `fvdsel::tag::dot` and the definition of `fvdsel::dotop<LExprT,RExprT>` the meta-function that matches inner product expression or dispatches transforms.

Listing 8: free function and meta-function associated to fvdsel::tag::grad

```
//! grad metafunction
template<typename A>
typename proto::result_of::make_expr< fvdsel::tag::grad,
                                      FVDSLDomain,
                                      A const &
                                    >::type
grad(A const &a)
{
    return proto::make_expr<fvdsel::tag::grad,
                            FVDSLDomain>(boost::ref(a));
}

//! 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;

  // member classes/structs/unions
  template<typename Expr,
           typename State,
           typename Data>
  struct impl
  : proto::pass_through<gradop>::template impl<Expr,
                                               State,
                                               Data>
  {};
};
```

Listing 9: Free function and meta-function associated to fvdsel::tag::dot

```
template<typename L,typename R>
typename
proto::result_of::make_expr<
                             fvdsel::tag::dot
                             , FVDSLDomain
                             , L const &
                             , R const &
                            >::type
dot(L const &l,R const& r)
{
  return proto::make_expr< fvdsel::tag::dot,
                           FVDSLDomain >(boost::ref(l),
                                         boost::ref(r));
}

template<typename LeftT,typename RightT>
struct dotop :  proto::transform< dotop<LeftT,RightT> >
{
  // types
  typedef proto::expr< fvdsel::tag::dot,
                       proto::list2< LeftT,
                                     RightT >
                     >                                type;
  typedef proto::basic_expr< fvdsel::tag::dot,
                             proto::list2<LeftT,
                                          RightT>
                           >                          proto_grammar;

  // member classes/structs/unions
  template<typename LExpr,
           typename RExpr,
           typename State,
           typename Data>
  struct impl
  : proto::pass_through<dotop>::template impl<LExpr,
                                              RExpr,
                                              State,
                                              Data>
  {};
} ;
```

Table 3 lists the main keywords with their associated tags, free functions and meta-functions.

Grammar definition.

The *Grammar* of our language is based on the production rules detailed in §3.3.1. Proto provides a set of tools that enable to implements each production rule in a user friendly declarative way. Terminal structures are detected with the meta-function defined in listing 10. Each production rule is implemented by a grammar structure composed with other grammar structures, proto pre-defined transforms (cf table 2) or some of our specific transforms (cf table 3).

Listing 10: terminal meta-function

```
template<typename T> struct is_base_type ;
template<typename T> struct is_mesh_var ;
template<typename T> struct is_mesh_group ;
template<typename T> struct is_function ;
template<typename T> struct is_test_function ;
template<typename T> struct is_trial_function ;

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>
            >
{};
```

In listing 11 we can compare the implementation of the DSEL grammar with the `BaseTypeGrammar`, `MeshVarTypeGrammar`, `TestFunctionTerminal`, `TrialFunctionTerminal`, `CoefExprGrammar` and `BilinearGrammar` structures to the EBNF definition of the production rules 2, 3, 4, and 5 specifying bilinear expressions.

Listing 11: Bilinear expression grammar

```
namespace fvdsel {

  struct BaseTypeGrammar
  : proto::terminal< proto::convertible_to<Real> >
  {} ;

  struct MeshVarTypeGrammar
  : proto::and_< proto::terminal<proto::_>,
                 proto::if_< fvdsel::is_mesh_var<proto::_value>() > >
  {} ;

  struct TestFunctionTerminal
  : proto::and_< FunctionTerminal,
                 proto::if_< fvdsel::is_test_function<proto::_value>() > >
  {} ;

  struct TrialFunctionTerminal
  : proto::and_< FunctionTerminal,
                 proto::if_< fvdsel::is_trial_function<proto::_value>() > >
  {} ;

  struct CoefExprGrammar ;

  struct CoefExprGrammar
  : proto::or_<
                BaseTypeGrammar,
                MeshVarTypeGrammar,
                proto::plus<CoefExprGrammar,
                            CoefExprGrammar>,
                proto::multiplies<CoefExprGrammar,
                                  CoefExprGrammar>,
                proto::divides<CoefExprGrammar,
                               CoefExprGrammar>
              >
  {} ;

  struct TrialExprGrammar
    : proto::or_< TrialFunctionTerminal,
                  proto::multiplies<CoefExprGrammar,
                                    TrialExprGrammar>,
                  fvdsel::jumpop<TrialExprGrammar>,
                  fvdsel::avgop<TrialExprGrammar>,
                  fvdsel::gradop<TrialExprGrammar>,
                  fvdsel::traceop<TrialExprGrammar>
                >

   {} ;

   struct TestExprGrammar
      : proto::or_< TestFunctionTerminal,
                    proto::multiplies<CoefExprGrammar,
                                      TestExprGrammar>,
                    fvdsel::jumpop<TestExprGrammar>,
                    fvdsel::avgop<TestExprGrammar>,
                    fvdsel::gradop<TestExprGrammar>,
                    fvdsel::traceop<TestExprGrammar>
                  >
   {} ;

   struct BilinearGrammar ;

   struct PlusBilinear
      : proto::plus< BilinearGrammar, BilinearGrammar >
    {};

    struct MinusBilinear
      : proto::minus< BilinearGrammar, BilinearGrammar >
    {};

   struct MultBilinear
     : proto::multiplies< CoefExprGrammar, BilinearGrammar >
   {};

  struct BilinearGrammar
  : proto::or_<
                proto::multiplies<TrialExprGrammar,
```

```
                                    TestExprGrammar>,
                fvdsel::dotop<TrialExprGrammar,
                              TestExprGrammar>,
                PlusBilinear,
                MinusBilinear,
                MultBilinear
              >
  {} ;
}
```

3.3.3 Evaluation contexts and transforms

Language back ends : expression evaluation, algorithm implementation.

The DSEL back ends are composed of algebraic structures (matrices, vectors, linear combinations) used in different kind of algorithms based mesh entities iterations, matrices, vectors evaluation or assembly operations. The implementation of theses algorithms will be based on the evaluation and manipulation of `FVDSLDomain` expressions. Such evaluations are based on two kind of Proto concepts : *Context* and *Transform* structures.

- A *Context* is like a function object that is passed along with an expression to the `proto::eval()` function. It associates behaviors with node types. `proto::eval()` walks the expression and invokes your context at each node.
- A *Transform* is a way to associate behaviors, not with node types in an expression, but with rules in a Proto grammar. In this way, they are like semantic actions in other compiler-construction toolkits.

Algorithms are implemented as specific expression tree evaluation, as a sequence of piece of algorithms associated to the behaviour of *Evaluation context* on each node or on *Transforms* that match production rules. Theses pieces of algorithm are written respectively in the `operator()()` of the structure `Context::eval` for *Context* objects, in the `operator()()` of *callable transforms* objects for . *Transforms.*

For instance, in the expression defined in listing 1, `allCells(Th)`, `K`, `u`, `v` are terminals of the language. **integrate**, **dot** and **grad** are specific keywords of the language associated to the tags `fvdsel::tag::integrate`, `fvdsel::tag::dot` and `fvdsel::tag::grad`. The binary operator `*` is associated to the tag `proto::tag::mult`

At evaluation, the expression is analyzed as follows :

1. The root node of the tree, associated to the tag `tag::integrate` is composed of an `MeshGroup` expression (`allCells(Th)`) and the `BilinearTerm` expression (`dot(K*grad(u),grad(v))`);
2. The integration algorithm consists in iterating on the elements of the `allCells(Th)` collection and in evaluating the bilinear expression on each cell. This bilinear expression is composed of: (i) a `TrialExpr` expression : `K*grad(u)`; (ii) a `TestExpr` expression : `grad(v)` (iii) a binary operator associated to the tag : `tag::dot` The evaluation of the `TrialExpr` expression and of the `TestExpr` expression on a cell return two linear combination objects which, associated to the binary operator tag lead to a bilinear contribution which is a local matrix contributing to the global linear system of the linear context with a factor equal to the measure of the cell.

To implement the integration algorithm associated to linear variational formulation, we have used both *Context* and *Transform* structures. A `BilinearContext` object, referencing a linear system back end object used to build the global linear system with different linear algebra packages has been developped to evaluate the global expression. On an Integrate node, this object call a `IntegratorOp` transform on the expression tree. In listing 12, we detail the implementation of this transform that matches in our example the expression with the tag `fvdsel::tag::integrate`, the `MeshGroup` expression `allCells(Th)` and the term `dot(K*grad(u),grad(v))`.

Listing 12: Integrator transform

```
struct Integrator : proto::callable
{
  //... callable object that will use a BilinearIntegrator transform on
  // a bilinear expression
  typedef int result_type;

  template<typename ZoneT,
           typename ExprT,
           typename StateT,
           typename DataT>
  int
  operator()(ExprT const& expr,
             ZoneT const& zone,
             StateT& state,
             DataT const& data) const
  {
     //call a transform that analyze ExprT
     //and dispatch to the appropriate transform
     return 0 ;
  }
} ;

struct IntegratorOp
: proto::or_<
       proto::when<
                    fvdsel::IntegratorGrammar,
                    fvdsel::Integrator(proto::_child_c<2>,
                                       proto::_child_c<1>,
                                       proto::_state,
                                       proto::_data
                                       )
                  >
       proto::when<
                   proto::plus<IntegratorOp,IntegratorOp>,
                   IntegratorOp(proto::_left,
                                IntegratorOp(proto::_right,
                                             proto::_state,
                                             proto::_data
                                            ),
                                proto::_data)
                 >
             >
  {};
```

In the *callable transform* `Integrator`, analyzing the integrate expression term, when a bilinear expression is matched, another transform `BilinearIntegrator` (listing 13) matching a `DotExpr` associated to `fvdsel::tag::dot` and the production rules matching the test and trial part of the bilinear expressions. The algorithm (listing 14) is called by the *callable transform* `DotIntegrator`. Note that the `BilinearContext` is passed along the expression tree with the `proto::_data` structure.

Listing 13: BilinearIntegrator transform

```
struct MultIntegrator : proto::callable
{
  typedef int result_type;
  template<typename TrialExprT,
           typename TestExprT,
           typename StateT,
           typename DataT>
  int
  operator()(TrialExprT const& lexpr,
             TestExprT const& rexpr,
             StateT& state,
             DataT const& data) const
  {
```

```
    // call integrate algorithm
    // with tag proto::tag::mult

    return integrate<proto::tag::mult>(getMesh(data),
                                       getGroup(data),
                                       lexpr,
                                       rexpr,
                                       GetContext(data) ) ;
  }
} ;

struct DotIntegrator : proto::callable
{
  typedef int result_type;
  template<typename TrialExprT,
           typename TestExprT,
           typename StateT,
           typename DataT>
  int
  operator()(TrialExprT const& lexpr,
             TestExprT const& rexpr,
             StateT& state,
             DataT const& data) const
  {
    // call integrate algorithm
    // with tag proto::tag::dot
    return integrate<proto::tag::dot>(getMesh(data),
                                      getGroup(data),
                                      lexpr,
                                      rexpr,
                                      GetContext(data) ) ;
    }
} ;

struct BilinearIntegrator
: proto::or_<
      proto::when< proto::multiplies<TrialExprGrammar,
                                     TestExprGrammar>,
                   MultIntegrator(proto::_left,          //! lexpr
                                  proto::_right,         //! rexpr
                                  proto::_state,         //! state
                                  proto::_data           //! con-
text
                                  )>,
      proto::when< fvdsel::dotop<TrialExprGrammar,
                                 TestExprGrammar>,
                   DotIntegrator(proto::_child_c<0>, //! left
                                 proto::_child_c<1>, //! trial
                                 proto::_state,      //! state
                                 proto::_data        //! con-
text
                                )
                  >,
      proto::when< proto::plus<BilinearGrammar,
                               BilinearGrammar>,
                   BilinearIntegrator(proto::_right,
                           BilinearIntegrator(proto::_left,
                                              proto::_state,
                                              proto::_data),
                                      proto::_data
                                     )
                  >
          >
{} ;
```

Listing 14 is a simple assembly algorithm. We iterate on each entity of the mesh group and evaluate the test and trial expression on each entity. For such evaluation, we have defined different kind of context objects. The structure `EvalContext<ItemT>` enables to compute the linear combination objects that return the evaluation of test or trial expression, which associated to the binary operator tag lead to a bilinear contribution, a local matrix contributing to the global linear system of the linear context with a factor equal to the measure of the cell. Note that the `BilinearContextT` is parametrized by a `phase_type` parameter that enables to optimize and factorize global linear system construction : intermediate computations can be stored in a cache system and be reused. For instance when a global linear system is built, the global system dimensions setting phase, the sparse structure matrix definition and the matrix filling phase can be separated. The first two phases can be easily factorized for several filling phases in iterative algorithms.

Listing 14: Integration assembly algorithm

```
template<typename ItemT,
         typename TestExprT,
         typename TrialExprT,
         typename tag_op,
         typename BilinearContextT>
void integrate(Mesh const& mesh,
               GroupT<ItemT> const& group,
               TrialExprT const& trial,
               TestExprT const& test,
               BilinearContextT& ctx)
{
  static const Context::ePhaseType phase =
  BilinearContextT::phase_type;
  auto matrix = ctx.getMatrix();
  for( auto cell : group )
  {
    EvalContext<Item> ctx(cell) ;       //! eval context on mesh item
    auto lu = proto::eval(trial,ctx) ;//! trial linear combination
    auto lv = proto::eval(test,ctx)); //! test linear combination
    BilinearContribution<tag_op> uv(lu,lv) ;
    assemble<phase>(matrix,                //! matrix
                    measure(mesh,cell),//! cell measure
                    uv ) ;                 //! bilinear contribution


  }
}
```

In the same way the evaluation of a linear form expression with a linear context leads to the construction of the righ hand side of a global linear system.

Once the global linear system built, it can be solved with a linear system solver provided by the linear algebra layer.

4. APPLICATIONS

Our benchmark is based on the following exact solution for the diffusion problem (2):

$$u(\mathbf{x}) = \sin(\pi x)\sin(\pi y)\sin(\pi z), \qquad \kappa = \begin{bmatrix} 1 & 0 & 0 \\ 0 & 1 & 0 \\ 0 & 0 & 1 \end{bmatrix},$$

on the square domain $\Omega = [0,1]^3$ with $f(x,y,z) = 3\pi sin(\pi x)sin(\pi y)sin(\pi z)$.

We compare the following methods: (i) the DSEL implementations of the ccG method (5) provided in Listing 15; (ii) the DSEL implementation of the SUSHI method with face unknowns (6) provided in Listing 16; (iii) the DSEL implementation of the G method (17) provided in Listing 17

Listing 15: ccG method implementation

```
MeshType Th;
Real K;
auto Vh = newCCGSpace(Th);
auto u = Vh->trial("U");
auto v = Vh->test("V");
auto lambda = eta*val(gamma)/val(H(Th));
BilinearForm a =
  integrate(allCells(Th),dot(K*grad(u),grad(v))) +
  integrate(allFaces(Th),jump(u)*dot(N(Th),avg(grad(v))) -
                                     dot(N(Th),avg(K*grad(u)))*jump(v) +
                                     lambda*jump(u)*jump(v);
LinearForm b =
  integrate(allCells(Th),f*v);
```

Listing 16: SUSHI method implementation

```
MeshType Th;
Real K;
auto Vh = newSUSHISpace(Th);
auto u = Vh->trial("U");
auto v = Vh->test("V");
BilinearForm a =
  integrate(allCells(Th),dot(K*grad(u),grad(v))) ;
LinearForm b =
  integrate(allCells(Th),f*v);
```

Listing 17: G method implementation

```
MeshType Th;
Real K;
auto Uh = newGSpace(Th);
auto Vh = newP0Space(Th);
auto u = Vh->trial("U");
auto v = Vh->test("V");
BilinearForm a =
  integrate(allFaces(Th),dot(N(Th),avg(K*grad(u)))*jump(v)) ;
LinearForm b =
  integrate(allCells(Th),f*v);
```

The codes is compiled with the `gcc` 4.5 compiler with the following compile options:

```
-03 -fno-builtin
-mfpmath=sse -msse -msse2 -msse3
-mssse3 -msse4.1 -msse4.2
-fno-check-new -g -Wall -std=c++0x
--param -max-inline-recursive-depth=32
--param max-inline-insns-single=2000
```

The benchmark test cases are run on a work station with a quad-core Intel Xeon processor GenuineIntel W3530, 2.80GHz, 8MB for cach size.

In our numerical tests we consider a families of h-refined meshes with h decreasing from 0.1 to 0.0125.

The linear systems are solved using the PETSc library[6] with the BICGStab solver preconditioned by the euclid ILU(0) preconditioner, with relative tolerance set to 10^{-6}.

The benchmarks monitor various metrics:

(i) *Accuracy.* The accuracy of the methods is evaluated in terms of the L^2 norm of the error. For the methods of §2, the L^2-norm of the error is evaluated using the cell center as a quarature node, i.e.,

$$\|u - u_h\|_{L^2(\Omega)} \approx \left(\sum_{T \in \mathcal{T}_h} |T| (u(\mathbf{x}_T) - u_T)^2 \right)^{\frac{1}{2}}.$$

The convergence order of a method is classically expressed relating the error to the mesh size h.

(ii) *Memory consumption.* When comparing methods featuring different number of unknowns and stencils, a more fair comparison in terms of system size and memory consumption is obtained relating the error to the number of DOFs (N_{DOF}) and to the number of nonzero entries of the corresponding linear system (N_{nz}).

(iii) *Performance.* The last set of parameters is meant to evaluate the CPU cost for each method and implementation. To provide a detailed picture of the different stages and estimate the overhead associated to the DSEL, we separately evaluate

- t_{init}, the time to build the discrete space;
- t_{ass}, the time to fill the linear systems (local/-global assembly). When DSEL-based implementations are considered, this stage carries the additional cost of evaluating the expression tree for bilinear and linear forms;
- t_{solve}, the time to solve the linear system.

The accuracy and memory consumption analysis is provided in Figure 2. We can check the expected linear convergence behaviour of the methods. A super linear convergence effect for the G method can be observed due to the regularity of our meshes.

The CPU cost analysis is provided in Figure 4. The cost of each stage of the computation is related to the number of DOFs in Figure 3 to check that the expected complexity is achieved. This is the case for all the methods considered.

A comparison in terms of absolute computation time is provided in Figure 4 on the 2D version of the test case with $h = 0.006125$. The overhead of the DSEL is estimated by comparing the times results of the ccg methods of the DSEL version the the `fvC++` implementation, a hand written `stl`-based implementation of the back-end discussed in §3.1. The difference in t_{ass} is due to the fact that in the hand written implementation computation stages used several times in the assembly phase are naturally pre-computed and stored while in our primary DSEL implemntation, cach mechanisms for such computations are not already available.

5. CONCLUSION AND PERSPECTIVES

Our DSEL for lowest-order methods enables to describe and solve various non trivial academic problems. Different numerical methods were implemented with a high level language close to one used in the unified mathematical framework. The analysis of the performance results of our study cases shows that the overhead of the language is not important regarding standard hand written codes.

In some future work, we plan to extend our DSEL to take into account: (i) various types of boundary conditions, (ii) the non linear formulation hiding the complexities of derivaties computation.

Within the HAMM[2] project (Hybrid architecture and multi-level model), we plan to handle multi-level methods and illustrate the interest of our approach to take advantage seamless of the performance of new hybrid hardware architecture with GP-GPU.

6. REFERENCES

[1] ISO/IEC 14977, 1996(E).

[2] HAMM Web page, 2010. http://www.agence-nationale-recherche.fr/en/anr-funded-project/?tx_lwmsuivibilan_pi2[CODE]=ANR-10-COSINUS-009.

[3] D. Abrahams and L. Gurtovoy. C++ template metaprogramming : Concepts, tools, and thechniques from boost and beyond. C++ in Depth Series. Addison-Wesley Professional, 2004.

[4] L. Agélas, D. A. Di Pietro, and J. Droniou. The G method for heterogeneous anisotropic diffusion on general meshes. *M2AN Math. Model. Numer. Anal.*, 44:597–625, 2010.

[5] P. Aubert and N. Di Césaré. Expression templates and forward mode automatic differentiation. Computer and information Science, chapter 37, pages 311-315. Springer, New York, NY, 2001.

[6] S. Balay, J. Brown, K. Buschelman, W. D. Gropp, D. Kaushik, M. G. Knepley, L. Curfman McInnes, B. F. Smith, and H. Zhang. PETSc Web page, 2011. http://www.mcs.anl.gov/petsc.

[7] F. Brezzi, K. Lipnikov, and M. Shashkov. Convergence of mimetic finite difference methods for diffusion problems on polyhedral meshes. *SIAM J. Numer. Anal.*, 45:1872–1896, 2005.

[8] F. Brezzi, K. Lipnikov, and V. Simoncini. A family of mimetic finite difference methods on polygonal and polyhedral meshes. *M3AS*, 15:1533–1553, 2005.

[9] D. A. Di Pietro. Cell-centered Galerkin methods. *C. R. Math. Acad. Sci. Paris*, 348:31–34, 2010.

Table 1: DSEL keywords

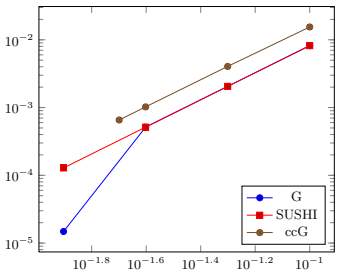
keyword	meaning
integrate(.,.)	$\int(.)$ integration of expression
dot(.,.)	$(.\cdot.)$ vector inner product
jump(.)	$[\![.]\!]$ jump accross a face
avg(.)	$\{.\}$ average accross a face

Table 2: Proto standard tags and meta-functions

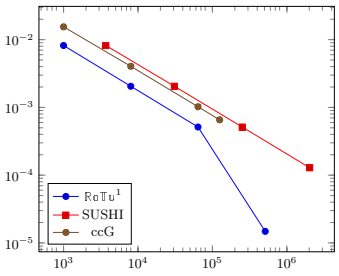
operator	narity	tag	meta-function
+	2	proto::tag::**plus**	proto::**plus**<.,.>
-	2	proto::tag::**minus**	proto::**minus**<.,.>
*	2	proto::tag::**mult**	proto::**mult**<.,.>
/	2	proto::tag::**div**	proto::**div**<.,.>

Table 3: DSEL keywords

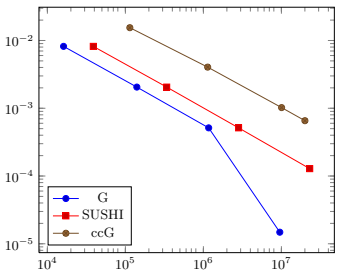
operator	narity	tag	meta-function
integrate(.,.)	2	fvdsel::tag::**integrate**	**integrateop**<.,.>
grad(.)	1	fvdsel::tag::**grad**	**gradop**<.>
jump(.)	1	fvdsel::tag::**jump**	**jumpop**<.>
avg(.)	1	fvdsel::tag::**avg**	**avgop**<.>
dot(.,.)	2	fvdsel::tag::**dot**	**dotop**<.,.>



(a) L^2-error vs. h

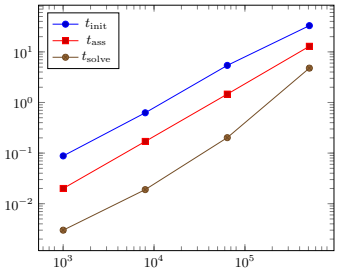


(b) L^2-error vs. N_{DOF}

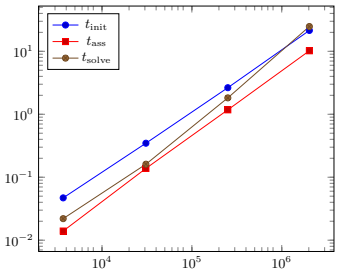


(c) L^2-error vs. N_{nz}

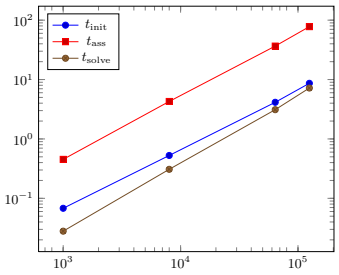
Figure 2: Accuracy and memory consumption analysis for the example of Sect. 4



(a) G : CPU cost vs. h



(b) SUSHI : CPU cost vs. h



(c) ccG : CPU cost vs. h

Figure 3: Performance analysis for the example of Sect. 4

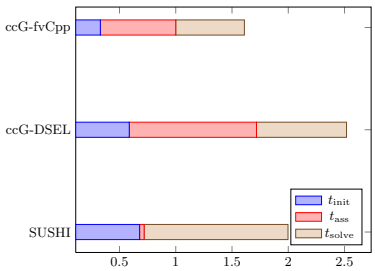


Figure 4: Comparison of different methods and implementation for the 2D test case of §4 ($h = 0.00625$)

[10] D. A. Di Pietro. A compact cell-centered Galerkin method with subgrid stabilization. *C. R. Acad. Sci. Paris, Ser. I.*, 348(1–2):93–98, 2011.

[11] D. A. Di Pietro. Cell centered Galerkin methods for diffusive problems. *M2AN Math. Model. Numer. Anal.*, 46(1):111–144, 2012.

[12] D. A. Di Pietro and J.-M. Gratien. Lowest order methods for diffusive problems on general meshes: A unified approach to definition and implementation. In J. Fovrt, J. Furst, J. Halama, R. Herbin, and F. Hubert, editors, *Finite Volumes for Complex Applications VI*, pages 3–19. Springer–Verlag, 2011.

[13] J. Droniou, R. Eymard, T. Gallouët, and R. Herbin. A unified approach to mimetic finite difference, hybrid finite volume and mixed finite volume methods. *M3AS*, 20(2):265–295, 2010.

[14] R. Eymard, T. Gallouët, and R. Herbin. Discretization of heterogeneous and anisotropic diffusion problems on general nonconforming meshes SUSHI: a scheme using stabilization and hybrid interfaces. *IMA J. Numer. Anal.*, 30:1009–1043, 2010.

[15] G. Grospellier and B. Lelandais. The Arcane development framework. In *Proceedings of the 8th workshop on Parallel/High-Performance Object-Oriented Scientific Computing*, POOSC '09, pages 4:1–4:11, New York, NY, USA, 2009. ACM.

[16] E. Niebler. *`boost::proto` documentation*, 2011. http://www.boost.org/doc/libs/1_47_0/doc/html/proto.html.

[17] T. Veldhuizen. Using C++ template metaprograms. C++ report, 7(4):36-43, May 1995. reprinted in C++ Gems, ed. Stanley Lippman, 1995.