HAL
archives-ouvertes.fr

Implementing Lowest-Order Methods for Diffusive Problems with a DSEL

Jean-Marc Gratien

▶ To cite this version:

Jean-Marc Gratien. Implementing Lowest-Order Methods for Diffusive Problems with a DSEL. Modelling and Simulation in Fluid Dynamics in Porous Media, Springer, pp.155–175, 2012, 10.1007/978-1-4614-5054-2_10 . hal-00788327

HAL Id: hal-00788327
https://hal-ifp.archives-ouvertes.fr/hal-00788327

Submitted on 14 Feb 2013

HAL is a multi-disciplinary open access archive for the deposit and dissemination of scientific research documents, whether they are published or not. The documents may come from teaching and research institutions in France or abroad, or from public or private research centers.

L'archive ouverte pluridisciplinaire **HAL**, est destinée au dépôt et à la diffusion de documents scientifiques de niveau recherche, publiés ou non, émanant des établissements d'enseignement et de recherche français ou étrangers, des laboratoires publics ou privés.

Implementing Lowest-Order Methods for Diffusive Problems with a DSEL

J.-M. Gratien

Abstract Industrial simulation software have to manage: *(i)* the complexity of the underlying physical models, *(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. Nowadays, some frameworks offer a number of advanced tools to deal with the complexity related to parallelism in a transparent way. However, high level complexity related to discretization methods and physical models lack of tools to help physicists to develop complex applications. Generative programming and domain-specific languages (DSL) are key technologies allowing to write code with a high level expressive language and take advantage of the efficiency of generated code for low level services. Their application to Scientific Computing has been up to now limited to Finite Element (FE) methods and Galerkin methods, for which a unified mathematical framework has been existing for a long time (see projects like `Freefem++`, `Getdp`, `Getfem++`, `Sundance`, `Feel++` [3], `Fenics` project). In reservoir and basin modeling, lowest order methods are promising methods allowing to handle general meshes. Extending Finite Volume (FV) methods, Aavatsmark, Barkve, Bøe and Mannseth propose consistent schemes for non orthogonal meshes while stability problems are solved with the Mimetic Finite Difference method (MFD) and the Mixte/Hybrid Finite Volume methods (MHFV) [1]. However the lack of a unified mathematical frame was a serious limit to the extension all of these methods to a large variety of problems. In [2], the authors propose a unified way to express FV multi-points scheme and DFM/VFMH methods. This mathematical frame allows us to extend the DSL used for FE and Galerkin methods to lowest order methods. We focus then on the capability of such language to allow the description and the resolution of various and complex problems with different lowest-order methods. We validate the design of the DSL that we have embedded in C++, on the implementation of several academic problems. We present some convergence results and

J.-M. Gratien
IFP Energies nouvelles, 1 et 4 av Bois Préau 92500 Rueil-Malmaison, France
e-mail: jean-marc.gratien@ifpen.fr

compare the performance of their implementation with the DSEL to their hand written counterpart.

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). 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 (figure 1).

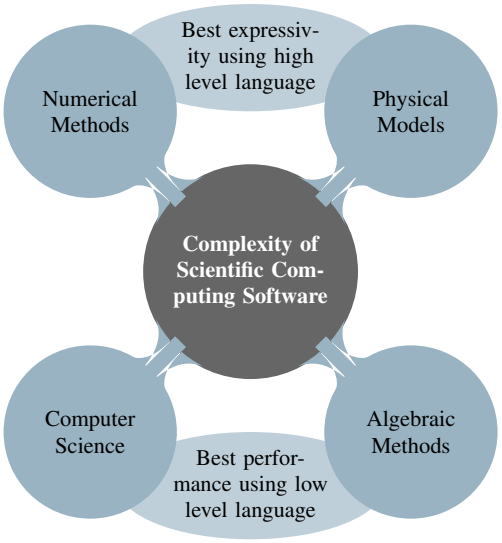


Fig. 1 Complexity management

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. Among such kind of framework, the Arcane Platform is a parallel

C++ framework, co-developped by CEA (Commissariat l'Energie Atomique) and IFP Energies Nouvelles, designed to develop applications based on 1D, 2D, 3D unstructured grids. It provides services to manage meshes, groups of mesh elements, discrete variables representing discrete fields on mesh elements, parallelism, network communication between processors and IO services. A linear algebra layer developped above this plateform, provides also a unified way to handle standard parallel linear solver packages such as Petsc, Hypre, MTL4, UBlas and IFPSolver an in house linear solver package.

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 (figure 1). 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. They are used for various reasons, teaching purposes, designing complex problems or rapid prototyping of new methods, schemes or algorithms, the main goal being always to hide technical details behind software layers and providing only the relevant components required by the user or programme [23, 24].

We try to extend this kind of approach to lowest order methods to solve the PDE systems of geo modeling applications. These kind of methods seem to be very promising for geoscience application as they allow to handle general meshes which is an important issue for reservoir and basin modeling. The first extension of FV methods is due to Aavatsmark, Barkve, Bøe and Mannseth [1, 2, 3, 4] and to Edwards and Rogers [17, 18]) in reservoir simulation. The main idea is to transform the classical two-points flux approximation into a multi-points flux approximation. This idea solves the consistency problem for non orthogonal meshes but does not guaranty the stability of the resulting method. This can be solved with Mimetic Finite difference method (MFD) [9, 8] and Mixed/hybrid Finite Volume methods (MHFV) [15, 19, 16]. These methods are elaborated, adding face unknowns and using a variational formulation approach instead of the classical conservative balance approach on each cell. However the lack of a unified mathematical frame was a serious limit to the extension all of these methods to a large variety of problems. A partial answer was recently proposed by Di Pietro in [21, 14, 12], by introducing a new class of methods inspired from non conforming finite element, see also Di

Pietro and Gratien [22]. These formulations enable to express in a unified way VF multi-points scheme and DFM/VFMH methods and allow to extend them to various problems in fluid and solid mechanics. This consistent unified mathematical frame allows a unified description of a large family of lowest-order methods. It is possible then, as for FE methods, to design of a high level language inspired from the mathematical notation, that could help physicist to implement their application. We have developped a language based on that frame, that we have embedded in the C++ language. This approach, used in projects like `Feel++`, `Fenics` or `Sundance` has several advantages over generating a specific language. Embedded in the C++ language, (i) it avoids the compiler construction complexities, taking advantage of the generative paradigm of the C++ language and allowing grammar checking at compile time; (ii) it allows to use other libraries concurrently which is often not the case for specific languages, our implementation heavily relies, in particular, on the tools provided by the `boost` library; (iii) it exploits the optimization capabilities of th C++ compiler, thereby allowing to tackle large study cases which is not possible with interpreted language; (iv) it allows to mix the object oriented programming and the functional programming paradigm. New concepts provided by the standard C++0x (the keyword `auto`, lambda functions, ...), make C++ very competitive as its syntax becomes comparable to that of interpreted languages like `Python` or `Ruby` used in projects like `FreeFem++` or `Fenics`, while performance issues remain preserved thanks to compiler optimisations.

The proposed DSEL has been developped on top of Arcane plateform [20]. It is based on useful concepts inspired from the unified mathematical frame. We focus on their capability 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 different methods on two diffusion problems. We present some convergence results and analyze some performance criteria.

2 Mathematical setting

Let $\Omega \subset \mathbb{R}^d$, $d \geq 2$, and $\mathscr{T}_h = \{T\}$ a given mesh partitioning Ω. Mesh faces with a $(d-1)$-dimensional measure, defined by $T_1, T_2 \in \mathscr{T}_h$ such that $F \subset \partial T_1 \cap \partial T_2$ (interface) or $T \in \mathscr{T}_h$ such that $F \subset \partial T \cap \partial\Omega$ (boundary), are respectively collected in the set $\mathscr{F}_h^i$ and $\mathscr{F}_h^b$. Let $\mathscr{F}_h \stackrel{\text{def}}{=} \mathscr{F}_h^i \cup \mathscr{F}_h^b$.

For all $k \geq 0$, we define the broken polynomial spaces of total degree $\leq k$ on $\mathscr{S}_h$,

$$\mathbb{P}_d^k(\mathscr{S}_h) \stackrel{\text{def}}{=} \{v \in L^2(\Omega) \mid \forall S \in \mathscr{S}_h,\, v|_S \in \mathbb{P}_d^k(S)\},$$

with $\mathbb{P}_d^k(S)$ given by the restriction to $S \in \mathscr{S}_h$ of the functions in $\mathbb{P}_d^k$.

We introduce trace operators which are of common use in the context of nonconforming FE methods. Let v be a scalar-valued function defined on Ω smooth enough to admit on all $F \in \mathscr{F}_h$ a possibly two-valued trace. To any interface $F \subset \partial T_1 \cap \partial T_2$

we assign two non-negative real numbers $\omega_{T_1,F}$ and $\omega_{T_2,F}$ such that

$$\omega_{T_1,F} + \omega_{T_2,F} = 1,$$

and define the jump and weighted average of v at F for a.e. $\mathbf{x} \in F$ as

$$[\![v]\!]_F(\mathbf{x}) \stackrel{\text{def}}{=} v|_{T_1} - v|_{T_2}, \qquad \{v\}_{\omega,F}(\mathbf{x}) \stackrel{\text{def}}{=} \omega_{T_1,F} v|_{T_1}(\mathbf{x}) + \omega_{T_2,F} v|_{T_2}(\mathbf{x}). \tag{1}$$

If $F \in \mathscr{F}_h^b$ with $F = \partial T \cap \partial\Omega$, we conventionally set $\{v\}_{\omega,F}(\mathbf{x}) = [\![v]\!]_F(\mathbf{x}) = v|_T(\mathbf{x})$. The index ω is omitted from the average operator when $\omega_{T_1,F} = \omega_{T_2,F} = \frac{1}{2}$, and we simply write $\{v\}_F(\mathbf{x})$. The dependence on both the point $\mathbf{x}$ and the face F is also omitted from both the jump and average trace operators if no ambiguity arises.

The unified mathematical frame presented in [13, 22] 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)-Galerkin 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, we consider the following heterogeneous diffusion model problem:

$$\begin{aligned} -\nabla\cdot(\kappa\nabla u) &= f \quad \text{in } \Omega, \\ u &= 0 \quad \text{on } \partial\Omega, \end{aligned} \tag{2}$$

with source term $f \in L^2(\Omega)$. Here, κ denotes a uniformly elliptic tensor field piecewise constant on the mesh $\mathscr{T}_h$.

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

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

with

$$a(u,v) \stackrel{\text{def}}{=} \int_\Omega \kappa\nabla u\cdot\nabla v,$$
$$b(v) \stackrel{\text{def}}{=} \int_\Omega f * v$$

In this framework, a specific lowest-order is defined by (i) setting $U_h(\mathscr{T}_h)$ and $V_h(\mathscr{T}_h)$ a trial and a test function space, (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 setting of a discrete function space U_h is based on three main ingredients:

- $\mathscr{T}_h$ the mesh partitioning Ω, $\mathscr{S}_h$ a submesh of $\mathscr{T}_h$ where $\forall S \in \mathscr{S}_h, \exists T_S \in \mathscr{T}_h, S \subset T_S$ (we will considere two choices: the identity $\mathscr{S}_h = \mathscr{T}_h$, and the pyramidal $\mathscr{S}_h = \mathscr{P}_h$ where cells $S \subset T_S$ are built with the center of T and a face $F \subset \partial T_S$);

- $\mathbb{V}_h$ the space of vector of degree of freedoms where the components of the vectors can be indexed by the mesh entities (cells, faces or nodes);
- $\mathfrak{G}_h$ a linear gradient operator that defines for each vector $v \in V_h$ a constant gradient on each element of $\mathscr{S}_h$ a submesh of $\mathscr{T}_h$.

The key idea to get a unifying perspective is to consider lowest-order methods as nonconforming methods based on incomplete broken affine spaces that are defined starting from the space of degrees of freedom (DOFs) $\mathbb{V}_h$. More precisely, we let

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

and consider the following choices:

$$\mathbb{V}_h = \mathbb{T}_h \text{ or } \mathbb{V}_h = \mathbb{T}_h \times \mathbb{F}_h. \tag{3}$$

The choice $\mathbb{V}_h = \mathbb{T}_h$ corresponds to cell centered finite volume (CCFV) and cell centered Galerkin (ccG) methods, while the choice $\mathbb{V}_h = \mathbb{T}_h \times \mathbb{F}_h$ leads to mimetic finite difference (MFD) and mixed/hybrid finite volume (MHFV) methods.

The key ingredient in the definition of a broken affine space is a piecewise constant linear gradient reconstruction $\mathfrak{G}_h : \mathbb{V}_h \to [\mathbb{P}^0_d(\mathscr{S}_h)]^d$ with suitable properties. We emphasize that the linearity of $\mathfrak{G}_h$ is a fundamental assumption for the implementation discussed in section 3.

Using the above ingredients, we can define the linear operator $\mathfrak{R}_h : \mathbb{V}_h \to \mathbb{P}^1_d(\mathscr{S}_h)$ such that, for all $\mathbf{v}_h \in \mathbb{V}_h$,

$$\forall S \in \mathscr{S}_h, S \subset T_S \in \mathscr{T}_h, \forall \mathbf{x} \in S, \qquad \mathfrak{R}_h(\mathbf{v}_h)|_S = v_{T_S} + \mathfrak{G}_h(\mathbf{v}_h)|_S \cdot (\mathbf{x} - \mathbf{x}_{T_S}). \tag{4}$$

The operator $\mathfrak{R}_h$ maps every vector of DOFs $\mathbf{v}_h \in \mathbb{V}_h$ onto a piecewise affine function $\mathfrak{G}_h(\mathbf{v}_h)$ belonging to $\mathbb{P}^1_d(\mathscr{S}_h)$. Hence, we can define a broken affine space as follows:

$$V_h = \mathfrak{R}_h(\mathbb{V}_h) \subset \mathbb{P}^1_d(\mathscr{S}_h). \tag{5}$$

With this framework, the model problem (2) can be solved with various methods:

- The G-method, see [6] is based on a space V_h^{g} defined setting $\mathbb{V}_h = \mathbb{T}_h$, $\mathscr{S}_h = \mathscr{P}_h$ and giving an operator $\mathfrak{G}_h^{\mathrm{g}}$ based on a L-construction .
  This method leads is to the following Petrov-Galerkin method:

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

  where $a_h^{\mathrm{g}}(u_h, v_h) \stackrel{\text{def}}{=} \sum_{F \in \mathscr{F}_h} \int_F \{\kappa \nabla_h u_h\} \cdot \mathrm{n}_F [\![v_h]\!]$ with ∇_h broken gradient on $\mathscr{P}_h$.
- The cell centered Galerkin method, see [10, 11, 14] is based on a space V_h^{ccg} defined setting $\mathbb{V}_h = \mathbb{T}_h$, $\mathscr{S}_h = \mathscr{T}_h$ and giving an operator $\mathfrak{G}_h^{\mathrm{ccg}}$ obtained with the green formula and a trace operator.
  The method reads

$$\text{Find } u_h \in V_h^{\text{ccg}} \text{ s.t. } a_h^{\text{ccg}}(u_h, v_h) = \int_\Omega f v_h \text{ for all } v_h \in V_h^{\text{ccg}}. \tag{6}$$

where,

$$\begin{aligned} a_h^{\text{ccg}}(u_h, v_h) \stackrel{\text{def}}{=} & \int_\Omega \kappa \nabla_h u_h \cdot \nabla_h v_h \\ & - \sum_{F \in \mathscr{F}_h} \int_F [\{\kappa \nabla_h u_h\}_\omega \cdot \mathrm{n}_F [\![v_h]\!] + [\![u_h]\!] \{\kappa \nabla v_h\}_\omega \cdot \mathrm{n}_F] \\ & + \sum_{F \in \mathscr{F}_h} \eta \frac{\gamma_F}{h_F} \int_F [\![u_h]\!][\![v_h]\!], \end{aligned} \tag{7}$$

with:

- ∇_h broken gradient on $\mathscr{T}_h$
- the weights in the average operator defined as follows: For all $F \in \mathscr{F}_h^i$ such that $F \subset \partial T_1 \cap \partial T_2$,

$$\omega_{T_1,F} = \frac{\lambda_2}{\lambda_1+\lambda_2}, \qquad \omega_{T_2,F} = \frac{\lambda_1}{\lambda_1+\lambda_2},$$

  where $\lambda_i \stackrel{\text{def}}{=} \kappa|_{T_i} \mathrm{n}_F \cdot \mathrm{n}_F$ for $i \in \{1,2\}$;
- $\gamma_F = \frac{2\lambda_1\lambda_2}{\lambda_1+\lambda_2}$ on internal faces $F \subset \partial T_1 \cap \partial T_2$,
- $\gamma_F = \kappa|_T \mathrm{n}_F \cdot \mathrm{n}_F$ on boundary faces $F \subset \partial T \cap \partial \Omega$
- and η is a (strictly positive) penalty parameter.

- The hybrid finite volume method, recovering the SUSHI scheme, see[19], [16] and [9, 8], is based on the space V_h^{hyb} defined setting $\mathbb{V}_h = \mathbb{T}_h \times \mathbb{F}_h$, $\mathscr{S}_h$ pyramidal and giving the operator $\mathfrak{G}_h^{\text{hyb}}$ obtained with the green formula.
  This method reads:

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

  with $a_h^{\text{sushi}}(u_h, v_h) \stackrel{\text{def}}{=} \int_\Omega \kappa \nabla_h u_h \cdot \nabla_h v_h$ and ∇_h broken gradient on $\mathscr{P}_h$.

As in all of these lowest order methods, gradient reconstructions are piecewise constant, integrals appearing in the defined bilinear forms are evaluated exactly using the barycenter of the mesh item (cell or face) as a quadrature node. This remark is an important point in the implementation details of forms in section 3.

3 Implementation

The framework described in section 2 allows a unified description for a large family of lowest methods and as for FE/DG methods 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 problem but also to solve and find its solution.

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 (6) with the programming counterpart in listing 1.

Listing 1 Diffusion problem 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());
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);
```

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 section 2, which constitute the back-end of the DSEL presented in section 3.3.

3.1.1 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 developed 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>)` are provided to manipulate the mesh, to extract different parts of the mesh.

3.1.2 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.

3.1.3 Linear combination, linear and bilinear contribution

The point of view presented in section 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 linear combination
`template<ValueT,ItemT> class LinearCombT`. Associated to an efficient linear algebra, this concept enables 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

3.2.1 Function spaces

Incomplete broken polynomial spaces defined by (5) are mapped onto `C++` types according to the `FunctionSpace` concept. The key role of `FunctionSpace` is to bridge the gap between the algebraic representation of DOFs and the functional representation used in the methods of section 2. This is achieved by the functions **grad** and **eval**, which are the `C++` counterparts of the linear operators $\mathfrak{G}_h$ and $\mathfrak{R}_h$ respectively; see section 2. More specifically,

(i) for all $S \in \mathscr{S}_h$, **grad** (S) returns a vector-valued linear combination corresponding to the (constant) restriction $\mathfrak{G}_h|_S$;
(ii) for all $S \in \mathscr{S}_h$ 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 (4).

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

A function space types also defines the subtypes `TestFunctionType`, `FunctionType`, `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 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 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.

3.2.2 Bilinear and linear forms

Bilinear and linear forms described in 2 result from the integration of respectively bilinear and linear terms on groups of mesh items. A **BilinearForm** and a **LinearForm** concept have been developped to represent these forms . They enable to store mesh item groups, expressions built with test, trial functions, unary and binary operators. They are the link between the numerical representation of the problem with forms and its algebraic representation with a matrix and a vector.

3.3 DSEL implementation

The **Key ingredients** to design a DSEL are:

1. Meta-programming techniques that consist in writing programs that transform types at compile time
2. Generic programming techniques that consist in designing generic components composed of abstract programs with generic types
3. Generative programming techniques that consist in generating concrete programs, transforming types with meta-programs to create concrete types to use with abstract programs of generic components
4. Expression template techniques [5, 7, 25] that consist in representing problems with expression tree and using tools to describe, parse and evaluate these trees.

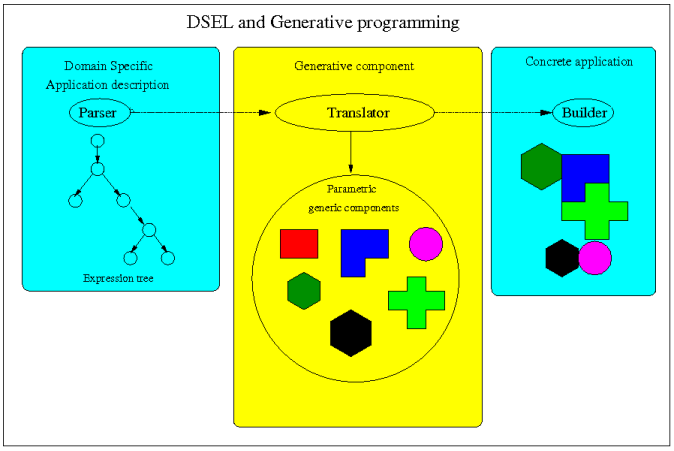


Fig. 2 Generative programming

Applying all these techniques, it is possible to represent a problem with an expression tree. Parsing this tree at compile time, using meta programming tools to introspect the expression, it is possible to select generic components, to link them together to assemble and generate a concrete program 2. The execution of this program consists in evaluating the tree at the run time, executing the concrete instance of the selected components to build a linear system, solve it to find the solution of the problem.

Using these principles, we have designed a DSEL that enables to express and define linear and bilinear forms. The terminals of our language are composed of symbols representing C++ objects with base types (Real or Integer) and with types representing discrete variables, functions, test and trial functions. Our language uses the standard C++ binary operators (+,-,*,/), the binary operator **dot** `(.,.)` repre-

senting the scalar product of vector expressions, unary operators representing standard differential operators like **grad**`(.)` and **div**`(.)`. The language is completed by a number a specific keywords **integrate**`(.,.)`, **N**`()` and **H**`()`.

The **integrate**`(.,.)` keyword associates a collection of mesh entities to linear and bilinear expression.

N`()` and **H**`()` are free functions returning discrete variable containing respectively the pre-computed values of n_F and h_F of the mesh faces of $\mathscr{T}_h$.

Our DSEL has been implemented with tools and concepts provided by the Boost::Proto template library, a powerful framework to build DSEL in C++ based on Expression templates techniques. This library provides a collection of generic concepts and meta functions that help to design a DSEL, its grammar and tools to parse and evaluate expressions with a tree representation. We used these tools to design the DSEL front end that enables to create expressions with terminals, unary and binary operators, and predefined free functions used as specific keywords. The grammar of the DSEL is based on tag structures and on meta functions allowing the introspection of the nodes of the expression tree. The DSEL back ends are composed of algebraic structures (matrices, vectors, linear combinations) used in algorithms. We use `Evaluation Context` object, kind of function objects that are passed along expression trees. They associate behaviors to node types, in other words they enable to call specific piece of algorithm regarding the type of the node expression. When an expression is evaluated, the context is invoked at each node of the tree. Algorithms are then implemented as specific expression tree evaluation, as a sequence of piece of algorithms associated to the behavior of the `Evaluation Context` on each node.

Algorithms associated to linear variational formulation were implemented with `LinearContext` and `BilinearContext` objects. These objects, with a reference to a linear system back end object, allow to build a global linear system with different linear algebra packages.

Let us consider for instance the bilinear form defined in listing 2:

Listing 2 Expression defining a bilinear form

```
BilinearForm ah = integrate(allCells(Th), dot(K*grad(u),grad(v)) ;
```

`allCells(Th)`, `K`, `u`, `v` are terminals of the language. **integrate**, **dot** and **grad** are specific keywords of the language. The expression defined in listing 2 has a tree structure which has the following representation.

When the expression is evaluated, the behavior associated to these context objects can be described as follows:

1. The root node of the expression tree is associated to the tag `tag::`**integrate** composed of an expression (`allCells(Th)`) and the expression (**dot**`(K*`**grad**`(u),`**grad**`(v))`);

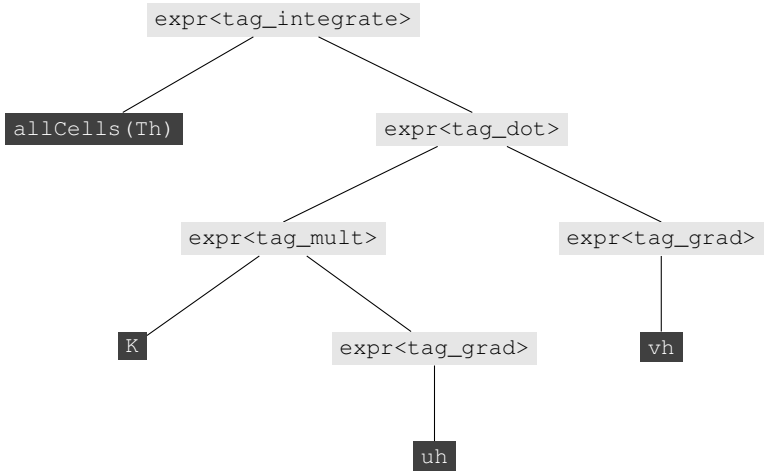


Fig. 3 Expression tree for the bilinear form 2. Expressions are in *light gray*, language terminals in *dark gray*

2. The integration algorithm consists in iterating on the cell elements of the `allCells(Th)` and evaluating the bilinear expression on each cell. This bilinear expression is composed of:
   - a trial function expression: `K*`**grad**`(u)`;
   - a test function expression: **grad**`(v)`
   - a binary operator associated to the tag: `tag::`**dot**

   With a linear context object, the evaluation of the trial function and of the test function 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.

This algorithm is generated with the simple following template function

```
template<typename ItemT,
         typename BilinearExprT,
         typename LinearContextT>
void integrate(Mesh const& mesh,
               GroupT<ItemT> const& group,
               BilinearExprT const& expr,
               LinearContextT& ctx)
{
  static const Context::ePhaseType
       phase = LinearContextT::phase_type;
  typedef tag_of<BilinearExprT>::type tag_op;
  auto test = TestFunction(expr);
  auto trial = TrialFunction(expr);
```

```
  auto system = ctx.getSystem();
  std::for_each(group.begin(),
                group.end(),
                [&system,&mesh](ItemT& cell)
  {
    assemble<tag_op,phase>(system,                    //! linear system
                           measure(mesh,cell), //! cell measure
                           proto::eval(trial,cell), //! trial linear com-
bination
                           proto::eval(test,cell)); //! test linear com-
bination
  }
}
```

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

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

3.3.1 Boundary condition management

In section 2 we have presented only homegeneous boundary conditions. In fact most of these methods are easily extended to more general boundary conditions. Let $\partial\Omega_d \subset \partial\Omega$ and $\partial\Omega_n \subset \partial\Omega$, let consider the following conditions:

$$u = g \text{ on } \partial\Omega_d,\ g \in L^2(\partial\Omega_d) \tag{8}$$

$$\frac{\partial u}{\partial n} = h \text{ on } \partial\Omega_n,\ h \in L^2(\partial\Omega_n) \tag{9}$$

To manage such conditions, we introduce: (i) extra degree of freedoms on boundary faces, (ii) constraints on the bilinear form or (iii) extra terms in the linear form. These constraints and terms lead to add or remove some equations in the matrix and to add extra terms in the right hand side of the linear system.

In our DSEL, the key words **trace**`(u)` enable to recover degree of freedoms on mesh elements, and **on**`(.,.)` enable to had constraints on group of on mesh elements. For example, with the hybrid method the boundary conditions 8 and 9 are expressed with the expressions of listing 3

Listing 3 boundary conditions management

```
BilinearForm ah = integrate(allCells(Th), dot(K*grad(u),grad(v)) ;
LinearForm bh = integrate(allCells(Th),f*v) ;

//Dirichlet condition on ∂Ω_d
ah += on(boundaryFaces(Th,"dirichlet"),trace(u)=g) ;

//Neumann condition on ∂Ω_n
bh += integrate(boundaryFaces(Th,"neumann"),h*trace(v)) ;
```

4 Applications

In this section we validate the design of our DSEL. We implement and compare different lowest order methods on a pure diffusion problem then present the results of a diffusion problem with a heterogenous permeability field coming from a more realistic reservoir model.

The prototypes implemented are 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 Genuine Intel W3530, 2.80GHz, 8MB for cache size.

4.1 Pure diffusion

We present for the diffusion problem different variational discrete formulations that we compare to their programming counterpart.

We present some numerical results, run on a family of meshes of increasing sizes $h \in \mathscr{H}$. We list in tables the value of different error norms regarding an analytical solution. For each kind of error, we estimate the order of convergence as $\text{order} = d \ln(e_1/e_2) / \ln\left(\text{card}(\mathscr{T}_{h_2})/\text{card}(\mathscr{T}_{h_1})\right)$, where e_1 and e_2 denote, respectively, the errors committed on $\mathscr{T}_{h_1}$ and $\mathscr{T}_{h_2}$, $h_1, h_2 \in \mathscr{H}$. We check the theoretical convergence results detailled in [13].

To evaluate errors, we consider the norms

$$\|u\|_{L^2}^2 = \int_\Omega u^2 \quad \text{and} \quad \|u\|_L^2 = \sum_{T\in\mathscr{T}_h} \sum_{F\in\mathscr{F}_T} \int_F \frac{1}{d_{F,T}^2} \mathfrak{T}(u)^2$$

where $\mathfrak{T}(u)$ is a trace operator on mesh faces, and $d_{F,T}$ is the distance of the center of a cell T to a face $F \subset \partial T$.

To analyze the performance of the framework, we evaluate the overhead of the language, the relative part of algebraic computations (defining, building and solving linear systems) and linear combination computations, studying the following criteria:

- t_{start} the time to precompute trace and gradient operators, to build the expression tree describing linear and bilinear forms;
- t_{def} the time to compute the linear system profile;
- t_{build} the time to fill the linear system evaluating the expression tree;
- t_{solve} the time to solve le linear system with linear algebra layer;
- N_{it} the number of iterations of the linear solver, the ILU0 preconditioned BiCGStab algorithm with 10^{-6} tolerance;
- N_{nz} the number on non zero entries of the linear system of the test case.

We compare all these times in seconds to $t_{ref} = \frac{t_{solver}}{N_{it}}$ the average time of one solver iteration approximatively equal to a fixed number of matrix vector multiplication operations.

In iterative methods (time integration, non linear solver), t_{start} and t_{def} correspond to computation phases only done once before the first iterative step, while the t_{build} corresponds to a computation phase done at each step. A careful attention is paid to the t_{build} results specially for such algorithms.

Nnz is an important criterion to evaluate the amount of memory used by the method.

We consider the problem:

$$\begin{cases} -\triangle u = 0 & \text{in } \Omega \subset \mathbb{R}^3, \\ u = g & \text{on } \partial\Omega. \end{cases}$$

4.1

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

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

The discrete formulations of the problem with the G-method, the ccG-method and the Hybrid-method defined in section 2 are represented by the definition of the bilinear forms a_h^{g}, a_h^{ccg}, a_h^{hyb}. We can compare them to their programming counterpart in listings 4,5 and 6

Listing 4 C++ implementation of a_h^{g} and b_h

```
MeshType Th; // declare T_h
BoundaryFaceVarType g; // declare boundary values
auto Vh = newP0Space(Th);
auto Uh = newGSpace(Th);
auto u = Uh->trial("U");
auto v = Vh->test("V");
BilinearForm ah_g =
 integrate(allFaces(Th), dot(N(), avg(grad(u))) * jump(v));
```

```
ah_g += on(boundaryfaces(Th),trace(u)=g) ;
```

Listing 5 C++ implementation of a_h^{ccg}

```
MeshType Th; // declare T_h
BoundaryFaceVarType g; // declare boundary values
auto Uh = newCCGSpace(Th);
auto u = Uh->trial("U");
auto v = Uh->test("V");
auto lambda = eta*gamma/H();
BilinearForm ah_ccg =
  integrate(allCells(Th),dot(grad(u),grad(v))) +
  integrate(allFaces(Th),-jump(u)*dot(N(),avg(grad(v)))
                         -dot(N(),avg(grad(u)))*jump(v)
                         +lambda*jump(u)*jump(v);
ah_ccg += on(boundaryfaces(Th),trace(u)=g) ;
```

Listing 6 C++ implementation of a_h^{hyb}

```
MeshType Th; // declare T_h
BoundaryFaceVarType g; // declare boundary values
auto Uh = newHybridSpace(Th);
auto u = Uh->trial("U");
auto v = Uh->test("V");
BilinearForm ah_hyb =
 integrate(allFaces(Th),dot(grad(u),grad(v));
ah_hyb += on(boundaryfaces(Th),trace(u)=g) ;
```

We consider the analytical solution $u(x,y,z) = sin(\pi x)sin(\pi y)sin(\pi z)$ of the diffusion problem (4.1) on the square domain $\Omega = [0,1]^3$ with $f(x,y,z) = 3\pi u(x,y,z)$.

Table 1, 2 and 3 list the errors in the L^2 and L norm of respectively the G method, the ccG method and the hybrid method.

Table 1 Diffusion test case: G method

card($\mathscr{T}_h$)	h	$\|u-u_h\|_L$	order	$\|u-u_h\|_{L^2(\Omega)}$	order
1000	1.00e-01	1.58e-02		2.92e-03	
8000	5.00e-02	3.96e-03	2.	7.28e-04	2.
64000	2.50e-02	9.89e-04	2.	1.82e-04	2.
125000	1.25e-02	6.32e-04	2.	1.16e-04	2.

In Figure 4, we compare convergence error of the G-method, the ccG method, the Hybrid-method and a standard hand written L-Scheme FV method.

In the tables 4, 5, 6 and 7, we compare the performance of each methods.

The analysis of these results shows that the G-method is comparable to the hand written FV method and the language implementation does not contribute to extra

Table 2 Diffusion test case: ccG method

card($\mathscr{T}_h$)	h	$\|u-u_h\|_L$	order	$\|u-u_h\|_{L^2(\Omega)}$	order
1000	1.00e-01	3.1474e-02		5.3866e-03	
8000	5.00e-02	7.8977e-03	1.99	1.4257e-03	1.92
64000	2.50e-02	1.9763e-03	2.	3.6157e-04	1.95
512000	1.25e-02	1.2649e-03	2.	2.3180e-04	1.95

Table 3 Diffusion test case: Hybrid method

card($\mathscr{T}_h$)	h	$\|u-u_h\|_L$	order	$\|u-u_h\|_{L^2(\Omega)}$	order
1000	1.00e-01	1.58e-02		2.92e-03	
8000	5.00e-02	3.95e-03	2.	7.28e-04	2.01
64000	2.50e-02	9.87e-04	2.	1.82e-04	2.
512000	1.25e-02	6.32e-04	2.	1.16e-04	2.

Table 4 Diffusion test case: G-method performance results

card($\mathscr{T}_h$)	N_{it}	N_{nz}	t_{start}	t_{def}	t_{build}	t_{solve}	t_{ref}	t_{start}/t_{ref}	t_{def}/t_{ref}	t_{build}/t_{ref}
1000	4	16120	8.8987e-02	1.1998e-02	7.9980e-03	3.0000e-03	7.50e-04	118.65	16.00	10.66
8000	8	140240	6.0191e-01	1.0398e-01	6.4990e-02	1.6997e-02	2.12e-03	283.30	48.94	30.59
64000	14	1168480	4.8033e+00	8.4787e-01	6.2190e-01	2.0097e-01	1.44e-02	334.61	59.06	43.32
125000	25	2300600	7.0929e+00	1.7137e+00	1.1738e+00	5.9191e-01	2.37e-02	299.58	72.38	49.58

Table 5 Diffusion test case: ccG-method performance results

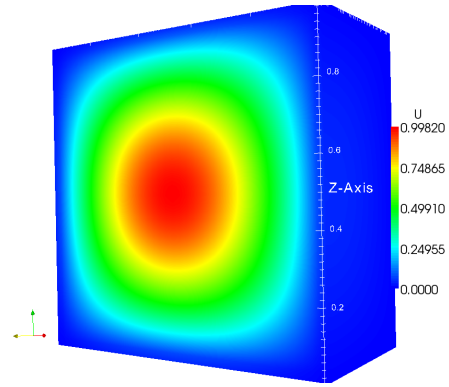
card($\mathscr{T}_h$)	N_{it}	N_{nz}	t_{start}	t_{def}	t_{build}	t_{solve}	t_{ref}	t_{start}/t_{ref}	t_{def}/t_{ref}	t_{build}/t_{ref}
1000	3	117642	6.5990e-02	3.5495e-01	9.2986e-02	3.0995e-02	1.03e-02	6.39	34.36	9
8000	5	1145300	5.2292e-01	3.4685e+00	8.0088e-01	2.9596e-01	5.92e-02	8.83	58.6	13.53
64000	8	10114802	4.1344e+00	2.9890e+01	6.9929e+00	3.0625e+00	3.83e-01	10.8	78.08	18.27
125000	10	20017250	8.1658e+00	6.0900e+01	1.3516e+01	6.3850e+00	6.39e-01	12.79	95.38	21.17

Table 6 Diffusion test case: Hybrid method performance results

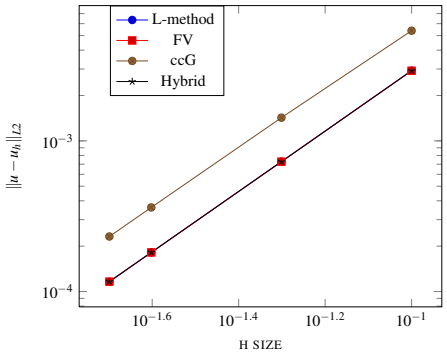
card($\mathscr{T}_h$)	N_{it}	N_{nz}	t_{start}	t_{def}	t_{build}	t_{solve}	t_{ref}	t_{start}/t_{ref}	t_{def}/t_{ref}	t_{build}/t_{ref}
1000	7	16120	4.6993e-02	1.0999e-02	4.0000e-03	2.1997e-02	3.14e-03	14.95	3.5	1.27
8000	17	140240	3.4095e-01	1.1898e-01	2.5996e-02	1.6098e-01	9.47e-03	36.01	12.56	2.75
64000	33	1168480	2.8686e+00	1.1128e+00	2.1197e-01	2.4106e+00	7.30e-02	39.27	15.23	2.9
125000	50	5563700	5.2122e+00	2.0507e+00	3.8094e-01	4.5323e+00	9.06e-02	57.5	22.62	4.2

cost. The G-method and the Hybrid-method have equivalent convergence order. A closer look to the N_{nz} column shows that the ccG method requires much more nonzero entries for the linear system than the G-method and the hybrid-method, and we can see the effect on the cost of the linear system building phase which is more important for the ccG method than for the G-method.

The inspection of the columns t_{start}/t_{ref} and t_{build}/t_{ref} shows that the implementation remains scalable regarding the size of the problem.



(a) 3D view



(b) convergence curves

Fig. 4 Diffusion problem

Table 7 Diffusion test case: standard hand written performance results

card($\mathscr{T}_h$)	N_{it}	N_{nz}	t_{start}	$t_{def+build}$	t_{solve}	t_{ref}	t_{start}/t_{ref}	t_{build}/t_{ref}
1000	4	16120	4.899e-02	3.399e-02	3.998e-03	1.00e-03	49.01	34.01
8000	7	140240	3.519e-01	2.149e-01	3.399e-02	4.86e-03	72.47	44.26
64000	13	1168480	2.786e+00	1.861e+00	3.489e-01	2.68e-02	103.8	69.34
125000	16	9536960	5.338e+00	3.893e+00	7.688e-01	4.81e-02	111.09	81.02

4.2 SPE10 test case

The test case is based on data taken from the second model of the 10^{th} SPE test case [**?**]. The geological model is a $1200 \times 2200 \times 170\,ft$ block discretized with a regular Cartesian grid with $60 \times 220 \times 85$ cells. This model is a part of a Brent sequence. The maps of porosity, horizontal and vertical permeability can be downloaded from the web site of the project [**?**].

Let Ω be the domain define by the layer 85 of the grid, $\partial\Omega_{xmin}$ and $\partial\Omega_{xmax}$, the left and right boundary of the layer. We consider the following heterogeneous diffusion model problem :

$$\begin{aligned} -\nabla\cdot(\kappa\nabla u) &= 0 && \text{in } \Omega, \\ u &= P_{min} = -10 && \text{on } \partial\Omega_{xmin}, \\ u &= P_{max} = 10 && \text{on } \partial\Omega_{xmax}, \\ \frac{\partial u}{\partial n} &= 0 && \text{on } \partial\Omega \ \partial\Omega_{xmin} \cup \partial\Omega_{xmax} \end{aligned} \tag{10}$$

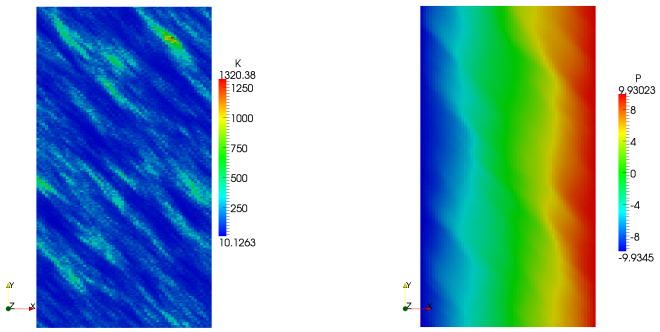
where κ is associated to the map of the horizontal permeability field of the layer 85.

The discrete formulations of this problem (10) have been implemented with the Hybrid-method defined in section 2 as in listing 7

Listing 7 C++ implementation of a_h^{hyb}

```
MeshType Th; // declare T_h
Real Pmin = -10, Pmax=10 ;
auto Uh = newHybridSpace(Th);
auto u = Uh->trial("U");
auto v = Uh->test("V");
BilinearForm ah_hyb =
 integrate(allFaces(Th),k*dot(grad(u),grad(v));
 ah_hyb += on(boundaryFaces(Th,"xmin"),trace(u)=Pmin) ;
 ah_hyb += on(boundaryFaces(Th,"xmax"),trace(u)=Pmax) ;
```

In figure 5, we have a 2D view of the permeability field and of the solution of the problem.



(a) 3D view (b) 3D view

Fig. 5 SPE10 permeability field and pressure solution

5 Conclusion and perspectives

Our DSEL for lowest-order methods allows to describe and solve various diffusion problems. Different numerical methods recovering standard methods (L-scheme, ccG, Sushi method) have been implemented with a high level language close to the 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 future works, we plan to extend our DSEL to take into account non linear formulations hiding the complexities of derivatives computation with Frechet's derivatives and to address new business applications with linear elasticity, poro-mechanic or dual medium model.

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

References

1. Aavatsmark, I., Barkve, T., Bøe, Ø. and Mannseth, T.: Discretization on non-orthogonal, curvilinear grids for multi-phase flow. In *Prov. of the 4th European Conf. on the Mathematics of Oil Recovery*, volume D, Røros, Norway, 1994.
2. Aavatsmark, I., Barkve, T., Bøe, Ø. and Mannseth, T.: Discretization on non-orthogonal, quadrilateral grids for inhomogeneous, anisotropic media. *J. Comput. Phys.*, 127:2–14, 1996.
3. Aavatsmark, I., Barkve, T., Bøe, Ø. and Mannseth, T.: Discretization on unstructured grids for inhomogeneous, anisotropic media, Part I: Derivation of the methods. *SIAM J. Sci. Comput.*, 19(5):1700–1716, 1998.
4. Aavatsmark, I., Barkve, T., Bøe, Ø. and Mannseth, T.: Discretization on unstructured grids for inhomogeneous, anisotropic media, Part II: Discussion and numerical results. *SIAM J. Sci. Comput.*, 19(5):1717–1736, 1998.
5. Abrahams, Davis and Gurtovoy, leksey: C++ template metaprogramming : Concepts, tools, and thechniques from boost and beyond. C++ in Depth Series. Addison-Wesley Professional, 2004.
6. Agélas, L., Di Pietro, D. A. and Droniou, J.: The G method for heterogeneous anisotropic diffusion on general meshes. *M2AN Math. Model. Numer. Anal.*, 44(4):597–625, 2010. http://dx.doi.org/10.1051/m2an/2010021.
7. Aubert, Pierre and Di Césaré, Nicolas: Expression templates and forward mode automatic differentiation. Computer and information Science, chapter 37, pages 311-315. Springer, New York, NY, 2001.
8. Brezzi, F., Lipnikov, K. and Shashkov, M.: Convergence of mimetic finite difference methods for diffusion problems on polyhedral meshes. *SIAM J. Numer. Anal.*, 45:1872–1896, 2005.
9. Brezzi, F., Lipnikov, K. and Simoncini, V.: A family of mimetic finite difference methods on polygonal and polyhedral meshes. *M3AS*, 15:1533–1553, 2005.
10. Di Pietro, D. A.: Cell centered Galerkin methods. *C. R. Acad. Sci. Paris, Ser. I*, 348:31–34, 2010. http://dx.doi.org/10.1016/j.crma.2009.11.012.
11. Di Pietro, D. A.: Cell-centered Galerkin methods. *C. R. Math. Acad. Sci. Paris*, 348:31–34, 2010.
12. Di Pietro, D. A.: Cell centered Galerkin methods for diffusive problems. Preprint available at http://hal.archives-ouvertes.fr/hal-00511125/en/, September 2010. Submitted.

13. Di Pietro, D. A.: A compact cell-centered Galerkin method with subgrid stabilization. Submitted. Preprint available at http://hal.archives-ouvertes.fr/hal-00476222/en/, April 2010.
14. Di Pietro, D. A.: A compact cell-centered Galerkin method with subgrid stabilization. *C. R. Acad. Sci. Paris, Ser. I.*, 348(1–2):93–98, 2011.
15. Droniou, J. and Eymard, R.: A mixed finite volume scheme for anisotropic diffusion problems on any grid. *Num. Math.*, 105(1):35–71, 2006.
16. Droniou, J., Eymard, R., Gallouët, T. and Herbin, R.: A unified approach to mimetic finite difference, hybrid finite volume and mixed finite volume methods. *M3AS*, 20(2):265–295, 2010.
17. Edwards, M.G. and Rogers, C.F.: A flux continuous scheme for the full tensor pressure equation. In *Prov. of the 4th European Conf. on the Mathematics of Oil Recovery*, volume D, Røros, Norway, 1994.
18. Edwards, M.G. and Rogers, C.F.: Finite volume discretization with imposed flux continuity for the general tensor pressure equation. *Comput. Geosci.*, 2:259–290, 1998.
19. Eymard, R., Gallouët, Th. and Herbin, R.: Discretization of heterogeneous and anisotropic diffusion problems on general nonconforming meshes SUSHI: a scheme using stabilization and hybrid interfaces. *IMA J. Numer. Anal.*, 2010. Published online.
20. Grospellier, Gilles and Lelandais, Benoit: 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.
21. Di Pietro, D. A.: Cell centered galerkin methods for diffusive problems. *M2AN Math. Model. Numer. A nal.*, 2010. Published online.
22. Di Pietro, D. A. and Gratien, J-M.: Lowest order methods for diffusive problems on general meshes: A unified approach to definition and implementation. In *FVCA6 proceedings*, 2011.
23. Prud'homme, C.: A domain specific embedded language in C++ for automatic differentiation, projection, integration and variational formulations. *Scientific Programming*, 2(14):81–110, 2006.
24. Prud'homme, C.: Life: Overview of a unified c++ implementation of the finite and spectral element methods in 1d, 2d and 3d. In Workshop On State-Of-The-Art In Scientific And Parallel Computing, Lecture Notes in Computer Science, page10. Springer-Verlag, 2007.
25. Veldhuizen, Todd: Using c++ template metaprograms. C++ report, 7(4):36-43, May 1995. reprinted in C++ Gems, ed. Stanley Lippman, 1995.