A Domain Specific Embedded Language in C++ for Automatic Differentiation, Projection, Integration and Variational Formulations

Christophe Prud'homme*

October 2, 2006

Abstract

In this article, we present a domain specific embedded language in C++ that can be used in various contexts such as numerical projection onto a functional space, numerical integration, variational formulations and automatic differentiation. Albeit these tools operate in different ways, the language overcomes this difficulty by decoupling expression constructions from evaluation. The language is implemented using expression templates and meta-programming techniques and uses various Boost libraries. The language is exercised on a number of non-trivial examples and a benchmark presents the performance behavior on a few test problems.

1 Introduction

Numerical analysis tools such as differentiation, integration, polynomial approximations or finite element approximations are standard and mainstream tools in scientific computing. Many excellent libraries or programs provide a high level programming interface to these methods : *(i)* programs that define a specific language such as the Freefem software family [12, 21], the Fenics project [18, 17], Getdp [11] or Getfem++ [23], or *(ii)* libraries or frameworks that supply some kind of domain specific language embedded in the programming language — hereafter called DSEL — such as LifeV (C++) [1, 20], Sundance (C++) [19], Analysa (Scheme, which is suited for embedding sub-languages like other Lisp based languages) [5].

These high level interfaces or languages are desirable for several reasons: teaching purposes, solving complex problems with multiple physics and scales or rapid prototyping of new methods, schemes or algorithms. The goal is always to hide (ideally all) technical details behind software layers and provide only the relevant components required by the user or programmer.

The DSEL approach has advantages over generating a specific language like in case *(i)* : compiler construction complexities can be ignored, other libraries can concurrently be used which is often not the case of specific languages which would have to also develop their own libraries and DSELs inherit the capabilities

*Université Joseph Fourier, LMC/LJK, 51, rue des Mathématiques, BP53X 38041 Grenoble Cedex 9, France, christophe.prudhomme@imag.fr

of the language in which they are written. However, DSELs are often defined for one particular task inside a specific domain [24] and implementation or parts of implementation are not shared between different DSELs.

This article proposes a DSEL for automatic differentiation, projection, integration or variational formulations. The language implementation uses expression templates [24] and other meta-programming techniques [2]. Related works are [20] and [19], but they differ with the proposed DSEL in many aspects: the former was designed only for variational formulations and requires to write the expression object by hand which can become complicated and error prone, while the latter implements the DSEL in an object oriented way without relying on meta-programming or expression templates.

Other objectives of our DSEL implementation are that it should *(i)* be efficient enough to integrate high performance/parallel software, *(ii)* be generic enough to accommodate different numerical types — for example arbitrary precision, see [22] but we won't discuss these aspects here. — A performance benchmark is available in section 4.1.

To illustrate further what the DSEL achieves, here is a comparison between a mathematical formulation of a bilinear form (1) and its programming counterpart, see listing 1.

$$\begin{aligned} a : X_h \times X_h &\to \mathbb{R} \\ (u, v) &\to \int_\Omega \nabla u \cdot \nabla v + uv \end{aligned} \tag{1}$$

Listing 1: Variational Formulation in *C++*; the `t` extension of `gradt` and `idt` identifies the trial functions

```
// a mesh of Ω ⊂ R^d, d = 1,2,3
Mesh mesh;
// Finite element scalar space P_K, K = 1,2,3,...
Space<Mesh,FEM_PK<d,K> > Xh(mesh);
// two elements of the Space Xh
Space<Mesh,FEM_PK<d,K> >::element_type u(Xh), v(Xh);
// A matrix in CSR format
csr_matrix_type M;
// bilinear form with M as its matrix representation
// with integration over all elements of the mesh
// and a method for exact integration of polynomials of
// degree ≤ K (IM_PK<d,K>)
BilinearForm<Xh,Xh> a(Xh,Xh,M);
a = integrate( elements(mesh),
               dot(gradt(u),grad(v))+idt(u)*id(v),
               IM_PK<N,K>() );
```

We clearly identify in listing 1 the variational formulation stated in equation (1). We shall describe the various steps to achieve this level of expression with as little overhead as possible. In section 2, we present some concepts concerning mainly integration and variational formulations, then in section 3 we present the main points about the DSEL. Finally in section 4, we present some non-trivial examples to exercise the language.

This article contains many listings written in C++ however most of them are not correct C++ in order to simplify the exposition. In particular, C++ keywords like `typename` or `inline` are often not present. Also many numerical ingredients such as polynomial approximations, numerical integration methods used in this article are not described or only very roughly, another publication will cover the mathematical kernel used by the DSEL in more details [22].

2 Preliminaries on Variational Forms

In what follows, we consider a domain $\Omega \subset \mathbb{R}^d$, $d = 1, 2, 3$ and its associated mesh $\mathcal{T}$ — out of d-simplices and product of simplices.

2.1 Mesh

We present first some tools that will be used later, namely how to extract parts of a mesh and the geometric mapping that maps a convex of reference — where polynomial sets and quadratures are constructed — to any convex of the mesh.

2.1.1 Mesh Parts Extraction

While applying integration and projection methods, it is common to be able to extract parts of the mesh. Hereafter we consider only elements of the mesh and elements faces. We wish to extract easily subsets of convexes out the total set consituting $\mathcal{T}$.

To do this out mesh data structure which is by all means fairly standard uses the Boost.Multi_index library[1] to store the elements, elements faces, edges and points. This way the mesh entities are indexed either by their ids, their markers — material properties, boundary ids…, — their location — whether the entity is internal or lies on the boundary of the domain. — Other indices could be certainly defined, however those three allow already a wide range of applications[2].

Thanks to Boost.Multi_index, it is trivial to retrieve pairs of iterators over the entities — elements, faces, edges, points — containers depending on the usage context. The pairs of iterators are then turned into a range, see Boost.Range[3], to be manipulated by the integration and projection tools that will be presented later.

A number of free functions are available that hide all details about the mesh class to concentrate only on the relevant parts.

- `elements(<mesh>)` the set of convexes constituting the mesh
- `idedelement(<mesh>, <id>)` the convex with id `<id>`
- `idedelements(<mesh>, <lower bound>, <upper bound>)` iterator range of convexes whose ids are in the range given by the predicates `<lower bound>` and `<upper bound>`, for example `idedelements(mesh,1000<=_1,_1<5000)`[4]
- `markedelements(<mesh>, <marker>)` iterator range over elements marked with `marker`

[1] http://www.boost.org/libs/multi_index/doc/index.html
[2] Another useful type of indexation could be the process id in a parallel framework.
[3] http://www.boost.org/libs/range/index.html
[4] `_1` is part of Boost.Lambda.

- `markedelements(<mesh>, <lower bound>, <upper bound>)` iterator range over elements whose markers are in the range given by the predicates `<lower bound>` and `<upper bound>`, for example

  ```
  markedelements(mesh,1<=_1,_1<5)
  ```

- `faces(<mesh>)` iterator range over all `mesh` element faces
- `markedfaces(<mesh>,<marker>)` iterator range over `mesh` element faces marked with `marker`
- `markedfaces(<mesh>,<lower bound>, <upper bound>)` iterator range over `mesh` element faces whose markers are in the range given by the predicates `<lower bound>` and `<upper bound>`, for example

  ```
  markedfaces(mesh, 1<=_1,_1<5)
  ```

- `bdyfaces(<mesh>)` iterator range over all boundary `mesh` element faces
- `internalfaces(<mesh>)` iterator range over all internal `mesh` element faces

2.1.2 Geometric Mapping

Functional spaces and quadrature methods, for example, are derived from polynomial sets or families that have to be constructed over the convexes of $\mathcal{T}$. Instead of doing this, it is common to construct these polynomials over a reference convex $\hat{T}$— segment, triangle, quadrangle, tetrahedron, hexahedron, prism or pyramid — and provide a geometric mapping or transformation from the reference convex $\hat{T}$ to any convex $T \in \mathcal{T} \subset \mathbb{R}^P, P \leq d$. We need to be able also to transform subentities, such as faces or points, of the reference element to the corresponding entities, faces and points respectively, in the real element.

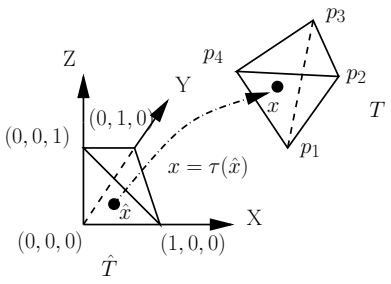


Figure 1: Geometric mapping τ from the reference tetrahedron $\hat{T}$ to a real tetrahedron T in 3D.

From now on, we denote with a ($\hat{}$) the quantities defined on the reference element. We define $\tau : \mathbb{R}^P \to \mathbb{R}^N$ that maps $\hat{T}$ to T. We shall denote K_τ its gradient, B_τ its pseudo-inverse and J_τ its jacobian. The geometric mapping is described by *(i)* a n_g components polynomial vector $\{\phi_g(\hat{x})\}_{g=1\ldots n_g}$ and *(ii)* the geometric points $\{p_g\}_{g=1\ldots n_g}$ of T such that

$$x = \tau(\hat{x}) = \sum_{g=1\ldots n_g} \phi_g(\hat{x})\, p_g \qquad (2)$$

We denote by $G = (p_1, \ldots, p_{n_g})$ the $N \times n_g$ matrix of geometric nodes. Equation (2) and quantities mentioned above are computed as follows, for any $\hat{x} \in \hat{T}$

$$\begin{aligned} x = \tau(\hat{x}) &= G\ \phi(\hat{x}) \\ K_\tau(\hat{x}) &= G\ \nabla\ \phi(\hat{x}) \\ J_\tau(\hat{x}) &= \begin{cases} \det(K_\tau(\hat{x})) & \text{if } P = N \\ \det(K_\tau^t(\hat{x}) K_\tau(\hat{x}))^{1/2} & \text{if } P \neq N \end{cases} \\ B_\tau(\hat{x}) &= \begin{cases} K_\tau^{-t}(\hat{x}) & \text{if } P = N \\ K_\tau(\hat{x})(K_\tau^t(\hat{x}) K_\tau(\hat{x}))^{-1} & \text{if } P \neq N \end{cases} \end{aligned} \qquad (3)$$

where $K_\tau^t(\hat{x})$ denotes the transpose of $K_\tau(\hat{x})$.

Equipped with the geometric mapping concept, we compute an integral on T as an integral on $\hat{T}$: if f is a function defined on T,

$$\int_T f(x)\mathrm{d}x \;=\; \int_{\hat{T}} f(\tau(\hat{x}))J_\tau(\hat{x})\mathrm{d}\hat{x} \tag{4}$$

and using a quadrature formula:

$$\int_{\hat{T}} f(\tau(\hat{x}))J_\tau(\hat{x})\mathrm{d}\hat{x} \;\approx\; \sum_{q=1\ldots Q} \hat{w}_q \; f(\tau(\hat{x}_q))J_\tau(\hat{x}_q) \tag{5}$$

where $\{\hat{x}_q, \hat{w}_q\}_{q=1\ldots Q}$ are quadrature nodes and quadrature weights defined in the reference element.

In our framework, the geometric mapping is not used directly by the developer but rather what we call the geometric mapping context which is a subclass of the geometric mapping class. The geometric mapping context is linked to an element T of the mesh such that, given a set of points $\{\hat{x} \in \hat{T}\}$, it provides information for each point in the set $\{x; x \in T \text{ and } x = \tau(\hat{x})\}$ such as the jacobian value $J_\tau(\hat{x})$, the gradient $K_\tau(\hat{x})$ of the mapping, the pseudo-inverse $B_\tau(\hat{x})$ of the gradient; or if the point $\hat{x}$ is on a face of $\hat{x}$, then x is on a face of T and the context provides the normal to the face at this point. A shortened interface of the `Context` class is presented in the listing 2.

Listing 2: Geometric mapping context class

```
class GeoMap::Context {
public:
  ...
  /**
   * constructor
   * G is column oriented matrix: each column contains
   * the coordinate of a geometric point of T
   * P is column oriented matrix: each column contains
   * the coordinate of the points in reference element
   */
  Context( matrix_node_type G, matrix_node_type P );
  /** return the dimension of the real element */
  int  N ();
  /** return the dimension of the reference element */
  int  P ();
  /** get the q-th node of P in the reference element */
  node_type const&  xRef ( int q ) const;
  /** get the q-th node in the real element */
  node_type const&  xReal ( int q ) const;
  /**  get the value of the jacobian at the q-th node
       in the reference element*/
  double J ( int q ) const;
  /** get the value of the gradient at q-th node */
  matrix_type const&  K ( int q ) const;
  /**get the value of the pseudo-inverse at q-th node */
  matrix_type const& B ( int q );
  /** get the coordinates of the geometric nodes */
  matrix_node_type const&  G ();};
```

Another subclass of the geometric mapping class is `Inverse` which as its name state does the inverse of the transformation : given a point x in $T \subset \mathbb{R}^N$ computes its location in the reference element $\hat{x}$ in $\hat{T} \subset \mathbb{R}^P$. `Inverse` is particularly useful for interpolation purposes.

We define a bilinear form $a : X \times Y \to \mathbb{R}$ and a linear form $\ell : X \to \mathbb{R}$, where X and Y are suitable function spaces defined on Ω. The finite element method discretizes X and Y using polynomials spaces defined on $\mathcal{T}$. We denote by $\mathcal{N}_X$ and $\mathcal{N}_Y$ the dimension of the discrete spaces X and Y[5] and by $\{\psi_i\}_{i=1\ldots\mathcal{N}_X}$ and $\{\varphi_i\}_{i=1\ldots\mathcal{N}_Y}$ a basis for X and Y respectively. For any $v \in X$, we have $v = \sum_{i=1\ldots\mathcal{N}} v_i\psi_i$, and similarly for the functions of Y. We can then write the entries $A_{ij}, i = 1\ldots\mathcal{N}_X, j = 1\ldots\mathcal{N}_X$ of the matrix A associated with a and the entries $L_i, i = 1\ldots\mathcal{N}_X$ of the vector L associated with ℓ as follows:

$$\begin{array}{rcll} A_{ij} & = & a(\psi_i, \varphi_j) & i = 1\ldots\mathcal{N}_X, j = 1\ldots\mathcal{N}_Y \\ L_i & = & \ell(\psi_i) & i = 1\ldots\mathcal{N}_X \end{array} \tag{6}$$

To construct A and L, we follow a standard assembly process that iterates over the elements T of $\mathcal{T}$ since $a(v, u)$ can be written as $\sum_{T\in\mathcal{T}} a_T(v, u)$ and similarly with ℓ and L. We then introduce *(i)* the restriction of the basis functions to T, $\{\psi_i^T\}_{i=1\ldots N_X}$ and $\{\varphi_i^T\}_{i=1\ldots\mathcal{N}_Y}$ where i is a local numbering over T, and *(ii)* the local to global mappings $\iota_X(\cdot,\cdot)$ and $\iota_Y(\cdot,\cdot)$ between the local numbering of the degrees of freedom and the global one. For example $\iota_X(T, i)$ is the global degree of freedom to which the i-th local degree of freedom of T contributes to. The assembly process is described in the algorithm 1.

Algorithm 1 Standard assembly procedure for A and L.

$A = 0$
for $T \in \mathcal{T}$ **do**
  for $i = 1\ldots N_X$ **do**
    for $j = 1\ldots N_Y$ **do**
      $A_{\iota_X(T,i)\iota_Y(T,j)} = A_{\iota_X(T,i)\iota_Y(T,j)} + a_T(\psi_i^T, \varphi_j^T)$
    end for
    $L_{\iota_X(T,i)} = L_{\iota_X(T,i)} + \ell_T(\psi_i^T)$
  end for
end for

In standard finite element software, the assembly is often split into two steps : *(i)* the local matrix $A_T = (a_T(\psi_i^T, \varphi_j^T))_{i=1\ldots N_X, j=1\ldots N_Y}$ and vector $L_T = (\ell_T(\psi_i^T))_{i=1\ldots N_X}$ are first constructed and *(ii)* the local to global mapping is used to add the contribution of the element T to A and L. This splitting is often used to optimize the local to global mappings [7] or optimize the local matrix and vector computation [17]. We will also follow this strategy in the remaining sections.

2.2 Construction of $a_T(\psi_i^T, \varphi_j^T)$ and $L_T(\psi_i^T)$

We focus now on the construction of the elementary contribution $a_T(\psi_i^T, \varphi_j^T)$ and $\ell_T(\psi_i^T)$ which is the heart of our methodology.

[5] X and Y will also be named as the test space and the trial space respectively.

2.2.1 Basis Functions

We turn to the treatment of the basis functions in our framework, and in particular we describe the computation of $f(\tau(\hat{x}))$ for any $\hat{x} \in \hat{T}$ as in equation (5). We define the finite element basis functions on the reference element. If f belongs to X, then we have for a given $T \in \mathcal{T}$ and its associated geometric mapping τ :

$$f(\tau(\hat{x})) = \sum_{i=1,\ldots,N_X} f_i \ \psi_i(x) \tag{7}$$

$$= \sum_{i=1,\ldots,N_X} f_i \ \hat{\psi}_i(\hat{x}) \tag{8}$$

$$= \underbrace{F^t}_{\text{Expansion coefficients}} \qquad \underbrace{\hat{\psi}(\hat{x})}_{\text{Computation on } \hat{T}} \tag{9}$$

where $F^t = [f_1, \ldots, f_{N_X}]$ and $\hat{\psi}(\hat{x}) = [\hat{\psi}_1(\hat{x}), \ldots, \hat{\psi}_{N_X}(\hat{x})]^t$. The gradient reads

$$\nabla \ f(\tau(\hat{x})) = F^t \nabla \psi(x) \tag{10}$$

$$= F^t \ B_\tau(\hat{x}) \qquad \underbrace{\hat{\nabla} \ \hat{\psi}(\hat{x})}_{\text{Computation on } \hat{T}} \tag{11}$$

Similar computations, albeit more involved, might be derived for the second order derivatives.

The basis function concept we developed is similar to the geometric mapping. In our framework, the degrees of freedom are associated with the elements of the mesh. More precisely they are ordered with respect to the geometric subentities of the elements — vertices, edges, faces and volumes — for global continuous functions to ensure a continuous expansion whereas in the case of global discontinuous functions it does not matter how the degress of freedom are ordered or organized within the element. This allows for flexible construction of polynomial sets such as Lagrange, Raviart-Thomas or modal basis with global continuous expansion or not. The article [22] presents these aspects in details. Essentially the `Basis` base class, see listing 3, provides an interface for obtaining the value of the basis function and its derivatives at any given point in the reference element. Similar to the geometric mapping, we also define a `Context` subclass that provides information on the basis functions at a given set of points $\{\hat{x}; \hat{x} \in \hat{T}\}$.

Listing 3: Basis functions interface

```
class Basis
{
  public:
  // access to precomputed basis functions values and
  // derivatives at a given set of points
  class Context
  {
    public:
    ...
    // value of the i-th basis function at the q-th node
    // in the reference element : φ^i(x̂_q)
    double phi( int q, int i );
    // gradient of the i-th basis function at the q-th node
```

```
    // B_τ^q(x̂_q) ∇̂ ψ^i(x̂_q)
    node_Type const& dphi( int i );
    ...
  }
};
```

Equipped with these tools and concepts and if we consider a function $f \in X$, we have

$$\begin{aligned} \int_T f(x)\,\mathrm{d}x &= \sum_{q=1\ldots Q} \hat{w}_q \sum_{i=1\ldots N_X} f_i\ \hat{\psi}_i(\hat{x}_q)\ J_\tau(\hat{x}_q) \\ \int_T \nabla f(x)\,\mathrm{d}x &= \sum_{q=1\ldots Q} \hat{w}_q \sum_{i=1\ldots N_X} f_i\ B_\tau(\hat{x}_q)\ \hat{\nabla}\hat{\psi}_i(\hat{x}_q)\ J_\tau(\hat{x}_q) \end{aligned} \tag{12}$$

2.2.2 Approximation Space

We define the notion of approximation space in C++ that maps closely the mathematical counterpart. An approximation space is a template class parametrized by a mesh class and the basis functions type — for example the standard Lagrange finite elements. — An approximation space wraps the mesh, the table of degrees of freedom (DoF), the basis function type and provides access to all them. Note that the geometric mapping is provided by the mesh class.

Listing 4: Approximation Space Interface

```
template<typename Mesh, typename Basis_t>
class Space
{
  // get the mesh
  Mesh const& mesh() const;
  // get the dof data structure
  Dof const& dof();
  // get the basis function data structure
  Basis_t const& basis() const;
  // get the geometric mapping
  Mesh::GeoMap const& geomap() const;
};
// ℙ1 finite element space in 2D
Space<Mesh, FEM_PK<2,1,scalar> > Xh( mesh2d );
// ℙ2^3 finite element space in 3D
Space<Mesh, FEM_PK<3,2,vectorial> > Xh( mesh3d );
```

A `Space` defines its own element type as a subclass: it ensures coherence and consistency when manipulating finite element functions. An `Element` derives from your preferred numerical vector type: we use `uBLAS`[6] for our linear algebra data structures and algorithms. The interface is roughly described in the listing 5.

Listing 5: Approximation Space Interface

```
enum ComponentType { X = 0, Y, Z };
template<typename Mesh, typename Basis_type>
class Space
{
```

[6] http://www.boost.org/libs/numeric/ublas/doc/index.htm

```
...
// Element subclass
template<typename T>
class Element : public ublas::vector<T>
{
  public:
  ...
  space_type const& space() const;
  component_space_type const& compSpace() const;

  // get the component of the vector
  component_type comp( ComponentType ) const;
  // interpolation of the function at a given point
  T operator()( node_type const& ) const;

  // Sobolev norms
  double normL2() const;
  double normH1() const;
  ...
};
};
```

An extension of the `Space` concept, is the `MixedSpace` which is a product of two spaces. This can actually be extended to a product of several spaces of different types — implemented using the MPL [2]. — This concept is useful for mixed formulations. `MixedSpace` defines also its own element type with some extra member to retrieve the underlying space elements.

Listing 6: Mixed Space Example

```
// P2 - P1 approximation space in 2D
typedef mpl::vector<FEM_PK<2,2,vectorial>,
                    FEM_PK<2,1,scalar> > SpaceList;
typedef MixedSpace<Mesh, SpaceList > space_t;
space_t Vh( mesh );
space_t::element_type U( Vh );
// FEM_PK<2,2,vectorial>
space_t::element_1_type u = U.element1();
// FEM_PK<2,1,scalar>
space_t::element_2_type u = U.element2();

// extract a view of the X component of u
space_t::element_1_type::component_type ux = u.comp(X);
```

At the moment, `MixedSpace` is concept for a product of two functional spaces. Extending this concept to a product of N functional spaces would be useful.

2.2.3 Linear and Bilinear Forms

One last concept needed to have the language expressive is the notion of forms. They follow closely their mathematical counterparts: they are template classes with arguments being the space or product of spaces they take their input from and the representation we can make out of these forms. In what follows, we consider only the case where the linear and bilinear forms are represented by vectors and matrices respectively. In a future work, we will eventually propose

the possibility to have vector-free and matrix-free representations: that would require to *store* the definition of the forms.

Listing 7 displays the basic interface and usage of the form classes.

Listing 7: Forms

```
Mesh mesh;
// P3(Ω ⊂ R^3)
typedef Space<Mesh,FEM_PK<3,3> > Space_t;
Space_t X_h(mesh);
Space_t::element_type u(X_h), v(X_h);

// Linear forms
template<typename Space, typename Rep>
class LinearForm
{
  LinearForm( Space const&, Rep& rep ) {...}
  LinearForm& operator=( Rep const& ) {...}
  template<typename Expr>
  LinearForm& operator=( Expr const& ) {...}
};


typedef ublas::vector<T> linearform_rep;
Linearform_rep F;
LinearForm<Space_t,linearform_rep> f(X_h,F);
f=integrate(elements(mesh),id(v));

// BiLinear forms
template<typename Space1, typename Space2,
         typename Rep>
class BiLinearForm
{
  BilinearForm( Space1 const&, Space2 const&,
                Rep& rep ) {...}
  BilinearForm& operator=( Rep const& ) {...}
  template<typename Expr>
  BilinearForm& operator=( Expr const& ) {...}
};


typedef ublas::compressed_matrix<T> bilinearform_rep;
Bilinearform_rep M;
BilinearForm<Space_t,Space_t,bilinearform_rep> a(X_h,X_h,
                                                 M);
a=integrate(elements(mesh),idt(u)*id(v));
```

Note that the linear and bilinear form classes are the glue between their representation and the mathematical expression given by `Expr`, it will

- fill the matrix with non-zero entries depending on the approximation space(s) and the mathematical expression;
- allow a per-component/per-space construction(blockwise);
- check that the numerical types of the expression and the representation are consistent

- when `operator=( Expr const& )` is called, the expression is evaluated and will fill the representation entries

The concepts of `MixedLinearForm` and `MixedBilinearForm` that would correspond to mixed linear and bilinear forms respectively — taking their values in the product of two functional spaces — exist also and follow the same ideas.

With the high level concepts described we can now focus on the language.

3 Language

The expression template technique won't be described as it is nowadays a mainstream technique [24, 2, 3, 4, 20]. The construction of the expression template objects in the coming sections is standard.

3.1 Expression Evaluation at a Set of Points in a Convex

Let C be a convex in $\mathbb{R}^d, d \leq 1, 2, 3$ — a n-simplex $n \leq d$ like lines, triangles or tetrahedrons or products of simplices like quadrangles, hexahedrons or prisms, — and $\hat{C}$ be a convex of reference in $\mathbb{R}^d, d \leq 1, 2, 3$ associated to C where we define quadrature points for integration or points to construct polynomials for finite elements and other approximation methods, see [10, 15].

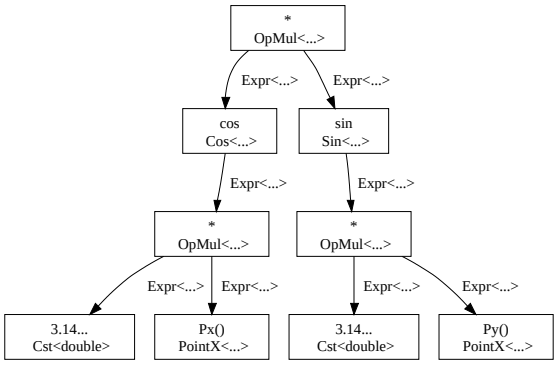


Figure 2: Expression template graph for $f : x \longrightarrow cos(\pi x)sin(\pi y)$

We wish to evaluate $f(x), \forall x \in \tau(\hat{S}_P) = \{x_1, \ldots, x_P\} \subset C$, $\hat{S}_P = \{\hat{x}_1, \ldots, \hat{x}_P\} \subset \hat{C}$, f is a real-value function $C \longrightarrow \mathbb{R}$ and τ is the geometric mapping $\hat{C} \longrightarrow C$, see 2.1.2.

In our code f is represented by an expression template – and not a standard C++ function or a functor, — see [24]. For example, consider $f : x \in C \longrightarrow \cos(\pi x)\sin(\pi y)$, we write it in C++ as `cos(π*Px())*sin(π*Py())`. The expression graph is shown on figure 2. Here `Px()` and `Py()` are free functions that construct objects that are evaluated as the `x` and `y` coordinates of the points $x; x \in C$.

Constructing the C++ object that represents the expression is done with standard expression template approach. However evaluating the expression is problematic as some ingredients are not known yet to the expression object such as the geometric mapping. So using a standard expression template approach certainly allow high level expressivity but cannot be applied to evaluate the expression.

To remedy this issue, we propose a very simple but very powerful solution which delegates the evaluation of the expression to another object than the expression object itself. In our case, the evaluation is delegated to a sublass of each object of the expression.

The `Expression` class, which is the glue between the various object types forming the expression template, is roughly sketched in listing 8.

Listing 8: Evaluation Delegation to Subclass

```
template<typename Expr>
class Expression
{
  typedef Expression<Expr> self_type;
  Expr const& expression() const { return expr; }

  template<typename GeoContext_t>
  class Eval
  {
    typedef Expr::Eval<GeoContext_t> evaluator_type;

    // construct the evaluator for expression Expr.
    Eval( self_type const& _expr,
          GeoContext_t const& gmc )
    : evaluator(_expr.expression,gmc)  {}

    // evaluatate at the q-th node used to build the
    // the finite element
    double operator()( int q ) const { return eval( q ); }
    evaluator_type eval; }; // Eval
  Expr expr; }; // Expression
```

`Expression<Expr>::Eval` is a template class parametrized by the geometric mapping context associated with each geometric element of the mesh. The constructor takes the expression `Expr` and the geometric mapping context as arguments to pass geometric data — coordinates of the current point, normals, measure of the element — down to all objects of the expression so that they can use it as needed. As already mentioned, `Px()` constructs a *C++* class that returns the `x` coordinate of the points where the evaluation is effected. Its implementation is presented in2 listing 9.

Listing 9: Current Evaluation Point Coordinates

```
class PointX
{
  typedef PointX self_type;

  template<typename GeoContext_t>
  class Eval
  {
    // construct the evaluator for expression Expr.
    Eval( self_type const& /*_expr*/,
          GeoContext_t const& gmc )
    : _M_gmc( gmc )  {}

    // returns the x coordinate of the q-th node
    // stored in the geometric mapping context
    double operator()( int q ) const
    { return gmc.xReal(q)[0]; }
    GeoContext_t gmc;
  }; // Eval
```

```
}; // Expression
Expression<PointX> Px()
{ return Expression<PointX>( PointX() ); }
```

`Py()` is implemented in a similar way. Regarding the mathematical functors `cos` and `sin`, they also follow the same idea as shown in listing 10.

Listing 10: Current Evaluation Point Coordinates

```
template<typename Expr>
class Cos
{
  typedef Cos<Expr> self_type;

  template<typename GeoContext_t>
  class Eval
  {
    typedef Expr::Eval<GeoContext_t> eval_expr_type;
    // construct the evaluator for expression Expr.
    Eval( Expr const& _expr,
          GeoContext_t const& _gmc )
    : eval_expr( _expr,gmc )  {}

    // returns the cosinus of the expression at the q-th
    // node stored in the geometric mapping context
    double operator()( int q ) const
    { return cos( eval_expr( q ) ); }
    eval_expr_type eval_expr;
  }; // Eval
}; // Expression
template<typename Expr>
Expression<Cos<Expr> > cos( Expr const& e )
{ return Expression<Cos<Expr> >( Cos<Expr>( e ) ); }
```

3.2 Nodal Projection

We described the mechanism to evaluate an expression at a set of points in a convex, we now turn to nodal projection of a function f onto an approximation space X — for example $X = \{u \in \mathbb{P}_k(\mathcal{T})\}$ where $\mathcal{T}$ is a triangulation of Ω and $\mathbb{P}_k$ is the set spanned by the Lagrange polynomials of degree $\leq k$. — We denote $\pi_X f$ the nodal projection of f onto X.

The nodal projection is an extension of the previous section at a set of convexes and $\hat{S}_P$ being the set of coordinates of the degrees of freedom (DoF) associated with X. The nodal projection is described by algorithm 2.

We define a free function `project(<space>,[elements,]<expression>)` that takes two or three arguments : the approximation space onto which we project the function, the expression representing the function we want to project and optionally a range of elements that restricts the projection to this set of elements, see section 2.1.1. `project()` constructs an template class parametrized by the arguments types passed to `project()`, see listing 11.

Listing 11: Nodal Projection

```
// Space_t type of approximation space
```

Algorithm 2 Nodal projection on X

ι_X is the local/global correspondance table
for $T \subset \mathcal{T}$ **do**
  $\hat{p}_i = 1, \ldots, N_X$ points coordinates associated with the DoF in T
  $c \leftarrow \{T, G, (\hat{p}_i)_{i=1 \ldots N_X}\}$ geometric mapping context, see 2.1.2
  for $i = 1, \ldots, N_X$ points coordinates associated with the DoF **do**
    $c.\hat{x} \leftarrow \hat{p}_i$
    $\pi_X f_{\iota_X(T,i)} = f(c.x)$
  end for
end for

```
// Erange range of iterators over the elements that
//        restricts the projection
// Expr_t expression to project
template<typename Space_t, typename Erange,
         typename Expr_t>
class Projector
{
  Projector( Space_t const& X, Erange const& erange,
             Expr_t const& E );
  Space_t::element_type operator()() const {...};
};
template<typename Space_t, typename Expr_t>
Space_t::element_type
project( Space_t const& X, Expr_t const& E )
{
  // projection of E over all elements of the mesh
  return project( X, elements(X.mesh()), E );
}
template<typename Space_t, typename ERange,
         typename Expr_t>
Space_t::element_type
project( Space_t const& X, ERange const& erange,
         Expr_t const& E )
{
  Projector<Space_t, Expr_t> P(X, erange, E);
  return P();
}
```

Listing 12 shows an example of nodal projection.

Listing 12: Nodal Projection Example

```
Mesh mesh; // mesh of Ω ⊂ ℝ^d
// Space of cubic lagrange element
Space<Mesh,FEM_PK<d,3> > X(mesh);
X::element_type u;
// project cos(πx)sin(πy) on ℙ3(Ω)
u = project( X, cos(π*Px())*sin(π*Py());
```

Other types of projection like L_2 or H_1 projections require other ingredients presented in the coming sections.

3.3 Numerical Integration

We now turn to numerical integration of $\int_\Omega f(x)\mathrm{d}x$ where f is the function to be integration over Ω. Numerical integration requires the evaluation of the function f at quadrature points in the convexes of the mesh associated to Ω. In our code, we used the quadrature constructions presented in [15] for n-simplices and simplices products.

The integration process is described by algorithm 3.3

Algorithm 3 Integration over a mesh $\mathcal{T}$ of a domain $\Omega \subset \mathbb{R}^d$ using a Quadrature Method

$(\hat{w}_q, \hat{x}_q)_{q=1,\dots Q}$ be the set of quadrature nodes and weights
for $T \subset \mathcal{T}$ **do**
  Set $c \leftarrow \{T, G, (\hat{x}_q)_{q=1\dots Q}\}$ geometric mapping context, see section 2.1.2
  $\int_\Omega f(x) += \sum_{q=1,\dots Q} \hat{w}_q f(\tau(\hat{x}_q))$ $\{\tau(\hat{x}_q)$ is given by `c.xReal(q)`$\}$
end for

We introduce a new keyword to reflect the integration action, see listing 13, which is a free function instantiating an integrator parametrized by *(i)* the set of geometric elements, see section 2.1.1, where the integration is done, *(ii)* the expression to integrate and *(iii)* the integration method, see listing 14.

Listing 13: `integrate` prototype

```
integrate( <elements>, <expr>, <integration method> )
```

Listing 14: `Integrator` class and `integrate` free function

```
template<typename EIter, typename Expr_t,typename Im_t>
class Integrator
{
  Integrator( Elist const& elist, Expr_t const& E,
              Im_t const& im );
  double operator()() const {...};
};
template<typename EIter, typename Expr_t,typename Im_t>
double
integrate( EIter const& eiter,
           Expr_t const& e,
           Im_t const & im )
{
  Integrator<EIter, Expr_t, Im_t> I(eiter, e, im );
  return I();
}
```

Listing 15 shows an example of the syntactic sugar brought by the language.

Listing 15: Integration Syntax

```
// mesh of [0,1]^d, d = 1,2,3 made of simplices
// of dimension d
Mesh mesh;
// Gauss Legendre integration of Order 10
// over simplex of dimension d
IM_PK<d,10> im;
```

```
// ∫_[0,1]^d cos(‖x‖_2) where ||·||_2 is the Euclidean norm
// elements(mesh) provides the list of all elements
// in the mesh data structure
v=integrate( elements(mesh), cos(norm2(P())), im );
// ∫_∂[0,1]^d cos(‖x‖_2)
// bdyfaces(mesh) provides the list of the element faces
// on the boundary of the mesh
v=integrate( bdyfaces(mesh), cos(norm2(P())), im );
```

3.4 Variational Formulations

The framework, presented in the last sections, can be extended to handle variational formulation with only minor changes to the evaluation class. We consider for now a convex $C \subset \mathbb{R}^d, d = 1, 2, 3$ and its associated reference convex $\hat{C}$, an approximation space X — for example $\mathbb{P}_k(C)$ — a bilinear form a $X \times X \to \mathbb{R}$ defined by

$$a(u, v) = \int_C u\ v \quad \forall\ u,\ v \in X \tag{13}$$

Listing 16 shows the C++ counterpart of equation (13).

Listing 16: Variational Integration

```
// integrate over element with id 1
integrate( idedelement(mesh,1), idt(u)*id(v) );
```

The `t` in `idt(.)` allows to distinguish trial and test functions: for example, `id(.)` identifies the test function values whereas `idt(.)` identifies the trial function values.

Given $u, v \in X$, we wish to compute the value $a(u, v)$ which can be approximated as follows

$$a(u, v) \approx \sum_{i=1\ldots N_X} \sum_{j=1\ldots N_X} u_i\ v_j \sum_{q=1\ldots Q} \hat{w}_q\ \hat{\psi}_i(\hat{x}_q)\ \hat{\psi}_j(\hat{x}_q) J(\hat{x}_q) \tag{14}$$

where $(\hat{w}_q, \hat{x}_q)_{q=1\ldots Q}$ are the quadrature nodes and weights defined in $\hat{C}$, $J(\hat{x}_q)$ is the jacobian of the geometric transformation between $\hat{C}$ and C at the point $\hat{x}_q$ and $(\hat{\psi}_i)_{i=1\ldots N_X}$ is the basis of X.

Recall section 2.2.1, we have the basis context subclass that allows to store values and derivatives of basis functions at a set of points. In the case of equation (14), the basis context subclass stores and provides an interface to $(\hat{\psi}_i(\hat{x}_q))_{q=1\ldots Q, i=1\ldots N_X}$.

Again the basis functions context is not known to the expression object. In order to accommodate the language with these concepts, it suffices to add new template parameters to the evaluation subclass of each classes allowed in an expression. However these parameters have default values that allows to handle the case of the previous section as well as linear forms and bilinear forms, see listing 17. In particular test basis functions context type are defaulted to `boost::none_t`[7] and trial basis functions context type to the test ones. If `boost::none_t` is used, then no language keyword may be used in the expression that will need basis functions operators such as `id(.)` or `idt(.)`.

[7] See `boost/none_t.hpp`

Listing 17: Evaluation subclass modifications for Variational Formulations

```
template<typename Expr>
class Expression
{
  typedef Expression<Expr> self_type;
  Expr const& expression() const { return expr; }

  template<typename GeoContext_t,
           typename Basis_test_t = boost::none_t,
           typename Basis_trial_t = Basis_test_t>
  class Eval
  {
    typedef Expr::Eval<GeoContext_t> evaluator_type;


    // construct the evaluator for expression Expr.
    Eval( self_type const& _expr,
          GeoContext_t const& gmc,
          Basis_trial_t const& u
          Basis_test_t const& v )
    : eval(_expr.expression,gmc,u,v)  {}

    // construct the evaluator for expression Expr.
    Eval( self_type const& _expr,
          GeoContext_t const& gmc,
          Basis_test_t const& v )
    : eval(_expr.expression,gmc,v)  {}

    ...

    // evaluatate at the ith-basis function and
    // j-th basis function at the q-th node used
    // to build the finite element
    double operator()( int q, int i, int j ) const
    { return eval( q, i, j ); }

    // evaluatate at the ith-basis function and
    // j-th basis function at the q-th node used
    // to build the finite element
    double operator()( int q, int i, int j ) const
    { return eval( q, i ); }

    ...

    evaluator_type eval; }; // Eval
  Expr expr; }; // Expression
```

Let's examine for example the implementation of the operator `dx(<element:u>)` which provides the first component of the first derivative of the basis function associated with `u`, see listing 18.

Listing 18: Operator `dx(.)`

```
template<typename Element>
class OpDx
```

```
{
  typedef Element element_type;
  typedef Element::basis_type basis_type;

  template<typename GeoContext_t,
           typename Basis_test_t,
           typename Basis_trial_t = Basis_test_t>
  class Eval
  {
    typedef Expr::Eval<GeoContext_t> evaluator_type;

    // construct the evaluator for expression Expr.
    Eval( self_type const& _expr,
          GeoContext_t const& gmc,
          Basis_trial_t const& u
          Basis_test_t const& v )
    : test_basis(v)  {}
    // construct the evaluator for expression Expr.
    Eval( self_type const& _expr,
          GeoContext_t const& gmc,
          Basis_test_t const& v )
    : test_basis(v)  {}
    ...
    // if the data type for test basis functions
    // are the same then
    //  return the first component of the first
    //  derivative
    // otherwise return 0
    double operator()( int q, int i, int j ) const
    {
      // the if disappears at compile time since
      // the condition is known
      if ( boost::is_same<basis_type,
                          Basis_test_t>::value )
        return test_basis.dphi(q,j)[0];
      else
        return 0;
    }
    double operator()( int q, int i ) const
    { same as above }
    ...
    Basis_test_t test_basis; }; // Eval
  Expr expr; }; // Expression
```

The types of the test and trial basis functions, `Basis_test_t` and `Basis_trial_t` are tested with respect to the basis function type `basis_type` of the element passed to the operator. If the types are not the same then at *compile time* the evaluation of this operator returns 0. At first glance for standard scalar equations — heat equation say — it allows just to ensure that the evaluation makes sense, however when dealing with mixed formulation — e.g. Stokes equations — this feature becomes very important if not crucial for correctness and performance-wise. Indeed it will disable automatically the terms associated with the basis functions which are not evaluated during local assembly of the mixed formulation : for example when filling the block corresponding to the

velocity space, all the others terms — velocity-pressure and pressure terms — are evaluated to 0.

An immediate remark is that we may have lots of computation for nothing, i.e. computing 0. However a simple check with `g++ -O2 --save-temps` shows that `g++` optimizes out expressions containing 0 *at compile time* and removes the corresponding code. The *compile time check* of the basis functions types is then crucial to get the previous `g++` optimization[8].

As mentionned in the introduction, DSEL inherits the capabilities of the underlying language; we have shown here a very good illustration of this statement.

Now that we know how to do integrate a variational form on a convex, it is easy to extend it to a set of convexes.

3.5 Automatic Differentiation

Automatic differentiation as described in [4, 3] operates differently from the previous algorithms when evaluating an expression template. We are no longer evaluating an expression at a set of points but rather dealing with expressions that manipulate a differentiation numerical type `ADType<P,N>` — P is the number of parameters and N the order of differentiation — holding a value and corresponding derivatives — in pratice up to order $N = 2$ and $P \leq 100$ —

To implement the automatic differentiation engine, we construct the expression object as before. However the evaluation is changed: in our case we create a new subclass for evaluating the differentiation. Each class which could be possibly used for automatic differentiation need to implement this subclass. Listing 19 shows an excerpt of the implementation of the expression templates glue class `Expression`.

Listing 19: Automatic differentiation subclass

```
template<typename Expr>
class Expression
{
  typedef Expression<Expr> self_type;
  Expr const& expression() const { return expr; }

  // evaluation of expression at a set of point
  template<...> class Eval {};

  // differentiate expression
  class Diff
  {
    typedef Expr::Diff diff_type;

    // construct the differentiator for expression Expr.
    Diff( self_type const& _expr )
    : diff(_expr.expression())  {}

    double value() const
```

[8] This `g++` optimization is a combination of at least two optimization strategies: *(i)* constant folding and *(ii)* algebraic simplifications & Reassociation, see http://www.redhat.com/software/gnupro/technical/gnupro_gcc.html for more details

```
    { return diff.value(); }

    double grad( int __ith ) const
    { return diff.grad( __ith ); }

    double hessian( int __i, int __j ) const
    { return diff.hessian( __i, __j ); }

    diff_type diff; }; // Diff
  Expr expr; }; // Expression
```

The automatic differentiation evaluation engine does not need to be much further discussed as it follows ideas already published elsewhere, see [4]. We just demonstrate here that decoupling expression construction from its evaluation allows to share the code constructing the expression template object.

A Remark on mixing automatic differentiation and variational formulations We could actually push the envelop quite a lot and use the automatic differentiation type within a variational formulation to differentiate with respect to some parameters for sensitivity analysis, optimization or control of engineering components. That is to say, mix the two evaluation engines : first the integral expression is evaluated and during the integral evaluation, the automatic differentiation language takes over to finalize it. The enabler of the second stage in the evaluation would be the automatic differentation data type and its `operator=( Expr const& )`. This is truly what meta-programming is about — code that generates code that generates code... — However this has not been tested in the examples shown in section 4 and in particular performance could be a concern depending on the quality of the generated code. This is one of the topics for future research as it may open important areas of applications for the language such as the ones mentionned previously — sensitivity analysis, optimization and control.

This leads to another advantage of sharing the expression object construction: it ensures that we have the same set of supported mathematical functions or functors for automatic differentation on the one hand and projection, integration on the other hand which means less development work and less bugs.

3.6 Overview of the Language

The implementation of the language itself, the construction of the expression object is done in a very standard way conceptually, see [24]. However from a technical point, it uses state of the art tools such as the Boost.Mpl [2] and Boost.Preprocessor [2, 16].

In particular the Boost.preprocessor library is extremely useful when it comes to generate the objects and functions of the language from a variety of numerical types and operators — using for example the macro

`BOOST_PP_LIST_FOR_EACH_PRODUCT`.

Regarding the grammar of the language, it follows the C++ one for a specific set of keywords wich are displayed in tables 1, 2 and 3. The keywords choice follows closely the Freefem++ language, see [12].

`H()`	diameter of the current element
`Hface()`	diameter of the current face
`Emarker()`	current element marker
`Eid()`	current element id
`N()`	unit outward normal at the current node of the current face
`Nx(),Ny(),Nz()`	x, y and z component of the unit outward normal
`P()`	coordinates of the current node
`Px(),Py(),Pz()`	x, y and z component of current node

Table 1: Tables of geometric/element operators available at the current element, face and node

`*,+,/,-,!,||,&&,...`	standard unary and binary operations
`cos(<expr>)`	trigonometric/hyperbolic functions
`sin(<expr>)`	
`...`	
`exp(<expr>)`	exponential function
`log(<expr>)`	logarithmic function
`abs(<expr>)`	absolute value of expression
`floor(<expr>)`	floor of expression
`ceil(<expr>)`	ceil of expression
`chi(<expr>)`	Heaviside step function
`min(<expr>,<expr>)`	min/max
`max(<expr>,<expr>)`	
`pow(<expr>,<expr>)`	expression to the power
`<expr>^(<expr>)`	
`dot(<expr>,<expr>)`	dot product of two vectorial expressions
`jump(<expr>)`	jump of an expression across a face of an element
`avg(<expr>)`	average of an expression across a face of an element

Table 2: Tables of mathematical functions that can be applied to expressions

`id(<element:u>)`	basis functions associated to `u`
`grad(<element:u>)`	gradient of the basis functions associated to `u`
`dx(<element:u>)` `dy(<element:u>)` `dz(<element:u>)`	x,y and z components of the gradient of the basis functions associated to `u`
`dxx(<element:u>)` `dyy(<element:u>)` ...	components of the hessian of the basis functions associated to `u`
`grad(<element:u>)`	gradient operators of basis functions associated to `u`
`div(<element:u>)`	divergence operator of basis functions associated to `u`
`idt(<element:u>)...`	the previous operators with a suffix `t` to specify the trial basis functions

Table 3: Tables of Operators for variational formulations

A note on the supported numerical types The language supports the standard ones available in C++— boolean, integral and floating-point types — and some additional ones

- `std::complex<>` complex data type
- `ADtype<.,.>` the automatic differentiation type
- `dd_real` double-double precision data type from the QD library, see [13]
- `qd_real` quad-double precision data type from the QD library
- `mp_real` arbitrary precision data type from the ARPREC library, see [6]

Other data types can be relatively easily supported thanks to Boost.Preprocessor. There are plans for example to support an interval arithmetic data type — the one provided by Boost.Interval, — which would be interesting to measure, for example, the impact of small perturbations or uncertainty on the results.

4 Test Cases

We present now various examples to illustrate the techniques developed previously. We shall show only the relevant part of each examples. Also, for each example, we shall present different possible formulations.

4.1 Performance

In the following section, various results are presented with some timings for the construction of bilinear and linear forms. The calculations have been conducted on an AMD64 opteron [9] running linux `2.6.9` and GNU/Debian/Linux.

The `g++-4.0.1` compiler was used to compile the code with the following options

[9] AMD Opteron(tm) Processor 248, 2.2Ghz, 1MB cache, 8GB ram

Listing 20: g++ options for optimization

```
# standard options
-march=opteron -fast-math -O2
-funroll-loops
# aliasing
-fstrict-aliasing -fargument-noalias-global
```

These options are relatively standard and give good results all the time. Other optimization options have been tried but did not yield significant performance improvements.

We consider various Advection-Diffusion-Reaction problems to evaluate the performance of our framework proposed in [20].

$$\mu\Delta u = 0 \quad D, \tag{15}$$

$$\mu\Delta u + \sigma u = 0 \quad DR, \tag{16}$$

$$\mu\Delta u + \beta\Delta\nabla u + \sigma u = 0 \quad DAR \tag{17}$$

on a unit square and cube domain.

For each problem we consider both the case of constant and space-dependent coefficients. The following expressions were assumed for the space-dependent coefficients

$$\mu(x,y,z) = \begin{cases} x^3+y^2 & \textbf{(2D)} \\ x^3+y^2z & \textbf{(3D)} \end{cases} \tag{18}$$

$$\beta(x,y,z) = \begin{cases} (x^3+y^2z, x^3+y^2) & \textbf{(2D)} \\ (x^3+y^2z, x^3+y^2, x^3) & \textbf{(3D)} \end{cases} \tag{19}$$

$$\sigma(x,y,z) = \begin{cases} x^3+y^2 & \textbf{(2D)} \\ x^3+y^2z & \textbf{(3D)} \end{cases} \tag{20}$$

The corresponding C++ code for the 3D is presented in listing 21. The C++ for the 2D cases is very similar but simpler.

Listing 21: Benchmark for elliptic problems

```
#define gradugradv(u,v) (dxt(u)*dx(v) + dyt(u)*dy(v) + \
                         dzt(u)*dz(v))
#define agradu(a,b,c,u) ((a)*dxt(u) + (b)*dyt(u) + \
                         (c)*dzt(u))

// D
D = integrate(elements(mesh), gradugradv( u,v ) );
D = integrate(elements(mesh), (Px()^(3)+Py()^(2)*Pz())*
                               gradugradv(u,v));

// DR
DR = integrate( elements(mesh), gradugradv(u,v )  +
                                 idt(u)*id(v));
DR = integrate( elements(mesh), (Px()^(3)+Py()^(2)*Pz())*
                (gradugradv(u,v) + idt(u)*id(v)));
```

```
// DAR
DAR = integrate( elements(mesh), gradugradv(u,v) +
                 idt(u)*id(v) + agradu(1,1,1,u)*id(v));
DAR = integrate( elements(mesh),
               (Px()^(3)+Py()^(2)*Pz())*(gradugradv(u,v) +
               idt(u)*id(v))+
               agradu((Px()^(3)+Py()^(2)*Pz()),
                      (Px()^(3)+Py()^(2)),
                      (Px()^(3)),u) * id(v) );
```

4.1.1 Results

The benchmark table, see 4, reports the timings for filling the matrix entries associated with the problems D, DR and DAR. Lagrange element of order 1 — 3 dof in 2D, 4 in 3D — and order 2 — 6 dof in 2D, 10 dof in 3D — have been tested. The integration rule changes depending on the polynomial order we wish to integrate exactly. For $\mathbb{P}1$ Lagrange elements 3 quadrature points are used in 2D, 4 in 3D and for $\mathbb{P}2$ Lagrange elements 4 are used in 2D and 15 in 3D.

Dim	NEls	$\mathbb{P}$k	NDoF	D		DR		DAR	
				const	xyz	const	xyz	const	xyz
2D	20742	$\mathbb{P}1$	10570	0.08	0.13	0.08	0.13	0.13	0.17
		$\mathbb{P}2$	41881	0.2	0.23	0.2	0.23	0.25	0.29
	82460	$\mathbb{P}1$	41 629	0.36	0.48	0.37	0.48	0.51	0.66
		$\mathbb{P}2$	165717	0.81	0.99	0.83	1.02	1.07	1.42
	330102	$\mathbb{P}1$	165850	1.47	2	1.5	2	2.06	2.6
		$\mathbb{P}2$	661801	3.38	4.06	3.42	4.22	4.22	5.45
	517454	$\mathbb{P}1$	259726	2.44	3.25	2.48	3.24	3.29	4.28
		$\mathbb{P}2$	1036905	5.5	6.57	5.86	7.17	6.98	8.78
3D	8701	$\mathbb{P}1$	1723	0.06	0.07	0.06	0.07	0.08	0.11
		$\mathbb{P}2$	12825	0.42	0.46	0.43	0.48	0.49	0.58
	69602	$\mathbb{P}1$	12239	0.49	0.73	0.5	0.75	0.69	1.09
		$\mathbb{P}2$	96582	3.47	3.77	3.6	4.22	4.3	4.9
	554701	$\mathbb{P}1$	92071	4.05	5.88	4.55	6.36	5.96	8.55
		$\mathbb{P}2$	748575	31.27	34.42	32.35	36.16	35.82	42.08
	1085910	$\mathbb{P}1$	178090	8.59	12.32	8.96	12.62	11.74	17.54

Table 4: Language Performances in $[0,1]^2$ and $[0,1]^3$. Timings are in seconds.

4.1.2 Analysis

In order to facilitate the study of these timing results, they are displayed on figures 3 and 4 for the 2D cases and 3D cases respectively with each time the matrix assembly time with respect to the number of elements along with the ratio between the `cst` timings and `xyz` timings for $\mathbb{P}1$ and $\mathbb{P}2$ cases. We can observe a few things:

- matrix assembly time versus number of elements is linear (in $\log - \log$) which is no surprise,
- the difference of performances between the different equations D, DR and DAR are quite small even with the DAR and the 3D cases which means that we do a good job at sharing as much computations as possible between the different terms of the equation,
- the overhead due to non-constant coefficients is very small in all cases. which is not the case for example in [20]. In particular, as shown by the ratio figures, the ratios are always less than 2 and in most cases — among them the most expensive ones — they are in $[1.1; 1.3]$. In [20] they used functions or functors to treat the non constant coefficients, they get factors between 2 and 5 (5 in the worst case 3D $\mathbb{P}_2$). This means that the expression templates mechanism and the `g++` optimizer do a very good job at optimizing out the expression evaluation.

These results illustrate very well the pertinence of using meta-programming in general and expression templates in particular in demanding scientific computing codes. Also note that there was no particular optimization based on some knowledge of the underlying equation terms with respect to the local matrix assembly, so there is room for improvements — for example the symmetry of the bilinear form or the precomputation of stiffness, mass and convection matrices on the reference element, see [17].

Regarding compilation time, it is certainly true that using expressions template and meta-programming has an impact on the compile time. However if most of the library is templatized then the compile time cost is usually paid when compiling the end-user application and no more when compiling the library itself. To give an idea, the performance benchmark code was compiled with all cases : $\mathbb{P}1$ and $\mathbb{P}2$ in 2D and 3D for all problems D, DR and DAR. In other words, a single executable was generated to run all the cases presented earlier. On an AMD64 opteron [10], the compilation takes between 2 and 3 minutes using the compiler options from listing 20. The book by David Abrahams and Aleksey Gurtovoy [2] provides a complete discussion on meta-programming and its impact on compile time.

4.2 A Variationnal Inequality

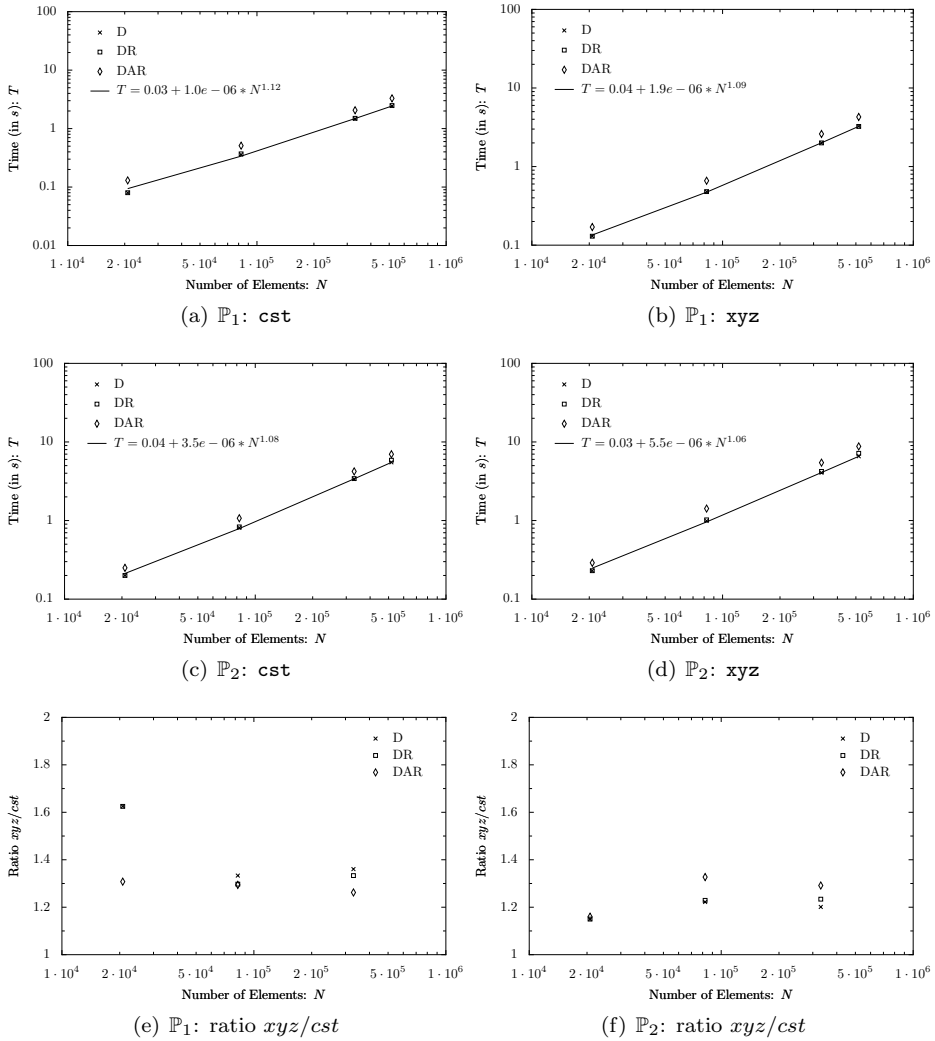
This test case is described in [9]. We consider a rectangular tank of length $a = 0.1$m, and height $b = 0.05$m. A cylindrical tube crosses it with a diameter of 0.015m. The tank is filled with ice and there is a thin layer of solid/liquid polymer on top of it. For symmetry reason, we consider only half of the tank — $a = 0.05$m. — The problem is formulated as follows: The temperature θ is solution of the parabolic equation

$$\int_\Omega \beta(\theta) \frac{\partial \theta}{\partial t} v - \nabla \cdot (\mu(\theta) \nabla \theta) = 0 \tag{21}$$

where

$$\mu(\theta) = \chi_{y<b-3p}[\mu_1 \chi_{\theta>\epsilon} + \mu_2 \chi_{\theta<0}] + \mu_3 \chi_{y>b-3p}, \tag{22}$$

[10] AMD Opteron(tm) Processor 248, 2.2Ghz, 1MB cache, 8GB ram

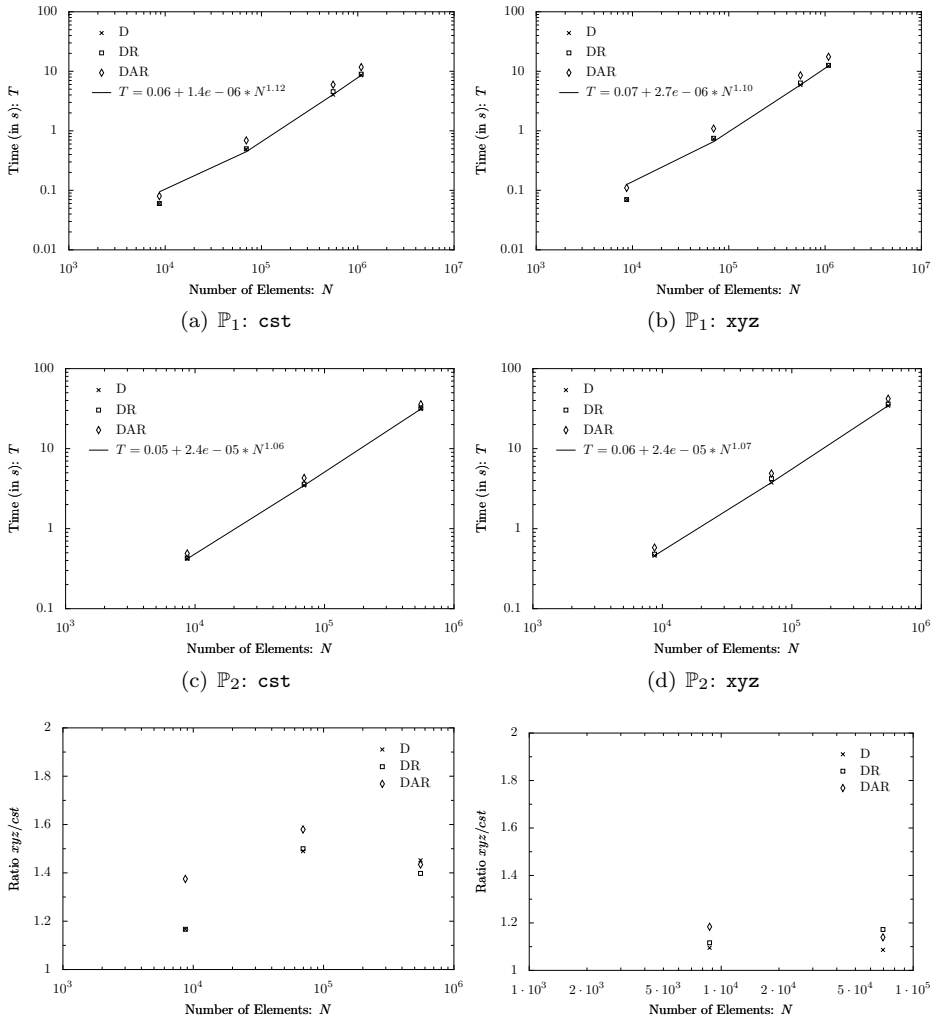


(a) $\mathbb{P}_1$: `cst` (b) $\mathbb{P}_1$: `xyz`

(c) $\mathbb{P}_2$: `cst` (d) $\mathbb{P}_2$: `xyz`

(e) $\mathbb{P}_1$: ratio xyz/cst (f) $\mathbb{P}_2$: ratio xyz/cst

Figure 3: Cases $\mathbb{P}_1$ and $\mathbb{P}_2$ in 2D. Matrix assembly time versus number of elements and ratios between non-constant coefficients assembly and constant coefficients assembly



(a) $\mathbb{P}_1$: `cst` (b) $\mathbb{P}_1$: `xyz`

(c) $\mathbb{P}_2$: `cst` (d) $\mathbb{P}_2$: `xyz`

(e) $\mathbb{P}_1$: ratio xyz/cst (f) $\mathbb{P}_2$: ratio xyz/cst

Figure 4: Cases $\mathbb{P}_1$ and $\mathbb{P}_2$ in 3D. Matrix assembly time versus number of elements and ratios between non-constant coefficients assembly and constant coefficients assembly

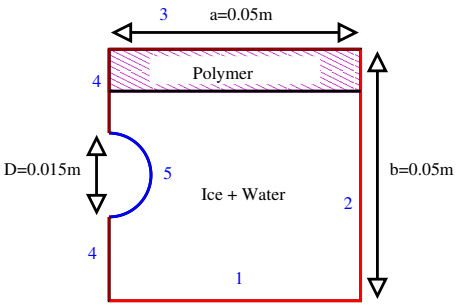


Figure 5: Variational Inequality

$$\beta(\theta) = \chi_{y<b-3p}[c_1\chi_{\theta>\epsilon} + c_2\chi_{\theta<0} + \frac{1}{\epsilon}L_1\chi_{0<\theta<\epsilon}] + \chi_{y>b-3p}[c_3\chi_{\theta>\epsilon} + c_4(\chi_{\theta>\epsilon} + \chi_{\theta<0}) + \frac{1}{\epsilon}L_2\chi_{0<\theta<\epsilon}], \quad (23)$$

and $\epsilon = 0.05$. $\beta(\theta)$ is the volumetric heat capacity and $\mu(\theta)$ is the thermal conductivity. Finally, χ represent the characteristic function.

We impose the following conditions:

- $\theta = 0$ on boundaries 1 and 2
- $\frac{\partial\theta}{\partial n} = 0$ on boundary 3 and 4
- $\theta = -0.1 + 0.05t$ on boundary 5 where t represents the time

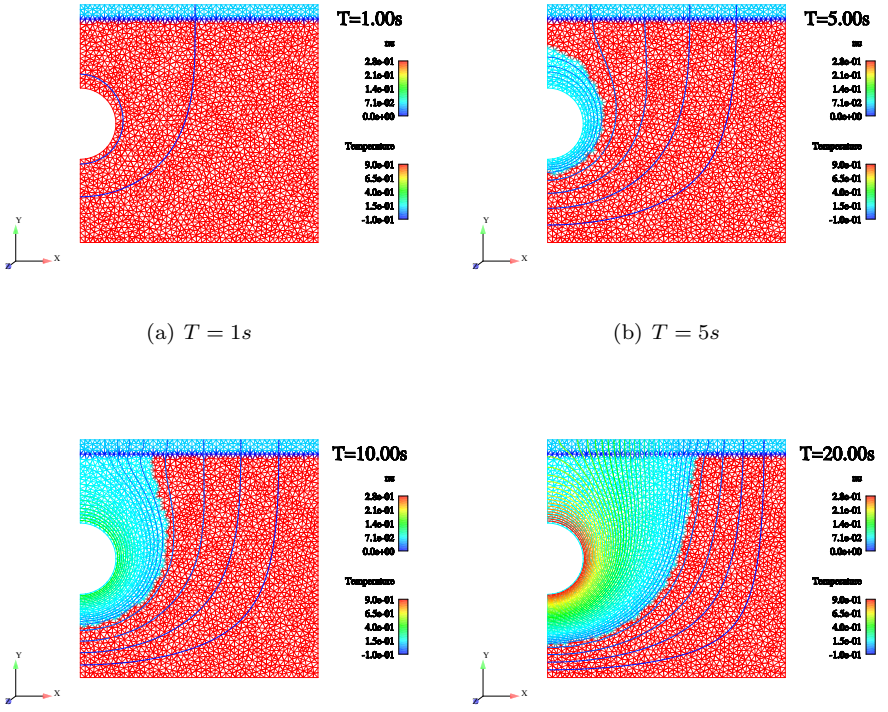
The actual code reads as follows :

Listing 22: Variational Inequality

```
Mesh mesh;
fespace_type P1;
P1::element_type u,v;
LinearForm<fespace_type> f( P1, F );

f = integrate( elements(mesh),
// term: β(θ) θ v
          id(P1,θ)*id(v)*
// χ_{y<b-3p}[c1 χ_{θ>ε} + c2 χ_{θ<0} + (L1/ε) χ_{0<θ<ε}]
           (chi(Py()<b-3*p)*
          (c1*chi(id(P1,θ)>epsilon)+
           c2*chi(id(P1,θ)<0)+
           (L1/epsilon)*(chi(id(P1,θ)>0)*
                         chi(id(P1,θ)<epsilon)))+
// χ_{y>b-3p}[c3 χ_{θ>ε} + c4(χ_{θ>ε} + χ_{θ<0}) + (L2/ε) χ_{0<θ<ε}]
           chi(Py()>=b-3*p)*(c3*chi(id(P1,θ)>0.5+epsilon)+
           c4*chi(id(P1,θ)<0.5)+
          (L2/epsilon)*chi(id(P1,θ)>0.5)*
                       chi(id(P1,θ)<0.5+epsilon))));

BilinearForm<fespace_type> a( u, v, A );
a = integrate( elements(mesh),
// term: β(θ) θ v
          id(θ)*id(v)*
// χ_{y<b-3p}[c1 χ_{θ>ε} + c2 χ_{θ<0} + (L1/ε) χ_{0<θ<ε}]
           (chi(Py()<b-3*p)*
          (c1*chi(id(P1,θ)>epsilon)+
           c2*chi(id(P1,θ)<0)+
           (L1/epsilon)*(chi(id(P1,θ)>0)*
                         chi(id(P1,θ)<epsilon)))+
// χ_{y>b-3p}[c3 χ_{θ>ε} + c4(χ_{θ>ε} + χ_{θ<0}) + (L2/ε) χ_{0<θ<ε}]
           chi(Py()>=b-3*p)*(c3*chi(id(P1,θ)>0.5+epsilon)+
```

```
           c4*chi(id(P1,θ)<0.5)+
          (L2/epsilon)*chi(id(P1,θ)>0.5)*
                       chi(id(P1,θ)<0.5+epsilon)))+

// term: dt μ(θ) ∇ θ · ∇ v
          dt*dot(grad(θ),grad(v))*(
// χ_{y<b−3p}[μ1 χ_{θ>ε} + μ2 χ_{θ<0}] + μ3 χ_{y>b−3p}
           (chi(Py()<b-3*p)*(\mu_1*chi(id(P1,θ)>epsilon)+
           mu2*chi(id(P1,θ)<0))+
           mu3*chi(Py()>b-3*p)));
// boundary conditions
a += on(1,θ,F,-0.1)+
     on(2,θ,F,-0.1)+
     on(5,θ,F,-0.1+0.05*t);
```

Figure 6 shows the contour lines of the temperature at various time steps and the mesh colored by the thermal conductivity over the entire domain: we see that the ice is melting and transformed into water — in red the ice and light blue the water.



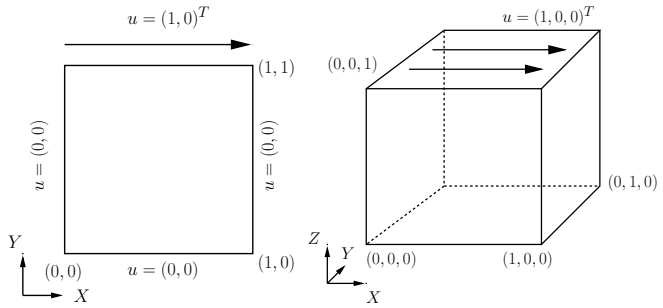
(a) $T = 1s$ (b) $T = 5s$

(c) $T = 10s$ (d) $T = 20s$

Figure 6: Temperature and thermal conductivity in the tank at various time steps

4.3 Stokes and Navier-Stokes

We consider here the standard driven cavity test case with the following setting described by the figure 7.



(a) 2D (b) 3D

Figure 7: Driven Cavity

First we use a stable mixed approximation for the velocity and pressure spaces — say the Taylor-Hood element $(\mathbb{P}_2 - \mathbb{P}_1)$. — The variational formulation reads as follows:
Find $(u, p) \in X_h \times M_h$ such that for all $(v, q) \in X_h \times M_h$

$$\int_\Omega \nabla u \cdot \nabla v - \int_\Omega \nabla \cdot v \; p = 0 \tag{24}$$

$$\int_\Omega \nabla \cdot u \; q = 0 \tag{25}$$

$$u_{|Y=1} = (1,0)^T \quad \text{in 2D} \tag{26}$$

$$u_{|Z=1} = (1,0,0)^T \quad \text{in 3D} \tag{27}$$

Listing 23: Mixed Variational Formulation

```
// mixed finite element space (P2 velocity, P1 pressure) in 3D
typedef MixedSpace<Mesh_t,
                   FEM_PK<3,2,vectorial>,
                   FEM_PK<3,1,scalar> > space_type;
space_type V_h;
V_h U,V;
// views for U and V
typedef space_type::space_1_type X_h;
typedef space_type::element_2_type M_h;
space_type::element_1_type u = U.element1();
space_type::element_2_type p = U.element2();
space_type::element_1_type v = V.element1();
space_type::element_2_type q = V.element2();
//
csr_matrix_type A;
MixedBilinearForm<space_type> a( V_h, V_h, A );
// block wise construction
a( u, v ) = integrate( elements(mesh), dot(grad(u), grad(v)) );
```

```
a( p, v ) = integrate( elements(mesh), -id(p)*div(v) );
a( u, q ) = integrate( elements(mesh), id(q)*div(u) );
a( p, q ) = integrate( elements(mesh), 1e-6*id(p)*id(q) )+
            // 10 identifies the lid
            on( 10, u, F, oneX() )+
            // 20 = ∂Ω\10
            on( 20, u, F, 0 );
// or could be done this way. less expensive:
// only one loop over the elements
a = integrate( elements(mesh),
               dot(gradt(u), grad(v)) - idt(p)*div(v) +
               id(q)*divt(u) + 1e-6*idt(p)*id(q) )+
    // 10 identifies the lid
    on( 10, u, F, oneX() )+
    // 20 = ∂Ω\10
    on( 20, u, F, 0 );
```

We can also use an equal order approximation — say $\mathbb{P}1 - \mathbb{P}1$ — and add a stabilization term like the jump of the pressure gradient over the internal faces as proposed in [8]. The formulation reads then as follows: Find $(u, p) \in X_h \times M_h$ such that $\forall\, (v, q) \in X_h \times M_h$

$$\int_\Omega \nabla u \cdot \nabla v - \int_\Omega \nabla \cdot v \; p = 0 \tag{28}$$

$$\int_\Omega \nabla \cdot uq + \sum_{\mathcal{F} \in \Omega_h} \int_{\mathcal{F}} [\frac{\partial p}{\partial n}][\frac{\partial q}{\partial n}] = 0 \tag{29}$$

$$u_{|Y=1} = (1,0)^T \quad \text{in 2D} \tag{30}$$

$$u_{|Z=1} = (1,0,0)^T \quad \text{in 3D} \tag{31}$$

where $[\cdot]$ denotes the jump of the quantity across a face. Codewise we operate just a slight modification of listing 23, see the listing 24.

Listing 24: Mixed Variational Formulation

```
// can be of equal order for velocity and pressure spaces
MixedBilinearForm<space_type> a( V_h, V_h, A, false );
a = integrate( elements(mesh),
               dot(gradt(u), grad(v)) - idt(p)*div(v) +
               id(q)*divt(u) + 1e-6*idt(p)*id(q) )+
  // add the jump of the normal derivatives of the pressure
  // where Γ_p is a constant. One can take Γ_p = 2.5e^-2
  integrate(internalfaces(mesh),Γ_p*(Hface()^3)*jump(dnt(p))*
                                jump(dn(q)) )+
  // 10 identifies the lid
  on( 10, u, F, oneX() )+
  // 20 = ∂Ω\10
  on( 20, u, F, 0 );
```

4.4 Particule in a Shear Flow

We consider now a rigid particule in a shear Stokes flow treated using a penalization method, see [14].

Denote $\Omega = [-1,1]^2$, we consider a cirular particule positionned at $(x_p = 0, y_p = 0) \in \Omega$ of radius $r_p = 0.1$ and a penalization parameter ϵ. Denote χ_p the characteristic function of the particule defined as $\chi_p = \chi((x-xp)^2+(y-yp)^2 > r_p)$. We seek $(u,p) \in X_h \times M_h$ such that $\forall (v,q) \in X_h \times M_h$

$$\int_\Omega (1+\chi_p/\epsilon)(\nabla u + \nabla u^T)(\nabla v + \nabla v^T) - \int_\Omega \nabla \cdot v\, p = 0 \tag{32}$$

$$\int_\Omega \nabla \cdot u\, q = 0 \tag{33}$$

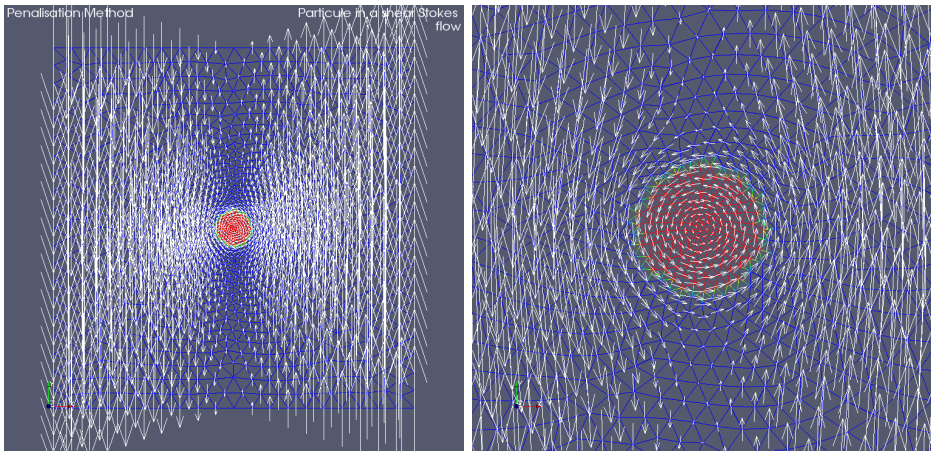
$$u_{|\partial\Omega} = (0, x) \tag{34}$$

where X and M are sub-spaces of $H_1(\Omega)$ and $L_2(\Omega)$ respectively. Listing 25 shows the corresponding C++ code.

Listing 25: Particular in a shear Stokes flow

```
// (xp,yp) are the coordinates of the center of the particule
// rp is the radius of the particule
//  ∫_[-1,1]^2 (1+χ_{(x-xp)^2+(y-yp)^2>r}/ε)(∇u+∇u^T)(∇v+∇v^T)
MixedBilinearForm<Space_t> a(P2P1,P2P1,A);
a = integrate( elements(mesh),
               (1+chi(rp^2>(Px()-xp)^2+(Py()-yp)^2)/eps)*
               dot(gradt(u)+gradTt(v),grad(v)+gradT(v))-
               idt(p)*div(v) + idt(q)*divt(u),
               IM_PK<2,2>() )+
// Dirichlet boundary conditions for the shear flow
    on( 7, uy, F, Px() )+
    on( 7, ux, F, 0 );
```

Figure 8 shows the velocity vector field and identifies the particule in the mesh using its characteristic function χ_p.



(a) Velocity vector field (b) Close-up

Figure 8: Particule in shear Stokes flow

Conclusion

We have developed a unified DSEL for different aspects of numerical analysis, namely differentiation, integration, projection and variational formulations. The main results of this article are that *(i)* such a DSEL is feasible: an implementation has been done and exercised with some non-trivial test cases and *(ii)* decoupling the expression object construction from its evaluation using a delegate subclass allows for very powerful notations in C++: one unique engine is used for the expression construction and as many engines as needed for the expression evaluation — we have seen two different engines: one for projection, integration and variational formulations and one for automatic differentiation. — This technique can certainly be successfully used in other contexts.

Future developments will include a study whether the work done in [17] can be applied to accelerate the assembly steps during integration. Also, although vectorial notations in the language can be used, they need to be formalized within the language to avoid ambiguities that would yield wrong results.

Acknowledgments

The DSEL was developed within a development branch of the LifeV project, see [22, 1].

References

[1] Lifev: a finite element library. http://www.lifev.org.

[2] David Abrahams and Aleksey Gurtovoy. *C++ Template Metaprogramming : Concepts, Tools, and Techniques from Boost and Beyond*. C++ in Depth Series. Addison-Wesley Professional, 2004.

[3] Pierre Aubert and Nicolas Di Césaré. Expression templates and forward mode automatic differentiation. In George Corliss, Christèle Faure, Andreas Griewank, Laurent Hascoët, and Uwe Naumann, editors, *Automatic Differentiation of Algorithms: From Simulation to Optimization*, Computer and Information Science, chapter 37, pages 311–315. Springer, New York, NY, 2001.

[4] Pierre Aubert, Nicolas Di Césaré, and Olivier Pironneau. Automatic differentiation in C++ using expression templates and application to a flow control problem. *Computing and Visualisation in Sciences*, 2000. Accepted.

[5] Babak Bagheri and Ridgway Scott. Analysa. http://people.cs.uchicago.edu/ ridg/al/aa.ps, 2003.

[6] David H. Bailey, Yozo Hida, Karthik Jeyabalan, Xiaoye S. Li, and Brandon Thompson. C++/fortran-90 arbitrary precision package. http://crd.lbl.gov/~dhbailey/mpdist/.

[7] Satish Balay, Kris Buschelman, Victor Eijkhout, William D. Gropp, Dinesh Kaushik, Matthew G. Knepley, Lois Curfman McInnes, Barry F. Smith, and Hong Zhang. PETSc users manual. Technical Report ANL-95/11 - Revision 2.1.5, Argonne National Laboratory, 2004.

[8] Erik Burman and Peter Hansbo. Edge stabilization for the generalized Stokes problem: a continuous interior penalty method. *Comput. Methods Appl. Mech. Engrg.*, 2005. in press.

[9] Nicolas Di Césaré and Oliver Pironneau. Hatfem, une bibliothèque de manipulation des fonctions chapeaux. http://nicolas.dicesare.free.fr/Fac/R97033.ps.gz and http://www.ann.jussieu.fr/~pironneau/.

[10] Philippe G. Ciarlet. *The finite element method for elliptic problems*, volume 40 of *Classics in Applied Mathematics*. Society for Industrial and Applied Mathematics (SIAM), Philadelphia, PA, 2002. Reprint of the 1978 original [North-Holland, Amsterdam; MR0520174 (58 #25001)].

[11] Patrick Dular and Christophe Geuzaine. Getdp: a general environment for the treatment of discrete problems. http://www.geuz.org/getdp.

[12] Frédéric Hecht and Olivier Pironneau. *FreeFEM++ Manual*. Laboratoire Jacques Louis Lions, 2005.

[13] Yozo Hida, Xiaoye S. Li, and David H. Bailey. Quad-double arithmetic: Algorithms, implementation, and application. Technical Report LBNL-46996, Lawrence Berkeley National Laboratory, Berkeley, CA 9472, Oct. 2000. http://crd.lbl.gov/~dhbailey/mpdist/.

[14] Joao Janela, Aline Lefebvre, and Bertrand Maury. A penalty method for the simulation of fluid - rigid body interaction. In *ESAIM proceedings*, 2005. submitted.

[15] George Em Karniadakis and Spencer J. Sherwin. *Spectral/hp element methods for CFD*. Numerical Mathematics and Scientific Computation. Oxford University Press, New York, 1999.

[16] Vesa Karvonen and Paul Mensonides. The boost library preprocessor subset for c/c++. http://www.boost.org/libs/preprocessor/doc/.

[17] Robert C. Kirby and Anders Logg. A compiler for variational forms. Technical report, Chalmers Finite Element Center, 05 2005. www.phi.chalmers.se/preprints.

[18] A. Logg, J. Hoffman, R.C. Kirby, and J. Jansson. Fenics. http://www.fenics.org/, 2005.

[19] Kevin Long. Sundance: Rapid development of high-performance parallel finite-element solutions of partial differential equations. http://software.sandia.gov/sundance/.

[20] Daniele A. Di Pietro and Alessandro Veneziani. Expression templates implementation of continuous and discontinuous galerkin methods. *Submitted to Computing and Visualization in Science*, 2005.

[21] Stéphane Del Pino and Olivier Pironneau. *FreeFEM3D Manual*. Laboratoire Jacques Louis Lions, 2005.

[22] Christophe Prud'homme. A modern and unified c++ implementation of finite element and spectral element methods in 1d, 2d and 3d. In preparation.

[23] Yves Renard and Julien Pommier. Getfem++: Generic and efficient c++ library for finite element methods elementary computations. http://www-gmm.insa-toulouse.fr/getfem/.

[24] Todd Veldhuizen. Using C++ template metaprograms. *C++ Report*, 7(4):36–43, May 1995. Reprinted in C++ Gems, ed. Stanley Lippman.