Computing and Visualization in Science manuscript No.
(will be inserted by the editor)

D. A. Di Pietro · **A. Veneziani**

Expression Templates Implementation of Continuous and Discontinuous Galerkin Methods

Received: date / Revised version: date

Abstract Efficiency and flexibility are often mutually exclusive features in a code. This still prompts a large part of the Scientific Computing community to use traditional procedural language. In the last years, however, new programming techniques have been introduced allowing for a high level of abstraction without loss of performance. In this paper we present an application of the Expression Templates technique introduced in [13] to the assembly step of a finite element computation. We show that a suitable implementation, such that the compiler has the role of parsing abstract operations, allows for user-friendliness and gain in performance with respect to more traditional techniques. Both the cases of conforming and discontinuous Galerkin finite element discretization are considered. The proposed implementation is finally applied to a number of problems entailing different kind of complications.

Keywords Galerkin methods, Finite elements implementation, Object-Oriented programming, Expression templates

1 Introduction

Object-Oriented (OO) programming has become an important approach in Computer Science for solving complex problems in an effective and elegant way. One of the most relevant features is *the high level of abstraction* (generic programming) supported by OO languages like C++ (see [12]). Abstraction together with encapsulation and operator overloading can make the implementation of a problem closer to its mathematical formulation and, at the same time, improve the maintainability of a code.

In principle, these features are very well suited for applications in Scientific Computing, and in particular for the approximate solution of Partial Differential Equation (PDE) problems arising in different fields (Mathematical Physics, Biology, Economy, etc.). The difficulty of implementing a general purpose solver in this context lies in the difference between the mathematical formulation of a differential problem and its implementation (see [9]). The support for user-defined types therefore appears the ideal tool for the implementation of mathematical structures. However, what seemed to be a "*natural union* has historically been more of a *stormy relationship*" ([4]). The application of OO programming has been limited in the context of Numerical Computing by efficiency concerns: the extensive use of virtual functions and operator overloading can strongly reduce the performance of a code. As a matter of fact, resolution of operator overloading is typically done run time, which can be unacceptable when dealing with complex problems (as *e.g.* in fluid mechanics). All these reasons (and, of course, historical ones) still compell a part of the Scientific Computing community to use traditional procedural languages (Fortran and C), ensuring better efficiency. Nevertheless, several efforts have been done to provide user-friendly interfaces to general purpose finite elements (FE) solvers. With no claim of completeness, we quote FreeFem (`www.freefem.org`), DiffPack ([8]), Open Foam (`www.openfoam.org`).

In the last years, special programming techniques have been developed with the goal of providing both elegance and efficiency in the OO framework. In particular, in [13] a technique called *Expression Templates* has been proposed for the effective handling of mathematical expression passed as arguments to subroutines and vector operations. The basic idea is to regard an abstract expression as a template that can be resolved by the compiler, *i.e.* not run-time. The effectiveness of Expression Templates technique in handling different, quite simple, mathematical problems is illustrated *e.g.* in [7]. In this paper, we aim at extending the use of Expression Templates to the implementation of a finite element library for PDE. The Expression Templates technique will

Send offprint requests to: Alessandro Veneziani, `alessandro.veneziani@mate.polimi.it`

Daniele A. Di Pietro
Dipartimento di Ingegneria Industriale, Università degli Studi di Bergamo, Viale Marconi 5, I - 24030 Dalmine (Bg), Italy

Alessandro Veneziani
MOX (Modeling and Scientifi c Computing), Dipartimento di Matematica "F. Brioschi",Politecnico di Milano,Via Bonardi 9, I - 20133 Milano, Italy

be used to build the discrete version of a differential operator which can be viewed as the composition of elementary operators and coefficients. The resulting code is therefore really *user-friendly*, since the user can define the problem in a way close to its mathematical formulation and, on the other hand, still *effective* thanks to the Expression Templates approach.

Our main concern will be the so-called *assembly step* of the FE solver, where the matrices resulting from the discretization of the differential operator are built. In this respect, our use of Expression Templates is quite different from the one proposed in [9], where the so-called *matrix-free* approach is considered and the Expression Templates technique is used as an effective and user friendly tool for managing algebraic operations in solving the discretized problem. We will address both the case of conforming and non-conforming FE approximation. Particular importance will be given to discontinuous Galerkin (DG) methods, which have received more and more attention in the last years because of their better properties in hyperbolic and convection-dominated problems. We will finally show how the Expression Templates technique can be easily adapted to the implementation of different kind of FE solvers.

The outline of the paper is the following. In §2.1 we will recall the basics of the Expression Templates technique. In §2.2 we will correspondingly recall basics concepts of conforming Galerkin methods focusing on continuous FE. In §3 we actually illustrate how to effectively implement the assembling phase for a generic differential operator `oper`. For the sake of simplicity, in the exposure we'll refer to an advection diffusion problem featuring constant coefficients (§3.1). We will then illustrate the extension to the more general case of non-constant coefficients of different type (tensors, scalars, etc.) (§3.2). In §3.3 we will extend the approach for the implementation of advection stabilization techniques, which are mandatory for solving advection dominated problems within the context of Galerkin methods. In §4 we will extend the Expression Templates technique to the implementation of DG methods. In particular, in §4.1.1 we will address some recent development in the DG framework, namely the *Interior Penalty* method, and its Expression Templates coding.

Finally, in §5 we report some numerical results, providing quantitative confirmation of the effectiveness of the Expression Templates technique. In §6 we will draw some conclusions.

2 Basic Facts

2.1 Basics of the Expression Templates

We briefly recall some basics concept about the Expression Templates technique. A complete description can be found in the original paper by Veldhuizen [13] and in [5], [4]. See also [7]. Suppose that $\mathbf{x}$ is a vector of n real numbers storing the abscissas where you need to evaluate a generic function $f(x)$. For the linear function $f(x) = ax + x/b$, the vector formulation of our task reads:

$$\mathbf{y} = a\mathbf{x} + \frac{1}{b}\mathbf{x}. \tag{1}$$

The required operation has actually to be interpreted componentwise and it refers to the execution of n scalar operations, namely:

$$y_i = ax_i + x_i/b, \quad (i = 1, \ldots, n) \tag{2}$$

which can be considered as the *scalar core* of the vector expression. Now let `y`, `a`, `x` and `b` be the variables storing all the terms in (1). The computation of `y` can be carried in C++ in a generic way (with respect to `f`) by using a pointer to a callback function or a suitably defined functor class. Alternatively, the task can be accomplished by operator overloading. Once all the involved operations (`+`, `*`, `/`, `=`) are suitably defined, the following instruction will be legal:

```
y = a * x + x / b;
```

Although this solution is elegant and clear, the operator overloading is not effective. The actual meaning of the above instruction will be indeed parsed run-time. Moreover, the expression is decomposed into the binary operations at hand. A first loop will be devoted to the computation of `a * x` and the result will be stored in a temporary vector; another loop will then compute `x / b`, storing the result in a second temporary vector; finally, a third temporary vector will receive the result of the sum of the previous ones. The operation will end up with the assignment of the third temporary to vector `y`. The net result is that an operation which could be efficiently completed with a single loop on the scalar core (2) will require three loops and three temporary vectors. In addition, the use of a function pointer or a functor is not really satisfactory, since the call to the functions will be iterated inside the loop generating a lot of overhead.

Expression Templates technique is based on the idea of building suitable classes so that the expression to be evaluated can be considered as a templatized argument. The expression type will be built by the compiler through an implicit type interpretation. In this way, the only run-time operation will be the computation of the expression for each value of the index `i`, that will become a simple call to `operator()`. This will allow the evaluation of the entire scalar core of the expression, without temporaries. All these tasks can be accomplished through the following steps:

1. Define the fundamental types that are involved in the expression at hand; in our example, vectors of reals and scalars (double).
2. Define a *wrapper* class to make objects of different type homogeneous, allowing their composition without requiring the explicit definition of all the possible combinations of operands.
3. Define the operations of interest in such a way that they can be passed as a template argument for building the expression at hand.

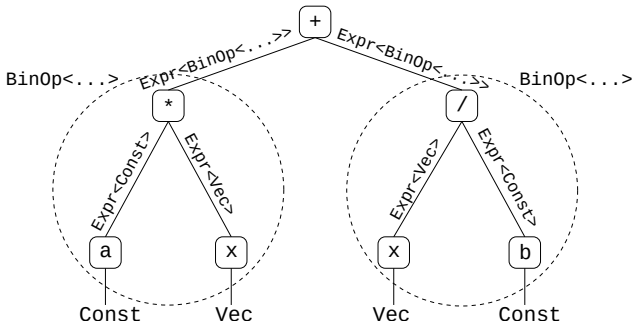


Fig. 1 Parse tree for the expression `f = a * b + x / b`.

Every step is extensively illustrated in [13] and [5]. In the case of our example, we need to instantiate an object of type:

```
Expr<BinOp<
      BinOp<Expr<Const>,Expr<Vec>,OpMult>,
      BinOp<Expr<Vec>,Expr<Const>,OpDiv>,
      OpAdd>>
```

where `Expr<A>` is the wrapper class, `BinOp<A,B,Op>` is the basic class for binary operations, `Vec` and `Const` refer to the fundamental types involved in the expression and `Op...` stands for the class defining the corresponding operation. This class is the inline formulation of the parse tree depicted in Fig. 1. It is important to point out that the construction of such class is not matter of the user, nor it is solved run-time, but it is interpreted from the expression by the compiler.

2.2 Basics of the Finite Element Method

In this section we will give some basic concepts about continuous Galerkin methods. A comprehensive introduction to these topics can be found in [10] and [6]. In the exposure we will refer to a steady advection-diffusion-reaction problem featuring constant coefficients: most of what follows, however, can be extended to a general linear (or linearized) differential operator. We will therefore consider the following problem: *given a bounded domain* $\Omega \in \mathbb{R}^3$, *find* $u(\mathbf{x})$ *such that*:

$$-\mu\triangle u+\beta\cdot\nabla u+\sigma u=f,\quad \mathbf{x}\in\Omega. \tag{3}$$

For the moment being, we assume that μ,σ are given constants and β is a constant vector. The forcing term $f(\mathbf{x})$ is assigned too. Different conditions can be prescribed on the boundary $\partial\Omega$: for the sake of simplicity, we can assume that on the boundary we have so called *(homogeneous) Neumann boundary conditions*:

$$\mu\nabla u\cdot\mathbf{n}=\mathbf{0}, \tag{4}$$

being $\mathbf{n}$ the unit outward normal vector to the boundary. Equations (3) and (4) define the so-called *strong formulation* of the differential problem. Such a formulation is not suitable in some real cases, *e.g.* when the forcing term is not regular. It is therefore worthwhile resorting to a more general, *weak* or *variational* formulation. Denote by V the functional space the unknown u is assumed to belong to. The space V will be, in general, infinite-dimensional. The variational formulation of the problem reads: *find* $u(\mathbf{x})\in V$ *such that for each function* $\varphi\in V$

$$a(u,\varphi)=\mathscr{F}(u), \tag{5}$$

where:

$$a(u,\varphi)\equiv\int_\Omega\mu\nabla u\cdot\nabla\varphi\,\mathrm{d}\omega+\int_\Omega\beta\cdot\nabla u\varphi\,\mathrm{d}\omega+\int_\Omega\sigma u\varphi\,\mathrm{d}\omega,$$
$$\mathscr{F}(u)\equiv\int_\Omega f\varphi\,\mathrm{d}\omega.$$

Equation (5) stems from the application of well known variational principles and of the Green formula (for the diffusive term); φ is usually called a *test function*. In order to find a numerical solution for problem (5), we need to approximate it with a finite dimensional one. The Galerkin class of methods is based on the introduction of a *finite dimensional subspace* V_h of V such that $V_h\subset V$ and $V_h\to V$ when $h\to 0$. In particular, we denote by $\{\varphi_i\}$ for $i=1,2,\dots N_h$ a basis functions set of V_h, such that every function $v_h\in V_h$ can be written as a linear combination of the φ_i. The finite dimensional formulation of problem (5) therefore reads: *find* $u_h(\mathbf{x})\in V_h$ *such that, for every function* φ_i $(i=1,2,\dots N_h)$

$$a(u_h,\varphi_i)=\mathscr{F}(\varphi_i). \tag{6}$$

Since $u_h\in V_h$ we can express it in terms of the basis function as:

$$u_h=\sum_{j=1}^{N_h}U_j\varphi_j. \tag{7}$$

In view of (7), the discrete problem (5) can be rewritten in the algebraic form $A\mathbf{U}=\mathbf{F}$, where $\mathbf{U}$ is the vector of the unknowns U_i, A is the matrix whose entries are:

$$a_{ij}=\int_\Omega\mu\nabla\varphi_j\cdot\nabla\varphi_i\,\mathrm{d}\omega+\int_\Omega\beta\cdot\nabla\varphi_j\varphi_i\,\mathrm{d}\omega+\int_\Omega\sigma\varphi_j\varphi_i\,\mathrm{d}\omega, \tag{8}$$

while $\mathbf{F}=[f_i]=[\int_\Omega f\varphi_i\,\mathrm{d}\omega]$.

The construction of the matrix A and vector $\mathbf{F}$, which will be considered here in the framework of Expression Templates , is the so called *assembly step*.

Different methods belonging to the Galerkin class are obtained by different choices for the basis function set $\{\varphi_i\}$. In particular, in continuous (or conformal) FE, V_h is given by the span of piecewise polynomial functions over a suitable triangulation (mesh) $\mathscr{T}_h$ of the domain Ω. Namely, for a selected polynomial degree k:

$$V_h\equiv\left\{v_h\in C^0(\overline{\Omega}):\,v_h|_{K_j}\in\mathbb{P}^k,\ \forall K_j\in\mathscr{T}_h,\right\}$$

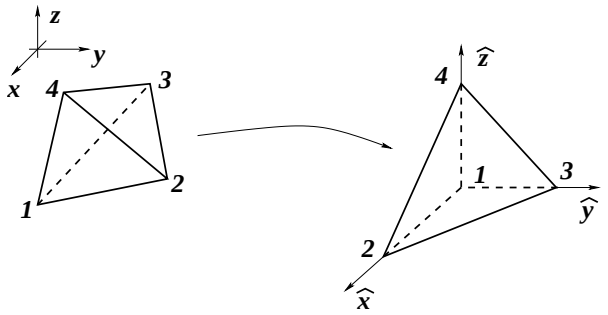


Fig. 2 Map of a generic tetrahedron into the refrence unit 3D simplex.

Matrix A assembling is typically carried out with a loop over the elements of the triangulation. More precisely, every element of the mesh is mapped onto a reference element where the explicit expression of the basis functions is known. We denote by $\{\widehat{\varphi}_i\}$ the basis function set in the reference frame $O\widehat{x}\widehat{y}\widehat{z}$. For a 3D tetrahedral mesh, *e.g.*, every terahedron is mapped into the unit 3D simplex (see Fig. 2). Denote by $\texttt{oper}_{ij}$ the function to be integrated in the problem at hand, i.e.

$$\texttt{oper}_{ij} = \mu \nabla\varphi_j \cdot \nabla\varphi_i + \beta \cdot \nabla\varphi_j\varphi_i + \sigma\varphi_j\varphi_i.$$

The computation of the entry a_{ij} reads

$$a_{ij} = \int_\Omega \texttt{oper}_{ij}\, \mathrm{d}\omega = \sum_{k=1}^{\texttt{Ne}} \int_{T_k} \texttt{oper}_{ij}\, \mathrm{d}\omega$$
$$= \sum_{k=1}^{\texttt{Ne}} \int_{\widehat{T}} \widehat{\texttt{oper}}_{ij} |J_k|\, \mathrm{d}\widehat{\omega}$$

where Ne is the number of elements, J_k is the jacobian of the map of T_k onto $\widehat{T}$, $|J_k|$ its determinant and $\widehat{\texttt{oper}}_{ij}$ is the reformulation of $\texttt{oper}_{ij}$ with respect to the reference variables $\widehat{x},\widehat{y},\widehat{z}$, namely:

$$\widehat{\texttt{oper}}_{ij} = \mu \left(J^{-1}\widehat{\nabla}\widehat{\varphi}_j\right) \cdot \left(J^{-1}\widehat{\nabla}\widehat{\varphi}_i\right) + \beta \cdot \left(J^{-1}\widehat{\nabla}\widehat{\varphi}_j\right) \widehat{\varphi}_i + \sigma\widehat{\varphi}_j\widehat{\varphi}_i.$$

The assembling of the matrix A is therefore split into a *local computation* over the reference terhaedron, yielding a $N_{\text{dof}} \times N_{\text{dof}}$ matrix A_{loc} and a *global update* of the matrix A, in which the entries of the global matrix A are updated by the corresponding entries of the local one. A possible snapshot of the assembly loop is therefore:

```
for(int i = 1; i <= mesh.numVolumes(); ++i) {
    // compute |J| and J^-1 grad
    fe.updateFirstDeriv(mesh.volumeList(i));

    // initialize local matrix:
    Aloc.zero();

    // compute local matrix
    compute_mat(Aloc,oper,fe);

    // update global matrix
    assemb_mat(A,Aloc,fe,dof);
}
```

The computation of the integrals over $\widehat{T}$ is usually carried out by means of suitable quadrature formulae:

$$\int_{\widehat{T}} \widehat{\texttt{oper}}_{ij} |J| d\widehat{\omega} \approx \sum_{l=1}^{\texttt{nq}} oper_{ij}(\widehat{x}_l,\widehat{y}_l,\widehat{z}_l) w_l,$$

where nq is the number of quadrature nodes, with coordinates $\widehat{x}_l,\widehat{y}_l,\widehat{z}_l$ and w_l is the associated weight. A possible snapshot of the corresponding loop is:

```
for(int i = 0; i < fe.nbNode; i++) {
    for(int j = 0; j < fe.nbNode; j++) {
        s = 0;

        for(int l = 0;l < fe.nbQuadPt; l++) {
            // (*)
            s += oper(i,j,l) * fe.weightDet(l);
            // (*)
        }
    loc_mat(i,j) += s;
    }
}
```

These code is generic with respect to the actual differential operator oper to be solved. Actually, for the specific problem at hand, we should substitute the lines of code delimited by (*) with:

```
for(int ic = 0; ic < 3; ic++) {
 s += mu * phiDer(i,ic,l) * phiDer(j,ic,l) +
      beta(ic)* phiDer(i,ic,l) * phi(j,l);
}
s += sigma * phi(i,l) * phi(j,l);
```

where we assumed that phi(i,l) and phiDer(i,ic,l) store respectively $\widehat{\varphi}_i(\widehat{x}_l)$ and $\partial_{\widehat{x}_{ic}}\widehat{\varphi}_i(\widehat{x}_l)$. The goal of the present paper is exactly to show that, thanks to Expression Templates , we can write a generic code retaining the abstract operator oper, without loss of efficiency and the parsing of the portion of code surrounded by (*) can be carried out by the compiler.

3 Assembly Through Expression Templates

3.1 Construction of the Scalar Core

In this subsection we deal with the expression templates implementation of matrix assembly step. The code line:

```
s += oper(i,j,l) * fe.weightDet(l);
```

corresponds to the *scalar core* of matrix assembly and it's therefore the counterpart of the componentwise operation in the algebraic problem (2). In view of this remark, the first step to take is to specify the basic types for the leaves of the parse tree, which we call *Elementar Operators*. In solving the reference problem (3), we are interested in three basic operators which combine to give the differential problem:

1. stiffness operator (diffusion);
2. gradient operator (convection);
3. mass operator (reaction).

Elementar operators can be implemented as functors: a snapshot of the code is given hereafter.

```
// Diffusion
class Stiff{
public:
  Stiff(CurrentFE* fe):_fe(fe) {
  }
  CurrentFE* fe_ptr() {
      return _fe;
  }

  Real operator()(int i, int j, int l) {
   Real s = 0.;
    for(int ic = 0; ic < _fe->nbCoor; ic++) {
        s += _fe->phiDer(i,ic,l) *
             _fe->phiDer(j,ic,l);
    }
    return s;
  }
private:
   CurrentFE* _fe;
};

// Convection
template<int coor>
class Grad{
public:
    Grad<coor>(CurrentFE* fe):_fe(fe) {
    }
    CurrentFE* fe_ptr() {
        return _fe;
    }

    Real operator()(int i, int j, int l) {
        return _fe->phi(i,l) *
               _fe->phiDer(j,coor,l);
    }
private:
    CurrentFE* _fe;
};

// Reaction
class Mass{
public:
    Mass(CurrentFE* fe):_fe(fe) {
    }
    CurrentFE* fe_ptr() {
        return _fe;
    }

    Real operator()(int i, int j, int l) {
        return _fe->phi(i,l) * _fe->phi(j,l);
    }
private:
    CurrentFE* _fe;
};
```

Since we want to handle algebraic expressions involving combinations of all these operators, we need a *wrapper* class `EOExpr` (*Elementar Operator Expression*), whose implementation is given below.

```
template<typename P, typename A>
class EOExpr{
private:
    A _a;
public:
    EOExpr(const A& eo):_a(eo) {
    }

    P operator()(int i, int l) {
        return _a(i,l);
    }
};
```

In this code, the type returned by the elementar operator is passed as a template class. This is not stricty necessary in the example, where the scalar kernel invariably returns a real value. However, the abstraction can be useful for the extension of the approach to vector or even tensor operators. In the case of constant coefficients, the only possible operations are the multiplication or the division of an operator by a scalar coefficient. We therefore introduce abstract operations involving elementar operators and real numbers. These operations are coded into specific classes which will be suitably wrapped in order to become nodes of the parse tree.

```
// Composition of two operators
template <typename P, typename A,
          typename B, typename Op>
class EOBinOp{
private:
    A _a;
    B _b;
public:
    EOBinOp(const A& a, const B& b):_a(a), _b(b) {
    }

    P operator()(int i, int j, int l) {
        return Op::apply(_a(i,j,l),_b(i,j,l));
    }
};

// Real-Operators operations (Multiply)
template <typename P, typename A, typename Op>
class EORBinOp{
private:
    A _a;
    Real _b;
public:
    EORBinOp(const A& a, const Real b)
        :_a(a), _b(b) {
    }
```

```
    EORBinOp(const Real  b,const A& a)
        :_a(a), _b(b) {
    }

    P operator()(int i, int j, int l) {
        return Op::apply(_a(i,j,l),_b);
    }
};
```

Now, we need to introduce the actual definition of the operations by specifying the `apply` methods:

```
// Sum
template<typename P>
class EOAdd{
public:
    EOAdd(){
    }

    static inline P apply(P a, P b) {
        return a + b;
    }
};

// Scalar multiplication
template<typename P>
class EORMult{
public:
    EORMult() {
    }
    static inline P apply(P a, Real b) {
    return a * b;
    }
};
```

Finally, operators + and * can be overloaded in a way similar to the one proposed in [13] for algebraic operations:

```
// Operator+
template<typename P, typename A, typename B>
EOExpr<P, EOBinOp<P,EOExpr<P,A>,EOExpr<P,B>,
                  EOAdd<P> > >
operator+(const EOExpr<P,A>& a,
          const EOExpr<P,B>& b) {
    typedef EOBinOp<P,EOExpr<P,A>,
                    EOExpr<P,B>,
                    EOAdd<P> > ExprT;
    return EOExpr<P,ExprT>(ExprT(a,b));
}

// Operator*
template<typename P, typename A>
EOExpr<P,EORBinOp<P,EOExpr<P,A>,EORMult<P> > >
operator*(const Real b, const EOExpr<P,A>& a) {
    typedef EORBinOp<P,EOExpr<P,A>,
                     EORMult<P> > ExprT;
    return EOExpr<P,ExprT>(ExprT(a,b));
}
```

For the sake of simplicity, we assume for the moment that the coefficients β_2 and β_3 are both zero; the general case will be discussed later on in this section. An example of code using the classes described above to assemble problem matrices could be:

```
// Coefficients
Real mu = 2.;
Real sigma = 0.05;
Real beta1 = 0.1;

// Types for the EO incapsulation of operators
typedef EOExpr<Real, Stiff>  EOStiff;
typedef EOExpr<Real, Mass>   EOMass;
typedef EOExpr<Real, Grad<0> >  EOGradx;

// Operators...
Stiff Ostiff(&fe);
Mass Omass(&fe);
Grad<0> Ogradx(&fe);

// ...and their wrappings into EO
EOStiff stiff(Ostiff);
EOMass mass(Omass);
EOGradx gradx(Ogradx);

assemble(mu*stiff+beta1*gradx+sigma*mass,
         mesh, fe, dof, phifct, A, F);
```

The `assemble` function builds the problem matrix `A` and the right hand side `F` given an expression for the operator and all the finite element data, whose exact definition lies outside of the scope of the present work. The `compute_mat` method is listed below.

```
template<typename Oper>
void compute_mat(ElemMat& Aloc, Oper& oper,
                 const CurrentFE& fe) {
    Real s;
    for(int i = 0; i < fe.nbNode; i++) {
        for(int j = 0; j < fe.nbNode; j++) {
            s = 0;
            for(int l = 0; l < fe.nbQuadPt; l++) {
                s += oper(i,j,l) * fe.weightDet(l);
            }
            Aloc(i,j) += s;
        }
    }
}
```

Method `assemble` receives an extremely complicated expression for a simple problem. Actually, the type of `oper` for the problem at hand, corresponding to the given parsing tree, is:

```
EOExpr<Real,EOBinOp<Real,
 EOExpr<Real,EOBinOp<Real,
  EOExpr<Real,EORBinOp<Real,EOExpr<Real,Stiff>,
        Real,EORMult>>,
  EOExpr<Real,EORBinOp<Real,EOExpr<Real,
                                   Grad<0> >,
        Real,EORMult>>,
```

```
 EOAdd>,
  EOExpr<Real,EORBinOp<Real,EOExpr<Real,Mass>,
         Real,EORMult>>,
 EOAdd>>
```

However, it is worthwhile remarking that:

1. the type for the expression is not built by the programmer;
2. expression parsing is done at compile time, which improves the efficiency of the code.

From the viewpoint of the user this leads to a more readable code without efficiency loss with respect to other general purpose methods, which typically can pay a higher price for the abstraction. A more classical way to obtain similar flexibility, but renouncing the abstraction related to expression handling, would require the introduction of a set of basic functions to build *Elementar Matrices* and then assemble the desired operator as the sum of such basic bricks:

```
ElemMat Aloc(fe.nbNode,1,1);
ElemVec Floc(fe.nbNode,1);

for(int i = 1; i <= mesh.numVolumes(); i++) {
fe.updateFirstDerivQuadPt(mesh.volumeList(i));

 stiff(mu, Aloc, fe);
 grad(0, beta, Aloc, fe)
 mass(sigma, Aloc, fe)
 source(sourceFct, elvec, fe);

 assemb_mat(A, Aloc, fe, dof);
 assemb_vec(F, Floc, fe,dof);
}
```

As an example, we give the implementation of the `grad` operator:

```
void grad(const int ic, Real coef,
      ElemMat& Aloc, const CurrentFE& fe) {
  Real s;
  for(int i = 0; i < fe.nbNode; i++) {
      for(int j = 0; j < fe.nbNode; j++) {
          s = 0;
          for(int l = 0; l < fe.nbQuadPt; l++)
              s += fe.phi(j,l) *
                   fe.phiDer(i,ic,l) *
                   fe.weightDet(l);
            Aloc(i,j) += coef*s;
        }
    }
}
```

In this case we have an outer loop on the elements and as many loops on the quadrature nodes as elementar operators. Expression Templates implementation allows to avoid this redundancy, since the scalar kernel is indeed a functor parsed by the compiler. As will be shown later, numerical experiments show that this feature is reflected in performance. Moreover, as already mentioned, the readibility of the code is greatly improved and the distance between the mathematical formulation of a problem and its implementation is strongly reduced, which has been considered by some authors (see [9]) a crucial point in the set up and diffusion of general purpose libraries for solving PDE.

3.1.1 Vector Operators

The implementation of the gradient operator is not completely satisfactory yet since every component has to be added by hand. In this paragraph we give a glance of the modifications needed in order to be able to combine the vector operator `grad` in the simple way:

```
mu * stiff + beta * grad + sigma * mass
```

Again,for the sake of simplicity we confine the discussion to the case of constant coefficients, postponing the generalization to the next subsection. The handling of such an expression requires the introduction of two more ingredients, *i.e.* vector operators and scalar multiplication. The former can simply be viewed as functors taking the component as an argument of `operator()`, while coefficient-vector products will be performed inside a suitable binary operation class (`EORVBinOp`).

The implementation of `operator()` in vector gradient operator is given hereafter.

```
Real vGrad::operator() (int i, int j,
                        int ic, int l) {
    return _fe->phiDer(j,ic,l) * _fe->phi(i,l);
}
```

In order for the wrapper class `EOExpr` to mimic the behaviour of a vector operator it is necessary to add an overloading of `operator()` matching the one above:

```
template<typename P, typename A>
P EOExpr<P, A>::operator()(int i, int j,
                           int ic, int l) {
    return _a(i, j, ic, l);
}
```

As mentioned above, we also need a class handling scalar multiplication of a vector operator times a vector coefficient. Assuming that a suitable definition of a constant coefficient vector class `RVect` is available, the implementation of `EORVBinOp` reads:

```
template<typename P, typename A, typename Op>
class EORVBinOp {
private:
    A _a;
    RVect _f;
public:
    EORVBinOp(const A& a, const RVect& f):
        :_a(a), _f(f){
    }

    EORVBinOp(const RVect& f, const A& a)
        :_a(a), _f(f){
```

```
    }

    P operator()(int i, int j, int l) {
        P s = 0.;
        for(int ic = 0; ic < NDIM; ic++)
            s += Op::apply(_a(i, j, ic, l),
                           _f(ic));
        return s;
    }
};
```

being `NDIM` the number of dimensions of the problem. The vector gradient operator will be defined in the main program by invoking:

```
vGrad Ograd(&fe);
EOExpr<Real, vGrad> grad(Ograd);
```

Once a suitable overloading of `operator*` is provided following the guidelines given above, the parsed type for the expression `beta * grad` will then be:

```
EOExpr<Real, EORVBinOp<Real,
                       RVect,
                       EOExpr<Real, vGrad> > >
```

3.1.2 Symmetric Operators

Reaction-diffusion problems (see (3) with $\beta = \mathbf{0}$) are symetric problems, in which matrix assembly can be effectively implemented by computing only one half of the local matrix entries. This feature can be exploited in the framework of our Expression Templates implementation by introducing a proper version of the function that computes local matrices:

```
template<typename Oper>
void compute_mat_symm(ElemMat& Aloc,
                      Oper& oper,
                      const CurrentFE& fe) {
    Real s;
    for(int i = 0; i < fe.nbNode; i++) {
        for(int j = i; j < fe.nbNode; j++) {
            s = 0;
            for(int l = 0; l < fe.nbQuadPt;
                l++) {
                s += oper(i,j,l) *
                     fe.weightDet(l);
            }
            Aloc(i,j) += s;
        }
    }
}
```

Suitable mechanisms can be introduced so that the most appropriate version of `compute_mat` is automatically called by the `assemble` function, but their description is outside the scope of the present work.

3.1.3 Boundary condition management

After the assembly step we end up with a matrix A which is the discrete version of the problem but which doesn't account for boundary conditions yet. For homogeneous boundary conditions like (4), no further operation is needed and the matrix is ready for the linear solver. Other kinds of conditions (non homogeneous Neumann, Dirichlet or Robin) require further work on A. Since we explicitly build the matrix, the application of boundary conditions can be done indepentently from matrix assembly. This is not the case in *matrix free* approaches, where the matrix is not explicitly stored, but a method is provided to perform matrix-vector operations. Such an implementation is considered, *e.g.* in [9].

3.2 Extension to Space Dependent Coefficients

An important extension is that to space (and possibly time) dependent coefficients. For the sake of simplicity, let's examine the case when the coefficients of our model problem (3) are space dependent, *i.e.*:

$$\mu = \mu(x,y,z),\ \beta_1 = \beta_1(x,y,z),\ \sigma = \sigma(x,y,z).$$

In this case, the scalar kernel of the assembling phase reads:

```
mu(x,y,z)    * stiff(i,j,l)  +
beta1(x,y,z) * grad(0,i,j,l) +
slma(x,y,z)  * mass(i,j,l)
```

where `x,y,z` are the coordinates of the `l`-th quadrature node. The coefficients `mu`, `beta` and `sigma` are defined by suitable functors, *e.g.*:

```
class Fmu : public Function {
public:
Real inline operator()(Real x, Real y, Real z){
      return 0.05*x;
    }
};
```

In order to handle operations involving a space-dependent coefficient and a differential operator we need to define a suitable `EOFBinOp` class, whose implementation reads:

```
template <typename P, typename A, typename Op>
class EOFBinOp {
private:
    A _a;
    Function _f;
public:
    EOFBinOp(const A& a, const Function& f) :
        _a(a), _f(f) {
    }

    EOFBinOp(const Function& f, const A& a) :
        _a(a), _f(f) {
    }

    P operator() (int i, int j, int l,
```

```
                    Real x, Real y, Real z ) {
        return Op::apply(_a(i, j, l),
                         _f(x, y, z) );
    }
};
```

A corresponding overloading of `operator()` must be added to `EOExpr` class so that it can mimic the behaviour of the `EOFBinOp` class:

```
template<typename P, typename A>
P EOExpr<P, A>::operator()(int i, int j, int l,
                           Real x, Real y,
                           Real z) {
    return _a(i, j, ic, l, x, y, z);
}
```

The final step to take is again to add a proper overloading of `operator*`:

```
template <typename P, typename A>
EOExpr<P, EOFBinOp<P, EOExpr<P, A>,
                   EORMult<P> > >
operator*(const Function& f,
          const EOExpr<P, A>& a) {
    typedef EOFBinOp<P, EOExpr<P, A>,
                     EORMult<P> > ExprT;
    return EOExpr<P, ExprT>( ExprT( a, f ) );
}
```

In §3.1.1 we outlined the modifications required by the original scheme in order to handle expressions involving constant vector coefficients and vector operators. Following the same strategy, and keeping in mind the discussion above, it is possible to figure out how to handle the case of function vector or tensor coefficients.

3.3 Stabilization of Advection-Diffusion Problems

The framework we set up allows to introduce any linear operator by simply adding its definition in the *Elementar Operator* set. The purpose of this section is to show how strongly consistent stabilization method can be implemented following this strategy.

3.3.1 A Short Introduction to Stabilization Methods

It is well known that Galerkin solution of advection-diffusion-reaction problems can lead to oscillating solutions when the convective term is quantitatively dominating (see e.g. [10]). Let us suppose that $\|\beta\| \gg \mu$. A general strategy to eliminate numerical oscillations is to add a numerical viscosity to the original formulation (5) of the problem. The so called *stabilized* formulation reads:

$$a(u_h,\varphi_i) + a_{\text{stab}}(u_h,\varphi_i) = \mathscr{F}(\varphi_i) + \mathscr{F}_{\text{stab}}(\varphi_i), \tag{9}$$

where $a_{\text{stab}}(u_h,\varphi_i)$ and $\mathscr{F}_{\text{stab}}(\varphi_i)$ are the stabilizing terms, for which several expressions are available. Among the others, *strongly consistent methods* have the property of achieving numerical stability without significantly affecting the asymptotic accuracy of the unperturbed finite element formulation of the problem. As a matter of fact, strongly consistent schemes share the following feature:

$$\mathscr{F}_{\text{stab}}(\varphi) - a_{\text{stab}}(u_{ex},\varphi) = 0, \quad \forall \varphi \in V,$$

so that the strong consistency or adherence of the numerical approximation to the original problem is maintained. More precisely, let us denote by $\mathscr{L}$ the advection-diffusion-reaction differential operator in its strong form:

$$\mathscr{L}u \equiv -\mu\triangle u + \beta\cdot\nabla u + \sigma u.$$

The original problem therefore reads $\mathscr{L}u = f$. The basic idea of strongly consistent methods is to introduce a perturbation proportional to the residual $\mathscr{L}u_h - f$, so that even for a fixed non-vanishing value of the discretization parameter h the perturbation vanishes when applied to the exact solution. To this aim, let us split the original operator into its symmetric and skew-symmetric components:

$$\begin{aligned} \mathscr{L}_S &= -\mu\triangle u + \left(\tfrac{1}{2}\nabla\cdot\beta + \sigma\right)u, \\ \mathscr{L}_{SS} &= \tfrac{1}{2}\nabla\cdot(\beta u) + \tfrac{1}{2}\beta\cdot\nabla u. \end{aligned} \tag{10}$$

so that we have $\mathscr{L} = \mathscr{L}_S + \mathscr{L}_{SS}$. The expressions for the stabilization terms in weak formulation read:

$$a_{\text{stab}}(u_h,\varphi_i) \equiv \sum_{K\in\mathscr{T}_h} \delta\left(Lu_h, \frac{h_K}{|\mathbf{b}|}(L_{SS}+\rho L_S)\varphi_i\right)_K, \tag{11}$$

$$\mathscr{F}_{\text{stab}}(\varphi_i) \equiv \sum_{K\in\mathscr{T}_h} \delta\left(f, \frac{h_K}{|\mathbf{b}|}(L_{SS}+\rho L_S)\varphi_i\right)_K. \tag{12}$$

Here K is a generic element of the triangulation $\mathscr{T}_h$, with diameter h_K; $(\cdot,\cdot)_K$ is the $L^2(K)$ scalar product (so that the strong elementwise formulation of the problem is mathematically well posed); $\delta > 0$ and $\rho > 0$ are parameters to be suitably tuned. Different methods are obtained according to the value of the parameter ρ. In particular, for $\rho = 0$ we recover the *Streamline Upwind/Petrov Galerkin* (SUPG) method, for $\rho = 1$ the *Galerkin Least Squares* (GALS) method, while for $\rho = -1$ the *Douglas-Wang/Galerkin* (DWG) method. The choice is a bit more delicate for the parameter δ, and many recipes are available in the literature. In our implementation we took:

$$\delta = \begin{cases} h_K|\beta|/2\|\beta\|_0 & \text{if } Pe_K > 1 \\ 0 & \text{otherwise} \end{cases}$$

being Pe_K the local Peclet number. An analysis of these methods can be found in [10], §8.4. As for the classic Galerkin methods, the error estimates show that the accuracy depends on the polynomial degree.

3.3.2 Expression Template Implementation

The aim of this section is to show how to implement stabilization techniques in such a way that the following expression is legal and makes sense:

```
mu * stiff + beta * grad + sigma * mass + stab
```

For simplicity of exposition we'll confine the discussion to problem (3) with zero right hand side ($f = 0$). In this case $\mathscr{F}_{\text{stab}}(\cdot) \equiv 0$ and we can focus on matrix assembly. Moreover, in order to avoid unuseful complications, we'll consider the case of constant coefficients: keeping in mind the discussion above, removing these hypotheses is a simple exercise left to the reader.

Consider the following `Stabilization` class:

```
template<typename mu_type,
         typename beta_type,
         typename sigma_type,
         int rho>
class Stabilization {
public:
  Stabilization(CurrentFE* fe,
          mu_type  mu,
          beta_type  beta,
          sigma_type sigma):
  _fe(fe),
  _mu(mu), _beta(beta), _sigma(sigma) {
  }

  CurrentFE* fe_ptr() {
    return _fe;
  }

  Real operator()(int i, int j, int l) const {
    Real h_K = _fe->diam;
    Real norm_2_beta = norm_2(beta);
    Real norm_L2_beta = norm_2_beta *
      _fe->meas();

    Real Pe_K = fabs( (.5 * h_K * norm_2_beta)
      / _mu);
    Real delta_K = Pe_K > 1 ?
      (.5 * h_K * norm_2_beta) /
      norm_L2_beta : 0.;

    Real LPhi_j = 0.;
    Real LSPhi_i = 0.;
    Real LSSPhi_i = 0.;

    for(int ic = 0; ic < _fe->nbCoor; ic++) {
      LPhi_j += - _mu *
         _fe->phiDer2(j,ic,ic,l)
        + _beta(ic) * _fe->phiDer(j,ic,l)
        + _sigma * _fe->phi(j,l);
      LSPhi_i += - _mu *
            _fe->phiDer2(i,ic,ic,l)
            + _sigma * _fe->phi(i,l);
      LSSPhi_i += _beta(ic) *
                  _fe->phiDer(i,ic,l);
    }

    return ((h_K / norm_2_beta) * delta_K *
           LPhi_j *
           (LSSPhi_i + rho * LSPhi_i));
}

protected:
  CurrentFE* _fe;
  mu_type  _mu;
  beta_type  _beta;
  sigma_type _sigma;
};
```

The key idea is to introduce a class sharing the same interface as a standard elementar operator but with additional properties allowing to compute the stabilization contribution.

In the case of strongly consistent methods, the `Stabilization` class should store all the coefficients and different specializations should be implemented according to the coefficients' nature. The resulting class can be used as a standard elementar operator except for the call to the non-standard constructor. An example of usage is given below:

```
// Stabilization type
typedef Stabilization<Real, RVect, 0> SUPG;

// Type for EO incapsulation
typedef EOExpr<Real, SUPG> EOSUPG;

// SUPG operator...
SUPG Osupg(&fe, mu, betaR);

// ...and its wrapping
EOSUPG supg(Osupg);
```

and, finally:

```
mu * stiff + beta * grad + sigma * mass + supg
```

We would like to point out that having parametrized the class with respect to `rho` allows to obtain any of the strongly consistent methods described above by simply choosing the proper value in the definition of the stabilization type.

4 Discontinuous Galerkin Method via ET

4.1 Discontinuous Galerkin Methods

Discontinuous Galerkin (DG) finite element methods were originally developed for nonlinear hyperbolic problems featuring discontinuous solutions even starting from a regular initial datum. The basic idea is to decouple the degrees of freedom belonging to each element, and to estabilish weak links by means of inter-element boundary terms. Renouncing continuity on element boundaries, a possible discontinuity in the solution can be resolved within a patch of a few

elements. In the remainder of this section we give a quick introduction to the DG approximation of a hyperbolic problem. For the sake of simplicity, we will refer to the linear case. For a more complete presentation we refer the reader to [3,1].

The model linear hyperbolic problem in divergence form reads:

$$\frac{\partial u}{\partial t} + \beta \cdot \nabla (u) = 0, \tag{13}$$

where β is a given continuous velocity field, possibly depending on space and time coordinates and such that $\nabla \cdot \beta = 0$. The weak formulation of the problem reads: *find* $u \in V$ *such that*:

$$\sum_{K \in \mathscr{T}_h} \int_K \frac{\partial u}{\partial t} v \, \mathrm{d}\mathbf{x} - \sum_{K \in \mathscr{T}_h} \int_K u \beta \cdot \nabla v + \sum_{K \in \mathscr{T}_h} \int_{\partial K} v \mathbf{F}^K \cdot \mathbf{n} \, \mathrm{d}\sigma \tag{14}$$

for all $v \in V$. In the previous formula we named $F \equiv \beta \, u|_K$ the flux of u through the element boundary ∂K. The DG finite dimensional approximation of this problem can be obtained by choosing V_h such that

$$V_h \equiv \left\{ v_h \in L^2(\Omega) \mid v_h|_K \in \mathbb{P}^k(K) \quad \forall K \in \mathscr{T}_h \right\}. \tag{15}$$

If we do not require inter-element continuity of test functions, a convenient choice for a base for V_h^k is a set of functions whose support is made up of exactly one element. If e denotes the face shared by elements K_1 and K_2 (see Fig. 3), the fluxes $\mathbf{F}^{K_1}$ and $\mathbf{F}^{K_2}$ across e in general will be different, *i.e.*:

$$\beta \, u|_{K_1} \neq \beta \, u|_{K_2}, \quad \mathbf{x} \in e = K_1 \cap K_2.$$

As a consequence, the integrals on the edges in general do not cancel out. A stable inter-element coupling can be obtained by replacing $\mathbf{F}$ with a conservative upwind flux $\mathbf{H}$. A convenient expression for $\mathbf{H}$ can be obtained after introducing the jump and average operators on internal faces defined as in [1]:

$$[\![f]\!] := f|_{K_1} \mathbf{n}_{K_1} + f|_{K_2} \mathbf{n}_{K_2}, \quad \{ f \} := \frac{1}{2} \left(f|_{K_1} + f|_{K_2} \right).$$

These definition can be extended to boundary faces by setting:

$$[\![f]\!] := f, \quad \{ f \} = \begin{cases} f \text{ on } \Gamma_{\text{in}} \\ g \text{ on } \Gamma_{\text{out}} \end{cases}$$

being g the Dirichelet datum on the inflow boundary. We refer the reader to the cited references for more details. The upwind flux $\mathbf{H}$ can be written as:

$$\mathbf{H} = \beta \{ u_h \} + \frac{1}{2} |\beta \cdot \mathbf{n}| + [\![u_h]\!]. \tag{16}$$

To obtain the final formulation we substitute $\mathbf{H}$ to $\mathbf{F}$ on every element and exploit the following property:

$$\begin{aligned} \sum_{K \in \mathscr{T}_h} \int_{\partial K} v_h|_K \mathbf{H} \cdot \mathbf{n}^K \, \mathrm{d}\sigma = & \sum_{e \in \mathscr{E}^0} \int_e \mathbf{H} \cdot [\![v_h]\!] \, \mathrm{d}\sigma \\ & + \sum_{b \in \mathscr{E}^\partial} \int_b \mathbf{H} \cdot [\![v_h]\!] \, \mathrm{d}\sigma \end{aligned} \tag{17}$$

where we named $\mathscr{E}^0$ and $\mathscr{E}^\partial$ the sets of internal and boundary element faces respectively and $\mathscr{E} = \mathscr{E}^\partial \cup \mathscr{E}^0$ their union. The DG weak approximation of the problem finally reads:

$$\begin{aligned} \sum_{K \in \mathscr{T}_h} \int_K \frac{\partial u}{\partial t} v \, \mathrm{d}\mathbf{x} + & \sum_{e \in \mathscr{E}^0} \int_e \mathbf{H} \cdot [\![v_h]\!] \, \mathrm{d}\sigma \\ & + \sum_{b \in \mathscr{E}^\partial} \int_b \beta \cdot [\![v_h]\!] \{ u \} \, \mathrm{d}\sigma = 0. \end{aligned} \tag{18}$$

The problem was finally discretized in time with a second order Crank-Nicolson method.

4.1.1 Recent Developments

More recently, DG finite element method has been successfully applied to problems involving elliptic terms. For simplicity of exposition we consider the Poisson problem:

$$-\Delta u = g$$

with homogeneous Dirichelet boundary conditions on $\partial\Omega$. In order to weakly impose continuity of the solution on element boundaries, different approaches can be pursued. We confine the discussion to the simplest one, which achieves the goal by introducing a penalty term on inter-element jumps. Provided suitable definitions of the trace operators are given on the domain boundary, a possible penalty term is the following:

$$\sum_{e \in \mathscr{E}} \int_e \eta_e [\![u_h]\!] \cdot [\![v_h]\!] \, \mathrm{d}\sigma$$

where we can choose $\eta_e = \frac{\mu}{h_e}$, being $\mu \in \mathbb{R}^+$ and h_e the measure of the face. The DG approximation of the model problem therefore reads:

$$\begin{aligned} \sum_K \int_K \nabla u_h \cdot \nabla v_h \, \mathrm{d}\mathbf{x} - & \sum_{e \in \mathscr{E}} \int_e \left([\![u_h]\!] \cdot \{ \nabla_h v_h \} + [\![v_h]\!] \cdot \{ \nabla_h u_h \} \right) \mathrm{d}\sigma \\ & + \sum_{e \in \mathscr{E}} \int_e \eta_e [\![u_h]\!] \cdot [\![u_h]\!] \, \mathrm{d}\sigma = \sum_K \int_K f v_h \, \mathrm{d}\mathbf{x}. \end{aligned}$$

We refer the interested reader to [1] for a complete survey. In the next sections we show how this method perfectly fits within the framework of Expression Templates implementation.

4.2 Implementation

DG methods differ from conventional FE ones in that inter-element coupling is achieved weakly through boundary terms. Consequently, the need for trace operators arises. The bilinear form providing the discretization of the PDE problem at hand (possibly after a suitable linearization) can therefore be conveniently regarded as the sum of three bilinear forms, dealing respectively with volume integrals, internal face integrals and boundary face integrals:

$$a(\varphi_i,\varphi_i) = a^K(\varphi_i,\varphi_i) + a^0(\varphi_i,\varphi_i) + a^\partial(\varphi_i,\varphi_j). \tag{19}$$

The contributions due to terms on internal and boundary faces are kept separated because the latter include weak boundary condition handling, which reflect on the different definition of the trace operators. In the implementation of matrix assembly routine we exploit formulas like (17) that allow to pass from a sum over elements to a sum over faces. In symbols:

$$\sum_{K\in\mathscr{T}_h}\left(\int_K \ldots + \int_{\partial K}\ldots\right) = \sum_{K\in\mathscr{T}_h}\int_K \ldots + \sum_{e\in\mathscr{E}^0}\int_e \ldots + \sum_{b\in\mathscr{E}^\partial}\int_e \ldots, \tag{20}$$

which is nothing but the expanded form of (19). This formulation allows to split matrix computation and assembling into three separate loops: one on volumes, one on internal faces and one on boundary faces. Volume integral contributions don't require further discussions, since they can be handled in the same way as in the conforming FEM case. Boundary integral terms, on the other side, deserve some more comments. We assume that two more current finite element classes are available (called `CurrentIFDG` and `CurrentBFDG`), storing the necessary information to compute boundary integrals. What is needed, in particular, are the values of adjacent elements' basis functions on face quadrature nodes, which we assume to be stored in members `phiK1` and `phiK2`. The general trace operator on internal faces will have the following interface:

```
class traceOperator {
public:
    traceOperator(CurrentIFDG* fe):_fe(fe){
    }

    Real operator()(int i, int j, int l,
                    int K1, int K2) {
        /* Function body */
    }

private:
    CurrentIFDG* _fe;
};
```

The most remarkable difference is that `operator()` now takes two element identifiers as arguments (`K1` and `K2`). The reason is that the shape functions of elements sharing internal face $e = \partial K_1 \cap \partial K_2$ can be conveniently re-numbered using two indices, the first being the element $K \in \{K_1, K_2\}$ coinciding with their support, the second the local DOF number on element K. The local matrix stemming from integrals on face e can therefore be evaluated by means of two inner loops on adjacent elements:

$$A^0_{\mathrm{loc},e} = \sum_{H\in\{K_1,K_2\}}\sum_{K\in\{K_1,K_2\}} a^0\left(\varphi_i^H,\varphi_j^K\right),$$

for $i = 1\ldots N_H$, $j = 1\ldots N_K$, being N_H and N_K the number of local degrees of fredoms on elements H and K respectively (which, in general, can differ from each other). The elementary matrix $A^0_{\mathrm{loc},e}$ is therefore built in a block-wise fashion, as shown in Fig. 3 Since expressions must mimic the behaviour of operators, a similar overloading for `operator()` is needed for the following classes: `EOExpr`, `EOBinOp`, `EORBinOp` and `EOFBinOp`. The implementation of boundary face trace operator is somehow easier, since every face $b \in \mathscr{E}^\partial$ only belongs to the boundary of one element, and only one more argument K has to be passed to `operator()`.

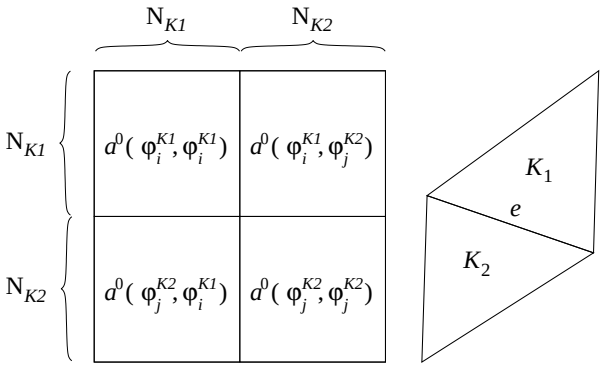


Fig. 3 Block structure of the elementary matrix associated to internal edge e.

In the light of the discussion above, it is worthwhile spending some words on jump and average unary trace operators, since most of the binary trace operators are just a combination of these two. The latter remark prompts for a separate implementation: for simplicity of exposition we'll list the code for `operator()` in `JumpIF` class, which implements the jump operator on internal faces.

```
Real JumpIF::operator()(int i, int ic,
                        int l, int H) {
  return (H == 0) ? _feIF->phiK1(i,l)
                    *_fe->normal(ic,l) :
                  -_fe->phiK2(i,l)
                    *_fe->normal(ic,l);
}
```

The operators `JumpBF` (jump trace operator on boundary faces), `AverageIF`, `AverageBF` (average trace operator on internal and boundary faces respectively) are implemented

in a similar way. Once available, this unary trace operators can be used as in the example given below, referring to the implementation of `operator()` in the interior penalty operator class (`IPIF`):

```
Real IPIF::operator()(int i, int j, int l,
                      int K, int H) {
    Real s = 0.;
    JumpIF jump(_fe);

    /* Computation of eta_e */

    for(int ic =0; ic < NDIM; ic++)
        s += eta_e * jump(i,ic,l,H)
                   * jump(j,ic,l,K);

return s;
}
```

Matrix assembly routines can be done in a similar way as before, except for the fact that now three loops (on volumes, internal and boundary faces) are needed.

5 Results

5.1 Continuous Finite Elements

Advection-Diffusin-Reaction problems In this paragraph we evaluate the performance of the proposed implementation using the more traditional approach sketched at the end of §3.1 as a benchmark. We consider the three problems defined by:

$$\begin{aligned} -\mu\triangle u = 0 \quad & \text{D}, \\ -\mu\triangle u + \sigma u = 0 \quad & \text{RD}, \\ -\mu\triangle u + \beta\cdot\nabla u + \sigma u = 0 \quad & \text{ADR} \end{aligned}$$

on a unit cube domain.

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

$$\begin{aligned} \mu\,(x,y,z) &= x^3+y^2z, \\ \beta\,(x,y,z) &= \left(x^3+y^2z, x^3+y^2, x^3\right), \\ \sigma\,(x,y,z) &= x^3+y^2z. \end{aligned}$$

All the results are collected in Tab. 1 for linear and quadratic finite elements, featuring respectively 1331 and 9261 DOFs. The expression template implementation proposed in this work proves very satisfactory in terms of efficiency, significantly reducing the assembly time in the case of space-dependent coefficients. This is mainly due to the fact that every redundancy is cancelled, as pointed out in §3.1.

A stabilized advection-diffusion problem As an example of stabilized computation, we considered the following problem:

$$\mu\triangle u + \beta\cdot\nabla u = 0,$$

N_{DOF}	Technique	D		RD		ADR	
		const	*xyz*	*k*	*xyz*	*k*	*xyz*
1331	ET	0.08	0.15	0.08	0.29	0.09	0.31
	EM	0.07	0.30	0.08	0.44	0.10	0.49
9261	ET	0.60	1.13	0.64	2.28	0.72	2.59
	EM	0.59	2.27	0.64	3.44	0.80	3.86

Table 1 Assembly time for expression template (ET) and elementar matrices (EM) techniques. The problems considered are stated in (21). The constant coeffi cient case is marked by const, while the space-dependent coeffi cient case is marked by the symbol *xyz*.

with coefficients:

$$\mu = 1\times 10^{-6}, \quad \beta = (x-1, y, 0)$$

and Dirichlet boundary conditions:

$$u|_{\partial\Omega} = g = \begin{cases} 1 & \text{if } x < \frac{1}{2}, \\ 0 & \text{otherwise.} \end{cases}$$

The discontinuity of the boundary datum induces oscillations on the numerical solution, as shown in Fig. 4(a). Adding the stabilization term allows to prevent oscillations, as shown in Fig. 4(b).

A non-linear problem As an example of a real problem, we solve within the Expression Templates code the problem:

$$-\mu\triangle u + \frac{au}{b+u} = 0, \quad \mathbf{x}\in\Omega$$

with $u(\partial\Omega) = g$. This problem comes from the investigations about Oxygen concentration in cells. In particular, Ω is a sphere centred in the origin and with radius R_2, μ is piecewise constant, namely $\mu = \mu_1$ for $x^2+y^2+z^2 \le R_1^2$ (with $R_1 < R_2$), a is picewise constant and in particular vanishes for $x^2+y^2+z^2 > R_1^2$. The cell corresponds to the cell with radius R_1 and the non linear term represents the Oxygen consumption due to the respiration, described by the so-called *Michaelis Menten law*. Outside the cell we have a gel where there is no respiration (for this model see e.g. [2]).

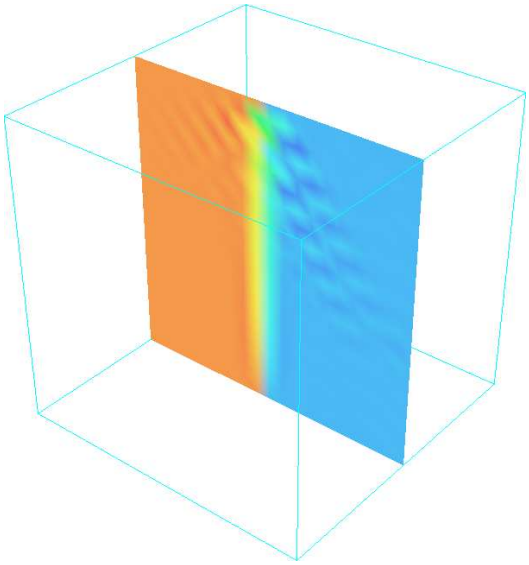
The nonlinear problem is linearized according to the following iterative fixed point scheme: given $u^{(0)}$, solve:

$$-\mu\triangle\widetilde{u}^{(k)} + \frac{a\widetilde{u}^{(k)}}{b+u^{(k-1)}} = 0, \quad \mathbf{x}\in\Omega$$

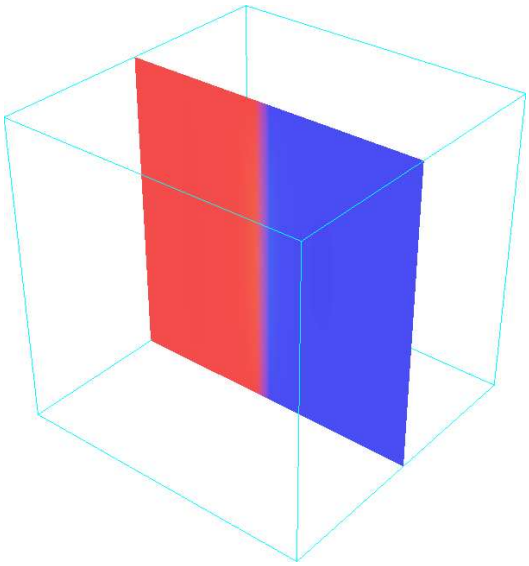
with $\widetilde{u}^{(k)}(\partial\Omega) = g$, and set:

$$u^{(k)} = \rho\widetilde{u}^{(k)} + (1-\rho)u^{(k-1)}.$$

for $k = 1,2,\ldots$ until convergence is reached. For a suitable choice of the relaxation parameter ρ, the method can be proved to converge. The values for the parameters are listed in Tab. 2. The partial pressure of the Oxygen is plot in Fig. 5. Some implementation details are mandatory. The problem



(a) Non-stabilized solution.



(b) Stabilized solution.

Fig. 4 Effect of the stabilization term on an advection-dominated problem. The plots show the solution value on the clip plane normal to axis z at $z = 0.5$.

Parameter	Interior value	Exterior value
μ	1.63×10^{-15}	3.30×10^{-15}
a	1.5×10^{-7}	0
b	0.44	–

Table 2 Values for the parameters of the example of the Oxygen in the cell.

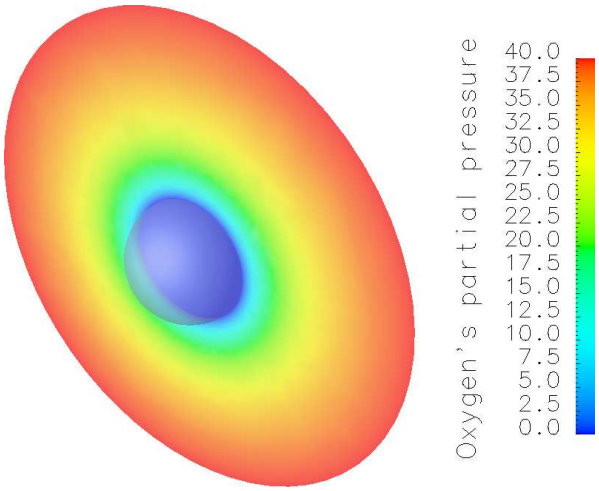


Fig. 5 Solution of the non-linear sample problem.

to be solved at every iteration can be viewed as a linear problem whose coefficient σ depend on the solution at previous iteration:

$$\sigma(u^{(k-1)}) = \frac{a}{b + u^{(k-1)}}.$$

A non-linear function was simply wieved as a functor whose `operator()` takes the vector of local solution values as an argument:

```
class sigmaU : public NLFunction {
 public:
  sigmaU(const Real R1, Real a1,
         const Real a2, const Real b1,
         const Real b2)
         :
         _R1(R1), _a1(a1), _a2(a2),
         _b1(b1), _b2(b2) {
        }

 Real operator() (CurrentFE& fe,
                  vector_type& loc_u,
                         int l ) {
  Real u_l = 0.;
  for( int k = 0; k < fe.nbNode; k++ )
       u_l += loc_u( k ) * fe.phi( k, l );

       Real x = fe.quadPt(l, 0);
       Real y = fe.quadPt(l, 1);
       Real z = fe.quadPt(l, 2);

       Real r = sqrt( x * x + y * y + z * z );

       Real a = ( r <= _R1 ) ? _a1 : _a2;
       Real b = ( r <= _R1 ) ? _b1 : _b2;

       return ( a / ( b + u_l ) );
       }
```

```
    private:
     Real _R1;
     Real _a1;
        Real _a2;
        Real _b1;
        Real _b2;
    };
```

Again this add requires the implementation of a class (called `EONLFBinOp`) handling binary operation involving a nonlinear coefficient and an elementar operator and a suitable overloading of `operator()`. For what concerns the viscosity, it can be simply viewed as a space-dependent coefficientstoring the necessary problem data (the radius R_1 and the two constant values μ_1 and μ_2). The main program will therefore read:

```
/* Definition of mu and mass as above */

sigmaU sigma(R1, a1, a2, b1, b2);
muxyz mu(R1, mu1, mu2);

do {
    // LHS and RHS initialization
    A.zeros();
    F = ZeroVector(dim);

    assemble_NL(mu * stiff +  sigma * mass,
                mesh, fe, dof, A, U);

    /* BC handling */

    // Residual computation
    res = norm_2(F - A * U);

    /* System solution: Utilde = A^-1 F */

    // Solution update
    U = rho * U + (1 - rho) * Utilde;

    nit++;
} while( res >= tol & nit <= max_nit);
```

5.2 A Discontinuous Galerkin Computation

As an example of DG computation we considered the problem of the deformation and stretching of a spherical interface of radius $R = 0.2$ after a given constant vorticity field. In the level set framework (see [11] for an introduction) this amounts to solving the following linear hyperbolic problem:

$$\frac{\partial u}{\partial t} + \beta \cdot \nabla u = 0,$$

with:

$$\beta = \begin{pmatrix} \beta_x \\ \beta_y \\ \beta_z \end{pmatrix} = \begin{pmatrix} \sin^2(2\pi x)(\sin(2\pi z) - \sin(2\pi y)) \\ \sin^2(2\pi y)(\sin(2\pi x) - \sin(2\pi z)) \\ \sin^2(2\pi z)(\sin(2\pi y) - \sin(2\pi x)) \end{pmatrix}.$$

The spherical interface is embedded as the zero level set of function u, which, at time $t = 0$, is taken to be the signed distance function from the interface, namely:

$$u_0(r) = \sqrt{x^2 + y^2 + z^2} - R.$$

No boundary condition specification is needed since the normal component of the velocity is zero on the domain boundary, and hence no inflow boundary is present. In order to perform the computation, it was necessary to define proper volume and trace operators, whose overloading of `operator()` is reported below. Exploiting a similar strategy as the one outline above for the stabilization methods, we decided to incorporate the analytical velocity as the property `_u` of the classes.

```
// Pure advection operator

// 1: Volume contribution
Real AdvecDG::operator()(int i, int j,
                         int ic, int l) {
    Real s = 0.;

    Real x = _fe.quadPt(0, l);
    Real y = _fe.quadPt(1, l);
    Real z = _fe.quadPt(2, l);

    for(int ic = 0; ic < NDIM; ic++)
        s += _u(x, y, z, ic) *
             _fe->phiDer(i, ic, l) *
             _fe->phi(j, l);
    return s;
}

// 2: Internal face contribution
Real AdvecIF::operator()(int i, int j, int l,
                         int K, int H) {
    Real s = 0.;
    Real un = 0.;

    JumpIF jump(_fe);
    AverageIF avg(_fe);

    for(int ic = 0; ic < NDIM; ic++)
        un += _u(x, y, z, ic) *
              _fe->normal(ic, l);

    for(int ic = 0; ic < NDIM; ic++)
        s += _u(x, y, z, ic) *
             jump(i, ic, l, H) *
             avg(j, l, K) +
             un * jump(i, ic, l, H) *
             jump(j, ic, l, K);

    return s;
}

// 3: Boundary face contribution
```

```
Real AdvecBF::operator()(int i, int j, int l,
                         int K, int H) {
    Real s = 0.;
    Real un = 0.;

    JumpBF jump(_fe);

    for(int ic = 0; ic < NDIM; ic++)
        un += _u(x, y, z, ic) *
              _fe->normal(ic, l);

    if(un)
        for(int ic = 0; ic < NDIM; ic++)
            s += un * jump(i, ic, l, H) *
                 jump(j, ic, l, K);

    return s;
}
```

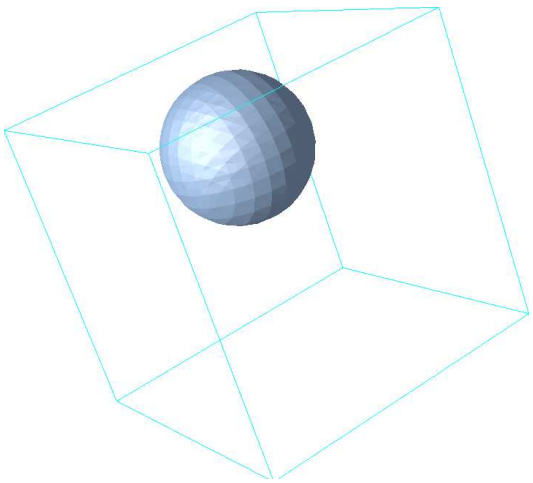
Assuming that a suitable velocity field `beta` has been defined, the `main()` reads:

```
// Types for EO incapsulation
typedef EOExpr<Real, AdvecDG> EOAdvecDG;
typedef EOExpr<Real, AdvecIF> EOAdvecIF;
typedef EOExpr<Real, AdvecBF> EOAdvecBF;

// Advection operators...
AdvecDG OadvecDG(&fe, beta);
AdvecIF OadvecIF(&fe, beta);
AdvecBF OadvecBF(&fe, beta);

// ...and their wrappings
EOAdvecDG advecDG(OadvecDG);
EOAdvecIF advecIF(OadvecIF);
EOAdvecBF advecBF(OadvecBF);

/* Definition of the BC handler BCh */

// Matrix assembly
assemble_DG(advecDG, advecIF, advecBF,
            mesh, BCh, fe, feIF, feBF,
            dof, sourcefct, A, F);

assemble(Mass, mesh, fe, dof, zero, M, F);

/* Problem solution */
```

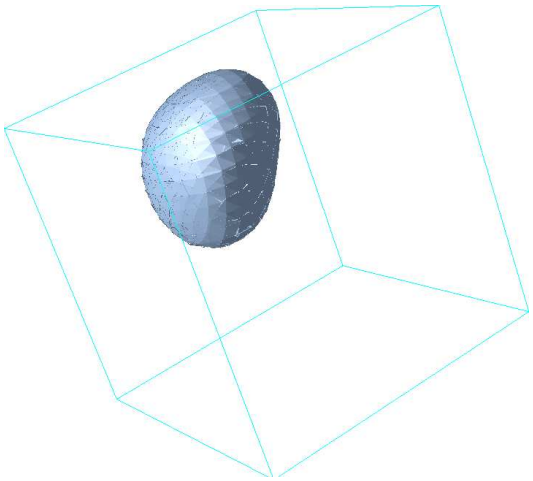
Again we would like to point out that the analytical advection field `beta` was provided in the constructor call when instantiating the operators. Some results are collected in Fig. 6 and Fig. 7.

6 Conclusions

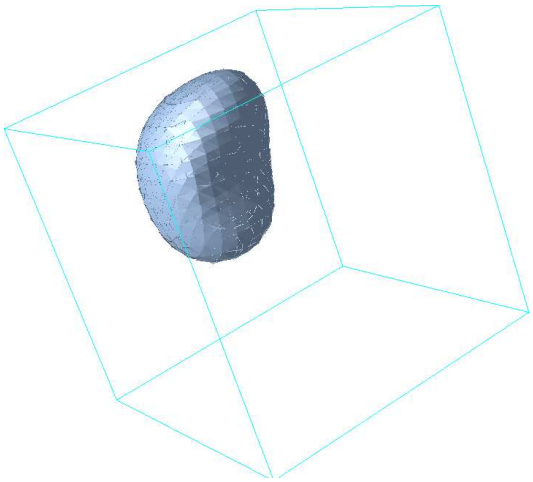
In this work we presented an Expression Templates implementation of the assembly step of a finite element computation. A suitable code design allows for great user-friendliness



(a) $t = 0$

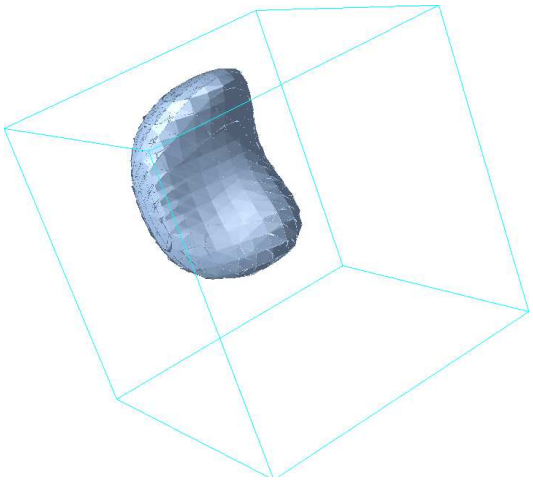


(b) $t = 0.25$

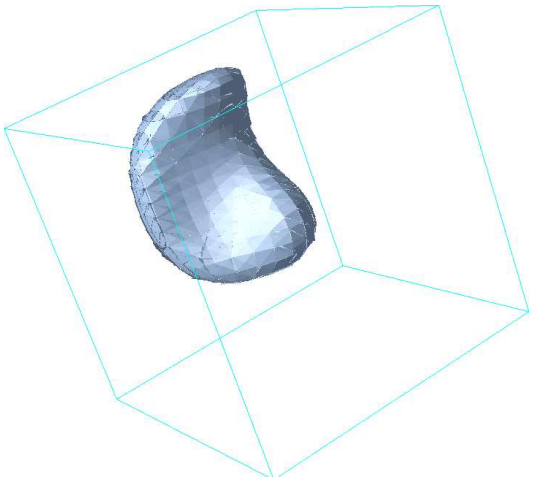


(c) $t = 0.5$

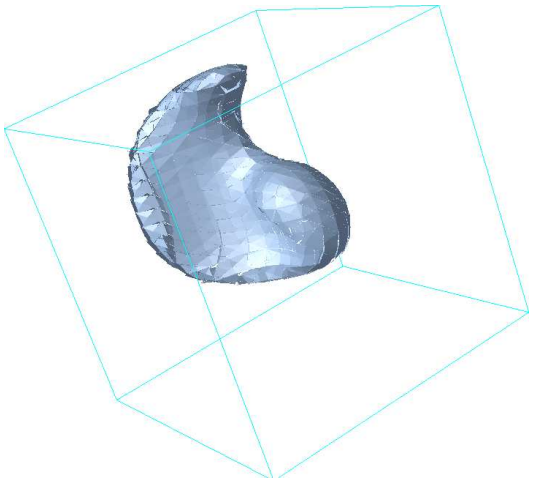
Fig. 6 Discontinuous Galerkin finite element solution sphere deformation problem: solution at different instants.



(a) $t = 0$



(b) $t = 0.25$



(c) $t = 0.5$

Fig. 7 Discontinuous Galerkin fi nite element solution sphere deformation problem: solution at different instants

and versatility of the code without abstraction penalty. Performance benchmark was provided by comparison with a more classical modular technique. Applications to conforming and discontinuous finite elements were considered, along with stabilized and non-linear problems and a number of examples were provided. In particular, the readibility and user easiness of the codes have been demonstrated on real interest problems.

We finally mention that the ideas in this paper have grown in the context of the development of an Object-Oriented Finite Element Library called `LifeV` (see `www.lifev.org`).

Acknowledgements

LifeV development joins researchers from EPF Lausanne, INRIA (Paris) and the MOX (Politecnico di Milano). The contributions of all the developers of the library are gratefully acknolewdged. We would like to mention, in particular, L. Formaggia, J.F. Gerbeau, M. Fernandez, A. Gauthier, D. Mastalli and C. Prud'homme.

References

1. D. N. ARNOLD, F. BREZZI, B. COCKBURN, AND D. MARINI, *Unifi ed analysis of discontinuous galerkin methods for elliptic problems*, SIAM J. Numer. Anal., 39 (2002), pp. 1749–1779.
2. E. AVGOUSTINIATOS AND C. COLTON, *Effect of external oxygen mass transfer resistances on viability of immunoisolated tissue*, Ann. NY Acad. Sci, 831 (1997), pp. 145–167.
3. B. COCKBURN AND C. W. SHU, *Runge-kutta discontinuous galerkin methods for convection-dominated problems*, J. Sci. Comp., 16 (2001), pp. 173–261.
4. G. FURNISH, *Disambiguated glommable expression templates*, Compuers in Physics, 11 (1997), pp. 263–269.
5. S. W. HANEY, *Beating the abstraction penalty in c++ using expression templates*, Computers in Physics, 10 (1996), pp. 552–557. Correction in Vol. 11,n. 14.
6. C. JOHNSON, *Numerical Solution of Partial Differential Equations by the Finite Element Method*, Cambridge University Press, Cambridge, 1987.
7. A. LANGER AND K. KREFT, *C++ expression templates*, C/C++ Users Journal, (2003).
8. H. LANGTANGEN, *Computational Partial Differential Equations*, Springer-Verlag, 1999.
9. C. PFLAUM, *Epression templates for partial differential equations*, Comp. Vis. Science, 4 (2001), pp. 1–8.
10. A. QUARTERONI AND A. VALLI, *Numerical approximation of partial differential equations*, no. 23 in Springer Series in Computational Mathematics, Springer-Verlag, Berlin, 1994.
11. J. A. SETHIAN, *Level set methods and fast marching methods, 2nd edition*, Cambdridge Press, New York, 1999.
12. B. STROUSTRUP, *The C++ Programming Language*, Addison Wesley Longman, Reading, MA, 2000.
13. T. VELDHUIZEN, *Expression templates*, C++ Report Magazine, 7 (1995), pp. 26–31. see also the web page `http://osl.iu.edu/ tveldhui` .