HAL
open science

Advances in Feel++: a domain specific embedded language in C++ for partial differential equations

Christophe Prud'Homme, Vincent Chabannes, Vincent Doyeux, Mourad Ismail, Abdoulaye Samake, Gonçalo Pena, Cécile Daversin, Christophe Trophime

To cite this version:

Christophe Prud'Homme, Vincent Chabannes, Vincent Doyeux, Mourad Ismail, Abdoulaye Samake, et al.. Advances in Feel++: a domain specific embedded language in C++ for partial differential equations. Eccomas'12 - European Congress on Computational Methods in Applied Sciences and Engineering, Sep 2012, Vienna, Austria. hal-00762442

HAL Id: hal-00762442

https://hal.science/hal-00762442

Submitted on 12 Dec 2012

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

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

European Congress on Computational Methods
in Applied Sciences and Engineering (ECCOMAS 2012)
J. Eberhardsteiner et.al. (eds.)
Vienna, Austria, September 10-14, 2012

ADVANCES IN FEEL++ : A DOMAIN SPECIFIC EMBEDDED LANGUAGE IN C++ FOR PARTIAL DIFFERENTIAL EQUATIONS

C. Prud'homme[1,2], V. Chabannes[1], V. Doyeux[3], M. Ismail[3], A. Samake[1], G. Pena[4], C. Daversin[5], C. Trophime[5]

[1] Laboratoire Jean Kuntzmann, Université Joseph Fourier Grenoble 1, BP53 38041 Grenoble Cedex 9, France, Tel.: +33476635497, Fax: +33476631263,
e-mail: christophe.prudhomme@ujf-grenoble.fr vincent.chabannes@imag.fr
abdoulaye.samake@imag.fr
[2] Université de Strasbourg / CNRS, IRMA / UMR 7501. Strasbourg, F-67000, France
[3]Université Grenoble 1 / CNRS, Laboratoire Interdisciplinaire de Physique / UMR 5588. Grenoble, F-38041, France
e-mail: vincent.doyeux@ujf-grenoble.fr mourad.ismail@ujf-grenoble.fr
[4] CMUC, University of Coimbra, Largo D. Dinis, Apartado 3008, 3001-454 Coimbra, Portugal,
e-mail: gpena@mat.uc.pt
[5] LNCMI-G, CNRS-UJF-UPS-INSA, 25 Av. des Martyrs, Grenoble, F-38042, France
e-mail: cecile.daversin@lncmi.cnrs.fr, christophe.trophime@lncmi.cnrs.fr

Keywords: Galerkin methods, finite element, parallel computing, domain specific embedded language, programming paradigms

Abstract. *This paper presents an overview of a unified seamlessly parallel framework for finite element and spectral element methods in 1D, 2D and 3D in C++ called* FEEL++. *The article provides a digression through the design of the library as well as the main abstractions handled by it, namely, meshes, function spaces, operators, linear and bilinear forms and an embedded variational language. In every case, the closeness between the language developed in* FEEL++ *and the equivalent mathematical objects is highlighted. More advanced mathematical methods e.g. the mortar, Schwartz (non)overlapping, three fields and two ficticious domain-like methods (the Fat Boundary Method and the Penalty Method) are described here [32].*

1 Introduction

Libraries to solve problems arising from partial differential equations (PDEs) through generalized Galerkin methods are a common tool among mathematicians and engineers. However, most libraries end up specializing in a type of equation, e.g. Navier-Stokes or linear elasticity models, or a specific type of numerical method, e.g. finite elements. The increasing complexity of differential models and the implementation of state of the art robust numerical methods, demand from scientific computing platforms general and clear enough languages to express such problems and provide a wealth of solution algorithms available in a minimal amount of code but maximum mathematical control. There are many freely available libraries which offer the capabilities described previously to a certain extent. To name a few: the Freefem software family [14, 31], the Fenics project [24, 25], Getdp [12] or Getfem++ [35], or libraries or frameworks such as deal.II (*C++*) [6], Sundance (*C++*) [26], Analysa (Scheme) [2]. Either they rely on a domain specific language (Python, the freefem language, ...) when it comes to describe the PDE to solve, or they are geometry or dimension dependent, or they are not so expressive with respect to the mathematics, i.e. the mathematics are hidden by programming details.

The library we present in this paper, called FEEL++, *Finite Element Embedded Language in C++*, see [33, 34] for the initial papers, provides also a clear and easy to use interface to solve complex PDE systems. It aims at bringing the scientific community a tool for the implementation of advanced numerical methods and high performance computing. Some recent applications of FEEL++ to multiphysics problems can be found in the literature, see [8, 11, 28–30].

Two main aspects in the design of the library are to (i) have the syntax, semantics and pragmatics of the library very close to the mathematics, and (ii) have a small manageable library that makes use wherever possible of established libraries (for linear system solves, for instance). While the first aims at creating a high level language powerful enough to describe solution strategies in a simple way, the second helps with the maintenance of the code delegating some procedures to frequently maintained third party libraries.

FEEL++ relies on a so-called *domain specific embedded language* (DSEL) designed to closely match the Galerkin mathematical framework. In computer science, DS(E)Ls are used to partition complexity and in our case the DSEL splits low level mathematics and computer science on one side leaving the FEEL++ developer to enhance them and high level mathematics as well as physical applications to the other side which are left to the FEEL++ user. This enables using FEEL++ for 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, provide only the relevant components required by the user or programmer and enforce the mathematical language computationnally between the users be they physicists, mathematicians, computer scientists, engineers or students. The DSEL approach has advantages over generating a specific *external* language: (i) interpreter/compiler construction complexities can be ignored, (ii) libraries can concurrently be used which is often not the case of specific languages which would have to also develop their own libraries and library system, (iii) DSELs inherit the capabilities of the host language (e.g.*C++*).

The DSEL on FEEL++ provides access to powerful, yet with a simple and seamless interface, tools such as interpolation or the clear translation of a wide range of variational formulations into the variational embedded language. Combined with this robust engine, lie also state of the art arbitrary order finite elements — including handling high order geomtrical approximations, — high order quadrature formulas and robust nodal configuration sets. The tools at the

user's disposal grant the flexibility to implement numerical methods that cover a large combination of choices from meshes, function spaces or quadrature points using the same integrated language and control at each stage of the solution process the numerical approximations.

Finally FEEL++ uses advanced C++ (e.g. template meta-programming) and in particular the latest standard C++ 11 that provides very useful additions such as type inference — `auto` and `decltype` keywords which are used throughout the paper. — FEEL++ also uses the essential Boost C++ libraries [1]: in this paper most scripts displayed use explicitely the Boost Parameter library[1] enabling a very powerful programming interface namely *named parameters* (required and optional) given in a random order to a class — template parameters —, a class member function or a free function. This library enhances tremendously readability, expressivity and ease of use of the code. Many other Boost libraries are used in FEEL++, some are listed in this document.

The paper describes an overview of the main abstractions and interfaces handled by the library. In section 2 we describe basic ideas behind the construction of the reference element, finite elements defined therein and the geometrical transformation. Section 3 shows how to handle and define meshes, function spaces, forms, functionals and operators. The variational embedded language is presented in section 4, with particular emphasis in the projection and integration procedures. Finally we present FEEL++ mesh adaptation interface to GMSH. Thanks to the language and computational framework, parallel computing using MPI is completely seamless and most of FEEL++ examples can be run either in sequential or parallel without the user having to do anything about MPI. More advanced examples using the mortar, Schwartz (non)overlapping, three fields, and ficticious domain-like methods to illustrate the flexibility of the language are available in [32].

2 Polynomial Library

The polynomial library is composed of various bricks: *(i)* the geometrical entities or convexes *(ii)* the prime basis in which we express subsequently the polynomials, *(iii)* the definition and construction of point sets in convexes (such as quadrature point sets) and finally *(iv)* polynomials and finite elements.

2.1 Convexes

The supported convexes are simplices and hypercubes of topological dimension n, $n = 1,2,3$ lying in $\mathbb{R}^d$ such that $n \leqslant d \leqslant 3$. The convexes are described geometrically in a standard way in terms of their subentities (vertices, edges, faces, volumes), see for example [20], and provide the ability to iterate over the entities of a convex or of the same topological dimension inside a convex, e.g. iterate over the edges of a tetrahedron.

2.2 Prime basis: L^2 Orthonormal Polynomials

In order to express polynomials in the convexes defined previously, we need to choose a *prime basis*, i.e., a basis in which all polynomial families are expressed. Often, the choice falls on the canonical basis (also known as the *moment* or *monomial basis*). However, recent work by R.C. Kirby [21–23] proposed to use the Dubiner polynomials as a prime basis on the simplex. We extended these ideas on the hypercubes using the Legendre polynomials. Other interesting examples of prime basis being used are the Bernstein polynomials. Our framework uses the

[1] `http://www.boost.org/doc/libs/1_50_0/libs/parameter/doc/html/index.html`

Dubiner or Legendre basis as the default prime basis. This choice simplifies the construction of finite elements due to the hierarchical and L^2 orthogonality properties these basis functions share. The choice of basis polynomials that are hierarchical allows for an easy extraction of a basis spanning a subspace of the polynomial space (which corresponds to extract a range of coefficients), whereas L^2 orthogonality simplifies some operations like numerical integration or the L^2 projection (which is explicit in this case). The use of these basis functions proved to provide much better numerical stability, see [28].

Details on the construction of the Dubiner polynomials can be found in [20] page 101. In practice, the prime basis is normalized.

2.3 Point Sets on Convexes

Now we turn to the construction of point sets $\mathbb{P}$ defined on a convex K. Point sets are represented algebraically by a matrix (rows are indexed by the coordinates while columns are indexed by the points) and they are parametrized by the associated convex and the numerical type. We recall that the convex is decomposed in vertices, edges, faces, volumes. A similar decomposition is done for the point sets: points are constructed and associated to their respective entities on which they reside. This is crucial when considering continuous and discontinuous Galerkin formulations.

The type of point sets supported are *(i)* the Equidistributed point set, *(ii)* the Warpblend point sets on simplices see [36], *(iii)* Fekete points in simplices, see [20], *(iv)* standard quadrature rules in simplices and finally *(v)* Gauss, Gauss-Radau and Gauss-Lobatto and combinations in simplices and hypercubes. It should be noted that the last family is constructed from the computation of the zeros of the Legendre polynomials on $[-1,1]$ including eventually the boundary vertices -1, 1 for the Radau and Lobatto flavors.

Warpblend and Fekete points are used with nodal basis on simplices which, when constructed at these points, present much better interpolation properties (lower Lebesgue constant, see [20]). Note that the Gauss-Lobatto points are the Fekete points in hypercubes.

2.4 Polynomial Set

After introducing in the previous sections the necessary bricks to the construction of polynomials on simplices and hypercubes, we now focus on the polynomial abstraction.

A polynomial set $\mathbb{P}$ is a template class parametrized by the prime basis in which it is expressed and the field type in which it has its values: scalar, vectorial or matricial. Its interface provides a number of operations such as evaluation and derivation at a set of points, extraction of polynomials or components (when the `FieldType` is `Vectorial` or `Matricial`) of a polynomial from a polynomial set .

One critical operation is the construction of the gradient of a polynomial (or a polynomial set) expressed in the prime basis. This usually requires solving a linear system where the matrix entries are given by the evaluation of the prime basis and its derivatives at a set of points. Again the choice of set of points is crucial here to avoid ill-conditioning and loss of accuracy. We choose Gauss-Lobatto points for hypercubes and Warpblend or Fekete points for simplicies as they provide a much better conditioning for the underlying system matrix (a generalized Vandermonde matrix, see [28]).

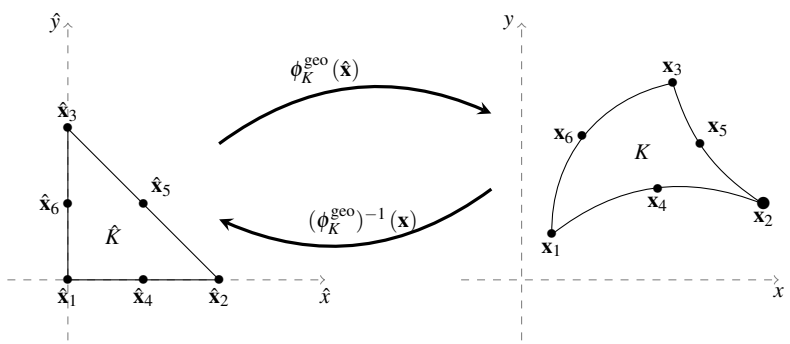
2.5 Finite Elements and Other Polynomial Basis

FEEL++ supports modal basis, e.g. Legendre or Dubiner, see [7, 20], as well as finite elements (FE) following the standard definition, set in [9], as a triplet $(K, \mathbb{P}, \Sigma)$ where K is a convex, $\mathbb{P}$ the polynomial space and Σ the dual space. We describe now some features of the finite element framework. The description of K and $\mathbb{P}$ has been presented previously and it remains to describe Σ. Σ is a set of functionals (which can be identified as degrees of freedom) defined in $\mathbb{P}$ with values in $\mathbb{R}$, $\mathbb{R}^d$ or $\mathbb{R}^{d\times d}$. Several types of functionals can then be instantiated which merely require basic operations like evaluation at a set of points, derivation at a set of points, exact integration or numerical integration. Some examples of functionals satisfying such requirements are 1. evaluation at a point $x \in K$, $\ell_x : p \to p(x)$, 2. derivation at a point $x \in K$ in the direction i, $\ell_{x,i} : p \to \frac{\partial p}{\partial x_i}(x)$, 3. moment integration associated with a polynomial $q \in \mathbb{P}(K)$, $\ell_q : p \to \int_K pq$.

A functional is represented algebraically by a vector whose entries result from the application of the functional to the prime basis in which we express the polynomials thanks to the bijection between $\mathscr{L}(\mathbb{P}, \mathbb{R})$ and $\mathbb{R}^{\dim(\mathbb{P})}$. Then applying the functional to a polynomial is just a scalar product between the coefficient of this polynomial in the prime basis by the vector representing the functional. For example the *Lagrange element* is the finite element $(K, \mathbb{P}, \Sigma = \{\ell_{x_i}, x_i \in X \subset K\})$ such that $\ell_{x_i}(p_j) = \delta_{ij}$ where p_j is a Lagrange polynomial and $X = \{x_i\}$ is a set of points defined in the convex K, for example the Equidistributed, Warpblend or Fekete point sets. Other FE such as $\mathbb{P}_{1,2}$-bubble, $\mathbb{RT}_k$ or $\mathbb{N}_k$ polynomials are constructed likewise though they require a more involved description.

2.6 Geometry

To conclude this section, one important object that is constructed with the help of the polynomial library is the *geometric transformation*. Indeed all polynomial set constructions are done on a reference convex, denoted $\hat{K}$, and the geometrical transformation maps it to a convex in the physical space which we denote K. This map, denoted ϕ_K^{geo}, is the C^1-diffeomorphism defined on $\hat{K} \subset \mathbb{R}^p, p = 1, 2, 3$ such that the image is $K \subset \mathbb{R}^d$, i.e. $\phi_K^{\text{geo}} : \hat{K} \longrightarrow K$ for $p \leqslant d \leqslant 3$. This map is contructed and associated to each convex K in a computational mesh $\mathscr{T}_h$, see section 3. Notice that this last condition over p and d covers a large spectrum of geometrical profiles. For instance, we handle lines or surfaces in $\mathbb{R}^3$. We refer the reader to section **??** for an example where this flexibility is exploited.



The geometric transformation is constructed as a suitable linear combination of Lagrange polynomials and therefore it can be a polynomial of arbitrary degree, allowing thus meshes

with elements that have curved edges/faces, see [29, 30]. Another consequence of ϕ_K^{geo} being a polynomial of a degree the user can choose, is the possibility to define isoparametric (or subparametric or surparametric) finite elements, see [28, 30]. Lets denote k_{geo} the polynomial order of the Lagrange basis in which ϕ_K^{geo} is expanded. If there is no ambiguity, we keep the notation ϕ_K^{geo}, otherwise we use the notation $\phi_{K,k_{\text{geo}}}^{\text{geo}}$.

The class that implements the definition and evaluation of the geometrical transformation also provides a function to evaluate its gradient, automatic consequence of ϕ_K^{geo} being an element belonging to a polynomial set. Another important transformation associated with ϕ_K^{geo} is its inverse, $(\phi_K^{\text{geo}})^{-1}$. In the case of an affine transformation, the inverse is calculated explicitely. However, if ϕ_K^{geo} is nonlinear, the evaluation/differentiation of $(\phi_K^{\text{geo}})^{-1}$ at a set of points is performed with the help of a nonlinear solver (we have used the nonlinear solver available in `PETSc` for these calculations, see section 3.2). The inverse transformation plays an essential role in providing an interpolation tool, all the advanced numerical methods presented in [32] use this tool and hence the inverse geometrical transformation.

3 Meshes, Function Spaces and Operators

In the previous section, we have described roughly the monodomain construction of polynomials. Now we turn to the ingredients for the multidomain construction and we start with the mesh mathematical description and associated data structures. Note that all these data structures presented are parallel (MPI) and work seamlessly in sequential and parallel computational environments.

3.1 Mesh Data Structures

Let $\Omega \subset \mathbb{R}^d$, $d \geq 1$, denote a bounded connected domain. We first need to introduce a suitable discretization of Ω, $\Omega_h \subset \Omega$. Note that if Ω is a polyhedral domain then $\Omega_h = \Omega$. We denote by $\mathscr{T}_h$ a finite collection of nonempty, disjoint open simplices or hypercubes $\mathscr{T}_h = \{K = \phi_K^{\text{geo}}(\hat{K})\}$ forming a partition of Ω_h such that $h = \max_{K \in \mathscr{T}_h} h_K$, with h_K denoting the diameter of the element $K \in \mathscr{T}_h$. We say that a hyperplanar closed subset F of $\overline{\Omega}$ is a mesh face if it has positive $(d-1)$-dimensional measure and if either there exist $K_1, K_2 \in \mathscr{T}_h$ such that $F = \partial K_1 \cap \partial K_2$ (and F is called an *internal face*) or there exists $K \in \mathscr{T}_h$ such that $F = \partial K \cap \partial \Omega_h$ (and F is called a *boundary face*). Internal faces are collected in the set $\mathscr{F}_h^i$, boundary faces in $\mathscr{F}_h^b$ and we let $\mathscr{F}_h := \mathscr{F}_h^i \cup \mathscr{F}_h^b$. For all $F \in \mathscr{F}_h$, we define $\mathscr{T}_F := \{K \in \mathscr{T}_h \mid F \subset \partial K\}$. For every interface $F \in \mathscr{F}_h^i$ we introduce two associated normals to the elements in $\mathscr{T}_F$ and we have $\mathrm{n}_{K_1,F} = -\mathrm{n}_{K_2,F}$, where $\mathrm{n}_{K_i,F}$, $i \in \{1,2\}$, denotes the unit normal to F pointing out of $K_i \in \mathscr{T}_F$. On a boundary face $F \in \mathscr{F}_h^b$, $\mathrm{n}_F = \mathrm{n}_{K,F}$ denotes the unit normal pointing out of Ω_h. We also introduce (i) the set of boundary elements $\mathscr{T}_h^b = \{K \in \mathscr{T}_h \mid \partial K \cap \partial \Omega \neq \emptyset\}$, (ii) the set of internal elements $\mathscr{T}_h^i = \mathscr{T}_h \backslash \mathscr{T}_h^b$, (iii) the set $\mathscr{N}_h$ which collects the nodes of the mesh, (iv) when $d = 3$, $\mathscr{E}_h$ which collects the edges of the mesh.

The collections $\mathscr{T}_h, \mathscr{F}_h, \mathscr{E}_h, \mathscr{N}_h$, as well as the internal and boundary collections, are provided by our mesh data structure and stored using the the Boost.Multi_index library[2]. The mesh entities (elements, faces, edges, nodes) are indexed either by their ids, the process id (i.e. the id given by MPI in a parallel context, by default the current process id) to which they belong, their markers (material properties, boundary ids...) or their location (whether the entity is internal or lies on the boundary of the domain). Other indices could certainly be defined, however those

[2] `http://www.boost.org/libs/multi_index/doc/index.html`

previous four already allow a wide range of applications. Thanks to Boost.Multi_index, it is trivial to retrieve pairs of iterators over the entity's 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, see section 4.1, and projection, see section 4.2, tools. Table 1 summarizes some of the available ranges in the library.

Range iterators	Description
`elements(<mesh>)`	range iterator over $\mathscr{T}_h$
`faces(<mesh>)`	range iterator over $\mathscr{F}_h$
`edges(<mesh>)`	range iterator over $\mathscr{E}_h$
`points(<mesh>)`	range iterator over $\mathscr{N}_h$
`markedelements(<mesh>, <marker (id|string)>)`	element range iterator over $\mathscr{T}_h$ marked by `marker`
`markedfaces(<mesh>, <marker (id|string)>)`	face range iterator over $\mathscr{T}_h$ marked by `marker`
`boundaryelements(<mesh>)`	element range iterator over $\mathscr{T}_h^b$
`internalelements(<mesh>)`	element range iterator over $\mathscr{T}_h^i$
`boundaryfaces(<mesh>)`	face range iterator over $\mathscr{F}_h^b$
`internalfaces(<mesh>)`	face range iterator over $\mathscr{F}_h^i$

Table 1: Some mesh range iterators

In *C++* the mesh data structure is defined through the type of geometrical entities (simplex or hypercube) and the geometrical transformation associated, see the listing below.

```
// 𝒯_h is a collection of simplices s.t. φ^geo_{K,k_geo} : K̂ ⊂ ℝ^p ⟶ K ⊂ ℝ^d
Mesh<Simplex<p, k_geo, d> > mesh;
// same as above except that we deal with a set of hypercubes
Mesh<Hypercube<p, k_geo, d> > mesh;
```

FEEL++ uses GMSH, see [13], to generate meshes in all 3 dimensions with $1 \leqslant k_{\text{geo}} \leqslant 5$ for $d = 2$ and $1 \leqslant k_{\text{geo}} \leqslant 4$ for $d = 3$. The listing below and the resulting Figure 1 for $k_{\text{geo}} = 1, 2, 3, 4$ illustrate the flexibility of FEEL++ regarding mesh handling.

```
typedef Mesh<Simplex<3,k_geo,3> > mesh_type;
// generate the mesh of the sphere using Gmsh
auto mesh = createGMshMesh( _mesh=new mesh_type,
                            _desc=domain(_shape="ellipsoid",_dim=3));
// generate a functionspace, see next section
FunctionSpace<mesh_type,bases<Lagrange<k_geo> > > Xh_type;
auto Xh = Xh_type::New( mesh );
// build the Lagrange interpolant u of degree k_geo of cos(5x)sin(5y)
auto u = project( _space=Xh, _range=elements(mesh), _expr=cos(5x)sin(5y) );
// export function to gmsh the mesh and u for high order visualisation
// gmsh requires that we use isoparametric elements
exporter( _format=gmsh,
```

[3] http://www.boost.org/libs/range/index.html

```
            _mesh=mesh,
            _name="ho_sphere" ).add( _data=u, _name="u" );
```

Finally the mesh data structure of topological dimension d allows to easily extract submeshes of topological dimension d and $d-1$. For example, the methods exposed in section **??** use the extraction of trace meshes as follows:

```
// Trace mesh composed of all boundary faces from 'mesh'
auto trace_mesh = mesh->trace(boundaryfaces(mesh));
// mesh of faces marked Gamma in 'mesh'
auto Gamma_mesh = mesh->trace(markedfaces(mesh,"Gamma"));
```

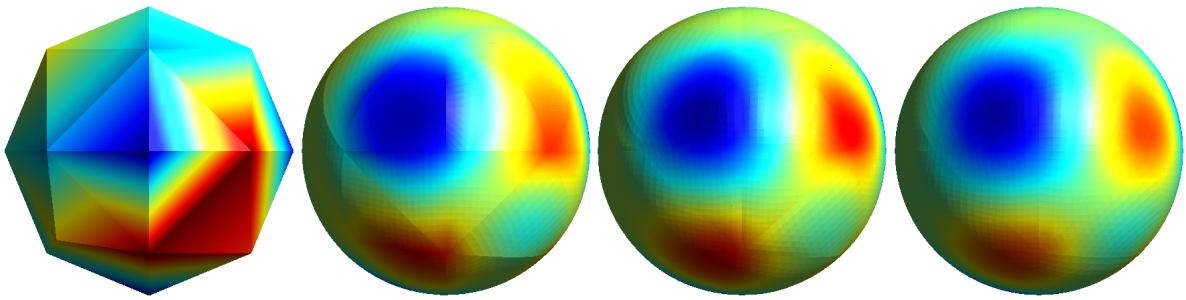


Figure 1: Sphere discretized with 31 elements. Visualization of a function in Gmsh using tetrahedral elements of order $N = 1,2,3,4$.

3.2 Algebraic representations

Algebraic representations are handled using a so-called *backend* which is a wrapper class that encapsulates several linear, nonlinear and eigenvalue algorithms as well as data structures like vectors and matrices. It provides all the algebraic data structure behind function spaces, operators and forms (see the following sections). In the case of linear functionals (also called linear forms or 1-forms), the representation is a vector and, in the case of linear operators and bilinear forms, the representation is a matrix.

The *backend* abstraction allows to write code that is independent of the libraries used in the assembly process or to solve the linear systems involved, thus hiding all the details of that algebraic part under the hood of the backend. Once a backend is set, the user can create in a transparent way vectors and matrices to be used during the assembly process, as is it shown is the following listing.

```
  // allocates a sparse matrix based on the degrees of freedom of the
  // trial functions space Xh and the test functions space Vh
  auto A = backend->newMatrix( _trial=Xh, _test=Vh );

  // allocates a vector based on the degrees of freedom of the
  // test functions space Vh
  auto b = backend->newVector( _test=Vh );
```

There are two available backends that provide an interface to `PETSc/SLEPc`, see [3–5, 15], and `Trilinos`, see [16–19]. The user can choose any of them, bearing in mind the tools and features available in each, such as parallelism, direct or iterative linear system solvers or preconditioners for these systems. Once a backend is defined, say `Backend<Trilinos>`,

the user is free to manipulate objects such as vectors and matrices (from the `Epetra` class, in this case), using most of the algebraic operations attainable from the original library. The backend also provides interfaces to linear system solvers (direct or iterative, with or without a preconditioner). The syntax for this function is illustrated in the next listing. Similar interfaces exist for non-linear and standard/generalized eigenvalue solvers.

```
// solve the linear system Au = F
backend -> solve( _matrix=A , _solution=u , _rhs=F );
```

3.3 Function Spaces and Functions

We now turn to the next crucial mathematical ingredient: the function space, whose definition depends on Ω_h — or more precisely its partitioning $\mathscr{T}_h$ — and the choice of basis function. Function spaces in FEEL++ follow the same definition, see listing 1, and FEEL++ provides support for continuous and discontinuous Galerkin methods and in particular approximations in L^2, H^1-conforming and H^1-nonconforming, H^2, $H(\text{div})$ and $H(\text{curl})$[4].

Listing 1: FunctionSpace

```
// space of continuous piecewise
// ℙ3 functions defined on a mesh
// of order 2 triangles in 3D
FunctionSpace<Mesh<Simplex<2,2,3>,
              bases<Lagrange<3> > > Xh;
```

The `FunctionSpace` class (i) constructs the table of degrees of freedom which maps local (elementwise) degrees of freedom to the global ones with respect to the geometrical entities, (ii) embeds the definition of the elements of the function space allowing for a tight coupling between the elements and their function spaces, (iii) stores an interpolation data structure (e.g. region tree) for rapid localisation of point sets (determining in which element they reside).

We introduce the following spaces

$$
\begin{aligned}
\mathbb{W}_h &= \{v_h \in L^2(\Omega_h) : \ \forall K \in \mathscr{T}_h, v_h|_K \in \mathbb{P}_K\}, \\
\mathbb{V}_h &= \mathbb{W}_h \cap C^0(\Omega_h) = \{v_h \in \mathbb{W}_h : \ \forall F \in \mathscr{F}_h^i \ [\![v_h]\!]_F = 0\} \\
\mathbb{H}_h &= \mathbb{W}_h \cap C^1(\Omega_h) = \{v_h \in \mathbb{W}_h : \ \forall F \in \mathscr{F}_h^i \ [\![v_h]\!]_F = [\![\nabla v_h]\!]_F = 0\} \\
\mathbb{CR}_h &= \{v_h \in L^2(\Omega_h) : \ \forall K \in \mathscr{T}_h, v_h|_K \in \mathbb{P}_1; \forall F \in \mathscr{F}_h^i \ \int_F [\![v_h]\!] = 0\} \\
\mathbb{R}\eth\mathbb{T}\cong_h &= \{v_h \in L^2(\Omega_h) : \ \forall K \in \mathscr{T}_h, v_h|_K \in \text{span}\left\{1, x, y, x^2 - y^2\right\}; \forall F \in \mathscr{F}_h^i \ \int_F [\![v_h]\!] = 0\} \\
\mathbb{RT}_h &= \{\boldsymbol{v}_h \in [L^2(\Omega_h)]^d : \ \forall K \in \mathscr{T}_h, v_h|_K \in \mathbb{RT}_k; \forall F \in \mathscr{F}_h^i \ [\![\boldsymbol{v}_h \cdot \mathrm{n}]\!]_F = 0\} \\
\mathbb{N}_h &= \{\boldsymbol{v}_h \in [L^2(\Omega_h)]^d : \ \forall K \in \mathscr{T}_h, v_h|_K \in \mathbb{N}_k; \forall F \in \mathscr{F}_h^i \ [\![\boldsymbol{v}_h \times \mathrm{n}]\!]_F = 0\}
\end{aligned}
\tag{1}
$$

where $\mathbb{RT}_k$ and $\mathbb{N}_k$ are respectively the Raviart-Thomas and Nédélec finite elements of degree k. The table 2 summarizes the supported approximation spaces.

The Legrendre and Dubiner basis yield implicitely discontinuous approximations, the Legendre and Dubiner boundary adapted basis, see [20], were designed to handle continuous approximations whereas the Lagrange basis can yield either discontinuous or continuous (default

[4] At the time of writing, H^2, $H(\text{div})$ and $H(\text{curl})$ approximations are in experimental support.

Continuous Solution	Discrete Approximation	Basis	Order	Dimension
L^2	$\mathbb{W}_h$	Lagrange Legendre Dubiner	any	1,2,3
H^1-conforming	$\mathbb{V}_h$	Lagrange Legendre boundary adapted Dubiner boundary adapted	any	1,2,3
H^1-nonconforming	$\mathbb{CR}_h$ $\mathbb{R}\eth\mathbb{T}\approxeq_h$	Crouzeix-Raviart Rannacher-Turek	1 2	2,3 2
$H(\text{div})$-conforming	$\mathbb{RT}_h$	Raviart-Thomas	any	2,3
$H(\text{curl})$-conforming	$\mathbb{N}_h$	Nédélec first kind	any	2,3
H^2-conforming	$\mathbb{H}_h$	Hermite	$\geqslant 2$	1,2,3

Table 2: FEEL++ approximations spaces

behavior) approximations. $\mathbb{RT}_h$ and $\mathbb{N}_h$ are implicitely spaces of vectorial functions $\boldsymbol{f}$ s.t. $\boldsymbol{f} : \Omega_h \subset \mathbb{R}^d \mapsto \mathbb{R}^d$. As to the other basis functions, i.e. Lagrange, Legrendre, Dubiner, etc., they are parametrized by their values namely `Scalar`, `Vectorial` or `Matricial`. Note that `FunctionSpace` handles also products of function spaces. This is very powerful to describe complex multiphysics problems when coupled with operators, functionals and forms described in the next section. Extracting subspaces or component spaces are part of the interface.

```
// continuous piecewise P3
// approximations
FunctionSpace<Mesh<Simplex<2> >,
  bases<Lagrange<3,Scalar,
                 Continuous> > > P3ch;
// discontinuous piecewise P3
// approximations
FunctionSpace<Mesh<Simplex<2> >,
  bases<Lagrange<3,Scalar,
                 Discontinuous> > > P3dh;
// mixed (P2 vectorial, P1 scalar,
// P1 Scalar) approximation
FunctionSpace<Mesh<Simplex<2»,
 bases<Lagrange<2,Vectorial>,
       Lagrange<1,Scalar>,
       Lagrange<1,Scalar»> P2P1P1;
```

The most important feature in `FunctionSpace` is that it embeds the definition of element which allows for the strict definition of an `Element` of a `FunctionSpace` and thus ensures the correctness of the code. An element has its representation as a vector — also in the case of product of multiple spaces. — The vector representation is parametrized by one of the linear algebra backends presented in section 3.2. Other supported operations are interpolation and extraction of components — be it a product of function spaces element or a vectorial/matricial

element, — see listing 3.3.

```
FunctionSpace<Mesh<Simplex<2> >,
  bases<Lagrange<3,Scalar, Continuous> > > P3ch;
// get an element from P3ch
auto u = P3ch.element();
FunctionSpace<Mesh<Simplex<2> >,
 bases<Lagrange<2,Vectorial>, Lagrange<1,Scalar>,
       Lagrange<1,Scalar> > > P2P1P1;
auto U = P2P1P1.element();
// Views: changing a view changes U and vice versa
// view on element associated to P2
auto u = U.element<0>();
// extract view of first component
auto ux = u.comp(X);
// view on element associated to 1st P1
auto p = U.element<1>();
// view on element associated to 2nd P1
auto q = U.element<2>();
```

Finally FEEL++ provides the Lagrange, $\mathscr{I}_{c,h}^{\mathrm{LAG}}, \mathscr{I}_{d,h}^{\mathrm{LAG}}$, Crouzeix-Raviart, $\mathscr{I}_h^{\mathrm{CR}}$, Raviart-Thomas, $\mathscr{I}_h^{\mathrm{RT}}$ and Nédélec, $\mathscr{I}_h^{\mathrm{N}}$ global interpolation operators. In abstract form, they read

$$\mathscr{I} : \mathbb{X} \ni v \mapsto \sum_{i=1}^{\dim \mathbb{X}} \ell_i(v)\phi_i \tag{2}$$

where $\mathbb{X}$ is the infinite dimensional space, $(\ell_i)_{i=1,\dots,\dim\mathbb{X}}$ are the linear forms and $(\phi_i)_{i=1\dots\dim\mathbb{X}}$ the basis function associated with the various approximations in table 2.

3.4 Functionals, Operators and Forms

We now introduce the necessary functional vocabulary, namely (i) functionals, $\ell : \mathbb{X} \mapsto \mathbb{R}$, (ii) operators, $\mathscr{A} : \mathbb{X} \mapsto \mathbb{Y}$ and (iii) bilinear forms, $a : \mathbb{X} \times \mathbb{Y} \mapsto \mathbb{R}$. Note that for a bilinear form $a : \mathbb{X} \times \mathbb{Y} \mapsto \mathbb{R}$ there exists a unique operator $\mathscr{A} : \mathbb{X} \mapsto \mathbb{Y}'$ s.t. $a(x,y) = \langle \mathscr{A}x, y\rangle$ where $\langle \cdot, \cdot \rangle$ denotes the duality product. The C++ counterpart of these mathematical objects follows closely their mathematical analog: they are template classes with arguments being the space (or product of spaces) they take as input and provide an interface to apply them on elements of the domain space. For operators, we can also apply their inverse to the elements of the image space. In the linear case, we provide also an algebraic representation : a vector for linear functionals or forms and a matrix for linear operators and bilinear forms — matrix- and vector-free representations are possible too. — The listing below excercises

```
auto Xh = Xh_type::New(mesh); Vh = Vh_type::New(mesh);
auto u = Xh->element(); auto v = Vh->element();
// operator T : Xh → Vh'
auto T = opLinear( _domainSpace=Xh, _imageSpace=dual(Vh)
                   [,_backend=backend] );
// definition of T, see eg section 4
T = integrate(_range=elements(mesh),_expr=gradt(u)*trans(grad(v)));
// linear functional f : Vh → ℝ
// f is a linear form
```

```
auto f = T.apply( u ); f.apply( v );
//or equivalently
auto f = funLinear( _domainSpace=V_h [, _backend=backend] );
f = T.apply(u); f.apply( v ) ;
// a is a bilinear form,
auto A=backend->newMatrix( _test=X_h, _trial=X_h );
auto a = form2( _test=X_h, _trial=X_h, _matrix=A );
// definition of a, see e.g. section 4
a = integrate(_range=elements(mesh),_expr=idt(u)*id(v));
// b is a linear form,
auto B=backend->newVector( _test=X_h );
auto b = form1( _test=X_h, _vector=B );
// definition of b, see e.g. section 4
b = integrate(_range=elements(mesh),_expr=id(v));
```

FEEL++ implements currently the following operators (i) interpolation operator, (ii) the lift and trace operators, and (iii) the projection operators (L^2, H^1, $H(\text{div})$, $H(\text{curl})$,...). We describe briefly the interpolation operator which is used later in this text. Note that the lift and trace operators are used also in some examples in section **??**.

The interpolation operator, $\mathscr{I} : X \longrightarrow Y$ where Y is a nodal basis and $\mathscr{I}(v)$, $v \in X$ is the interpolant of $v \in Y$, defined in the previous section. It is based on two fundamental tools: a localisation tool using a kd-tree data structure for fast localisation and the inverse geometrical transformation. This is a crucial tool for all advanced numerical methods presented later in this paper in [32].

Listing 2: Interpolation operator

```
auto X_h = X_h_type::New(mesh); V_h = V_h_type::New(mesh);
auto u = X_h->element(); auto v = V_h->element();
// operator I : X_h -> V_h
auto I = opInterpolation( _domainSpace=X_h, _imageSpace=V_h
                          [,_backend=backend] );
// compute v the interpolant of u in V_h
v = I(u);
```

4 A Variational Formulation Language Embedded in *C++*

The language has a variety of keywords — see appendix in [32] for an exhaustive list — that implement (i) access to geometrical data structure, e.g. exterior normal to the face of an element, (ii) standard mathematical functions and logical relations, (iii) operations related with the function spaces, such as element evaluation or differentiation, (iv) numerical integration. These keywords, with the help of algebraic operations such as sum, multiplication, division, etc, allow to build expressions which are used to define mathematical expressions then used by various objects such as forms or operators.

Some keywords show a common pattern in their nomenclature: for instance, the basic expressions `id(p)`, `grad(p)`, `jump(p)` or `div(p)` represent the evaluation, gradient, jump across a face or divergence of a test function, while the addition of `t` at the end of such expressions (`idt(p)`, `gradt(p)`, `jumpt(p)` or `divt(p)`) represents the same operators, but now concerning trial functions. Therefore, if `p` and `q` belong to the same functionspace, the construction of a bilinear form associated with the expression `idt(p)*id(q)` will result in

the assembly of the associated mass matrix. However, adding a `v` at the end of the keywords, allows to perform evaluation of the operator applied to the argument, ie, `idv(p)`, `gradv(p)`, `jumpv(p)` or `divv(p)` evaluate respectively the gradient of `p`, the jump of `p` across faces or the divergence of `p`.

FEEL++ supports scalar, vectorial and matricial operations in expressions. Indeed the *C++* operators follow the same rank semantics of their mathematical counterparts, see appendix in [32]. Listing 3 displays a *C++* code solving the linear elasticity equations, notice how the vectorial notations make it easy to express the variational formulation.

Often it is the case that we must integrate function space elements or basis functions (trial or test functions) that are defined on a mesh which is different from the integration mesh, see e.g. the mortar method or the Fat Boundary Method in [32]. Such a case is detected automatically and the interpolation procedure is called in the background of the integration or projection processes automatically to handle the necessary information transfer. All possible cases are actually handled by FEEL++.

Finally the variational language, integration and projection are parallel. There are still some complex operations that are not supported in parallel like handling fictitious domain methods such as the FBM, see [32]. Most standard methods are supported and already some complex FEEL++ applications are done in parallel such as Fluid Structure Interaction, see [8].

4.1 Integration

Integration is of course a fundamental tool in FEEL++. It is handled through the keyword `integrate` and is used to define forms, functionals and operators or just to compute integrals. It provides an extremely flexible and versatile interface in all these contexts:

```
integrate( _range=<mesh range iterators>, // integration domain
           // integrand
           _expr=<expression to integrate>,
           // quadrature
           _quad=_Q<polynomial order  to integrate exactly>(),
   ... // other parameters are available but not necessary here);
```

The parameters `_range` (domain of integration) and `_expr` (integrand) are required while `_quad` is optional. Indeed, by default, the DSEL computes an approximation of the polynomial order of the integrand and selects the quadrature that integrates it exactely (if the integrand is polynomial then the integration is exact). If this is not statisfactory then the user selects his own quadrature. Multiple `integrate` keywords can be accumulated using the + operator and returns an object which is handled automatically by forms, functionals and operators. There are two particular flavors: (i) integral evaluation which requires to use the `evaluate()` member function of the object returned by `integrate()` (ii) elementwise integral evaluation which requires to use of the `broken(<space>)` member function to return an element of `<space>` (typically a space of piecewise constant functions) containing the resulting integral computations.

```
typedef FunctionSpace<Mesh<Simplex<3>,
              bases<Lagrange<0,Scalar,Discontinuous> > > P0h_type;
auto P0h = P0h_type::New( mesh )
// evaluate ∫_{Ω_h} sin x = Σ_{K∈T_h} ∫_K sin x
auto v = integrate( _range=elements(mesh),
```

```
                    _expr=sin( Px() ) ).evaluate();
// store for all K ∈ 𝒯_h, ∫_K sin x in p0[ι_K] where ι_K is the
// index of element K
auto p0 = integrate( _range=elements(mesh),
                     _expr=sin( Px() ) ).broken( P0h );
for( auto ι_K : mesh.elements() ) {
  // print the element wise integrals
  std::cout << "p0["<< ι_K << "]=" << p[ι_K] << "\n";
}
```

Note that there is a special keyword in the context of bilinear forms and linear operators used conjointly with `integrate`, it is `on` which allows to eliminate Dirichlet "degrees of freedom".

4.2 Projection

Another important keyword is `project` which allows to compute the interpolant of an expression with respect to a nodal function space over a part of the mesh or the whole mesh. The interface is as follows

```
project( _space=<nodal function space in which the interpolant lives>
         _range=<domain range iterators>,
         _expr=<expression to be interpolated>, .... )
```

Here are some examples

```
typedef FunctionSpace<Mesh<Simplex<d>,
                      bases<Lagrange<1,Vectorial> > > Xhv_type;
auto Xhv = Xhv_type::New( mesh );
// build a piecewise ℙ_1 vectorial function in Xhv containing the
// coordinates of the vertices the mesh.
auto coord = project( _space=Xhv, _range=elements(mesh), _expr=P() );
// compute the x derivative of the coord function
auto dx_coord = project( _space=Xhv, _range=elements(mesh),
                         _expr=dxv(coord) );
auto dy_coord = project( _space=Xhv, _range=elements(mesh),
                         _expr=dyv(coord) );
```

4.3 A wrap up example

To finish, we present an example in linear elasticity which is dimension independent and takes only few lines of code to express.

Listing 3: Solving a linear elasticity problem on a campled beam

```
FunctionSpace<mesh_type,bases<Lagrange<1,Vectorial»> space_type;
auto Xh = space_type::New(mesh);
auto u = Xh->element(); v = Xh->element();
// strain tensor  0.5*(∇u+∇u^T)
auto F = backend->newVector( Xh );
auto force=project( _space=Xh, _range=markedface(mesh,"forceZ"),
                    _expr=vec(0,0,-1) );
form1( _test=Xh,  _vector=F ) =
              integrate( markedfaces(mesh,"ForceZ"),
```

```
                          trans(idv(force))*id(v) );
auto deft = sym(gradt(u));
auto def = sym(grad(u));
auto lambda=..., mu=....; // Lame coefficients
auto D = backend->newMatrix( Xh, Xh );
form2( _test=Xh, _trial=Xh, _matrix=D ) =
             integrate( elements(mesh),
                        lambda*divt(u)*div(v)  +
                        2*mu*trace(trans(deft)*def))+
             on( markedfaces(mesh,"clamped"), u, F, 0*one());
// solve
backend->solve( _matrix=D, _solution=u, _rhs=F );
// apply displacement to the mesh
movemesh( mesh, u );
```

5 Mesh adaptation

Since FEEL++ applications become more and more complex, mesh adaptation becomes crucial to control solution accuracy keeping a suitable computational cost. An elegant way is to use the notion of metric, by generating a mesh where the edges have unit length with respect to associated Riemannian metric space.

5.1 Metric-based mesh adaptation

Let's denote v_i and λ_i the eigenvectors and the eigenvalues of the metric tensor describing size and directional constraints of the adapted mesh. The generation of the unit mesh consists in applying the size $h_i = \frac{1}{\sqrt{\lambda_i}}$ on each direction v_i — anisotropic meshing —, or to use the smallest $h_i = \frac{1}{\sqrt{max(\lambda_i)}}$ on all directions — isotropic meshing —.

User can (i) use the metric built from his own function, (ii) use the metric built from the Hessian of the solution, as described in [27].

5.2 Mesh adaptation tool in Feel++

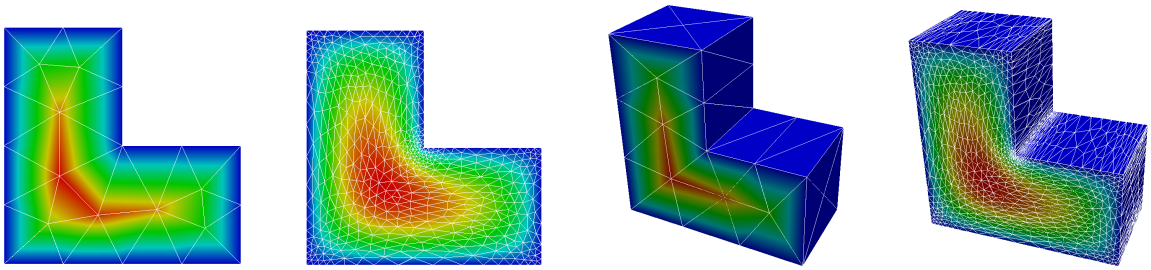
FEEL++ uses GMSH mesh adaptation possibilities, via GMSH API to avoid system calls and consequently enhance the efficiency. Metrics are stored in post processing files (.pos format), which allow GMSH to generate a background defining the associated constraints to apply on the new mesh. The member function `adaptMesh` of `MeshAdaptation` provides an interface allowing to easily build adapted mesh :

```
MeshAdaptation< <dim>,<approx order>,<geo order> > adaptation(<backend>);
adaptation.adaptMesh(_initMesh=<inital mesh>
                     _geofile=<path of geometry file (.geo)>
                     _adaptType=<"isotropic" / "anisotropic">
                     _var=<list of variables used to adapt mesh>
                     _metric=<list of metric used to adapt mesh>, ...);
```

`_var` and `_metric` are optional parameters, containing list of pairs (`element_type`, `"name"`). User has to set either `_var` to use metric built from Hessian of corresponding `element_type`, or `_metric` to use it directly as the metric tensor. When several metrics are specified, GMSH

uses metric intersection algorithm, taking minimum size on each node.

The following example shows the resolution of $-\Delta u = 1$ with homogeneous Dirichlet conditions on a L-shape geometry (2D/3D) with mesh adapted from solution :



(a) Initial 2D mesh (b) Anisotropic 2D (c) Initial 3D mesh (d) Anisotropic 3D

Figure 2: Mesh adaptation on 2D/3D L-shape

6 Conclusions

In this paper, we presented a versatile framework that allows to solve numerically a wide range of problems arising from PDE with an expressivity close to the mathematics. There remain however many challenges to overcome for example (i) automatic differentiation of variational forms, (ii) integration of multiscale methods, (iii) a full framework for domain decomposition including a preconditioner framework and some preconditioners implementation, (iv) the implementation of finite volume variational methods, see [10], (v) or integration model order reduction (e.g. certified reduced basis) into our framework. These are just a few of the challenges we are facing and for some of them we have made already good inroads. So far the framework has very nicely scaled with complexity — in terms of physical models, numerical methods or discretisation, solver and computer science — and retained very good properties of maintainability, code size and reproductibility.

Acknowledgements. The authors wish to thank D. Di Pietro and J.-M. Gratien from IFPEN as well as S. Bertoluzza from Imati/CNR/Pavia for very fruitful discussions. Mourad Ismail and Vincent Doyeux acknowledge the financial support from ANR MOSICOB. Vincent Chabannes and Christophe Prud'homme acknowledge the financial support of the Région Rhône-Alpes through the project ISLE/CHPID. Abdoulaye Samake and Christophe Prud'homme acknowledge the financial support of the project ANR HAMM. Gonçalo Pena ackowledges the financial support of the Calouste Gulbenkian Foundation through the project *Estímulo à Investigação*.

REFERENCES

[1] Boost c++ libraries. http://www.boost.org.

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

[3] 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.

[4] Satish Balay, Kris Buschelman, William D. Gropp, Dinesh Kaushik, Matthew G. Knepley, Lois Curfman McInnes, Barry F. Smith, and Hong Zhang. PETSc Web page, 2001. http://www.mcs.anl.gov/petsc.

[5] Satish Balay, Victor Eijkhout, William D. Gropp, Lois Curfman McInnes, and Barry F. Smith. Efficient management of parallelism in object oriented numerical software libraries. In E. Arge, A. M. Bruaset, and H. P. Langtangen, editors, *Modern Software Tools in Scientific Computing*, pages 163–202. Birkhäuser Press, 1997.

[6] Wolfgang Bangerth, Ralf Hartmann, and Guido Kanschat. `deal.II` *Differential Equations Analysis Library, Technical Reference*. `http://www.dealii.org`.

[7] Claudio Canuto, M. Yousuff Hussani, Alfio Quarteroni, and Thomas A. Zang. *Spectral Methods: Fundamentals in Single Domains*. Springer-Verlag, New York and Berlin, 2006.

[8] V. Chabannes, C. Prud'homme, and G. Pena. High order fluid structure interaction in 2D and 3D: Application to blood flow in arteries. *Journal of Computational and Applied Mathematics (Accepted)*, 2012.

[9] Philippe G. Ciarlet. *Finite Element Method for Elliptic Problems*. Society for Industrial and Applied Mathematics, Philadelphia, PA, USA, 2002.

[10] Daniele Antonio Di Pietro, Jean-Marc Gratien, and Christophe Prud'Homme. A domain-specific embedded language in C++ for lowest-order discretizations of diffusive problems on general meshes. *BIT*, 2012.

[11] V. Doyeux, V. Chabannes, C. Prud'homme, and M. Ismail. Simulation of two fluid flow using a level set method application to bubbles and vesicle dynamics. *Journal of Computational and Applied Mathematics (Accepted)*, 2012.

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

[13] C. Geuzaine and J.-F. Remacle. Gmsh: a three-dimensional finite element mesh generator with built-in pre- and post-processing facilities. *International Journal for Numerical Methods in Engineering*, 79(11):1309–1331, 2009.

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

[15] Vicente Hernandez, Jose E. Roman, and Vicente Vidal. SLEPc: A scalable and flexible toolkit for the solution of eigenvalue problems. *ACM Transactions on Mathematical Software*, 31(3):351–362, 2005.

[16] Michael Heroux, Roscoe Bartlett, Vicki Howle Robert Hoekstra, Jonathan Hu, Tamara Kolda, Richard Lehoucq, Kevin Long, Roger Pawlowski, Eric Phipps, Andrew Salinger, Heidi Thornquist, Ray Tuminaro, James Willenbring, and Alan Williams. An Overview of Trilinos. Technical Report SAND2003-2927, Sandia National Laboratories, 2003.

[17] Michael A. Heroux, Roscoe A. Bartlett, Vicki E. Howle, Robert J. Hoekstra, Jonathan J. Hu, Tamara G. Kolda, Richard B. Lehoucq, Kevin R. Long, Roger P. Pawlowski, Eric T. Phipps, Andrew G. Salinger, Heidi K. Thornquist, Ray S. Tuminaro, James M. Willenbring, Alan Williams, and Kendall S. Stanley. An overview of the trilinos project. *ACM Trans. Math. Softw.*, 31(3):397–423, 2005.

[18] Michael A. Heroux and James M. Willenbring. Trilinos Users Guide. Technical Report SAND2003-2952, Sandia National Laboratories, 2003.

[19] Michael A. Heroux, James M. Willenbring, and Robert Heaphy. Trilinos Developers Guide. Technical Report SAND2003-1898, Sandia National Laboratories, 2003.

[20] George Em Karniadakis and Spencer J. Sherwin. *Spectral/hp element methods for CFD*. Numerical Mathematics and Scientific Computation. Oxford University Press, New York, 2005.

[21] Robert C. Kirby. Algorithm 839: Fiat, a new paradigm for computing finite element basis functions. *ACM Trans. Math. Softw.*, 30(4):502–516, 2004.

[22] Robert C. Kirby. Optimizing fiat with level 3 blas. *ACM Trans. Math. Softw.*, 32(2):223–235, 2006.

[23] Robert C. Kirby. Singularity-free evaluation of collapsed-coordinate orthogonal polynomials. *ACM Trans. Math. Softw.*, 37(1):1–16, 2010.

[24] Robert C. Kirby and Anders Logg. A compiler for variational forms. *ACM Trans. Math. Softw.*, 32(3):417–444, 2006.

[25] Robert C. Kirby and Anders Logg. Efficient compilation of a class of variational forms. *ACM Trans. Math. Softw.*, 33(3):17, 2007.

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

[27] A. Loseille and F. Alauzet. Continuous mesh framework, part 1 : well-posed continuous interpolation error. *SIAM in Numerical Analysis*, 49(1), 2011.

[28] G. Pena. *Spectral Element Approximation of the incompressible Navier-Stokes equations evolving in a moving domain and applications*. PhD thesis, 2009. PhD Thesis, École Polytechnique Fédérale de Lausanne.

[29] G. Pena and C. Prud'homme. Construction of a high order fluid–structure interaction solver. *Journal of Computational and Applied Mathematics*, 234(7):2358–2365, August 2010.

[30] G. Pena, C. Prud'homme, and A. Quarteroni. High order methods for the approximation of the incompressible navier–stokes equations in a moving domain. *Computer Methods in Applied Mechanics and Engineering*, 209-212:197–211, February 2011.

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

[32] C. Prud'Homme, V. Chabannes, V. Doyeux, M. Ismail, A. Samake, G. Pena, et al. Feel++: A computational framework for galerkin methods and advanced numerical methods. *ESAIM Proc.*, 2012.

[33] Christophe Prud'homme. A domain specific embedded language in C++ for automatic differentiation, projection, integration and variational formulations. Scientific Programming, 14(2):81-110, 2006.

[34] Christophe Prud'homme. Life: Overview of a unified C++ implementation of the finite and spectral element methods in 1d, 2d and 3d. In Workshop On State-Of-The-Art In Scientific And Parallel Computing, Lecture Notes in Computer Science, page 10. Springer-Verlag, 2007.

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

[36] T. Warburton. An explicit construction of interpolation nodes on the simplex. *Journal of Engineering Mathematics*, 56(3):247–262, 11 2006.