HAL
open science

Feel++: A Computational Framework for Galerkin Methods and Advanced Numerical Methods

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

To cite this version:

Christophe Prud'Homme, Vincent Chabannes, Vincent Doyeux, Mourad Ismail, Abdoulaye Samake, et al.. Feel++: A Computational Framework for Galerkin Methods and Advanced Numerical Methods. ESAIM: Proceedings, 2012, 38, pp.429-455. 10.1051/proc/201238024 . hal-00662868v3

HAL Id: hal-00662868

https://hal.science/hal-00662868v3

Submitted on 27 Jan 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.

ESAIM: PROCEEDINGS, Vol. ?, 2012, 1-10
Editors: Will be set by the publisher

FEEL++: A COMPUTATIONAL FRAMEWORK FOR GALERKIN METHODS AND ADVANCED NUMERICAL METHODS

C. PRUD'HOMME[2], V. CHABANNES[1], V. DOYEUX[3], M. ISMAIL[3], A. SAMAKE[1] AND G. PENA[4]

Abstract **WARNING: — An English abstract is mandatory! —**

Abstract. This paper presents an overview of a unified framework for finite element and spectral element methods in 1D, 2D and 3D in *C++* called FEEL++. The article is divided in two parts. The first part 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. In the second part, examples using the mortar, Schwartz (non)overlapping, three fields and two ficticious domain-like methods (the Fat Boundary Method and the Penalty Method) are presented and numerically solved in the scope of the library.

Keywords: Galerkin methods and finite element and domain decomposition and ficticious domain and domain specific embedded language and programming paradigms

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 [23,43], the Fenics project [35,36], Getdp [18] or Getfem++ [48], or libraries or frameworks such as deal.II (*C++*) [6], Sundance (*C++*) [37], 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 [44,45] 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 [13,16,40–42].

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

[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

© EDP Sciences, SMAI 2012

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 is organized in two parts. The first part describes an overview of the main abstractions and interfaces handled by the library. In section 2 we describe the 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. The second part introduces several examples using the mortar, Schwartz (non)overlapping, three fields, see section 5, and ficticious domain-like methods, see section 6, to illustrate the flexibility of the language. These methods are being developed within FEEL++ to be used, possibly concurrently, to solve multiphysics and/or multiscale problems within high performance computing environments.

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 [31], 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 [32–34] proposed to use the Dubiner polynomials as a prime basis on the simplex. We extended these ideas on the hypercubes using the Legendre

[1] http://www.boost.org/doc/libs/1_48_0/libs/parameter/doc/html/index.html

polynomials. Other interesting examples of prime basis being used are the Bernstein polynomials. Our framework uses the 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 [40].

Details on the construction of the Dubiner polynomials can be found in [31] 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 [50], *(iii)* Fekete points in simplices, see [31], *(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 [31]). 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 [40]).

2.5. Finite Elements and Other Polynomial Basis

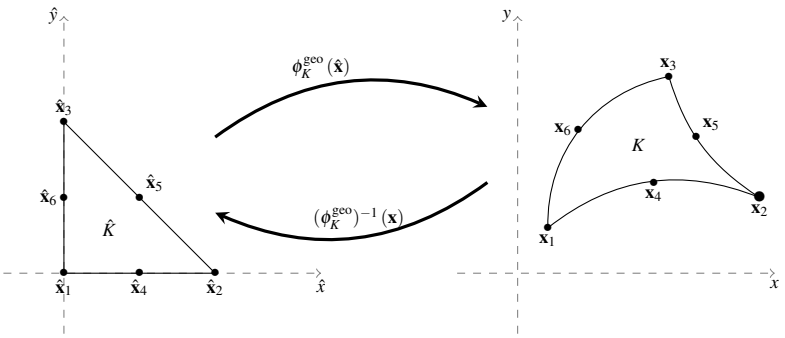
FEEL++ supports modal basis, e.g. Legendre or Dubiner, see [12,31], as well as finite elements (FE) following the standard definition, set in [14], as a triplet $(K,\mathbb{P},\Sigma)$ where K is a convex, $\mathbb{P}$ the polynomial space and Σ the dual space. We describe now some features of the finite 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 5.4 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 [41, 42]. 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 [40, 42]. 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 sections 5 and 6 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.

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 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 [19], 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,
          _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 5 use the extraction of trace meshes as follows:

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

```
// 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"));
```

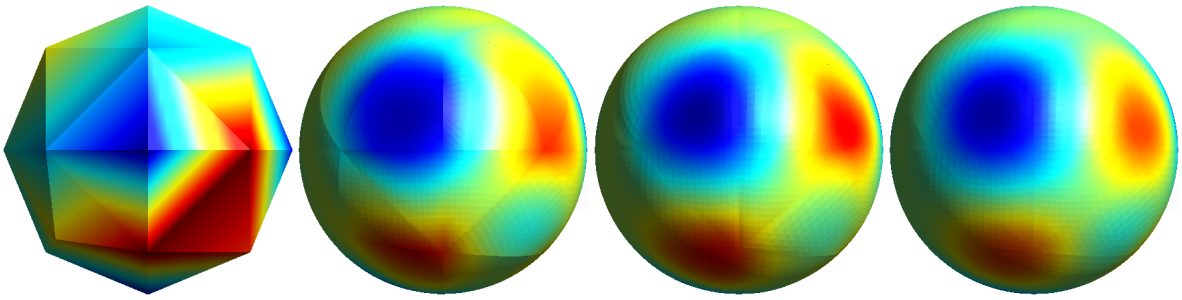


Figure 1. Sphere discretized with 31 elements. Visualization of a function in Gmsh using tetrahedral elements of order $N = 1,2,3,4$.

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, 24], and `Trilinos`, see [25–28]. 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].

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

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

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

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.

Continuous Solution Space	Discrete Approximation Space	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	$\mathbb{CR}_h$ $\mathbb{RaTu}_h$	Crouzeix-Raviart Rannacher-Turek	1 2	2,3 2
$H(\mathrm{div})$-conforming	$\mathbb{RT}_h$	Raviart-Thomas	any	2,3
$H(\mathrm{curl})$-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

The Legrendre and Dubiner basis yield implicitely discontinuous approximations, the Legendre and Dubiner boundary adapted basis, see [31], were designed to handle continuous approximations whereas the Lagrange basis can yield either discontinuous or continuous (default 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 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 T:X_h → V'_h
```

```
auto T = opLinear( _domainSpace=X_h, _imageSpace=dual(V_h) [,_backend=backend] );
// definition of T, see eg section 4
T = integrate(_range=elements(mesh),_expr=gradt(u)*trans(grad(v)));
// linear functional f:V_h → 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 5.

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 sections 5 and 6.

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 A 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`.

Finally FEEL++ supports scalar, vectorial and matricial operations in expressions. Indded the C++ operators follow the same rank semantics of their mathematical counterparts, see appendix A. 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 in section 5.2 or the Fat Boundary Method in section 6. 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++.

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
           _expr=<expression to integrate>, // integrand
           _quad=_Q<polynomial order  to integrate exactly>(), // quadrature
           ... // 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. Domain decomposition methods

We now turn to advanced numerical methods to exercise our framework and show that the embedding language (C++) goes little in the way of the expressivity and closeness to the mathematical language. We first investigate domain decomposition methods such as overlapping and nonoverlapping Schwartz methods, mortar and three fields methods. Domain decomposition methods provide techniques to find the solution of problems defined over domain from the solution of related problems defined on subdomains. They are also used in coupling discretisation methods or models and are suited for parallel computing.

Throughout this section we denote Ω a domain of $\mathbb{R}^d$, $d = 1,2,3$, and $\partial\Omega$ its boundary. We look for u the solution of the problem:

$$\begin{cases} Lu = f & \text{in} \quad \Omega \\ u = g & \text{on} \quad \partial\Omega \end{cases} \tag{3}$$

where L is a partial differential operator, and the functions f and g are given. For the sake of exposition we refer to the case of a domain Ω partitioned into two subdomains Ω_1 and Ω_2 such that $\overline{\Omega} = \overline{\Omega}_1 \cup \overline{\Omega}_2$. We denote $\Gamma_1 := \partial\Omega_1 \cap \overline{\Omega}_2$ and $\Gamma_2 := \partial\Omega_2 \cap \overline{\Omega}_1$, in the case of two overlapping subdomains, and $\Gamma := \partial\Omega_1 \cap \partial\Omega_2$, in the case of two nonoverlapping subdomains. The norm of $H^1(\Phi)$ will be denoted by $\|\cdot\|_{1,\Phi}$, while $\|\cdot\|_{0,\Phi}$ will indicate the norm of $L^2(\Phi)$ for all nonempty subset $\Phi \subseteq \Omega$.

5.1. Overlapping and nonoverlapping Schwartz methods

First we are interested in the overlapping and nonoverlapping Schwartz methods [46]. The overlapping multiplicative Schwartz algorithm with Dirichlet interface conditions at $(k+1)^{th}$ iteration, $k \geq 0$, is given by (4) where u_2^0 is known on Γ_1. The additive version of this algorithm is obtained by changing the interface condition $u_2^{k+1} = u_1^{k+1}$ on the second subdomain Ω_2 to $u_2^{k+1} = u_1^k$ in the second system of (4).

$$\begin{cases} Lu_1^{k+1} = f & \text{in} \quad \Omega_1 \\ u_1^{k+1} = g & \text{on} \quad \partial\Omega_1 \setminus \Gamma_1 \\ u_1^{k+1} = u_2^k & \text{on} \quad \Gamma_1 \end{cases} \qquad \begin{cases} Lu_2^{k+1} = f & \text{in} \quad \Omega_2 \\ u_2^{k+1} = g & \text{on} \quad \partial\Omega_2 \setminus \Gamma_2 \\ u_2^{k+1} = u_1^{k+1} & \text{on} \quad \Gamma_2 \end{cases} \tag{4}$$

The nonoverlapping Schwartz algorithm with Dirichlet and Neumann interface conditions at $(k+1)^{th}$ iteration, $k \geq 0$, are given by

Listing 4 Fixed point algorithm using Aitken acceleration for (4) and (5)

```
enum DDMethod {  DD = 0,/*Dirichlet-Dirichlet*/
                 DN = 1 /*Dirichlet-Neumann*/ };
auto accel =  aitken( _space=Xh2 );
accel.initialize( _residual=residual,_currentElt=lambda);
double maxIteration = 20;
while( !accel.isFinished() &&
       accel.nIterations() < maxIteration)
{
    // call the localProblem on the first subdomain Ω1
    localProblem(u1, idv(u2), DDMethod::DD);
    lambda = u2;
    // call the localProblem on the first subdomain Ω2
    if( ddmethod = DDMethod::DD )
        localProblem(u2, idv(u1), DDMethod::DD);
    else
        localProblem(u2, gradv(u1)*N(), DDMethod::DN);
    residual = u2-lambda;
    u2 = accel.apply( _residual=residual,_currentElt=u2);
    ++accel;
}
```

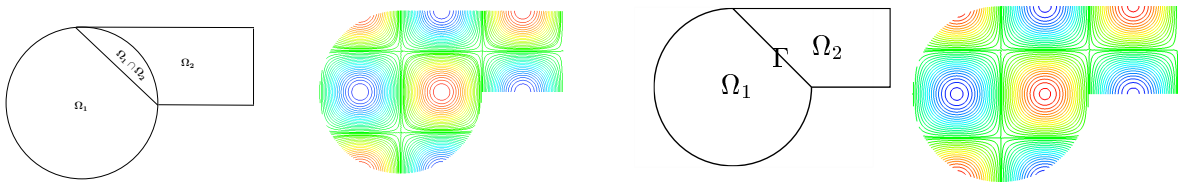
$$\begin{cases} Lu_1^{k+1} = f & \text{in} \quad \Omega_1 \\ u_1^{k+1} = g & \text{on} \quad \partial\Omega_1 \setminus \Gamma \\ u_1^{k+1} = \lambda^k & \text{on} \quad \Gamma \end{cases} \qquad \begin{cases} Lu_2^{k+1} = f & \text{in} \quad \Omega_2 \\ u_2^{k+1} = g & \text{on} \quad \partial\Omega_2 \setminus \Gamma \\ \dfrac{\partial u_2^{k+1}}{\partial n} = \dfrac{\partial u_1^{k+1}}{\partial n} & \text{on} \quad \Gamma \end{cases} \tag{5}$$

where $\lambda^{k+1} := \theta u_{2|_\Gamma}^{k+1} + (1-\theta)\lambda^k$, θ being a positive acceleration parameter which can be computed for example by an Aitken procedure, see listing 5, and λ^0 is given on Γ. The additive Schwartz method requires generally more iterations than the multiplicative method and is naturally parallelizable. The generalization of these algorithms to many subdomains is immediate.

The numerical solutions in Figures 2(b)-2(d) correspond to the partition of Ω into two subdomains Ω_1 and Ω_2 and the following configuration: (i) $g(x,y) = \sin(\pi x)\cos(\pi y)$ is the exact solution (ii) $f(x,y) = 2\pi^2 g$ is the right hand side of the equation (iii) we use $\mathbb{P}_2$ Lagrange approximation (iv) we use the maximal number of iteration equal to 10. Table 3 summarizes The L^2 and H^1 errors for both problems studied. Finally we present a 3D nonoverlapping Schwartz example using the same code in Figure 3.

	$\|\mathbf{u_1}-\mathbf{g}\|_{0,\Omega_1}$	$\|\mathbf{u_2}-\mathbf{g}\|_{0,\Omega_2}$	$\|\mathbf{u_1}-\mathbf{g}\|_{1,\Omega_1}$	$\|\mathbf{u_2}-\mathbf{g}\|_{1,\Omega_2}$
With overlap	2.52e-8	2.16e-8	4.07e-6	3.89e-6
Without overlap	1.37e-9	2.04e-8	1.32e-6	6.71e-6

Table 3. Numerical results for Schwartz methods

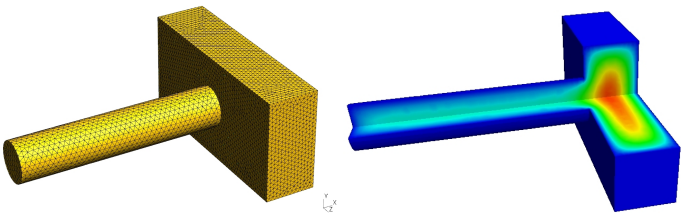


(a) Geometry (b) Solution (c) Geometry (d) Solution

Figure 2. Overlapping and nonoverlapping cases in 2D

Listing 5 Local problems for (4) and (5)

```
template<Expr> void localProblem(element_type& u, Expr expr, DDMethod ddmethod)
{
  auto Xh=u.functionSpace();
  auto mesh=Xh->mesh();
  auto v=Xh->element();
  auto F = M_backend->newVector(Xh);
  auto A = M_backend->newMatrix( Xh, Xh );

  // Assembly of the right hand side ∫_Ω fv
  form1( _test=Xh,_vector=F ) = integrate( elements(mesh), f*id(v) );

  // Assembly of the left hand side ∫_Ω ∇u·∇v
  form2( _test=Xh, _trial=Xh, _matrix=A ) =
      integrate( elements(mesh), gradt(u)*trans(grad(v)) );

  // Add Neumann contribution
  if ( ddmethod == DDMethod::DN )
      form1( _test=Xh,_vector=F ) += integrate( markedfaces(mesh,"Interface"),
                                                _expr=expr*id(v));
  else if( ddmethod == DDMethod::DD )
      form2( Xh, Xh, A ) += on( markedfaces(mesh,"Interface") , u,F,expr);

  // Apply the Dirichlet boundary conditions
  form2( Xh, Xh, A ) += on( markedfaces(mesh, "Dirichlet"), u,F,g);

  // Apply the Dirichlet interface conditions
  // solve the linear system Au=F
  M_backend->solve(_matrix=A,_solution=u,_rhs=F );
}
```



(a) Geometry (b) Solution

Figure 3. An example of nonoverlapping Schwartz method in 3D

5.2. Mortar domain decomposition method

Consider the problem (3) where $L := -\Delta$ and homogeneous Dirichlet boundary conditions. We assume that Ω is partitioned into two nonoverlapping subdomains and it is a d-dimensional domain ($d = 2,3$), with a Lipschitz boundary $\partial\Omega$. We also assume that f belongs to $L^2(\Omega)$. The main idea of this method is to enforce the weak continuity between the solutions on each subdomain. This is achieved by introducing a Lagrange multiplier corresponding to this connection constraint [7]. Problem (3) is equivalent to finding $u \in H_0^1(\Omega)$ such that

$$J(u) = \min_{v \in H_0^1(\Omega)} J(v), \quad J(v) = \frac{1}{2}\int_\Omega |\nabla v|^2 - \int_\Omega fv. \text{ We denote by } J_i(v) := \frac{1}{2}\int_{\Omega_i} |\nabla v_i|^2 - \int_\Omega f v_i \qquad i = 1,2.$$

The problem is now equivalent to find $(u_1,u_2) \in H$ such as

$$J_1(u_1)+J_2(u_2) = \min_{(v_1,v_2)\in H} J_1(v_1)+J_2(v_2),\ H = \Big\{(v_1,v_2) \in H^1(\Omega_1)\times H^1(\Omega_2):\ v_i = 0 \text{ on } \partial\Omega_i \setminus \Gamma,\ v_1 = v_2 \text{ on } \Gamma\Big\}.$$

We define the Lagrangian by $\mathscr{L}(v_1,v_2;\mu) := J_1(v_1)+J_2(v_2)+\int_\Gamma \mu(v_1-v_2)$ and we search for the saddle point solution (u_1,u_2,λ) such that

$$\mathscr{L}(u_1,u_2;\mu) \le \mathscr{L}(u_1,u_2;\lambda) \le \mathscr{L}(v_1,v_2;\lambda), \quad \forall\,(v_1,v_2)\in V_{1h}\times V_{2h} \quad \forall\,\mu\in W_h. \tag{6}$$

Let us denote by V_{ih} the finite element approximation space on Ω_i, of basis $(\psi_{i,j})_{j=1,\cdots N_i}, i=1,2$, and by W_h that of Γ, of basis $(\phi_k)_{k=1,\cdots K}$.
Then the saddle point (u_1,u_2,λ) can be approximated by the solution of the following discrete system

$$\begin{cases} A_1U_{1,h}+B_1^tL_h=\mathrm{f}_1 \\ A_2U_{2,h}-B_2^tL_h=\mathrm{f}_2 \\ B_1U_{1,h}-B_2U_{2,h}=0 \end{cases} \quad \text{where} \quad \begin{cases} (A_i)_{jl}=\displaystyle\int_{\Omega_i}\nabla\psi_{ij}\cdot\nabla\psi_{il} & j,l=1,\cdots,N_i \\ (B_i)_{kj}=\displaystyle\int_{\Gamma}\psi_{ij}\phi_k & k=1,\cdots,K, \quad j=1,\cdots,N_i \\ (\mathrm{f}_i)_j=\displaystyle\int_{\Omega_i} f\psi_{ij} & j=1,\cdots,N_i \end{cases} \tag{7}$$

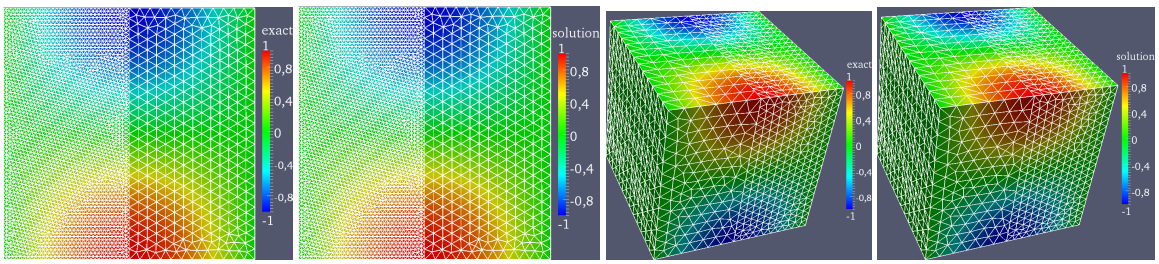
and $U_{1,h},U_{2,h}$ and L_h collect the degrees of freedom of $u_{1,h},u_{2,h}$ and λ_h, approximations to u_1,u_2 and λ.

In Listing 6 we display the terms corresponding to the jump matrices B_1 and B_2 in equation (7).

Listing 6 Jump terms in the global matrix

```
// function spaces and some associated elements(functions) for Ω1 and Ω2
auto Xh1 = space1_type::New( mesh1 );
auto u1 = Xh1->element();
auto Xh2 = space2_type::New( mesh2 );
auto u2 = Xh2->element();
// trace mesh and Lagrange multiplier space and associated function for Γ
auto Lh = lagmult_space_type::New(mesh1->trace( markedfaces(mesh1,gamma) ));
auto mu = Lh->element();
// assembly of jump matrices B1 and B2
auto B1 = M_backend->newMatrix( _trial=Xh1, _test=Lh );
form2( _trial=Xh1, _test=Lh, _matrix=B1 ) =
     integrate( elements(Lh->mesh()), idt(u1)*id(mu) );

auto B2 = M_backend->newMatrix( _trial=Xh2, _test=Lh );
form2( _trial=Xh2, _test=Lh, _matrix=B2) =
     integrate( elements(Lh->mesh()), idt(u2)*id(mu) );
```

Figures 4(b) and 4(d) correspond to the solution of (7) when Ω is partitioned into two nonoverlapping subdomains Ω_1 and Ω_2 and the following configurations are set: (i) $g(x,y)=\sin(\pi x)\cos(\pi y)$ is the exact solution (ii) $f(x,y)=2\pi^2 g$ is the right hand side of the equation (iii) we use $\mathbb{P}_2$ and $\mathbb{P}_3$ approximations, respectively, in Ω_1 and Ω_2 (iv) $h_{\Omega_1}=0.015$ and $h_{\Omega_2}=0.04$ in 2D (v) $h_{\Omega_1}=0.075$ and $h_{\Omega_2}=0.05$ in 3D.

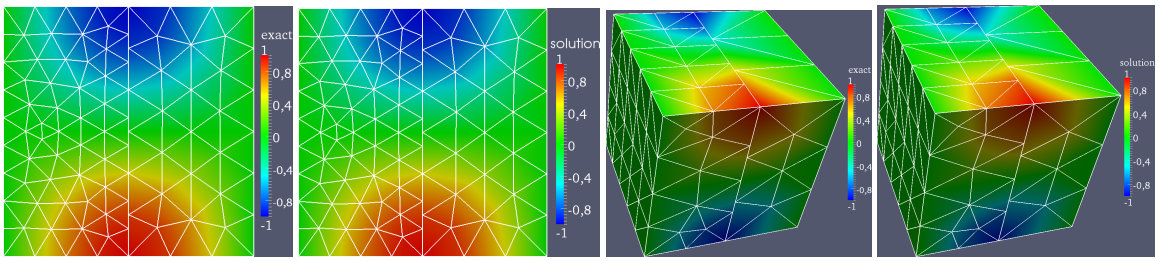


(a) Exact solution (b) Numerical solution (c) Exact solution (d) Numerical solution

Figure 4. Exact and Numerical solutions in 2D and 3D for mortar method

The numerical solutions 5(b) and 5(d) correspond to the same configuration used previously but we change the polynomial orders and mesh sizes as (i) $\mathbb{P}_7$ and $\mathbb{P}_6$ approximations, respectively, in Ω_1 and Ω_2, $h_{\Omega_1}=0.1$ and $h_{\Omega_2}=0.15$ in 2D (ii) $\mathbb{P}_3$ and $\mathbb{P}_4$ approximations, respectively, in Ω_1 and Ω_2, $h_{\Omega_1}=0.75$ and $h_{\Omega_2}=0.5$ in 3D.

Table 4 summarizes several error quantities for both problems studied and the global solution u_h defined on Ω as $u_h = u_{1,h}$ on Ω_1 and $u_h = u_{2,h}$ on Ω_2.



(a) Exact solution (b) Numerical solution (c) Exact solution (d) Numerical solution

Figure 5. Exact and Numerical solutions in 2D and 3D for mortar method using higher order approximations

	P-orders	$\|\mathbf{u_{1,h}} - \mathbf{g}\|_{0,\Omega_1}$	$\|\mathbf{u_{2,h}} - \mathbf{g}\|_{0,\Omega_2}$	$\|\mathbf{u_{1,h}} - \mathbf{g}\|_{1,\Omega_1}$	$\|\mathbf{u_{2,h}} - \mathbf{g}\|_{1,\Omega_2}$	$\|\lambda_\mathbf{h} - \mathbf{g}\|_{0,\Gamma}$	$\|\mathbf{u_h} - \mathbf{g}\|_{1,\Omega}$
2D case	$\mathbb{P}_2 - \mathbb{P}_3$	3.39695e-08	1.88289e-08	8.67241e-06	2.2959e-06	6.69075e-09	8.97116e-06
	$\mathbb{P}_7 - \mathbb{P}_6$	1.13996e-09	7.35774e-10	2.92219e-07	1.15223e-07	2.29634e-11	3.14115e-07
3D case	$\mathbb{P}_2 - \mathbb{P}_3$	8.74556e-06	8.40938e-06	1.17e-04	9.72e-05	1.52e-05	2.17e-04
	$\mathbb{P}_3 - \mathbb{P}_4$	9.28974e-07	8.97491e-07	3.55e-05	3.50e-05	1.03e-07	4.99e-05

Table 4. Numerical results for mortar methods in 2D and 3D

5.3. Three fields domain decomposition method

The weak formulation of (3) reads: find $u \in H_0^1(\Omega)$ such that $\int_\Omega \nabla u \cdot \nabla v = \int_\Omega f v, \quad \forall v \in H_0^1(\Omega)$. Denoting by u_i the restriction of the solution u to Ω_i, $i = 1,2$, the interface conditions on Γ are $\partial u_1/\partial \mathrm{n} = \partial u_2/\partial \mathrm{n}$ and $u_1 = u_2$ on Γ.

The three fields formulation [11] rewrites the original problem by relaxing the continuity requirements on both u and $\partial u/\partial \mathrm{n}$, at the expense of introducing two Lagrange multipliers for each subdomain. The new weak formulation allows independent approximations within the subdomains, including the possibility of using different methods and different meshes on both subdomains. First for all, let us define $\Lambda := \left\{ \eta \in H^{1/2}(\Gamma) \mid \eta = v_{|\Gamma} \text{ for a suitable } v \in H^1(\Omega) \right\}$ the trace space and Λ' its dual. The finite dimensional approximation of (8) can be devised by selecting suitable subspaces $V_{i,h}$ of $V_i := H^1(\Omega_i)$, $\Xi_{i,h}$ of Λ' and Λ_h of Λ. We denote by $(\psi_{i,j})_{j=1,\cdots N_i}$ a basis of $V_{i,h}$, $(\chi_{i,k})_{k=1,\cdots K_i}$ that of Λ' and $(\phi_k)_{k=1,\cdots K}$ that of Λ. In the following formulation, σ_i is the Lagrange multiplier that is used to 'glue' the value of u_i and λ on Γ, $i = 1,2$. As a matter of fact, the following equivalence result holds between the weak form of the unsplit problem (3) and the problem with Lagrange multipliers. The three fields formulation is given by: for $i = 1,2$ find $u_i \in V_i := H^1(\Omega_i)$, $\sigma_i \in \Lambda'$, $\lambda \in \Lambda$ such that

$$\begin{cases} \displaystyle\int_{\Omega_1} \nabla u_1 \cdot \nabla v_1 - \int_\Gamma \sigma_1 v_1 = \int_{\Omega_1} f v_1 & \forall\, v_1 \in H^1(\Omega_1) \\ \displaystyle\int_\Gamma \rho_1(\lambda - u_1) = 0 & \forall\, \rho_1 \in \Lambda' \\ \displaystyle\int_\Gamma (\sigma_1 + \sigma_2)\mu = 0 & \forall\, \mu \in \Lambda \\ \displaystyle\int_\Gamma \rho_2(\lambda - u_2) = 0 & \forall\, \rho_2 \in \Lambda' \\ \displaystyle\int_{\Omega_2} \nabla u_2 \cdot \nabla v_2 - \int_\Gamma \sigma_2 v_2 = \int_{\Omega_2} f v_2 & \forall\, v_2 \in H^1(\Omega_2). \end{cases} \tag{8}$$

Problem (8) can be stated in algebraic form as

$$\begin{cases} A_1U_{1,h} - B_1^tS_{1,h} & = \mathbf{f}_1 \\ -B_1U_{1,h} + C_1^tL_h & = \mathbf{0} \\ C_1S_{1,h} + C_2S_{2,h} & = \mathbf{0} \\ -B_2U_{2,h} + C_2^tL_h & = \mathbf{0} \\ A_2U_{2,h} - B_2^tS_{2,h} & = \mathbf{f}_2 \end{cases} \quad \text{where} \quad \begin{cases} (A_i)_{jl} = \displaystyle\int_{\Omega_i} \nabla\psi_{ij}\nabla\psi_{il} & j,l = 1,\cdots,N_i \\ (B_i)_{kj} = \displaystyle\int_{\Gamma} \psi_{ij}\chi_{i,k} & k = 1,\cdots,K_i, \quad j = 1,\cdots,N_i \\ (C_i)_{kj} = \displaystyle\int_{\Gamma} \chi_{i,j}\phi_k & j = 1,\cdots,K_i, \quad k = 1,\cdots,K \\ (\mathbf{f}_i)_j = \displaystyle\int_{\Omega_i} f\psi_{ij} & j = 1,\cdots,N_i \end{cases} \tag{9}$$

and $U_{i,h}, S_{i,h}, L_h$ collect the degrees of freedom of $u_{i,h}, \boldsymbol{\sigma}_{i,h}$ and λ_h, approximations to $u_i, \boldsymbol{\sigma}_i$ and λ. In Listing 7 we display the terms corresponding to the jump matrices B_1, C_1, B_2 and C_2 in equation (9).

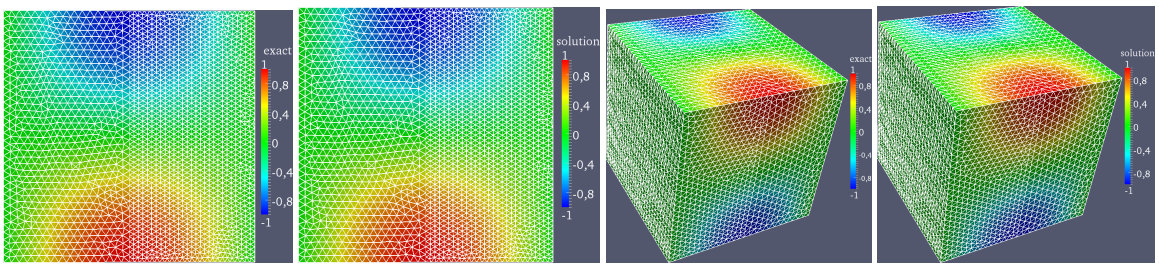
Listing 7 Jump terms in the global matrix

```
// function spaces and some associated elements(functions) for Ω1 and Ω2
auto Xh1 = space1_type::New( mesh1 );
auto u1 = Xh1->element();
auto Xh2 = space2_type::New( mesh2 );
auto u2 = Xh2->element();
// trace meshes and Lagrange multiplier spaces and associated elements
auto Lh1 = lagmult_space1_type::New( mesh1->trace( markedfaces(mesh1,gamma)));
auto mu1 = Lh1->element();
auto Lh2 = lagmult_space2_type::New( mesh2->trace( markedfaces(mesh2,gamma)));
auto mu2 = Lh2->element();

auto Lh = interface_space_type::New( interface_mesh );
auto mu = Lh->element();
// assembly of jump matrices B1 and B2
auto B1 = M_backend->newMatrix( _trial=Xh1, _test=Lh1 );
form2( _trial=Xh1, _test=Lh1, _matrix=B1) =
     integrate( elements(Lh1->mesh()), idt(u1)*id(mu1) );

auto C1 = M_backend->newMatrix( Lh, Lh1 );
form2( _trial=Lh, _test=Lh1, _matrix=C1) +=
     integrate( elements(Lh->mesh()), idt(mu)*id(mu1) );

auto B2 = M_backend->newMatrix( _trial=Xh2, _test=Lh2 );
form2( _trial=Xh2, _test=Lh2, _matrix=B2) =
     integrate( elements(Lh2->mesh()), idt(u2)*id(mu2) );

auto C2 = M_backend->newMatrix( Lh, Lh2 );
form2( _trial=Lh, _test=Lh2, _matrix=C2) +=
     integrate( elements(Lh->mesh()), idt(mu)*id(mu2) );
```

The following numerical solutions 6(b) and 6(d) correspond to partition of Ω into two nonoverlapping subdomains Ω_1 and Ω_2 and the following configurations (i) $g(x,y) = \sin(\pi x)\cos(\pi y)$ is the exact solution (ii) $f(x,y) = 2\pi^2 g$ is the right hand side of the equation (iii) $\mathbb{P}_2$, $\mathbb{P}_1$ and $\mathbb{P}_3$ approximations respectively in Ω_1, Ω_2 and Γ (iv) we set $h_{\Omega_1} = 0.03$, $h_{\Omega_2} = 0.02$ and $h_\Gamma = 0.01$ in 2D (v) we set $h_{\Omega_1} = 0.05$, $h_{\Omega_2} = 0.07$ and $h_\Gamma = 0.02$ in 3D. Table 5 summarizes several error quantities for both problems studied and the global solution u_h defined on Ω as $u_h = u_{1,h}$ on Ω_1 and $u_h = u_{2,h}$ on Ω_2.

	P-orders	$\|\mathbf{u_{1,h}} - \mathbf{g}\|_{0,\Omega_1}$	$\|\mathbf{u_{2,h}} - \mathbf{g}\|_{0,\Omega_2}$	$\|\mathbf{u_{1,h}} - \mathbf{g}\|_{1,\Omega_1}$	$\|\mathbf{u_{2,h}} - \mathbf{g}\|_{1,\Omega_2}$	$\|\lambda_\mathbf{h} - \mathbf{g}\|_{0,\Gamma}$	$\|\mathbf{u_h} - \mathbf{g}\|_{1,\Omega}$
2D case	$\mathbb{P}_2 - \mathbb{P}_1 - \mathbb{P}_3$	6.22203e-11	6.24881e-11	1.15144e-08	1.61313e-08	4.87515e-12	1.98192e-08
3D case	$\mathbb{P}_2 - \mathbb{P}_1 - \mathbb{P}_3$	2.65027e-07	2.24883e-07	2.21354e-05	1.39782e-05	2.45257e-08	2.61795e-05

Table 5. Numerical results in 2D and 3D using the three fields method



(a) Exact solution (b) Numerical solution (c) Exact solution (d) Numerical solution

Figure 6. Exact and Numerical solutions in 2D and 3D using the three fields method

5.4. Tools in Feel++: trace of trace and lift of lift

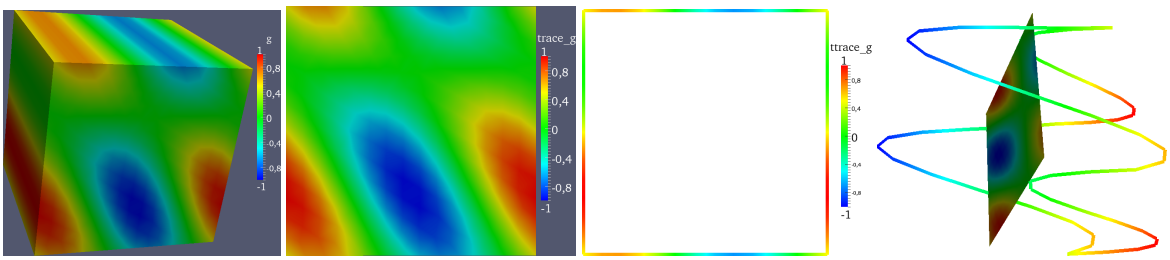
Trace of trace and lift of lift operations are needed in the construction of a substructuring preconditioner *e.g.* for the mortar method [10] which we are currently developping in 2D and 3D. Let Ω be a domain of $\mathbb{R}^3$, $\Sigma \subset \partial\Omega$ an open and nonempty subset and $\Gamma := \partial\Sigma$. We also recall that the trace space of $V := H^1(\Omega)$ on Σ is denoted by $H^{1/2}(\Sigma)$ and the trace space of $W := H^{1/2}(\Sigma)$ on Γ is indicated by $\Lambda := H^{1/2}_{\Sigma}(\Gamma)$ that is the trace space of trace space of V on Γ. It is necessary in this part to be able to manipulate the objects of real dimension equal to d and topological dimension ranging from 1 to d back and forth. Let $u \in V$, first we compute $v = u_{|\Sigma} \in W$ the trace of u and then $w = v_{|\Gamma} \in \Lambda$ the trace of v that is also the trace of trace of u. Reciprocally let $w \in \Lambda$. The extension of w by its mean $c := \dfrac{1}{|\Gamma|}\displaystyle\int_\Gamma w$ in W is given by $v \in W$ such that $v = w$ on Γ and $v = c$ in Σ. Now we compute the harmonic extension of v in V that is given by $u \in V$ such that $-\Delta u = 0$ in Ω and $u = v$ on Σ.

Listing 8 trace of trace and lift and lift implementation

```
auto Xh = space_type::New( mesh );
// trace function space associated to trace(mesh)
auto TXh = trace_space_type::New( mesh->trace(markedfaces(mesh,marker)) );
// trace function space associated to trace(trace(mesh))
auto TTXh = trace_trace_space_type::New( TXh->mesh()->trace(boundaryfaces(TXh->mesh())));
// Let g be an function given on 3D mesh
auto g = sin(pi*(2*Px()+Py()+1./4))*cos(pi*(Py()-1./4));
/* trace and trace of trace of g  */
// trace of g on the 2D trace_mesh
auto trace_g = vf::project( TXh, elements(TXh->mesh()), g );
// trace of g on the 1D trace_trace_mesh
auto trace_trace_g = vf::project( TTXh, elements(TTXh->mesh()), g );
/* lift and lift of lift of trace_trace_g  */
// extension of trace_trace_g by zero on 2D trace_mesh
auto zero_extension = vf::project( TXh,boundaryfaces(TXh->mesh()),idv(trace_trace_g) );
// extension of trace_trace_g by the mean of trace_trace_g  on trace_mesh
auto const_extension = vf::project( TXh, boundaryfaces(TTXh->mesh()),
                                         idv(trace_trace_g)-mean );
const_extension += vf::project( TXh, elements(TXh->mesh()), cst(mean) );
// harmonic extension of const_extension on 3D mesh
auto op_lift = operatorLift(Xh);
auto glift = op_lift->lift(_range=markedfaces(mesh,marker),_expr=idv(const_extension));
```

6. Some Fictitious domain-like methods

Fictitious domain methods (FDM) are a class of numerical methods widely used in the literature. The basic idea behind this class of methods is to use a computational domain which is generally larger than the real one and has a simple shape (hypercubes for example). This gives, in particular, the possibility to handle computations in complex geometries using a regular mesh thus making it possible to use fast solvers and/or efficient preconditioners. In



(a) volume and the function g (b) trace mesh and trace of g (c) wirebasket and trace trace of g (d) warp with respect the function

Figure 7. volume and wirebasket and the function g

recent years we have seen the emergence of many varieties of this class of methods based on this principle. See [20, 22] which are some of the first papers introducing and analyzing the Fictitious Domain Methods.

The efficiency of FDM has been proven in several papers and we can find in the literature many different simulations based on it. However, the main drawback of FDM is the loss of optimal convergence order when it is used with the finite element method. Indeed, if we replace the real domain by a simpler one containing it, the corresponding mesh does not match the real boundaries. Consequently, it leads naturally to the loss of optimal convergence order, see e.g. [20, 21, 47, 49].

We choose to present in this work two types of FDM-like methods : *(i)* the penalty method which is probably the oldest and the simplest one (from the point of view of implementation) in this category (see [21, 39] for example) and *(ii)* the Fat Boundary Method (FBM) which is more technical but has the great advantage to conserve the optimal order in the sense of classical finite element, and this, by adding an auxiliary local problem around the real boundaries [8, 9, 29, 38].

6.1. The Fat Boundary method

The Fat Boundary Method was introduced in [38] to solve elliptic problems in perforated domains and extended to complex fluid simulations in [9, 29]. As far as we know, it is known to be the only fictitious domain-like method that conserves the optimal order (see [8, 9]).

To fix ideas let us introduce this method in a simple case. Assume that we want to solve the Poisson equation with homogeneous Dirichlet boundary conditions in a perforated domain. To do this, we introduce a Lipschitz bounded domain $\Omega_{Glob} \subset \mathbb{R}^d$ containing a collection of N_b smooth sub-domains $B = \cup_{i=1}^{N_b} B_i$. The boundaries of Ω_{Glob} and B are respectively denoted by Γ and $\gamma = \cup_{i=1}^{N_b} \partial B_i$. The domain of interest is then the perforated one given by $\Omega_{Perf} = \Omega \setminus \bar{B}$ and its boundary is $\partial\Omega_{Perf} = \Gamma \cup \gamma$. A typical example of this domain is given by figure 8.

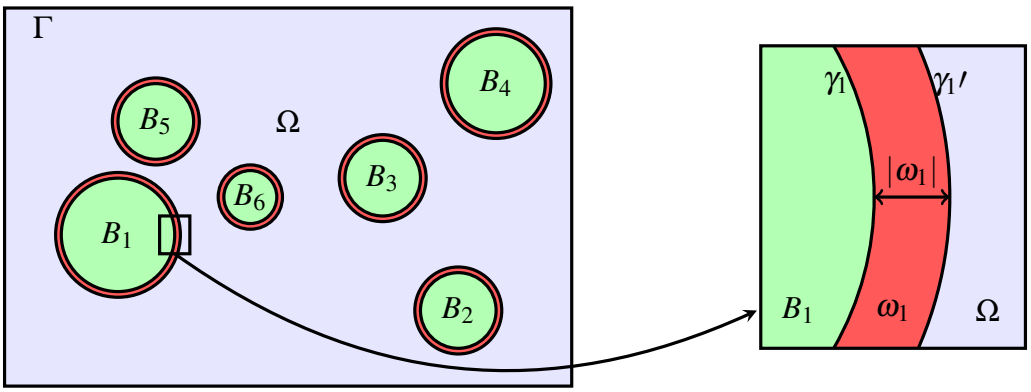


Figure 8. An example of a perforated domain.

Assume that we would like to solve the following problem : Given $f \in L^2(\Omega_{Perf})$, Find $u \in H_0^1(\Omega_{Perf})$ such that

$$-\Delta u = f \quad \text{in } \Omega_{Perf}. \tag{10}$$

Obviously the associated variational formulation reads : find $u \in H_0^1(\Omega_{Perf})$ such that

$$\int_{\Omega_{Perf}} \nabla u \cdot \nabla v = \int_{\Omega_{Perf}} f\, v, \quad \forall v \in H_0^1(\Omega_{Perf}). \tag{11}$$

Solving this problem by FBM consists in splitting it into two new ones : *(i)* a local one in a neighborhood of B, where we can use a fine mesh (in a thin layer around the holes), and a global one covering the whole domain Ω_{Glob}. The link between the global and the local problems is based on the interpolation of a globally defined field on an artificial boundary which delimits the local sub-domains, and the prescription of the jump of the normal derivative across the boundary of B. More precisely, we introduce a smooth artificial boundary γ' around B, and we denote by ω the (narrow) domain delimited by γ and γ' ($\partial\omega = \gamma \cup \gamma'$). A typical example of these different domains is given by figure 8.

In the case of the Poisson equation, we prove (see [38]) that problem (10) is equivalent to the following couple of problems : find $(u_{glob}, u_{loc}) \in H_0^1(\Omega_{Glob}) \times H_\gamma^1(\omega)$, such that

$$\left\{ \begin{array}{lll} (L): & \left\{ \begin{array}{rcll} -\Delta u_{loc} & = & f & \text{in} \quad \omega, \\ u_{loc} & = & u_{glob} & \text{on} \quad \gamma', \end{array} \right. \\ (G): & -\Delta u_{glob} = \overline{f} + \dfrac{\partial u_{loc}}{\partial n}\delta_\gamma & \text{in} \quad \Omega_{Glob}, \end{array} \right. \tag{12}$$

where $\overline{f}$ is the extension of f by 0 in B and $H_\gamma^1(\omega)$ is the set of functions in $H^1(\omega)$ with vanishing traces on γ.

The main advantage of the global problem (G) is to be set in a domain with a simple shape that makes it possible the use of fast solver. Concerning the local problem (L), its role is mainly to correct the solution in the neighborhood of the connected components of B and thus keep the optimal order. As the local domains are thin, the cost of the local computations should be negligible compared to the global one. Moreover, the structure of the local problems makes it easy to solve them in parallel.

As the problems (L) and (G) are coupled, one can solve them using a fix point strategy. We use the one given by the following three steps where U_h and L_h stand for the corresponding discrete finite element spaces and θ is a relaxation parameter in $[0,1[$.

In the following we present a sketch of the fix point algorithm and the corresponding FEEL++ codes. This shows the flexibility of the library and in particular its ability to handle nonstandard numerical methods even if it contains some unusual terms like the integral involving two different meshes (like the red term in equation 15). This is done in a transparent way for the user.

Loop (indexed by i).
Step 1 : Relaxation.

$$\mathscr{K}_i = \theta \mathscr{K}_{i-1} + (1-\theta)\, u_{glob}^{i-1}\,|_{\gamma'} \tag{13}$$

```
auto gamma0 = vf::project(_space=Xh_loc,
                          _range=markedfaces(Ω_Loc,γ'),
                          _expr=idv(u_loc) );
gamma.add( theta, gamma0 );
auto Ih=opInterpolation(_domainSpace=Uh,_imageSpace=Lh,
                        _range=markedfaces(Ω_Loc,γ'));
Ih->apply(u_glob,gamma0);
gamma.add( 1-theta, gamma0 );
```

Step 2 : Local problem.

Find $u_{loc}^i \in L_h$ such that $\forall v_{loc} \in L_h$:

$$\left\{ \begin{array}{r} \displaystyle\int_{\Omega_{Loc}} \nabla u_{loc}^i \cdot \nabla v_{loc} = \int_{\Omega_{Loc}} f\, v_{loc} \\ u_{loc}^i = \mathscr{K}_i \ \text{ on } \gamma' \\ u_{loc}^i = 0 \ \text{ on } \gamma \end{array} \right. \tag{14}$$

```
form2( _test=Lh, _trial=Lh, _matrix=MLoc )
   = integrate( _range=elements(Ω_Loc),
                _expr=gradt(u_loc)*trans(grad(v_loc)) );
form1( _test=Lh, _vector=FLoc)
   = integrate( _range=elements(Ω_Loc),
                _expr=f*id(v_loc) );
form2( _test=Lh, _trial=Lh, _matrix=MLoc)
   += on(_range=markedfaces(Ω_Loc,γ'),
         _element=u_loc, _rhs=FLoc,
         _expr=idv(𝒦) );
   += on( _range=markedfaces(Ω_Loc,γ),
          _element=u_loc, _rhs=FLoc,
          _expr=cst(0.) );
```

Step 3: Global problem.

Find $u_{glob}^i \in U_h$ such that $\forall v \in U_h$:

$$\left\{ \begin{array}{r} \displaystyle\int_{\Omega_{Glob}} \nabla u_{glob}^i \cdot \nabla v = \int_\gamma \frac{\partial u_{loc}^i}{\partial n}\, v + \int_{\Omega_{Glob}} \bar{f}\, v \\ u_{glob}^i = 0 \quad \text{on } \partial\Omega_{Glob} \end{array} \right. \tag{15}$$

```
form2( _test=Uh, _trial=Uh, _matrix=MGlob )
   = integrate( _range=elements(Ω_Glob),
                _expr=gradt(u_glob)*trans(grad(v_glob)) );
form1( _test=Uh, _vector=FGlob )
   = integrate( _range=elements(Ω_Glob),
                _expr=f*id(v_glob) );
form1( _test=Uh, _vector=FGlob)
   += integrate( _range=markedfaces(Ω_Loc, γ),
                 _expr=(gradv(u_loc)*N())*id(v_glob) );
```

```
form2( _test=Uh, _trial=Uh, _matrix=MGlob)
   += on( _range=boundaryfaces(Ω_Glob),
          _element=u_glob, _rhs=FGlob,
          _expr=cst(0.) );
```

In figure 9 we show an example of a global and local meshes used to solve the above scheme (13)-(14)-(15) with $f = 1$ in Ω_{Perf}. Recall that the two type of meshes (global and local ones) are generated independently and of course there is no constraint that they match. Figure 10 presents the iso-values of u_{glob} in the case of 2D simulation with $N_b = 20$ while figure 11 shows the iso-values of the 3D solutions u_{glob} in the case of one hole. Note that the values of u_{glob} inside the holes are not relevant (only the restriction of u_{glob} to Ω_{Perf} are of interest).

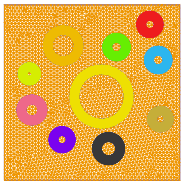


Figure 9. Global and Local Meshes in 2D.

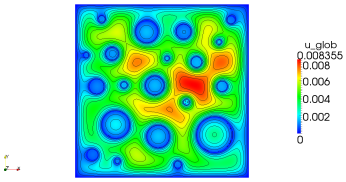


Figure 10. 2D-Problem with 20 holes.

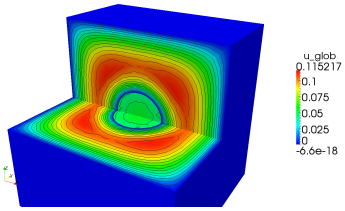


Figure 11. 3D-Problem with 1 hole.

6.2. Penalty method to simulate particle flow in capillaries

All the methods presented in this paper showed a complexity in implementation and thus are still under active research and mostly used in the mathematical field. We now turn to a somewhat simpler numerical method albeit used conjointly with physical experiments to understand particle flows in capillaries. Studies of flows in micro channels have been multiplied since several years because of the goal to create cheap and efficient lab-on-chips. An application that would like to achieve many groups with a good quality is the sorting of particles on lab-on-chips. For example sorting healthy / sick red blood cells. One of the building blocs of such microfluidic system is the splitting of a channel into two daughters channels. All the mechanisms involving the spreading of particles in such a system are not completely understood. It has been seen for example that, when a suspension of particles pass through a bifurcation having different flow rates at outlet, the branch having the higher flow rate see its volume fraction (concentration of particles) increase. This phenomenon is known as the *Zweifach - Fung effect* [51].

In [17], the authors used `FreeFem++` to run a 2D simulation of a microfluidic bifurcation in which a rigid and circular particle is put in the inlet channel and goes to one of the outlet channels. The choice of the particle to go to the higher or the lower flow rate channel depends on many parameters which have been studied in the paper such as the flow rate ratio between the two branches, the size of the particle, the size of the outlet channels or even the angle between the inlet and the outlet channels. The simulations were compared with experiments and showed a good agreement. These simulations and experiments shown an attracting hydrodynamical force toward the lower flow rate channel which was unknown until there. Moreover the increase in concentration in the highest flow rate branch has been explained by a geometrical effect of the distribution of particles in the inlet channel which was a good step forward in the understanding of the *Zweifach - Fung effect*.

We propose to implement the method used in [17] and have both 2D and 3D setups which is readily the case using FEEL++. Moreover, FEEL++ allows us to run the same simulation in 3D without re-coding.

The size of the channel is very small (typically 60 µm), so that we can assume that we are in a low Reynolds number regime ($Re = 10^{-1}$ in [17]). The particles are rigid and spherical (circular in 2D) and for the sake of simplicity, we consider only one particle in the bifurcation. The extension to many particles can be done with no special care since the particles do not touch. This assumption is valid since we are interested in dilute suspension and in this case the hydrodynamic force is generally sufficient to prevent contacts.

Let's call Ω_f the fluid domain, B the particle domain and $\Omega = \Omega_f \cup B$ the whole domain. As explained above, the fluid is assumed to be governed by Stokes equations

$$-2\nu\nabla\cdot\boldsymbol{D}(\boldsymbol{u}) + \nabla p = 0 \text{ in } \Omega_f, \tag{16}$$

$$\nabla\cdot\boldsymbol{u} = 0 \text{ in } \Omega_f, \tag{17}$$

$$\boldsymbol{u} = \boldsymbol{f} \text{ on } \partial\Omega_f \setminus \partial B, \tag{18}$$

where : $\boldsymbol{u}$ is the velocity of the fluid, $\boldsymbol{D}(\boldsymbol{u}) = \dfrac{\nabla \boldsymbol{u} + \nabla \boldsymbol{u}^T}{2}$ the deformation tensor, p is the pressure of the fluid and $\boldsymbol{f}$ is a given function related to the boundary conditions on $\partial\Omega$ (those on ∂B will be treated later by penalty). Moreover, we impose that the mean of the pressure is zero by adding a Lagrange multiplier λ. This approach allows us to use direct solvers such as UMFPack, MUMPS or PASTIX otherwise only iterative solvers would have been accessible. Thus, the variational formulation reads :

Find $(\boldsymbol{u}, p, \lambda) \in H^1(\Omega_f)^2 \times L^2_0(\Omega_f) \times \mathbb{R}$ such that $\forall (\boldsymbol{v}, q, \mu) \in H^1(\Omega_f)^2 \times L^2_0(\Omega_f) \times \mathbb{R}$:

$$
\begin{aligned}
2\nu \int_{\Omega_f} \boldsymbol{D}(\boldsymbol{u}) : \boldsymbol{D}(\boldsymbol{v}) - \int_{\Omega_f} p \nabla \cdot \boldsymbol{v} & \\
+ \int_{\Omega_f} \lambda q &= 0, \qquad (19)\\
\int_{\Omega_f} q \nabla \cdot \boldsymbol{u} &= 0, \qquad (20)\\
\int_{\Omega_f} \mu p &= 0, \qquad (21)\\
\boldsymbol{u} &= \boldsymbol{f} \text{ on } \partial\Omega.
\end{aligned}
$$

and the corresponding FEEL++ code is given by the listing 9.

The particle is taken into account following [30] in which, one imposes a rigid-body constraint by a penalty method. From the physics point of view, this method consists to consider the particle as a part of the fluid with a "huge viscosity" and constrained to have a rigid-body motion. Thus, instead of integrating in the fluid domain Ω_f, we rewrite our variational formulation in the whole domain Ω while taking into account the following constraint

$$
\boldsymbol{D}(\boldsymbol{u}) = 0, \text{ in } B. \qquad (22)
$$

Listing 9. Stokes equations

```
// define polynomial orders for fluid and pressure
const int VELOCITY_ORDER = 2;
const int PRESSURE_ORDER = 1;

//define basis
typedef Lagrange<VELOCITY_ORDER, Vectorial> basis_veloc_type;
typedef Lagrange<PRESSURE_ORDER, Scalar> basis_pressure_type;
typedef Lagrange<0, Scalar> basis_lag_type;
typedef bases<basis_veloc_type, basis_pressure_type,
              basis_lag_type> basis_type;
// space type for (u,p,λ)
typedef FunctionSpace<mesh_type, basis_type> space_type;

typedef typename space_type::element_type element_type;
element_type U;

// velocity and its test function
auto u = U.template element<0>();
auto v = V.template element<0>();

// pressure and its test function
auto p= U.template element<1>();
auto q= V.template element<1>();

// Lagrange multiplier and its test function
auto lambda= U.template element<2>();
auto mu= U.template element<2>();

// define D(u)
auto deft = sym(gradt(u));
// define D(v)
auto def  = sym(grad(v));

form2(Xh, Xh, D)= integrate(elements(mesh),
                    2 * nu * trace( deft * trans(def)) );
form2(Xh, Xh, D)+= integrate(elements(mesh),
                    - div(v) * idt(p) + id(q) * divt(u) );
// add Lagrange multiplier to impose vanishing mean pressure
form2(Xh, Xh, D)+= integrate(elements(mesh),
                    idt(lambda) * id(q) + id(mu) * idt(p) );

//   boundary conditions
// on walls u=0
form2(Xh, Xh, D)+= on(markedfaces(mesh, "Walls"),
                      u, F, cst(0.) * N()  );
// parabolic profile on inlet
auto inflow = (Py()+H/2)*(Py()-H/2);
form2(Xh, Xh, D)+= on(markedfaces(mesh, "Inflow"),
                      u, F, inflow * N());
// parabolic profile on outlet with Rq the flow rate ration
auto outflowtop = Rq * (Px()+L/2)*(Px()-(L/2));
form2(Xh, Xh, D)+= on(markedfaces(mesh, "Outflowtop"),
                      u, F, - outflowtop * N() );
```

This rigid-body constraint (22) is added to equation (19) using a small enough penalty coefficient ε. Finally the new variational formulation reads

Find $(\boldsymbol{u}, p, \lambda) \in H^1(\Omega)^2 \times L^2_0(\Omega) \times \mathbb{R}$ such that $\forall (\boldsymbol{v}, q, \mu) \in H^1(\Omega)^2 \times L^2_0(\Omega) \times \mathbb{R}$:

$$
\begin{aligned}
2\nu \int_{\Omega} \boldsymbol{D}(\boldsymbol{u}) : \boldsymbol{D}(\boldsymbol{v}) - \int_{\Omega} p \nabla \cdot \boldsymbol{v} & \\
+ \int_{\Omega} \lambda q + \frac{2}{\varepsilon} \int_B \chi(\boldsymbol{D}(\boldsymbol{u}) : \boldsymbol{D}(\boldsymbol{v})) &= 0, \qquad (23)\\
\int_{\Omega} q \nabla \cdot \boldsymbol{u} &= 0, \qquad (24)\\
\int_{\Omega} \mu p &= 0, \qquad (25)\\
\boldsymbol{u} &= \boldsymbol{f} \text{ on } \partial\Omega.
\end{aligned}
$$

Listing 10. Add penalty term

```
// caracteristic function χ
// a circle of radius R and center X = (xp,yp,zp)
auto carac=vf::project(Yh, elements(mesh),
                  chi(        R * R >
                      (Px()-xp)*(Px()-xp) +
                      (Py()-yp)*(Py()-yp) +
                      (Pz()-zp)*(Pz()-zp) );

// calculate the integral of χ on each element
auto p0 = integrate(_range=elements(mesh),
                    _expr=idv(carac)).broken(P0h);

// for all the elements
for(auto it=p0.begin(),en=p0.end(); it != en; ++it )
// if χ pass through the element, mark the element as 1
    { if ( abs(*it) > 1e-10 ) *it = 1; else *it = 0; }

// store the marker
mesh->updateMarker3( p0 );
```

```
C=backend->newMatrix( Xh, Xh );
// add penalty contribution only on the marked elements
form2(Xh, Xh, C)=
   integrate(marked3elements(mesh,1),
                2*idv(carac)*nu*trace(def*trans(deft))/epsilon );

// add the penalty contribution to the whole problem
C->addMatrix( D );
```

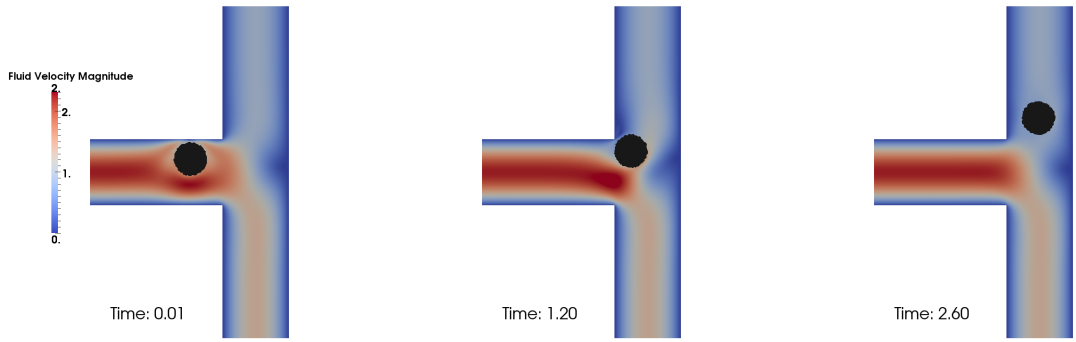
Note that the integral over B is performed by integrating over Ω while multiplying by the characteristic function χ (see listing 10). In practice, only the penalty term changes during time and has to be reassembled (e.g. the last term in equation (23)). Thus, only the penalty term is added to the whole problem at each iteration. Moreover, this term does vanishes on most elements of the mesh and instead of integrating the penalty term over all the elements and multiplying by χ, we only integrate on those crossed by χ using a special feature `broken()` of `integrate()` to localize these elements, see section 4.1.

Finally, we introduce the time step Δt and we denote by $(\boldsymbol{u}_n, p_n, \lambda_n)$ the solution of the system (23)-(24)-(25) at time $t_n = n\Delta t$. The velocity of the particle denoted by $\boldsymbol{V}_n$ is then calculated by the relation (26) and its position $\boldsymbol{X}_n$ by the relation (27).

$$\boldsymbol{V}_n = \frac{1}{\int_\Omega \chi} \int_\Omega \chi \, \boldsymbol{u}_n, \tag{26}$$

$$\boldsymbol{X}_{n+1} = \boldsymbol{X}_n + \Delta t \boldsymbol{V}_n, \tag{27}$$

Figure 12 shows a 2D simulation of a particle entering in the low flow rate branch while figures 13 and 14 display the results of the 3D simulation.



(a) $t = 0.01$ (b) $t = 1.20$ (c) $t = 2.60$

Figure 12. 2D simulation of a particle entering in the low flow rate channel.

7. 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) the seamless parallelisation of FEEL++— using both global computing resources thanks to MPI and local (compute node) available computing ressources thanks either to Intel TBB or CUDA/OpenCL, (ii) automatic differentiation of variational forms, (iii) integration of multiscale methods, (iv) a full framework for domain decomposition including a preconditioner framework and some preconditioners implementation, (v) the implementation of finite volume variational methods, see [15], (vi) 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.

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*

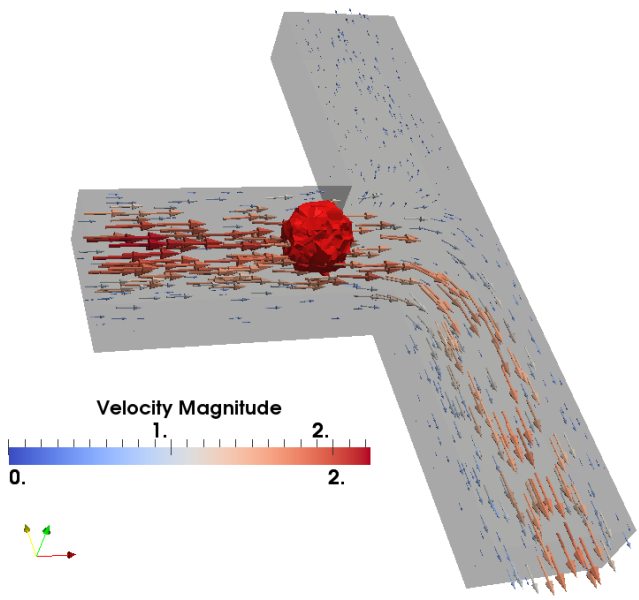
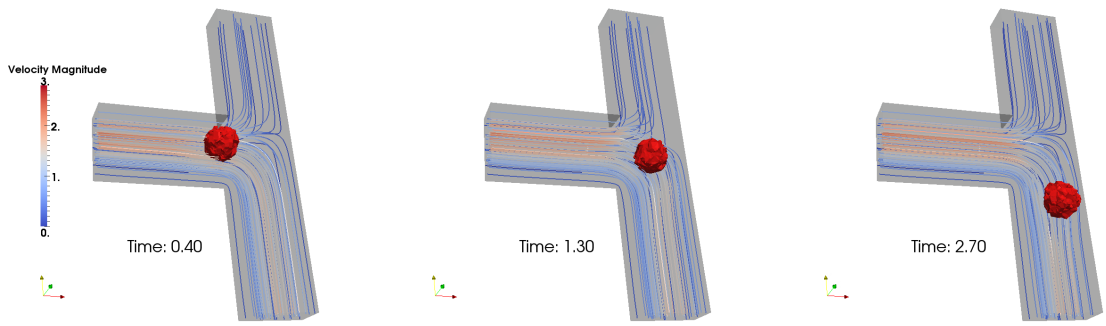


Figure 13. 3D simulation of the bifurcation problem. The velocity vectors are plotted and colored by the velocity magnitude.



(a) $t = 0.01$ (b) $t = 1.20$ (c) $t = 2.60$

Figure 14. 3D simulation of a particle entering in the high flow rate channel. Streamlines are represented and colored by the velocity magnitude.

à Investigação. Finally the authors wish to thank the Cemracs 2011 and its organizers which allowed us to develop within FEEL++ the numerical methods exposed in this paper.

A. FEEL++ LANGUAGE REFERENCE

Keyword	Math object	Description	Rank	$M \times N$
`P()`	$\overrightarrow{P}$	current point coordinates $(P_x, P_y, P_z)^T$	1	$d \times 1$
`Px()`	P_x	x coordinate of $\overrightarrow{P}$	0	1×1
`Py()`	P_y	y coordinate of $\overrightarrow{P}$ (value is 0 in 1D)	0	1×1
`Pz()`	P_z	z coordinate of $\overrightarrow{P}$ (value is 0 in 1D and 2D)	0	1×1
`N()`	$\overrightarrow{N}$	normal at current point $(N_x, N_y, N_z)^T$	1	$d \times 1$
`Nx()`	N_x	x coordinate of $\overrightarrow{N}$ at current point	0	1×1

Keyword	Math object	Description	Rank	$M \times N$
`Ny()`	N_y	y coordinate of $\overrightarrow{N}$ at current point (value is 0 in 1D)	0	1×1
`Nz()`	N_z	z coordinate of $\overrightarrow{N}$ at current point (value is 0 in 1D and 2D)	0	1×1
`J()`	$\nabla \phi_K^{\text{geo}}$	geometric mapping gradient associated with current K	2	$d \times n$
`invJT()`	$(\nabla \phi_K^{\text{geo}})^{-T}$	geometric mapping inverse gradient transposed	2	$d \times n$
`detJ()`	$\det \nabla \phi_K^{\text{geo}}$	geometric mapping inverse gradient transposed	0	1×1
`eid()`	e	index of Ω^e	0	1×1
`emarker()`	$m(e)$	marker of Ω^e	0	1×1
`h()`	h^e	size of Ω^e	0	1×1
`hFace()`	h_Γ^e	size of face Γ of Ω^e	0	1×1
`mat<M,N>(m_11,`	$\begin{pmatrix} m_{11} & m_{12} & \dots \\ m_{21} & m_{22} & \dots \\ \vdots & & \end{pmatrix}$	$M \times N$ matrix	2	$M \times N$
`m_12,...)`		entries being expressions		
`vec<M>(v_1,`	$(v_1, v_2, \dots)^T$	column vector with M rows	1	$M \times 1$
`v_2,...)`		entries being expressions		
`trace(expr)`	$\text{tr}(f(\overrightarrow{x}))$	trace of $f(\overrightarrow{x})$	0	1×1
`sym(A)`	$\frac{1}{2}(A + A^T)$	symmetric part of A	0	1×1
`antisym(A)`	$\frac{1}{2}(A - A^T)$	anti symmetric part of A	0	1×1
`abs(expr)`	$\lvert f(\overrightarrow{x}) \rvert$	element wise absolute value of f	$\text{rank}(f(\overrightarrow{x}))$	$m \times p$
`cos(expr)`	$\cos(f(\overrightarrow{x}))$	element wise cosinus value of f	$\text{rank}(f(\overrightarrow{x}))$	$m \times p$
`sin(expr)`	$\sin(f(\overrightarrow{x}))$	element wise sinus value of f	$\text{rank}(f(\overrightarrow{x}))$	$m \times p$
`tan(expr)`	$\tan(f(\overrightarrow{x}))$	element wise tangent value of f	$\text{rank}(f(\overrightarrow{x}))$	$m \times p$
`id(f)`	f	test function	$\text{rank}(f(\overrightarrow{x}))$	$m \times p$
`idt(f)`	f	trial function	$\text{rank}(f(\overrightarrow{x}))$	$m \times p$
`idv(f)`	f	evaluation function	$\text{rank}(f(\overrightarrow{x}))$	$m \times p$
`grad(f)`	∇f	gradient of test function	$\text{rank}(f(\overrightarrow{x})) + 1$	$p = 1,\ m \times$
`gradt(f)`	∇f	gradient of trial function	$\text{rank}(f(\overrightarrow{x})) + 1$	$p = 1,\ m \times$
`gradv(f)`	∇f	evaluation function gradient	$\text{rank}(f(\overrightarrow{x})) + 1$	$p = 1,\ m \times$
`div(f)`	$\nabla \cdot \overrightarrow{f}$	divergence of test function	$\text{rank}(f(\overrightarrow{x})) - 1$	1×1[6]
`divt(f)`	$\nabla \cdot \overrightarrow{f}$	divergence of trial function	$\text{rank}(f(\overrightarrow{x})) - 1$	1×1
`divv(f)`	$\nabla \cdot \overrightarrow{f}$	evaluation of function divergence	$\text{rank}(f(\overrightarrow{x})) - 1$	1×1
`curl(f)`	$\nabla \times \overrightarrow{f}$	curl of test function	1	$n = m, n \times$
`curlt(f)`	$\nabla \times \overrightarrow{f}$	curl of trial function	1	$m = n, n \times$
`curlv(f)`	$\nabla \times \overrightarrow{f}$	evaluation of function curl	1	$m = n, n \times$
`hess(f)`	$\nabla^2 f$	hessian of test function	2	$m = p = 1$
`jump(f)`	$[f] = f_0 \overrightarrow{N_0} + f_1 \overrightarrow{N_1}$	jump of test function	1	$m = 1, n \times$
`jump(f)`	$[\overrightarrow{f}] = \overrightarrow{f_0} \cdot \overrightarrow{N_0} + \overrightarrow{f_1} \cdot \overrightarrow{N_1}$	jump of test function	0	$m = 2, 1 \times$
`jumpt(f)`	$[f] = f_0 \overrightarrow{N_0} + f_1 \overrightarrow{N_1}$	jump of trial function	1	$m = 1, n \times$
`jumpt(f)`	$[\overrightarrow{f}] = \overrightarrow{f_0} \cdot \overrightarrow{N_0} + \overrightarrow{f_1} \cdot \overrightarrow{N_1}$	jump of trial function	0	$m = 2, 1 \times$
`jumpv(f)`	$[f] = f_0 \overrightarrow{N_0} + f_1 \overrightarrow{N_1}$	jump of function evaluation	1	$m = 1, n \times$
`jumpv(f)`	$[\overrightarrow{f}] = \overrightarrow{f_0} \cdot \overrightarrow{N_0} + \overrightarrow{f_1} \cdot \overrightarrow{N_1}$	jump of function evaluation	0	$m = 2, 1 \times$
`average(f)`	$f = \frac{1}{2}(f_0 + f_1)$	average of test function	$\text{rank}(f(\overrightarrow{x}))$	$m = n, n \times$
`averaget(f)`	$f = \frac{1}{2}(f_0 + f_1)$	average of trial function	$\text{rank}(f(\overrightarrow{x}))$	$m = n, n \times$
`averagev(f)`	$f = \frac{1}{2}(f_0 + f_1)$	average of function evaluation	$\text{rank}(f(\overrightarrow{x}))$	$m = n, n \times$

[5] Gradient of matrix value functions is not implemented, hence $p = 1$

[6] Divergence of matrix value functions is not implemented, hence $p = 1$

Keyword	Math object	Description	Rank	$M \times N$
`leftface(f)`	f_0	left test function	$\text{rank}(f(\overrightarrow{x}))$	$m = n, n \times$
`leftfacet(f)`	f_0	left trial function	$\text{rank}(f(\overrightarrow{x}))$	$m = n, n \times$
`leftfacev(f)`	f_0	left function evaluation	$\text{rank}(f(\overrightarrow{x}))$	$m = n, n \times$
`rightface(f)`	f_1	right test function	$\text{rank}(f(\overrightarrow{x}))$	$m = n, n \times$
`rightfacet(f)`	f_1	right trial function	$\text{rank}(f(\overrightarrow{x}))$	$m = n, n \times$
`rightfacev(f)`	f_1	right function evaluation	$\text{rank}(f(\overrightarrow{x}))$	$m = n, n \times$
`maxface(f)`	$\max(f_0, f_1)$	maximum of right and left test function	$\text{rank}(f(\overrightarrow{x}))$	$m \times p$
`maxfacet(f)`	$\max(f_0, f_1)$	maximum of right and left trial function	$\text{rank}(f(\overrightarrow{x}))$	$m \times p$
`maxfacev(f)`	$\max(f_0, f_1)$	maximum of right and left function evaluation	$\text{rank}(f(\overrightarrow{x}))$	$m \times p$
`minface(f)`	$\min(f_0, f_1)$	minimum of right and left test function	$\text{rank}(f(\overrightarrow{x}))$	$m \times p$
`minfacet(f)`	$\min(f_0, f_1)$	minimum of right and left trial function	$\text{rank}(f(\overrightarrow{x}))$	$m \times p$
`minfacev(f)`	$\min(f_0, f_1)$	minimum of right and left function evaluation	$\text{rank}(f(\overrightarrow{x}))$	$m \times p$
`-`	$-g$	element wise unary minus		
`-`	$!g$	element wise logical not		
`+`	$f+g$	tensor sum		
`-`	$f-g$	tensor substraction		
`*`	$f*g$	tensor product		
`/`	f/g	tensor division (g scalar field)		
`<`	$f<g$	element wise less		
`<=`	$f \leq g$	element wise less or equal		
`>`	$f>g$	element wise greater		
`>=`	$f \geq g$	element wise greater or equal		
`==`	$f=g$	element wise equal		
`!=`	$f \neq g$	element wise not equal		
`&&`	f and g	element wise logical and		
`||`	f or g	element wise logical or		

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] F. Ben Belgacem and Y. Maday. The mortar element method for three-dimensional finite elements. *R.A.I.R.O. Modél. Math. Anal.*, 31:289–302, 1997.

[8] S. Bertoluzza, M. Ismail, and B. Maury. The fat boundary method: semi-discrete scheme and some numerical experiments. *Domain decomposition methods in science and engineering*, pages 513–520, 2005.

[9] S. Bertoluzza, M. Ismail, and B. Maury. Analysis of the fully discrete fat boundary method. *Numerische Mathematik*, 118(1):49–77, 2011.

[10] S. Bertoluzza and M. Pennacchio. Analysis for substructuring preconditioners for mortar methods in an abstract framework. *Appl. Math. Lett.*, 35:131–137, 2007.
[11] F. Brezzi and L.D. Marini. A three-fields domain decomposition method. *Domain Decomposition Methods in Science and Engineering*, pages 27–34, 1994.
[12] Claudio Canuto, M. Yousuff Hussani, Alfio Quarteroni, and Thomas A. Zang. *Spectral Methods: Fundamentals in Single Domains*. Springer-Verlag, New York and Berlin, 2006.
[13] V. Chabannes, C. Prud'homme, and G. Pena. High order fluid structure interaction in 2D and 3D: Application to blood flow in arteries. unpublished, 2012.
[14] Philippe G. Ciarlet. *Finite Element Method for Elliptic Problems*. Society for Industrial and Applied Mathematics, Philadelphia, PA, USA, 2002.
[15] 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. December 2011.
[16] 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 (Submitted)*, 2012.
[17] V. Doyeux, T. Podgorski, S. Peponas, M. Ismail, and G. Coupier. Spheres in the vicinity of a bifurcation: elucidating the zweifach-fung effect. *Journal of Fluid Mechanics*, 674:359–388, 2011.
[18] Patrick Dular and Christophe Geuzaine. Getdp: a general environment for the treatment of discrete problems. http://www.geuz.org/getdp.
[19] 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.
[20] V. Girault and R. Glowinski. Error analysis of a fictitious domain method applied to a dirichlet problem. *Japan Journal of Industrial and Applied Mathematics*, 12(3):487–514, 1995.
[21] R. Glowinski and T.W. Pan. Error estimates for fictitious domain/penalty/finite element methods. *Calcolo*, 29(1):125–141, 1992.
[22] R. Glowinski, T.W. Pan, and J. Periaux. A fictitious domain method for dirichlet problem and applications. *Computer Methods in Applied Mechanics and Engineering*, 111(3-4):283–303, 1994.
[23] Frédéric Hecht and Olivier Pironneau. *FreeFEM++ Manual*. Laboratoire Jacques Louis Lions, 2005.
[24] 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.
[25] 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.
[26] 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.
[27] Michael A. Heroux and James M. Willenbring. Trilinos Users Guide. Technical Report SAND2003-2952, Sandia National Laboratories, 2003.
[28] Michael A. Heroux, James M. Willenbring, and Robert Heaphy. Trilinos Developers Guide. Technical Report SAND2003-1898, Sandia National Laboratories, 2003.
[29] M. Ismail. *Méthode de la frontière élargie pour la résolution de problèmes elliptiques dans des domaines perforés. Application aux écoulements fluides tridimensionnels*. PhD thesis, Université Pierre et Marie Curie, Paris, France, 2004.
[30] João Janela, Aline Lefebvre, and Bertrand Maury. A penalty method for the simulation of fluid - rigid body interaction. *ESAIM: Proc.*, 14:115–123, 2005.
[31] George Em Karniadakis and Spencer J. Sherwin. *Spectral/hp element methods for CFD*. Numerical Mathematics and Scientific Computation. Oxford University Press, New York, 2005.
[32] Robert C. Kirby. Algorithm 839: Fiat, a new paradigm for computing finite element basis functions. *ACM Trans. Math. Softw.*, 30(4):502–516, 2004.
[33] Robert C. Kirby. Optimizing fiat with level 3 blas. *ACM Trans. Math. Softw.*, 32(2):223–235, 2006.
[34] Robert C. Kirby. Singularity-free evaluation of collapsed-coordinate orthogonal polynomials. *ACM Trans. Math. Softw.*, 37(1):1–16, 2010.
[35] Robert C. Kirby and Anders Logg. A compiler for variational forms. *ACM Trans. Math. Softw.*, 32(3):417–444, 2006.
[36] Robert C. Kirby and Anders Logg. Efficient compilation of a class of variational forms. *ACM Trans. Math. Softw.*, 33(3):17, 2007.
[37] Kevin Long. Sundance: Rapid development of high-performance parallel finite-element solutions of partial differential equations. http://software.sandia.gov/sundance/.
[38] B. Maury. A fat boundary method for the poisson problem in a domain with holes. *Journal of scientific computing*, 16(3):319–339, 2001.
[39] B. Maury. Numerical analysis of a finite element/volume penalty method. *Partial Differential Equations*, pages 167–185, 2008.
[40] 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.
[41] 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.
[42] 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.
[43] Stéphane Del Pino and Olivier Pironneau. *FreeFEM3D Manual*. Laboratoire Jacques Louis Lions, 2005.
[44] 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.
[45] 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.

[46] A. Quarteroni and A. Valli. *Domain decomposition methods for partial Differential equations*, chapter The Mathematical Fundation of Domain Decomposition Methods, pages 1–39. Numerical Mathematics and Scientific Computation. Oxford University Press, New York, July 1999.

[47] I. Ramiere. Convergence analysis of the q1-finite element method for elliptic problems with non-boundary-fitted meshes. *International Journal for Numerical Methods in Engineering*, 75(9):1007–1052, 2008.

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

[49] PEJ Vos, R. van Loon, and SJ Sherwin. A comparison of fictitious domain methods appropriate for spectral/hp element discretisations. *Computer Methods in Applied Mechanics and Engineering*, 197(25):2275–2289, 2008.

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

[51] Yuan-Cheng and Fung. Stochastic flow in capillary blood vessels. *Microvascular Research*, 5(1):34 – 48, 1973.