Life: Overview of a Unified C++ Implementation of the Finite and Spectral Element Methods in 1D, 2D and 3D

Christophe Prud'homme[1]

Université Joseph Fourier, Laboratoire Jean Kuntzmann,
51 rue des Mathématiques, 38041 Grenoble, France,
christophe.prudhomme@ujf-grenoble.fr

Abstract. This article presents an overview of a unified framework for finite element and spectral element methods in 1D, 2D and 3D in *C++* called Life. The objectives of this framework are quite ambitious and could be expressed in various ways: *(i)* the creation of a versatile mathematical kernel allowing for solving easily problems using different techniques thus allowing testing and comparing methods, e.g. cG versus dG, *(ii)* the creation of a *small* and *manageable* library which shall nevertheless encompass a wide range of numerical methods and techniques and *(iii)* build mathematical software that follows closely the mathematical abstractions associated with the partial differential equations solved.

1 Introduction and Basic Principles

This article presents an overview of a unified framework for finite element and spectral element methods in 1D, 2D and 3D in *C++*, called Life. The objectives of this framework is quite ambitious and could be expressed in various ways such as *(i)* the creation of a versatile mathematical kernel easily solving problems using different techniques thus allowing testing and comparing methods, e.g. continuous Galerkin (cG) versus discontinuous Galerkin (dG) , *(ii)* the creation of a *small* and *manageable* library which shall nevertheless encompass a wide range of numerical methods and techniques, *(iii)* build mathematical software that follows closely the mathematical abstractions associated with partial differential equations (PDE) — which is often not the case, for example the design could be physics oriented, — and *(iv)* the creation of a library entirely in *C++* allowing to create complex multi-physics applications such as fluid-structure interaction or mass transport in haemodynamics — the rationale being that these applications are computing intensive and the use of an interpreted language such as Python would not be satisfying though in many simpler cases that would simplify and accelerate the development.

Now we describe a few requirements or features and general considerations that the library tries to satisfy:

- The syntax, the semantics and the pragmatics of the library are very close to the mathematics; the design and implementation should be mathematics

oriented. C++ supports very well multiple paradigms design and offers a wide range of solutions for a given problem. Generic programming, OO programming, meta-programming are such paradigms and they are definitely very useful when dealing with mathematical abstractions.
- The library shall remain small and manageable and make use wherever possible of established libraries. In particular, it uses most of the Boost[1] libraries. To name but a few: ublas, lambda, mpl, preprocessor, serialization, multi_index.
- The library should compare reasonably well with other similar frameworks performance wise. Partial specialization in C++ [2] allows to select efficient algorithms at compile time. Proper benchmarking shall be available soon.
- The library should achieve a certain level of numerical type independence: that is being able to use different numerical types such as the standard ones — `float`, `double`, `long double` or `std::complex<>`, — double-double and quad-double, see [13], arbitrary precision, see [6] or interval arithmetic; numerical classes are then parametrized by the numerical type and mathematical functions are available in a unified way. To achieve this, it uses the Boost.preprocessor library and provides through a `math::` namespace a set of mathematical functions for all supported numerical types. One should note that the library takes care of type deduction in operations using different numerical types.
- Wherever geometry is concerned, the dimension — up to 3D for now, but it is open to higher dimensions — is a template parameter of the data structures.
- Wherever possible, computing intensive operations are expressed in an algebraic way and use when possible appropriate efficient libraries, the computations should be mostly driven by linear algebra. For standard numerical types, libraries like BLAS and LAPACK can then be used for efficient calculations. Again partial specialization in C++ [2] is at work here.
- The library delegates linear algebra in the sense that well maintained third party libraries are used through unified interfaces using the Facade design pattern, see [10]. At the moment, the library supports gmm [24], Boost.ublas, PETSc [7] and Trilinos [12].
- The library should provide the tools to generate parallel applications. The parallelisation of the library is built on top of the PETSc and Trilinos frameworks, (Par)METIS [15] and it provides iterators over the partitioned mesh.

There are many freely available libraries which offer the capabilities described previously to a certain extent. To name but a few: the Freefem software family [11,22], the Fenics project [19,18], Getdp [9] or Getfem++ [24], or libraries or frameworks such as LifeV (C++) [1,21], deal.II(C++) [8], Sundance (C++) [20], Analysa (Scheme) [5]. 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.

[1] http://www.boost.org

The main part of the article will describe the general ideas in various areas that drove the design and implementation: first we shall present the polynomial library, then the function spaces, the operators or forms and the domain specific language embedded into C++ for variational formulations.

2 A 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 the simplices and quadrilaterals in $nD, n = 1, 2, 3$. Higher dimensions will eventually be implemented. Prisms and pyramids are not yet supported. The convexes are described geometrically in a standard way in terms of their subentities vertices, edges, faces, volumes. Then they are wrapped around two classes `Reference<>` and `Real<>` which define the interface for the reference convex and the convex in the real space. Again, the crucial albeit standard point is the decomposition in subentities and the capacity 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, we need to choose a prime basis. Often one chooses the canonical one — also known as the moment or monomial basis. — Recent work by R.C. Kirby [16,17] proposed to use the Dubiner polynomials as a prime basis on the simplex. We extended these ideas on the quadrilaterals using the Legendre polynomials. Other 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 thanks to the L_2 orthogonality property of these polynomials which allows for: *(i)* easily extracting a basis spanning a subspace of a polynomial space — which corresponds to extract a range of coefficients, — *(ii)* simplifying some operations like numerical integration or the L_2 projection which is now explicit and *(iii)* better numerical stability.

A brief digression on the Dubiner and Legendre polynomials: they are constructed by taking the products of principal functions based on the Jacobi polynomials denoted $P_p^{\alpha,\beta}$ on $[-1;1]$ where $\alpha, \beta > -1$ and satisfying the L_2 orthogonality relationship:

$$\int_{-1}^{1} (1-x)^{\alpha}(1+x)^{\beta} P_p^{\alpha,\beta}(x) P_q^{\alpha,\beta}(x) = C_{\alpha,\beta}\delta_{pq} \tag{1}$$

The case $\alpha = \beta = 0$ recovers the Legendre polynomials. The Dubiner polynomials are constructed by introducing a collapsed coordinate system and proper weighting functions associated with some Jacobi polynomials to recover the L_2 orthogonality, see [14] page 101 for more details. In practice, the prime basis is normalized.

2.3 Point Sets on Convexes

Now we turn to the construction of point sets in the reference convex. Point sets are represented by a matrix, they are parametrized by the associated convex and the numerical type. 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 Galerkin formulations for example.

The type of point sets supported are *(i)* the equidistributed point set, *(ii)* the warpblend point sets on simplices proposed by T. Warburton [25], *(iii)* Fekete points in simplices, see [14], *(iv)* standard quadrature rules in simplices and finally *(v)* Gauss, Gauss Radau and Gauss Lobatto and combinations in simplices and quadrilaterals. It should be noted that the last family is constructed from the computation of the zero of the Legendre polynomials on $[-1; 1]$ including eventually the boundary vertices -1, 1 for the Radau and Lobatto flavors. The kernel of the polynomial library is the construction of Jacobi polynomials and the computation of their zeros.

Warpblend and Fekete points are used with nodal basis which, when constructed at these points, present much better interpolation properties — lower Lebesgue constant, see [14] — Note that the Gauss-Lobatto points are Fekete points in the quadrilateral — i.e. they maximize the determinant of the associated generalized Vandermonde matrix.

2.4 Polynomial Set

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

They are template classes parametrized by the prime basis in which they are expressed and the field type in which they have their values: scalar, vectorial or matricial. Their interface provides a number of operations such as evaluation and derivation at a set of points, extraction of polynomials from a polynomial set or components of a polynomial or polynomial set when the `FieldType` is `Vectorial` or `Matricial`.

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 matrices 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 quadrilaterals and Warpblend or Fekete points for simplicies as they

provide a much better conditioning for the Vandermonde matrix. A trick is to do these computations using higher precision types, e.g. `dd_real`, and then fall back in the end to the required numerical type, e.g. `double`. This trick provides a appreciable gain in accuracy.

2.5 Finite Elements and Other Polynomial Basis

Now we turn to finite elements(FE) which we define and construct in a reference element. The reference FE are usually described by the triplet $(K, \mathbb{P}, \Sigma)$ where K is a convex, $\mathbb{P}$ the polynomial space and Σ the dual space. We have already seen the ingredients for K and $\mathbb{P}$, it remains to describe Σ. Σ is a set of functionals — degrees of freedom — taking their values in $\mathbb{P}$ with values in a scalar, vectorial or matricial field.

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. Here are some examples of functionals:

- Evaluation at a point $x \in K$, $\ell_x : p \to p(x)$
- Derivation at a point $x \in K$ in the direction d, $\ell_{x,d} : p \to \dfrac{\partial p}{\partial x_d}(x)$
- 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. 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 equispaced, warpblend or Fekete point sets. Other FE such as $\mathbb{P}_{1,2}$-bubble, $\mathbb{RT}_k$, $\mathbb{N}_k$ or polynomials are constructed likewise though they are more complex.

3 Meshes, Function Spaces and Operators

In the previous section, we have described roughly the mono-domain construction of polynomials, now we turn to the ingredients for the multi-domain construction. We start with some remarks on the mesh data, then the function spaces abstraction and finally the concept of forms.

3.1 Mesh Data Structures

While performing integration and projection, it is common to be able to extract parts of the mesh. We wish to extract easily subsets of convexes out of the total set constituting the mesh.

The mesh data structure uses the Boost.Multi_index library[2] to store the elements, elements faces, edges and points. This way the mesh entities 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..., — 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 entities — elements, faces, edges, points — containers depending on the usage context. The pairs of iterators are then turned into a range, see Boost.Range[3], to be manipulated by the integration and projection tools.

A number of free functions are available that hide all details about the mesh class to concentrate only on the relevant parts, here are two examples:

- `elements(<mesh>,<pid>)` the set of convexes constituting the `<mesh>` associated with the current process id
- `markedfaces(<mesh>,<marker>,<pid>)` the set of faces marked by `<marker>` of the `<mesh>` and belonging to the process id `<pid>`

3.2 Function Spaces and Functions

Recall that we want the framework to follow the mathematical abstraction as closely as possible. We therefore introduce naturally the `FunctionSpace` abstraction which is parametrized by the mesh type, the basis type and the numerical type, see listing 1.1. Its role is to construct the degrees of freedom table, embed the creation of elements of the space, and store interpolation data structures such as localization trees.

Functions spaces can be also defined as a product of other function spaces. This is very powerful to describe complex multiphysics problems when coupled with operators/forms described in the next section. Extracting subspaces or component spaces are part of the interface. The most important feature is that it embeds the definition of element which allows for 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 the introduction — gmm, PETSc or Trilinos — that will then be subsequently used to solve the PDE. Other supported operations are interpolation and extraction of components — be it a product of function spaces element or a vectorial/matricial element, — see listing 1.1 for an example of `FunctionSpace::Element`.

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

3.3 Operators and Forms

One last concept needed to have the variational formulation language mathematically expressive is the notion of forms. They follow closely their mathematical counterparts: they are template classes with arguments being the space or product of spaces they take as input and the algebraic representation of these forms. In what follows, we consider only the case where the linear and bilinear forms are represented by vectors and matrices respectively — we could consider also vector-free and matrix-free representations.
The crucial point is that the linear and bilinear form classes are the glue between their algebraic representation and the variational formulation which is stored in a complex tree-like data structure that we denote `Expr`, it will

- Fill the matrix with non-zero entries depending on the approximation space(s) and the mathematical expression;
- Allow a per-component/per-space construction(blockwise);
- Check that the numerical types of the expression and the representation are consistent
- When `operator=( Expr const& )` is called, the expression is evaluated and will fill the representation entries

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

4 A Variational Formulation Language Embedded in C++

In order to express the PDE problems, libraries or applications rely either on a domain specific language usually written in an interpreted language such as Python or a home made one or on high level interfaces. These interfaces or languages are desirable for several reasons: teaching purposes, solving complex problems with multiple physics and scales or rapid prototyping of new methods, schemes or algorithms. The goal is always to hide (ideally all) technical details behind software layers and provide only the relevant components required by the user or programmer.

In the library, a domain specific language *embedded* — later written DSEL — in C++ has been implemented. This language for numerical integration and projection has been proposed by the author in [23]. The DSEL approach has advantages over generating a specific *external* language like in case *(i)* : compiler construction complexities can be ignored, other libraries can concurrently be used which is often not the case of specific languages which would have to also develop their own libraries and DSELs inherit the capabilities of the language in which they are written. However, DSELs are often defined for one particular task inside a specific domain [26] and implementation or parts of implementation are not shared between different DSELs. Here, we have a language that shares the expression tree construction between integration, projection and automatic differentiation.

The implementation of the language relies on standard techniques such as expression templates [26,2,3,4,21] and an involved two stages evaluation of the expression tree. Without dwelling too much upon the internals of the language, it is interesting to note that it supports some optimisation with respect to the product of N function spaces. Indeed let's consider the statements in listing 1.1: depending on the block to be filled by the local/global assembly process, some terms must be equated to 0. The language detects that at compile time and thanks to the g++ compiler it will not consider the terms which are not applying to the currently assembled block, see [23] for more details.

5 Example: A Navier-Stokes Solver for the Driven Cavity

We wish to solve the 3D incompressible Navier-Stokes equations [14] in a domain $\Omega = [0,1]^3$ in parallel with a $\mathbb{P}_2/\mathbb{P}_1$ discretization for the velocity and pressure respectively, that is to say, $(u,p) \in X_h \times M_h$ such that for all $(v,q) \in X_h \times M_h$, $\int_\Omega \frac{\partial u}{\partial t} v + \nu \nabla u \cdot \nabla v + u \cdot \nabla u v - \nabla \cdot v \, p = 0$ with $\int_\Omega \nabla \cdot u \, q = 0$, $u_{\Gamma_2} = (1,0,0)^T$ on Γ_2 which is the top face of unit domain Ω and $u_{\Gamma_1} = (0,0,0)^T$ on $\Gamma_1 = \partial\Omega \backslash \Gamma_2$ and X_h and M_h being adequate spaces the velocity and pressure respectively. As a parallel linear algebra backend, Trilinos is used.

Listing 1.1: A 3D incompressible Navier-Stokes example

```
// mesh of linear tetrahedron
typedef Mesh<Simplex<3,1> > mesh_t;
// define and partition the mesh if in parallel
mesh_t mesh; mesh.partition();
typedef fusion::vector<Lagrange<3,2,Vectorial>,
                       Lagrange<3,1,Scalar> > basis_t;
typedef FunctionSpace<mesh_t,basis_t> space_type;
space_type V_h( mesh );
space_type::Element<trilinos> U( V_h ), V( V_h );
// views for U and V: velocity
space_type::element<0,trilinos>::type u = U.element<0>();
space_type::element<0,trilinos>::type v = V.element<0>();
// views for U and V: pressure
space_type::element<1,trilinos>::type p = U.element<1>();
space_type::element<1,trilinos>::type q = V.element<1>();
// integration method : Gauss points on the Simplex in 3D
// that integrates exactly polynomials of degree 5
IM<3,5,double,Simplex,Gauss> im;
for( double t = dt; t < T; t+= dt ) {
  // ops: id, dx,dy,dz, grad...
  // when no suffix, it identifies test basis functions
  // the t suffix identifies trial basis functions
  // the v suffix denotes the interpolated value
  // integrate(<set of elements>,<integration>,<expression>);
  backend<trilinos>::vector_type F;
  form( V_h, F) f =
    integrate( elements(mesh), im,
     (idv(u) - dt*gradv(p))*id(v) );
  backend<trilinos>::matrix_type A;
  form( V_h, V_h, A) a =
    integrate( elements(mesh), im,
```

```
      nu*dt*dot(gradt(u), grad(v)) + idt(u)*id(v) +
      dt*(dot( idv(u), gradt(u) ))*id(v)
      - idt(p)*div(v) + id(q)*divt(u) )+
     on( 10, u, F, oneX() )+ // 10 identifies Γ1
     on( 20, u, F, 0 ); // 20 identifies Γ2
  A.close(); // final assembly
}
```

6 Conclusion and Main Results

Life is a mathematical library with an emphasis on *(i)* approximation methods while the linear algebra is delegated to third party libraries and *(ii)* the ability to solve partial differential equations using a language embedded in *C++* very close to the mathematics. This allows the user to concentrate on the problem at hand and manipulate high level abstractions and not being bothered by the programming details. Life will also become the new mathematical kernel of the LifeV project (www.lifev.org) whose objective is to develop solvers for multi-scale multiphysics applications in particular in haemodynamics.

Acknowledgments

The author wishes to acknowledge the chair of Prof. A. Quarteroni at EPFL which funded this research, and E. Burman, C. Winkelmann, G. Steiner and G. Pena from EPFL for the many fruitful discussions we had.

References

1. Lifev: a finite element library. http://www.lifev.org.
2. David Abrahams and Aleksey Gurtovoy. *C++ Template Metaprogramming : Concepts, Tools, and Techniques from Boost and Beyond*. C++ in Depth Series. Addison-Wesley Professional, 2004.
3. Pierre Aubert and Nicolas Di Césaré. Expression templates and forward mode automatic differentiation. In George Corliss, Christèle Faure, Andreas Griewank, Laurent Hascoët, and Uwe Naumann, editors, *Automatic Differentiation of Algorithms: From Simulation to Optimization*, Computer and Information Science, chapter 37, pages 311–315. Springer, New York, NY, 2001.
4. Pierre Aubert, Nicolas Di Césaré, and Olivier Pironneau. Automatic differentiation in C++ using expression templates and application to a flow control problem. *Computing and Visualisation in Sciences*, 3(4):197–208, January 2001.
5. Babak Bagheri and Ridgway Scott. Analysa. http://people.cs.uchicago.edu/ ridg/al/aa.ps, 2003.
6. David H. Bailey, Yozo Hida, Karthik Jeyabalan, Xiaoye S. Li, and Brandon Thompson. C++/fortran-90 arbitrary precision package. http://crd.lbl.gov/~dhbailey/mpdist/.
7. Satish Balay, Kris Buschelman, Victor Eijkhout, William D. Gropp, Dinesh Kaushik, Matthew G. Knepley, Lois Curfman McInnes, Barry F. Smith, and Hong Zhang. PETSc users manual. Technical Report ANL-95/11 - Revision 2.1.5, Argonne National Laboratory, 2004.

8. Wolfgang Bangerth, Ralf Hartmann, and Guido Kanschat. `deal.II` *Differential Equations Analysis Library, Technical Reference*. `http://www.dealii.org`.
9. Patrick Dular and Christophe Geuzaine. Getdp: a general environment for the treatment of discrete problems. http://www.geuz.org/getdp.
10. E. Gamma, R. Helm, R. Johnson, and J. Vlissides. *Design Patterns*. Addison Wesley Professional Computing Series. Addison Wesley, 1995.
11. Frédéric Hecht and Olivier Pironneau. *FreeFEM++ Manual*. Laboratoire Jacques Louis Lions, 2005.
12. 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.
13. Yozo Hida, Xiaoye S. Li, and David H. Bailey. Quad-double arithmetic: Algorithms, implementation, and application. Technical Report LBNL-46996, Lawrence Berkeley National Laboratory, Berkeley, CA 9472, Oct. 2000. `http://crd.lbl.gov/~dhbailey/mpdist/`.
14. George Em Karniadakis and Spencer J. Sherwin. *Spectral/hp element methods for CFD*. Numerical Mathematics and Scientific Computation. Oxford University Press, New York, 2005.
15. George Karypis and Vipin Kumar. Parallel Multilevel k-way Partitioning Schemes for Irregular Graphs. Technical Report 036, Minneapolis, MN 55454, May 1996.
16. Robert C. Kirby. Algorithm 839: Fiat - a new paradigm for computing finite element basis functions. *ACM Trans. Math. Software*, 30(4):502–516, 2004.
17. Robert C. Kirby. Optimizing fiat with level 3 blas. *ACM Trans. Math. Software*, 2005.
18. Robert C. Kirby and Anders Logg. A compiler for variational forms. Technical report, Chalmers Finite Element Center, 05 2005. `www.phi.chalmers.se/preprints`.
19. A. Logg, J. Hoffman, R.C. Kirby, and J. Jansson. Fenics. `http://www.fenics.org/`, 2005.
20. Kevin Long. Sundance: Rapid development of high-performance parallel finite-element solutions of partial differential equations. http://software.sandia.gov/sundance/.
21. Daniele A. Di Pietro and Alessandro Veneziani. Expression templates implementation of continuous and discontinuous galerkin methods. *Submitted to Computing and Visualization in Science*, 2005.
22. Stéphane Del Pino and Olivier Pironneau. *FreeFEM3D Manual*. Laboratoire Jacques Louis Lions, 2005.
23. 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.
24. Yves Renard and Julien Pommier. Getfem++: Generic and efficient c++ library for finite element methods elementary computations. http://www-gmm.insa-toulouse.fr/getfem/.
25. T.Warburton. An explicit construction for interpolation nodes on the simplex. *Journal of Engineering Mathematics*, 2005. Submitted.
26. Todd Veldhuizen. Using C++ template metaprograms. *C++ Report*, 7(4):36–43, May 1995. Reprinted in C++ Gems, ed. Stanley Lippman.