SIAM J. SCI. COMPUT.
Vol. 40, No. 2, pp. C234–C257

FELICITY: A MATLAB/C++ TOOLBOX FOR DEVELOPING FINITE ELEMENT METHODS AND SIMULATION MODELING*

SHAWN W. WALKER†

Abstract. This paper describes a MATLAB/C++ finite element toolbox, called FELICITY, for simulating various types of systems of partial differential equations (e.g., coupled elliptic/parabolic problems) using the finite element method. It uses MATLAB in an object-oriented way for high-level manipulation of data structures in finite element codes, while utilizing a domain-specific language (DSL) and code generation to automate low-level tasks such as matrix assembly (via the MATLAB `mex` interface). We describe the fundamental functionality of the toolbox's MATLAB interface, such as using higher order Lagrange (simplicial) meshes, defining finite element spaces, allocating degrees-of-freedom, assembling discrete bilinear and linear forms, and interpolation over meshes. Moreover, we describe in-depth how automatic code generation is implemented in FELICITY. Two example problems and their implementation are provided to demonstrate the ability of FELICITY to solve coupled problems with interacting subdomains of different co-dimension. Future improvements are also discussed.

Key words. finite elements, coupled systems, geometric flows, code generation, MATLAB, open source software

AMS subject classifications. 68N30, 65N30, 65M60, 68N19, 68N20

DOI. 10.1137/17M1128745

1. Introduction.

The development of numerical methods to solve partial differential equations (PDEs) continues to advance to address new domain areas and problems of increasing complexity. With this, the number of software packages has increased to address the needs for researching new methods and simulating large problems. In particular, there are several finite element (FE) packages (of varying design) available to simulate a wide variety of PDEs, such as `GRINS` [14], `feelpp` [1], `FEniCS` [38, 8], `deal.II` [11], `DUNE` [26], `freeFEM++` [32], `GetFEM++` [2], `LifeV` [3], `mfem` [4], `milamin` [25], `SfePy` [24], `oomph-lib` [33], and `Vega` [5].

This paper describes the FE toolbox FELICITY: Finite ELement Implementation and Computational Interface Tool for You. It can be used to develop new finite element methods (FEMs) for coupled multiphysics problems and nonlinear problems, especially those that involve geometric information, such as surface tension-driven flows. Some highlights of the toolbox are as follows:

- It provides a general framework to tackle many types of problems.
- A domain-specific-language (DSL) easily allows development of a special purpose code for a single problem.
- The MATLAB interface provides a powerful numerical computing environment at a high level.
- Automatic code generation, combined with the `mex`/C++ interface, maintains performance (e.g., for matrix assembly).

1.1. Motivation.

Many FE packages are available, both commercial and free, so "why another FE toolbox?" The original purpose in designing FELICITY (beginning

*Submitted to the journal's Software and High-Performance Computing section May 4, 2017; accepted for publication (in revised form) January 19, 2018; published electronically March 15, 2018.
http://www.siam.org/journals/sisc/40-2/M112874.html
Funding: This work was supported by NSF grants DMS-1418994 and DMS-1555222.
†Department of Mathematics, Louisiana State University, Baton Rouge, LA 70803 (walker@math.lsu.edu).

in 2010) was for solving elliptic and parabolic PDEs, where subdomains of different co-dimension can interact. Some examples are as follows:

- geometric flow problems, such as mean curvature flow and Willmore flow (e.g., [54, 17]), where PDEs on surfaces are solved using surface FE approaches [29];
- problems where surface PDEs (e.g., Laplace–Beltrami) are coupled to PDEs in a bulk domain, e.g., diffusion of surfactants in droplets;
- multiphysics models that use Lagrange multipliers (defined on interfaces) to couple different physical models together;
- moving domain/interface problems that use front-tracking with conforming meshes to model moving boundaries.

In all these cases, one is confronted with defining multiple FE spaces over subdomains of, say, co-dimension 1 that must interact with FE spaces over the bulk domain through boundary integral terms. In addition, there may be mesh movement/deformation issues to deal with.

To the best of our knowledge, there were no FE toolboxes during the *initial* development of FELICITY that could easily handle FE spaces on multiple subdomains, perform mesh manipulation, and solve Laplace–Beltrami PDEs with iso-parametric elements without major effort. Only recently have all these items (simultaneously) become features in other publicly available FE toolboxes.

FELICITY has now expanded to include other features, which make it a viable tool for a variety of problems. It can be used to implement other methods to simulate free boundary problems, such as phase-field methods, as well as many "standard" problems, such as Poisson's equation, the Navier–Stokes equations, elasticity problems, etc. Moreover, FELICITY is developed in MATLAB, so it inherits the convenience of the MATLAB interface. However, it is not pure MATLAB; C++ code is used to maintain performance (see subsection 1.2).

Currently, the main motivation behind FELICITY is to provide a flexible high-level interface, a good level of modularity, and some low-level access to the underlying code to facilitate research of FE methods and development of simulation software for novel mathematical models. Most research-level problems require some kind of "non-standard" modification of preexisting methods. The main philosophy in FELICITY is to not hide *all* the details of implementing an FEM so that modifications can remain moderately easy. Some examples are the following:

- Nontrivial mesh modification. Performing topological changes of meshes in front-tracking is usually difficult with "black-box" FE packages because the mesh data structure is usually buried in the package.
- Coupled models. A typical example is when different types of FE spaces on different subdomains meet at an interface. If the coupling is nonlinear, then special care is needed to develop solvers/preconditioners. FELICITY provides direct access to the "blocks" of a multiphysics problem.
- New algorithms. For example, implementing multigrid methods for new types of problems can be difficult in black box packages. It is often assumed that one will use an "off-the-shelf" multigrid solver that was developed for a standard problem (which may be inappropriate). In contrast, a multigrid solver for Cahn–Hillard [19] was recently developed in FELICITY.

1.2. Design decisions. In creating FELICITY, the main design decision was to use a high-level scripting language as much as possible. For instance, managing degrees-of-freedom (DoFs) for an FEM often involves indexing tricks, which can be efficiently implemented through vectorization with, say, MATLAB or Python. More-

over, a high-level language allows easy access to other external packages. Other FE packages that do this are `milamin` (MATLAB), `SfePy` (Python), and `FEniCS` (Python). `freeFEM++` has its own custom scripting language, but it is more restrictive than the other options listed here.

The alternative to a high-level language is a standard language, such as C++. This allows more control and freedom of developing a code but is more labor intensive; this is the approach of `deal.II`. Moreover, the syntax of defining a problem is less "pretty" in a standard language than in a high-level one. On the other hand, a high-level language is very readable, easy to learn/use, and can still be very flexible in many situations. Unfortunately, high-level languages may have adverse inefficiencies in certain contexts (e.g., `for` loops in MATLAB).

Thus, the next design decision was to isolate certain computationally intensive tasks of an FEM and implement them in C++ code that is interfaced to MATLAB through the `mex` interface. Some of these tasks are well defined; one can straightforwardly implement a stand-alone C++ code to perform the task and then wrap it with the `mex` interface. Some examples are search trees (FELICITY includes interfaces to quadtrees and octrees), as well as mesh generation (which FELICITY also provides via the method in [49]).

However, other tasks are not so well defined. There is an immense variability possible in developing an FEM. In this case, FELICITY uses automatic C++ code generation to write special purpose C++ code that implements a very specific task in a given FEM, e.g., allocating DoFs, assembling matrices that represent discrete bilinear forms, interpolating FE functions and geometric data, and finding closest points in curved meshes. Automatic code generation is accomplished through a DSL written in a MATLAB function that uses special MATLAB classes included in FELICITY. Code generation provides a way to mediate the demands of a general software package with the flexibility of a special purpose code. In this regard, FELICITY takes inspiration from the `FEniCS` project [37, 39], though the details of the design and implementation are different. Indeed, `FEniCS` only does code generation for assembling FE forms. FELICITY demonstrates that code generation can be used to mitigate the implementation of other FE tasks (e.g., interpolation and searching curved surface meshes for closest points).

The next design decision was about manipulating (simplex) meshes. FELICITY builds on the MATLAB `triangulation` class with its own special purpose mesh classes, which provide easy access to mesh data, such as connectivity, point coordinates, neighbor information, etc., for simplex meshes in one, two, and three dimensions. In principle, other types of meshes can be implemented in FELICITY, but this has not been done yet.

These mesh classes allow users to directly manipulate meshes during FE simulations. Examples of this are in moving mesh problems, such as moving interface problems, shape optimization, and dynamic meshes that undergo topological changes [43]. Because of the high-level interface in FELICITY, users can move FE simulation data from one mesh to another without having to dive into low-level code. Other FE packages have this capability as well but, to the best of our knowledge, it is not clear whether they are comparably easy.

One particular design decision (or feature) in FELICITY, which is significantly different from most (if not all) other packages, is that local FE spaces are stored/represented in a "flat" m-file, i.e., the m-file stores the nodal basis functions, ordering of DoFs on the reference element, etc. The motivation here comes from [35, Chap. 5, Metaprogramming], where the rubric is "put abstractions in code, details in meta-

data," meaning a flat (i.e., simple to understand) file. In comparison, `FEniCS` completely hides this information. Our opinion is that the specific definition of the local FE space should not be hidden from the user, especially given that it is a critical part of the mathematical formulation of the FE discretization. Software designers cannot foresee all possible use cases, so as much information as possible should be open to the user.

On a related note, FELICITY implements different classes of elements in a modular fashion. For example, the automatic implementation of Lagrange (H^1) elements is based on an internal `H1_Trans` class that handles transformations of basis functions to physical elements. $H(\mathrm{div})$ elements, such as Raviart–Thomas, have an `Hdiv_Trans` class for transforming basis functions. There is also an `Hcurl_Trans` class for $H(\mathrm{curl})$ elements. These classes are generic in the sense that they know nothing about the nodal basis function definitions; they only implement transformation code. Hence, adding a new element that falls under one of these classes is essentially trivial; the user need only specify the flat m-file mentioned earlier. Unfortunately, implementing a new class of transformations requires knowledge of some of the inner workings of FELICITY.

In conclusion, FELICITY can be thought of as a library where the user learns the package, brings various pieces together (where some of the pieces are more automated than others), and creates a stand-alone program to simulate some desired phenomena. The user never has to interact with the C++ code directly; however, the generated code is readable and can (in principle) be modified if necessary. Furthermore, the user is free to develop some parts of the code themselves, while using a subset of the functionality of FELICITY.

1.3. What you should know to use FELICITY. The user is expected to be familiar with weak and variational formulations of PDEs and have some experience in the FEM, e.g., some knowledge of at least one of the references [18, 20, 21, 23, 34]. In addition, the user should know MATLAB at the level of [6].

1.4. Outline. The remainder of the paper describes the FELICITY framework, its capabilities, its implementation, and examples of its use. Section 2 gives an overview and highlights the main functionality of the FE toolbox. In section 3, we explain how the DSL and code generation aspect of FELICITY are implemented. Section 4 presents numerical solutions for two example PDE problems implemented in FELICITY; details of the implementation are given in supplementary section SM1. We conclude in section 5 by highlighting the successes of FELICITY and discussing areas of future improvement.

2. Overview. We describe how FELICITY is used to implement parts of an FE code in an object-oriented way and how other parts are automated by code generation.

2.1. Building blocks of an FEM. In order to develop an FE code to solve a particular PDE problem, the following steps are often taken:

1. Replace the continuous domain with a discrete domain (i.e., mesh generation).
2. Define FE function spaces on the discrete domain.
3. Replace the continuous (infinite dimensional) weak formulation of the PDE by a discrete (finite dimensional) version.
4. Compute discrete matrices that represent bilinear and linear forms in the discrete weak formulation.
5. Form the matrix system that represents the discrete weak formulation, including enforcement of boundary conditions.

6. Apply a solver to the linear system to find the discrete solution.
7. Postprocess the solution; this may require interpolation of discrete FE functions.

This sequence of steps may change, such as when solving a nonlinear problem with Newton's method. Furthermore, various data structures are required in implementing the above sequence of steps, which are outlined in the following subsections.

Note: in order to run the code of this paper in MATLAB, the user needs to have FELICITY installed; see the manual [52] or the wiki [53] for more details.

2.1.1. Meshes. The discrete domain is represented by a mesh or triangulation data. Simplicial meshes are implemented in FELICITY by taking advantage of the built-in MATLAB class `triangulation`. Indeed, FELICITY provides three mesh classes that (essentially) are subclasses of `triangulation`:

- `MeshInterval` for polygonal curve meshes.
- `MeshTriangle` for triangulated surface meshes.
- `MeshTetrahedron` for triangulated volume meshes consisting of tetrahedra.

To create a mesh object with FELICITY, the user must provide the mesh connectivity and the mesh point coordinates as standard MATLAB matrices. The general syntax is

```
Mesh = Mesh<TYPE>(Connect,Points,Name);
```

where `<TYPE>` is either `Interval`, `Triangle`, or `Tetrahedron`. `Connect` is an $M \times (t+1)$ matrix where M is the number of mesh elements and t is the topological dimension, `Points` is an $N \times d$ matrix where N is the number of points and d is the geometric dimension, and `Name` is a string representing the name of the domain. For example, suppose we have a mesh $\mathcal{T}$ of the unit square Ω in $\mathbb{R}^2$ consisting of two triangles defined as $\mathcal{T} = \{T_1, T_2\}$, where $T_1 = \{v_1, v_2, v_3\}$, $T_2 = \{v_1, v_3, v_4\}$, and the points (vertices) have the following coordinates: $v_1 = (0,0)$, $v_2 = (1,0)$, $v_3 = (1,1)$, $v_4 = (0,1)$. With the FELICITY toolbox, we create this mesh using the following commands at the MATLAB prompt:

```
Tri  = [1 2 3; 1 3 4];
Pts  = [0, 0; 1, 0; 1, 1; 0, 1];
Mesh = MeshTriangle(Tri,Pts,'Omega').
```

Geometrically, T_1 and T_2 are "straight" triangles, meaning that the sides are straight line segments (in $\mathbb{R}^2$) joining the vertices. Note that FELICITY only allows for *conforming* meshes (defined in subsection 2.1.5), e.g., no "hanging vertices."

Remark 2.1 (mesh generation). FELICITY provides several functions for generating meshes of simple domains:
`triangle_mesh_of_disk`, `triangle_mesh_of_sphere`,
`bcc_triangle_mesh` (of a square), `equilateral_triangle_mesh`,
`regular_tetrahedral_mesh` (of a cube), `bcc_tetrahedral_mesh`, which are located in the `Misc_Routines` subdirectory of FELICITY. In addition, FELICITY has an interface to a general three-dimensional mesh generator (TIGER) based on [49].

A nice feature of FELICITY is that subdomains can be stored within the mesh object. For example, the boundary of the unit square mesh, $\Gamma := \partial\Omega$, consists of four directed edges denoted as $E_1 = \{v_1, v_2\}$, $E_2 = \{v_2, v_3\}$, $E_3 = \{v_3, v_4\}$, $E_4 = \{v_4, v_1\}$. These are stored in the mesh object with the following commands:

```
Edges = [1 2; 2 3; 3 4; 4 1];
% alternatively: Edges = Mesh.freeBoundary();
Mesh = Mesh.Append_Subdomain('1D','Gamma',Edges);
```

where the first argument specifies the topological dimension of the subdomain and the second argument gives it a name, followed by the edge data. Note: the boundary edges can be extracted using the class method `freeBoundary`. Furthermore, users can store embedded subdomains, e.g., the diagonal of the unit square mesh:

```
DE = [1 3];
Mesh = Mesh.Append_Subdomain('1D','Diagonal',DE).
```

Remark 2.2. It is implicit in the mesh data structure above that each triangle (element) T in the mesh is linear or "straight," instead of curved. This implies the existence of an affine map $\mathbf{F}_T : T_{\text{ref}} \to T$, where T_{ref} is a standard reference domain [18, 23, 34] (e.g., the unit triangle in $\mathbb{R}^2$ with ordered vertices $(0,0)$, $(1,0)$, $(0,1)$). In subsection 2.1.5, we discuss how to handle meshes containing *curved* elements.

2.1.2. The reference finite element. FE spaces are concretely defined by first specifying a reference finite element in the sense of Ciarlet [23, 20], i.e., the reference domain T_{ref}, the space of shape functions $\mathcal{P}$, and the nodal variables $\mathcal{N}$. This is stored by FELICITY in several flat m-file scripts that define various elements, such as Lagrange elements, Raviart–Thomas elements, and Nedelec elements (of the first kind), up to degree 3. For example, the element definition for Lagrange elements of degree 3 on the standard reference tetrahedron is specified in the file `lagrange_deg3_dim3.m`. Executing the m-file delivers a MATLAB structure (i.e., `struct`) with several fields that specify the reference finite element $(T_{\text{ref}}, \mathcal{P}, \mathcal{N})$; e.g., the reference domain type is stored as a string, along with other information like the topological dimension.

Remark 2.3. FELICITY stores the reference element for each dimension and degree in separate files. The main reason is readability and openness to the user. This simple format makes it easy to include new elements, as long as the element falls under one of the transformation classes that FELICITY already has, such as H^1 (e.g., Lagrange), $H(\text{div})$ (e.g., Raviart–Thomas), and $H(\text{curl})$ (e.g., Nedelec).

The space of shape functions $\mathcal{P}$ is a linear space, so it is spanned by a minimal set of basis functions. Thus, FELICITY *identifies* $\mathcal{P}$ with the nodal basis set, where the Kronecker delta property is satisfied, i.e., $N_i(\phi_j) = \delta_{ij}$ for all $1 \le i, j \le \dim(\mathcal{P})$, where $\dim(\mathcal{P})$ is the dimension of $\mathcal{P}$ and $\mathcal{N}$ and $N_i \in \mathcal{N}$, $\phi_j \in \mathcal{P}$. The nodal variables $\mathcal{N}$ are stored in the m-file with enough information to allow the user to check the Kronecker delta property (if they wish) with the given basis functions.

The subscript j in the basis function ϕ_j is called the *local* degree-of-freedom (DoF) index. Each basis function is essentially stored as a string. Note: the local basis functions are connected to the global basis functions of the FE space by a DoF map, i.e., `DoFmap` (see subsection 2.1.4).

Each DoF (equivalently, nodal variable) is associated with a point in T_{ref}, with specific reference domain coordinates (stored as barycentric coordinates in the flat m-file). For Lagrange elements, the points correspond to the point evaluation of the associated nodal variable. For other element types, such as $H(\text{div})$, the nodal variable involves computing an inner product (integral) of the basis function against a specific (dual) Lagrange basis function [16]. Thus, the point coordinate corresponds to the dual Lagrange function. Note that the choice of the dual basis functions fully specifies the nodal variables.

The user can create a MATLAB class object for performing queries on the reference finite element, e.g., by executing the following at the MATLAB prompt:

```
RFE = ReferenceFiniteElement(lagrange_deg3_dim3()).
```

2.1.3. FE spaces. The global FE space, defined over a domain (say Ω), is obtained by gluing together local element spaces defined on each element T in $\mathcal{T}$ [18, 23, 34]. Thus, let $V(T)$ be an FE space defined on $T \in \mathcal{T}$ by $V(T) = \{v \in L^1(T) : v = \eta \circ \mathbf{F}_T^{-1}, \text{ where } \eta \in \mathcal{P}(T_{\text{ref}})\}$, where $\mathcal{P}(T_{\text{ref}})$ are the nodal basis functions defined over T_{ref}, and $\mathbf{F}_T : T_{\text{ref}} \to T$ is the affine map from the reference element to the "physical" element in $\mathcal{T}$. Note that, at this point, we are only considering meshes containing *straight* elements (see Remark 2.2); subsection 2.1.5 discusses FE spaces defined over meshes containing *curved* elements.

The global FE space then is

$$V(\Omega) = \{v \in C^0(\Omega) : v|_T = \eta \circ \mathbf{F}_T^{-1} \text{ for some } T \in \mathcal{T}, \eta \in \mathcal{P}(T_{\text{ref}})\}, \tag{1}$$

where, in this case, we have assumed a C^0 continuity requirement over the entire domain Ω. The linear vector space $V(\Omega)$ is spanned by a finite set of basis functions, i.e., $V(\Omega) = \{\phi_1, \phi_2, \ldots, \phi_J\}$. In this case, the index k in ϕ_k is the *global* DoF index of the unique basis function ϕ_k in (1).

2.1.4. Degree-of-freedom map. FELICITY must know how basis functions on a mesh element T are "connected" to basis functions on a neighboring mesh element T', in order to guarantee certain continuity requirements in the *global* FE space, e.g., C^0 for Lagrange elements [18, 23, 34]. This is done by defining a degree-of-freedom map (`DoFmap`) for each element T of $\mathcal{T}$:

$$\texttt{DoFmap} : \mathcal{T} \times \{1, 2, \ldots, \dim(\mathcal{P})\} \to \{1, 2, \ldots, J\}; \tag{2}$$

i.e., $k = \texttt{DoFmap}(T, j)$ is the global index of the jth basis function on element T. In FELICITY, this is implemented as a MATLAB matrix of size $M \times \dim(\mathcal{P})$, where M is the number of mesh elements.

The `DoFmap` data may be defined by either the user or FELICITY. For example, the `DoFmap` for a continuous piecewise linear FE space defined on the mesh in subsection 2.1.1 can simply be taken to be *identical* to `Tri`; this is because the DoFs correspond to the vertices (points) of the mesh. In the case of a piecewise quadratic space, the DoFs correspond to the mesh vertices and the midpoints of mesh edges. It is sometimes desirable to preserve this correspondence of the DoFs with mesh vertices and edges. For instance, it makes accessing the solution at the vertices trivial, which is convenient for plotting purposes; the alternative is to interpolate the solution, which requires some additional (minor) coding. In addition, it may be convenient to group the edge DoFs together for other processing tasks. For more complicated spaces, more care must be taken in specifying `DoFmap`. FELICITY provides a way to automate this (see subsection 2.2.3).

FELICITY provides the class `FiniteElementSpace` for querying an FE space. One can create an object of this class with the following commands:

```
DoFmap = <user defines or FELICITY allocates>;
V = FiniteElementSpace('V',RFE,Mesh,'Omega',k_Tuple);
V = V.Set_DoFmap(Mesh,DoFmap);
```

where the first argument names the space, the second argument is the reference finite element defined earlier, and the third argument is the mesh object defined earlier. The fourth argument is the name of the domain (or subdomain) on which the space is defined.

The last argument, `k_Tuple`, is the number of cartesian products to take of the base FE space, i.e., the FE space will consist of a k-tuple of the base FE space,

where k is given by `k_Tuple`. So, if `k_Tuple` $= 1$, then the `FiniteElementSpace` object corresponds to (1). If `k_Tuple` $= k > 1$, then the `FiniteElementSpace` object corresponds to

$$\begin{aligned}\mathbf{V}(\Omega) &= \{\mathbf{v} \in [C^0(\Omega)]^k : \mathbf{v} \cdot \mathbf{e}_j|_T = \eta \circ \mathbf{F}_T^{-1}, 1 \le j \le k, T \in \mathcal{T}, \eta \in \mathcal{P}(T_{\text{ref}})\} \\ &\equiv \underbrace{V(\Omega) \times \cdots \times V(\Omega)}_{k \text{ terms}}. \end{aligned} \tag{3}$$

This provides a convenient way to define "vector-valued" Lagrange FE spaces. Note that one can also take cartesian products of a base FE space containing intrinsically vector-valued basis functions, such as an $H(\text{div}, \Omega)$ FE space [16, 31].

Remark 2.4 (FE coefficient functions). A standard approach, which we adopt here, is to represent an FE function v in $V(\Omega)$ by a vector of coefficients `v`, which is an $N \times 1$ matrix (column vector), where N is the total number of DoFs in the FE space. Thus, `v(i)` is the value of the nodal variable for DoF index `i`.

For a cartesian product (k-tuple) space, e.g., (3), `v` consists of k column vectors, i.e., `v` is an $N \times k$ matrix, where column j gives the nodal values of $\mathbf{v} \cdot \mathbf{e}_j$. So N is the number of DoFs for *a single* component. It is often convenient to concatenate the column vectors using `v(:)`, which gives a $kN \times 1$ column vector.

Remark 2.5 (mixed element spaces). For problems involving mixed elements, such as in solving the Stokes equations, the user must define each FE space separately and manually build the block matrix system. A demo describing this for the Stokes equations is provided in FELICITY. In the future, FELICITY will provide a `MixedElementSpace` class (analogous to `FiniteElementSpace`) to handle this.

2.1.5. Higher order meshes. Piecewise affine simplicial meshes consist of line segments in one dimension, flat triangles with straight sides in two dimensions, tetrahedra with straight edges and flat faces in three dimensions, etc. The mesh example in subsection 2.1.1 consists of straight triangles. We now review meshes consisting of *curved* elements and how to define them in FELICITY.

To this end, we use the notation $\widehat{\mathcal{T}}$ to indicate a mesh consisting of piecewise affine elements (not curved), where $\widehat{T} \in \widehat{\mathcal{T}}$ indicates a specific affine element. Moreover, we write $\widehat{\Omega} = \text{int}\big(\cup_{\widehat{T} \in \widehat{\mathcal{T}}} \overline{\widehat{T}}\big)$, i.e., $\widehat{\Omega}$ is a piecewise linear approximation of the true, curved domain Ω_s. Note that $\widehat{T}$ and $\widehat{\Omega}$ are open sets and $\overline{\widehat{T}}$ is the closure of $\widehat{T}$. An FE space, defined on the piecewise linear approximate domain $\widehat{\Omega}$, is given by

$$V(\widehat{\Omega}) = \{v \in C^0(\widehat{\Omega}) : v|_{\widehat{T}} = \eta \circ \mathbf{F}_{\widehat{T}}^{-1} \text{ for some } \widehat{T} \in \widehat{\mathcal{T}}, \eta \in \mathcal{P}(T_{\text{ref}})\}. \tag{4}$$

Let T_{ref} have topological dimension t. Next, assume Ω_s is smooth, embedded in $\mathbb{R}^d$, and is parameterized by a set of nonoverlapping local charts $\{\mathbf{\Phi}_{T_s}\}$ [50], where $\mathbf{\Phi}_{T_s} : T_{\text{ref}} \to T_s \subset \Omega_s \subset \mathbb{R}^d$ is a nonlinear, bijective map such that $T_s := \mathbf{\Phi}_{T_s}(T_{\text{ref}})$ (where $d \ge t$). When $d = t$, we demand that $\mathbf{\Phi}_{T_s}$ have a positive Jacobian determinant [13]. This gives a partition of the true domain Ω_s into a mesh of nonlinear elements $\mathcal{T}_s = \{T_s\}$. Then a version of (4), defined on the true geometry, is given by

$$V(\Omega_s) = \{v \in C^0(\Omega_s) : v|_{T_s} = \eta \circ \mathbf{\Phi}_{T_s}^{-1} \text{ for some } T_s \in \mathcal{T}_s, \eta \in \mathcal{P}(T_{\text{ref}})\}. \tag{5}$$

However, this is not always practical, so we take an FE (Lagrange) approach to the local charts (e.g., iso-parametric [13]). Define the FE space

$$\mathbf{G}_k(\widehat{\Omega}) = \{\mathbf{v} \in [C^0(\widehat{\Omega})]^d : \mathbf{v}|_{\widehat{T}} = \boldsymbol{\eta} \circ \mathbf{F}_{\widehat{T}}^{-1} \text{ for } \widehat{T} \in \widehat{\mathcal{T}}, \boldsymbol{\eta} \in [\mathcal{L}_k(T_{\text{ref}})]^d\}, \tag{6}$$

where $\mathcal{L}_k(T_{\text{ref}})$, $k \geq 1$, is the space of (scalar) Lagrange polynomials of degree k on T_{ref}. For each affine approximation $\widehat{T}$ of $T_s \in \mathcal{T}_s$, we build a higher order version T:

$$T = \boldsymbol{\Phi}_T(T_{\text{ref}}), \quad \boldsymbol{\Phi}_T := \mathbf{g}_k \circ \mathbf{F}_{\widehat{T}} \text{ for some fixed } \mathbf{g}_k \in \mathbf{G}_k(\widehat{\Omega}). \tag{7}$$

Note that $\boldsymbol{\Phi}_T \in [\mathcal{L}_k(T_{\text{ref}})]^d$. We then obtain $\mathcal{T} = \{T\}$ and define a higher order approximation $\Omega := \text{int}(\cup_{T\in\mathcal{T}}\overline{T})$ of Ω_s. We assume the collection of maps $\{\boldsymbol{\Phi}_T\}_{T\in\mathcal{T}}$ are nonoverlapping and bijective, have positive Jacobian determinant when $d = t$, produce a continuous domain embedded in $\mathbb{R}^d$, and are *conforming* in the sense that

$$\begin{aligned} &\boldsymbol{\Phi}_{T_1}(T_{\text{ref}}) \cap \boldsymbol{\Phi}_{T_2}(T_{\text{ref}}) = \emptyset \text{ for all pairs } T_1, T_2 \in \mathcal{T}, \text{ such that } T_1 \neq T_2, \\ &\boldsymbol{\Phi}_{T_1}(\overline{T}_{\text{ref}}) \cap \boldsymbol{\Phi}_{T_2}(\overline{T}_{\text{ref}}) = \emptyset \text{ or } \boldsymbol{\Phi}_{T_1}(\omega), \text{ such that } T_1 \neq T_2 \text{ and } \omega \subset \overline{T}_{\text{ref}}, \end{aligned} \tag{8}$$

where T_{ref} is considered an open set, $\overline{T}_{\text{ref}}$ is the closure of T_{ref}, and ω is a topological entity of $\overline{T}_{\text{ref}}$ of dimension less than t, e.g., a vertex, edge, or facet. Note that FELICITY only allows for conforming meshes (straight or curved).

Using (6) and (7), we obtain a version of (5) defined over the higher order domain approximation Ω:

$$V(\Omega) = \{v \in C^0(\Omega) : v|_T = \eta \circ \boldsymbol{\Phi}_T^{-1} \text{ for some } T \in \mathcal{T}, \eta \in \mathcal{P}(T_{\text{ref}})\}. \tag{9}$$

The spaces $V(\widehat{\Omega})$, $V(\Omega)$, and $V(\Omega_s)$ differ only in that different local maps are used to represent the local element geometry; topologically, all three spaces are equivalent. When dealing with higher order domains, FELICITY provides a special class `GeoElementSpace`, which is a subclass of `FiniteElementSpace`. Creating an object of this class, which represents $\mathbf{G}_k(\widehat{\Omega})$, is accomplished by

```
Gk = GeoElementSpace('Gk',Gk_RefElem,Mesh);
```

where the first argument names the space, the second argument is the (Lagrange) reference finite element $\mathcal{L}_k(T_{\text{ref}})$ defined earlier, and the third argument is a piecewise linear mesh object. A function in `Gk` is a d-tuple valued (Lagrange) FE function; i.e., the number of vector components is equal to d. See supplementary section SM1.1 for an example.

2.1.6. Evaluating boundary and initial conditions. One can define where boundary conditions are enforced by using the `FiniteElementSpace` object. For example, suppose we want to fix all DoFs that are on the boundary of our unit square mesh. This is accomplished with the following command:

```
V = V.Append_Fixed_Subdomain(Mesh,'Gamma').
```

The user can then access the DoFs that are attached to the subdomain `Gamma` with

```
Fixed_DoFs = V.Get_Fixed_DoFs(Mesh);
```

where `Fixed_DoFs` is an array of indices. Note that one must usually supply the mesh when retrieving information about the FE space.

Furthermore, users can access the coordinates of the DoFs through the command:

```
XC = V.Get_DoF_Coord(Mesh);
```

where `XC` is an $N \times d$ matrix containing the coordinates of all DoFs in the FE space. Note: N is the number of DoFs, and d is the geometric dimension.

Next, we show how to set boundary conditions for an FE function represented by a coefficient vector. Continuing with our example, suppose we want to enforce $u = g$ on Γ, where u is a discrete FE function. Moreover, assume that we have an m-file `g.m` that will evaluate the function g given the coordinates. If `u` denotes the vector of coefficients representing u, then this can be accomplished with the following commands:

```
g_values = g(XC); % evaluates g at *all* DoF nodes
u(Fixed_DoFs) = g_values(Fixed_DoFs);
```

where we have taken advantage of the indexing abilities of MATLAB. Initial conditions can be set in a similar way, e.g.,

```
u_init = init_conditions(XC);
```

where `init_conditions.m` is a user supplied m-file to evaluate the desired initial condition function.

2.1.7. Other data. Computing FE matrices requires a loop over mesh elements, computing the local element map, applying various transformations to basis functions, etc. FELICITY takes care of this (see subsection 2.2.2). The matrices are stored in the standard sparse matrix format of MATLAB.

Any other data, such as parameters, can be stored using the usual MATLAB arrays and structs.

2.2. Automating FE implementation. FELICITY automates parts of building an FE code. A central goal in the design of FELICITY is to allow for some flexibility in "getting between" the modules to implement algorithmic aspects that do not fit a standard pattern. Note that FELICITY requires the MATLAB symbolic toolbox to take advantage of the automation aspects.

2.2.1. Defining forms/matrices. Suppose $\Omega \subset \mathbb{R}^2$, define the FE space $V = \{v \in C^0(\Omega) : v|_T \text{ is affine}\}$, and consider the following bilinear and linear forms:

$$m(u,v) = \int_\Omega uv, \quad a(u,v) = \int_\Omega \nabla u \cdot \nabla v, \quad l(v) = \int_\Omega v \ \text{ for all } u, v \in V. \tag{10}$$

Computing the matrix representation of bilinear and linear forms is a common task in FE codes, usually referred to as matrix assembly. FELICITY can automate the implementation of this task by first defining an m-file that defines the forms *abstractly*. An example m-file is as follows:

```
function MATS = MatAssem()

% define domain (2-D)
Omega = Domain('triangle');

% define finite element (FE) space
V = Element(Omega,lagrange_deg1_dim2);

% define functions on FE space
v = Test(V);
u = Trial(V);

% define FE matrices
M = Bilinear(V,V);
I1 = Integral(Omega, v.val * u.val);
M = M.Add_Integral(I1);

A = Bilinear(V,V);
I2 = Integral(Omega, v.grad' * u.grad);
A = A + I2;
```

```
L = Linear(V);
L = L + Integral(Omega, v.val);

% set the minimum number of quadrature points in quadrature rule
Quad_Order = 10;
% define geometry representation - Domain, (default is linear)
G_Space = GeoElement(Omega);
% define a set of matrices
MATS = Matrices(Quad_Order,G_Space);

% collect all of the matrices together
MATS = MATS.Append_Matrix(M);
MATS = MATS.Append_Matrix(A);
MATS = MATS.Append_Matrix(L);

end
```

This m-file uses a kind of DSL similar in spirit to the unified-form-language (UFL) of FEniCS [8]. First, the domain of the problem is defined (abstractly); i.e., it is a mesh of triangles with no other information given. The next command defines the FE space `V`. Then, `Test` and `Trial` functions (in the FE sense) are defined on the space `V`.

Next, we define the FE matrices by first specifying whether it is a `Bilinear` or `Linear` form on `V`. Then, the form is fully defined by specifying the integral computation to be performed involving the `Test` and `Trial` functions. We show three different ways to "add" an integral to a form. Note that the domain of the integral is specified. In particular, multiple integrals, *on different subdomains of varying co-dimension*, can contribute to the form.

In the remaining part of the m-file, we specify the minimum number of quadrature points to use, how the primary domain is represented (e.g., is the domain comprised of piecewise linear triangles or piecewise quadratic triangles?); in this case, the default is used (piecewise linear triangles). The `MATS` object serves to output all the information in the m-file. See subsection 3.1 and the FELICITY manual [52] for more details.

Remark 2.6. The abstract m-file may have input arguments, e.g.,
`function MATS = MatAssem(dim,deg).`
This allows for *parameterizing* the form definitions, which can be useful in setting up a general method to solve many kinds of problems.

2.2.2. Matrix assembly. FELICITY can take the m-file above and process it into a stand-alone matrix assembly C++ code, which is then compiled into a MATLAB mex file that is callable from the MATLAB prompt. Assuming the m-file is named `MatAssem.m`, the following command generates the mex file:

```
[status, Path_To_Mex] =...
    Convert_Form_Definition_to_MEX(@MatAssem,{},'mex_MatAssem').
```

The first argument is a function handle to `MatAssem` (this requires `MatAssem.m` to be in a directory that is in the MATLAB path). The second argument is a cell array of possible inputs to pass to `MatAssem` (see Remark 2.6). The last argument specifies the filename of the mex file, which is placed in the same directory as `MatAssem.m`. More options are available for the conversion script; see [53, 52] for more details.

Using the matrix assembly code requires the user to supply various pieces of information, such as the mesh, FE space (`DoFmap`), etc. Supplementary section SM1

shows examples of how to call the mex file.

2.2.3. Allocating DoFs. As mentioned in subsection 2.1.4, a `DoFmap` is a data structure used to store the "connectivity" of the basis functions in an FE space. It is certainly possible for users to define the `DoFmap` by writing their own special purpose code, but this can be cumbersome for elements more complicated than piecewise linear. Fortunately, FELICITY provides a way to do this automatically.

The following commands will create a mex file to generate a `DoFmap` for `Elem`:

```
% get struct of reference element data
Elem = lagrange_deg2_dim2(); % piecewise quadratic on triangles
Main_Dir = <user specified directory>;
[status, Path_To_Mex] =...
         Create_DoF_Allocator(Elem,'mexDoF_P2',Main_Dir).
```

Note: `lagrange_deg2_dim2.m` is contained in the directory `/FELICITY/Elem_Defn`. Running this will create a mex file (in `Main_Dir`) named `mexDoF_P2` (with a file extension dependent on your system type).

Running the mex file is done with the following command:

```
P2_DoFmap = mexDoF_P2(uint32(Tri));
```

where `Tri` is the connectivity matrix of the underlying triangular mesh. The output `P2_DoFmap` is of size $M \times \dim(\mathcal{P})$, where M is the number of mesh elements in `Tri` and $\mathcal{P}$ is the space of shape functions in the reference finite element. Examples of using the mex DoF allocator are given in supplementary section SM1.

2.2.4. Interpolation. Suppose we want to interpolate an expression involving a function p in V at a set of arbitrary points in Ω. For example, consider the expression $\nabla p \cdot \mathbf{x}$ defined on Ω, where $\mathbf{x}$ is the coordinate function on Ω. One can use a similar type of abstract m-file to define the interpolation and then use FELICITY to automatically generate a mex file to evaluate FE data at the interpolation points. A sample m-file for this expression follows:

```
function INTERP = Interpolate_FE_Data()

% define domain
Omega = Domain('triangle');

% define FE spaces
V = Element(Omega,lagrange_deg1_dim2,1);
% define functions on FE spaces
p = Coef(V);

% define a geometric function on 'Omega' domain
gf = GeoFunc(Omega);
% define expressions to interpolate
Interp_Expr = Interpolate(Omega, p.grad' * gf.X);

% define geometry representation - Domain, reference element
G1 = GeoElement(Omega);
% define a set of interpolation expressions to perform
INTERP = Interpolations(G1);

% collect all of the interpolations together
INTERP = INTERP.Append_Interpolation(Interp_Expr);
```

```
end
```

Generating the mex file is similar to the matrix assembly case. Assuming the m-file is named `Interpolate_FE_Data.m`, the mex file is generated by

```
[status, Path_To_Mex] =...
    Convert_Interp_Definition_to_MEX(...
            @Interpolate_FE_Data,{},'mex_Interpolate_FE_Data').
```

Using the mex file requires the user to supply various pieces of information, such as the mesh, FE space (`DoFmap`), etc. Note that the mex file requires the interpolation points to be given in a certain format.

Remark 2.7 (point searching). In order to interpolate an FE function, a user needs to search the mesh to locate which mesh element (e.g., triangle) contains a given point [7]. FELICITY offers utilities to do this as well. Moreover, curved meshes, where each mesh element is described by a Lagrange element of degree > 1, may be searched. Again, FELICITY utilizes code generation to generate a special purpose code for point searching. We omit the details for brevity (see [52] for more information).

2.3. Other software aspects. Solver support is somewhat limited to what MATLAB offers (such as backslash, generic iterative solvers, etc.). However, several other packages offer interfaces to MATLAB, e.g., MUMPS [9], Pardiso [45, 44], SuperLU [36], and AGMG [42, 41]. These can all be used in conjunction with FELICITY.

Other software pieces included in FELICITY are as follows: a tetrahedral mesh generator [49], performing closest point searches of higher order meshes (e.g., for a surface mesh embedded in three dimensions), available search-trees (bitree, quadtree, octree) implemented directly in C++ (which can enable fast nearest neighbor searches and speed up point searching in a mesh), simulation management (e.g., saving and loading data, visualization), and extensive help/documentation.

3. Under the hood. We describe some of the details of how the automation aspects of FELICITY were implemented.

3.1. Generating matrix assembly code. Automatically generating a matrix assembly code starts with the abstract form file; running the form file (as an m-function) creates an object called `MATS`, which collects all the relevant meta-data (subsection 3.1.1). The next step processes `MATS` further to put the meta-data into a more convenient format as well as optimize various aspects of the matrix assembly code (subsection 3.1.2). Then, the C++ code is automatically generated from the processed meta-data; this consists of "pasting" together snippets of standard code with customized portions that take advantage of the specificity of the form file (subsection 3.1.3). Finally, the C++ code is compiled with the MATLAB `mex` command to create an executable that is callable from the MATLAB prompt.

3.1.1. Define abstract forms. Subsection 2.2.1 showed how to abstractly define the forms in (10). The m-file `MatAssem.m` is actual MATLAB code that creates several simple *objects* which build on one another.

The first command in `MatAssem.m` uses a MATLAB class (provided by FELICITY) called `Domain`. So the command

```
Omega = Domain('triangle');
```

creates an object called `Omega` of type `Domain`, using the *constructor* of the class; the argument simply specifies the type of domain (in this case, the domain is a mesh of

triangles in $\mathbb{R}^2$). At this stage, no other information is given, such as the number of triangles, mesh geometry, etc.

The next command defines an FE space using the class `Element`; i.e., the command

```
V = Element(Omega,lagrange_deg1_dim1);
```

creates an object called `V` of type `Element` using the default constructor of the class. The first argument is a `Domain` object that (abstractly) indicates the domain on which the FE space is defined; the second argument is a struct output from running the command `lagrange_deg1_dim1`, which gives a complete specification of the local FE space. Hence, `V` is a continuous piecewise linear Lagrange FE space, defined over the mesh of triangles that represents `Omega`. Since no specific mesh is given, `V` is abstract; i.e., the specific form of the global basis functions is not known (because they depend on the mesh geometry, which is not known). Note that `V` stores a copy of `Omega` inside it.

Next, `v` is defined as an object from the class `Test` that abstractly represents a test function in `V`. It also keeps a copy of `V` inside, which is useful for error checking the form definition m-file. Similarly, we have `u` as an object of type `Trial` (i.e., a trial function in `V`).

Defining the FE forms follows a similar procedure. For example, `M` is an object of type `Bilinear` (i.e., a bilinear form) defined on the test space `V` and the trial space `V`. Note that `M` keeps an internal copy of the test and trial spaces, which is necessary for code generation and useful for error checking.

The next command

```
I1 = Integral(Omega, v.val * u.val);
```

is the most complex part of the m-file `MatAssem.m` in terms of its implementation; it specifies how to compute the value of `M` when given a test function and a trial function. The object `I1` is of type `Integral` created with the default constructor, which takes two arguments. The first argument is the (abstract) domain on which the integral is to be computed. The second is a MATLAB symbolic expression (from the Symbolic Computing Toolbox of MATLAB), which represents the integrand.

In other words, `v.val` and `u.val` output symbolic variables that represent point evaluation of the basis functions. In particular, `val` is a *method* of the `Test` and `Trial` classes (as well as `Coef`). The output of `u.val` is a MATLAB symbolic variable, with the name `u_v1_t1`, that captures the *signature* of the basis function *and* the specific quantity to evaluate. In general, for the `val` method, the format is

```
<function name>_v<vector component>_t<tuple index>
```

The "vector component" refers to the intrinsic vector component of the function, e.g., if the basis function is intrinsically vector valued, such as with Raviart–Thomas elements. The "tuple index" refers to a specific component in the cartesian product FE space (recall (3)). In the current example, the FE space has only one cartesian component, and the base FE space has only one component (i.e., scalar-valued). As another example, the output of `u.grad` is

```
u_v1_t1_grad1
u_v1_t1_grad2
```

i.e., it is a vector-valued symbolic variable whose components refer to the components of the gradient of `u`, so it represents the point evaluation of the gradient of the basis function `u`.

Hence, the syntax `v.val * u.val` results in the symbolic expression

```
v_v1_t1 * u_v1_t1
```

which completely encodes how to evaluate the bilinear form `M` given the test and trial

functions `v` and `u`. The expression could be quite complicated depending on the form, e.g., if nonlinear terms are present. MATLAB can convert a symbolic expression into optimized C code by using the command `ccode`; this is used by FELICITY during the C++ code generation segment.

By taking advantage of the MATLAB command `inputname`, as well as some text processing tricks (e.g., `regexpr`), the `Bilinear` and `Linear` forms (abstractly) know what the domain of integration is, what the integrand is (as a symbolic expression), *and* what functions appear in the integrand and the FE spaces they belong to.

The last main part of `MatAssem` is the command

```
G_Space = GeoElement(Omega);
```

which creates an object `G_Space` of class `GeoElement` that indicates how the geometry of `Omega` is represented. In this case, the default constructor is used, so the geometry is assumed to be piecewise linear (i.e., each triangle in `Omega` is flat with straight sides). Another, equivalent, way to create `G_Space` is

```
G_Space = GeoElement(Omega,lagrange_deg1_dim2);
```

where we explicitly define (in the second argument) the local FE space to use for parameterizing a triangle in the mesh. Higher order Lagrange elements may also be used for modeling curved elements. Note that only Lagrange elements are allowed.

After all forms are defined, the user creates a special catch-all object

```
MATS = Matrices(Quad_Order,G_Space);
```

that holds a copy of `G_Space` and the (minimum) number of quadrature points to use in numerically evaluating all integrals. The next several lines simply store copies of the (previously defined) forms inside the `MATS` object.

Hence, `Element`, `Test`, `Trial`, `Bilinear`, `Linear`, `Integral`, `GeoElement`, and `Matrices` are all MATLAB classes within the FELICITY toolbox. These classes are very lightweight; not much information is stored in each. The main purpose they serve is to collect all the "meta-data" necessary to completely determine a stand-alone C++ code that will assemble sparse matrices representing discrete versions of the forms on a given mesh. All the meta-data is stored in `MATS`, which is the only output from the `MatAssem.m` m-file.

Remark 3.1. The FEniCS project uses a custom parser to read a UFL code that defines FE forms, which has the advantage of not being limited in the kind of syntax that it allows. In comparison, FELICITY uses the MATLAB scripting language, execution engine, and lightweight classes to abstractly define FE forms. This has the advantage of doing immediate error checking of the syntax as well as the (relative) ease of creating data structures to contain the meta-data.

3.1.2. Process abstract form data. The next stage of generating code to build FE forms is an initial *compile* step. This starts by running the abstract m-file (e.g., `MatAssem.m`), as any MATLAB function, to build the `MATS` object, which is essentially a giant MATLAB `struct`.

Then, `MATS` is processed further; i.e., FELICITY makes multiple passes through `MATS` in order to correlate various items. For example, the `M` and `A` forms (in `MatAssem`) utilize the same `Test` and `Trial` FE spaces. Therefore, when generating the C++ code, the global basis functions can be computed once (on a given mesh element) no matter how many forms use those basis functions.

In addition, the integrands of the forms are processed to determine what actually needs to be computed for the basis functions, as well as any geometric information. For instance, it checks whether the gradient is needed, or whether the normal vector on a surface is needed, or whether the trace of a basis function on a subdomain is required,

etc. This is accomplished by processing the *signatures* (see subsection 3.1.1) that appear in the integrands. The main tools in making this work are text processing with `regexpr` and using MATLAB's symbolic toolbox. This processing gradually builds a list of various "quantities" that the C++ code must implement.

Furthermore, FELICITY checks for possible efficiencies that the C++ implementation can take advantage of, such as whether the form is symmetric or whether the mesh consists of piecewise linear elements (instead of curved triangles). It also checks whether the form is defined over a cartesian product FE space, i.e., the matrix corresponding to the discrete form may contain block submatrices, with some blocks identical. Other checks are made as well.

After the initial compile step, FELICITY (internally) produces several data structures containing detailed information that the C++ code generator will use.

3.1.3. Generate C++ code. Once FELICITY has processed the abstract form file, it takes this information and generates a stand-alone C++ code that assembles the specified matrices given certain information, such as the mesh data, FE space data (i.e., the `DoFmap`(s)), any FE coefficient data, etc.

First, it creates a specialized mesh class to read the mesh connectivity data, vertex coordinates, as well as compute local mesh element maps, and other quantities, e.g., the Jacobian of the map. Portions of the code that define this class never change from problem to problem. Other parts do depend on the problem, such as the specific FE space to use in representing the mesh geometry (as well as what quadrature points to evaluate the local element map).

In addition, it creates specialized classes to do the following:

- access subdomain data in the mesh,
- transform basis functions and FE coefficient functions,
- compute local FE matrices, and
- assemble global FE matrices (i.e., discrete forms) from local matrices.

In all cases, each class is a mix of standard (static) code, essentially stored as small snippets within various (internal) m-files, and custom generated code that takes advantage of the specific information of the problem. For example, the following quantities are fixed and known when the code is generated:

- number of quadrature points,
- number of basis functions,
- evaluation of basis functions at quadrature points,
- number and type of subdomains in the mesh,
- types of transformations to perform on basis function, and
- the block structure of FE matrices.

The generated code is tailored to the specific problem of interest by taking advantage of the above information. Specific classes are easier to implement, and the C++ compiler can take advantage of the known information, e.g., by using fixed length arrays, in contrast to developing a generic multipurpose C++ FE code. After the C++ code is generated, it is compiled into a mex file that can be called from MATLAB.

3.1.4. Sparse matrix assembly. Built into the C++ code is a generic sparse matrix assembly class that interfaces to the MATLAB sparse matrix format. It originally started from this code [15] but has diverged from it. It is more convenient to do the matrix assembly in C++, without the user's involvement, than to use the MATLAB `sparse` command to build the sparse matrices from array data. This is especially the case if iso-parametric (curved) elements are used with complicated FE basis functions.

The embedded matrix assembly class determines the sparsity structure as the local FE matrices are computed and inserted into the structure. Of course, this can be slow for large matrices. FELICITY provides a way to reuse the sparsity structure when reassembling matrices, such as in time-dependent problems. This is accomplished by calling the `mex` matrix assembly code with the (previously assembled) FE matrix `struct` as the first input.

The assembly strategy of FELICITY is to loop through the *elements* of a given subdomain and assemble those contributions separate from other subdomains. This is done for each subdomain. Note: FELICITY never traverses the "edges" in a subdomain; only the elements of the subdomain (those mesh entities with largest topological dimension in the subdomain) are traversed. Put differently, the only way FELICITY will traverse mesh edges is if they are defined as a subdomain and an FE form is defined over that subdomain.

In the case of $H(\mathrm{div})$ elements, where a choice of sign must be made on each facet in the mesh, FELICITY knows the chosen orientation of the facets. Hence, it is trivial to include appropriate sign changes of basis functions on a given element when transforming the basis functions on that element. This is in contrast to traversing a unique list of facets in the mesh.

A future alternative for sparse matrix assembly could be to use an open source linear algebra package, such as `armadillo` [48], to build the sparse matrix structure.

3.2. Allocating DoFs. Allocating DoFs (see subsection 2.2.3) for an FE space defined over a subdomain of a mesh requires a loop through the mesh elements. On each element, a new set of indices is allocated for the DoFs on that element. However, the DoF indices on neighboring elements must match in order to ensure certain continuity requirements of the FE space [23, 20].

The structure of the allocation algorithm is mostly generic, except when ensuring DoFs match, e.g., across facets, edges, and vertices of a tetrahedral mesh. For a specific element, the m-file that defines the reference element (see subsection 2.1.2) contains the barycentric coordinates of the DoFs on the reference domain. This information is used by FELICITY to generate a special purpose C++ code that will allocate DoFs for that element while ensuring that neighboring DoFs match. This requires the DoF coordinates to be invariant under permutations of the reference domain's vertices [20]. If this property is not satisfied, FELICITY will output an error message.

3.3. Interpolation. Defining the abstract definition file for an interpolation (recall subsection 2.2.4) is not much different from defining the abstract form definition (see subsection 2.2.1). The same techniques described in subsection 3.1 are used, such as lightweight MATLAB classes, text processing, and the symbolic toolbox. In fact, some of the exact same code generation utilities for forms are also used for interpolation.

4. Examples. We demonstrate how to implement with FELICITY two example problems in the following sections. In order to run these examples, FELICITY must be installed properly (see the wiki [53] for more info).

4.1. Example: Laplace–Beltrami with weak boundary conditions. We consider the Laplace–Beltrami problem on a smooth surface Γ_s with smooth boundary

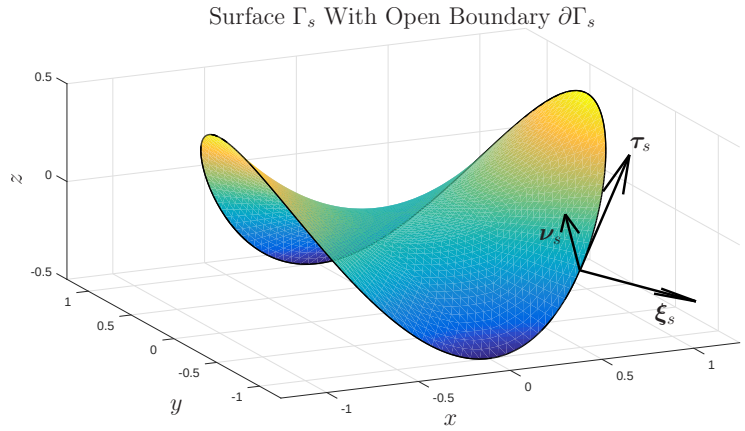

FIG. 1. *Surface diagram. Hypothetical surface Γ_s with boundary $\partial\Gamma_s$ highlighted by a thick black curve. The oriented normal vector of Γ_s is $\boldsymbol{\nu}_s$. The oriented tangent vector of the boundary curve $\partial\Gamma_s$ is $\boldsymbol{\tau}_s$. The co-normal vector (pointing out of the boundary) is denoted by $\boldsymbol{\xi}_s := \boldsymbol{\tau}_s \times \boldsymbol{\nu}_s$.*

$\partial\Gamma_s$, with given boundary condition g:

$$\begin{aligned} -\Delta_{\Gamma_s} u &= f \quad \text{in } \Gamma_s, \\ u &= g \quad \text{on } \partial\Gamma_s, \end{aligned} \tag{11}$$

where u is the solution, and f is the right-hand-side data; see Figure 1 for an example surface. Note that Δ_{Γ_s} is the Laplace–Beltrami operator defined by $\Delta_{\Gamma_s} := \nabla_{\Gamma_s} \cdot \nabla_{\Gamma_s}$, where ∇_{Γ_s} is the tangential (or "surface") gradient on Γ_s. The weak form of these equations is given as follows. Given data f in $H^{-1}(\Gamma_s)$, g in $H^{1/2}(\partial\Gamma_s)$, find u in $H^1(\Gamma_s)$, and λ in $H^{-1/2}(\partial\Gamma_s)$ such that

$$\begin{aligned} \int_{\Gamma_s} \nabla_{\Gamma_s} u \cdot \nabla_{\Gamma_s} v + \int_{\partial\Gamma_s} \lambda v &= \int_{\Gamma_s} f v \quad \forall v \in H^1(\Gamma_s), \\ \int_{\partial\Gamma_s} \mu u &= \int_{\partial\Gamma_s} \mu g \quad \forall \mu \in H^{-1/2}(\partial\Gamma_s), \end{aligned} \tag{12}$$

where we have imposed the boundary conditions via the Lagrange multiplier λ. If the solution is sufficiently smooth, then an integration by parts shows that $\lambda = -\boldsymbol{\xi}_s \cdot \nabla_{\Gamma_s} u$, where $\boldsymbol{\xi}_s$ is defined in Figure 1 (i.e., λ is given by the Neumann data). The formulation (12) is an example of how one can have separate function spaces defined on the domain Γ_s and its boundary $\partial\Gamma_s$.

The implementation of this example is given in supplementary section SM1.1; the output for this example is shown in Figure 2.

4.2. Example: EWOD. We consider a problem of droplet motion in microfluidics applications driven by electro-wetting-on-dielectric (EWOD) [22, 40, 55]. The model is essentially Hele–Shaw flow with surface tension and electric effects [56, 54]:

$$\begin{aligned} \alpha \partial_t \mathbf{u} + \beta \mathbf{u} + \nabla p = 0, \quad \nabla \cdot \mathbf{u} = 0 \quad &\text{in } \Omega(t), \\ p = \kappa(t) + E(t), \quad \partial_t \mathbf{x} \cdot \mathbf{n} = \mathbf{u} \cdot \mathbf{n} \quad &\text{on } \Gamma(t) \equiv \partial\Omega(t), \end{aligned} \tag{13}$$

where $\Omega(t) \subset \mathbb{R}^2$ is the time-dependent droplet domain, $\Gamma(t) \equiv \partial\Omega(t)$ is the liquid-gas interface (a closed one dimensional curve), $\mathbf{u}$ is the fluid velocity, p is the pressure,

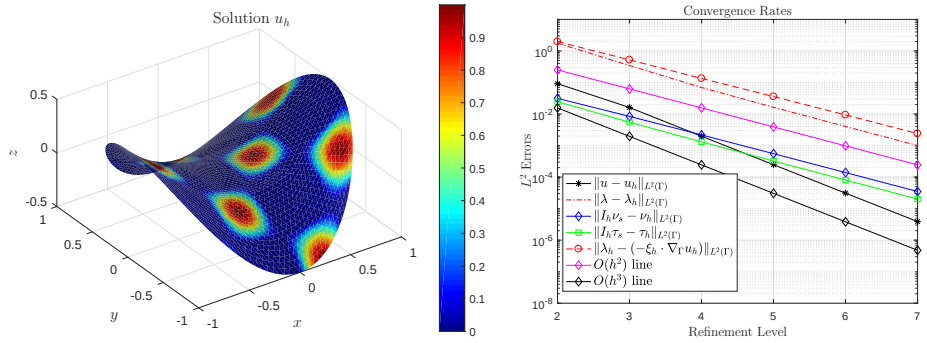

FIG. 2. *Visualization of the solution of* (SM4) *and convergence rates for the numerical error norms discussed in supplementary section* SM1.1.2.

$\kappa(t)$ is the signed curvature of $\Gamma(t)$, $E(t)$ is the electro-wetting force, $\mathbf{n}(t)$ is the outer unit normal vector of $\Gamma(t)$, and $\mathbf{x}(t)$ is the parameterization of $\Gamma(t)$; hence, this is a moving interface problem. The constants α, β are material parameters.

With this, we introduce a time-discrete weak formulation proposed in [30]. Given Lipschitz domains Ω^i, Γ^i, $\mathbf{x}^i \equiv \mathrm{id}_{\Gamma^i}$ in $H^1(\Gamma^i)$, and E^{i+1} in $H^{1/2}(\Gamma^i)$, find $\mathbf{u}^{i+1}$ in $H(\mathrm{div}, \Omega^i)$, p^{i+1} in $L^2(\Omega^i)$, $\mathbf{x}^{i+1}$ in $H^1(\Gamma^i)$, and λ^{i+1} in $H^{1/2}(\Gamma^i)$ such that

$$
\begin{aligned}
&\alpha \int_{\Omega^i} \frac{\mathbf{u}^{i+1} - \mathbf{u}^i}{\delta t} \cdot \mathbf{v} + \beta \int_{\Omega^i} \mathbf{u}^{i+1} \cdot \mathbf{v} \\
&\quad - \int_{\Omega^i} p^{i+1} \nabla \cdot \mathbf{v} + \int_{\Gamma^i} \lambda^{i+1} \mathbf{n}^i \cdot \mathbf{v} = - \int_{\Gamma^i} E^{i+1} \mathbf{n}^i \cdot \mathbf{v} \quad \forall \mathbf{v} \in H(\mathrm{div}, \Omega^i), \\
&\int_{\Omega^i} q \nabla \cdot \mathbf{u}^{i+1} = 0 \quad \forall q \in L^2(\Omega^i), \\
&\int_{\Gamma^i} \nabla_{\Gamma^i} \mathbf{x}^{i+1} \cdot \nabla_{\Gamma^i} \mathbf{y} - \int_{\Gamma^i} \lambda^{i+1} \mathbf{n}^i \cdot \mathbf{y} = 0 \quad \forall \mathbf{y} \in H^1(\Gamma^i), \\
&- \int_{\Gamma^i} \frac{\mathbf{x}^{i+1} - \mathbf{x}^i}{\delta t} \cdot \mathbf{n}^i \mu + \int_{\Gamma^i} \mathbf{u}^{i+1} \cdot \mathbf{n}^i \mu = 0 \quad \forall \mu \in H^{1/2}(\Gamma^i),
\end{aligned}
\tag{14}
$$

where we use a superscript i to denote the time-index. Note that we have implicitly used the following backward Euler method for updating $\mathbf{x}$:

$$
\mathbf{x}^{i+1}(s) = \mathbf{x}^i(s) + \delta t [\mathbf{u}^{i+1}(\mathbf{x}^i(s)) \cdot \mathbf{n}^i(\mathbf{x}^i(s))] \mathbf{n}^i(\mathbf{x}^i(s)) \tag{15}
$$

for all arc-length parameters s in Γ^i.

The curvature is approximated by λ^{i+1}, i.e., $\kappa^{i+1} \equiv \lambda^{i+1}$, where λ^{i+1} acts as a Lagrange multiplier enforcing the condition (15). Indeed, the third equation in (14) is the weak form of the curvature [28, 12]: starting from $-\Delta_\Gamma \mathbf{x} = \kappa \mathbf{n}$ and using a simple integration by parts [50] gives

$$
\int_\Gamma \kappa \mathbf{n} \cdot \mathbf{y} = - \int_\Gamma \Delta_\Gamma \mathbf{x} \cdot \mathbf{y} = \int_\Gamma \nabla_\Gamma \mathbf{x} \cdot \nabla_\Gamma \mathbf{y}, \tag{16}
$$

where κ is the curvature of the closed curve Γ and $\mathbf{n}$ is the normal vector. Note that, for a one dimensional curve, $\Delta_\Gamma \equiv \partial_s^2$, where ∂_s is the arc-length derivative. If Γ is a surface, then (16) holds with κ being the summed curvature of Γ.

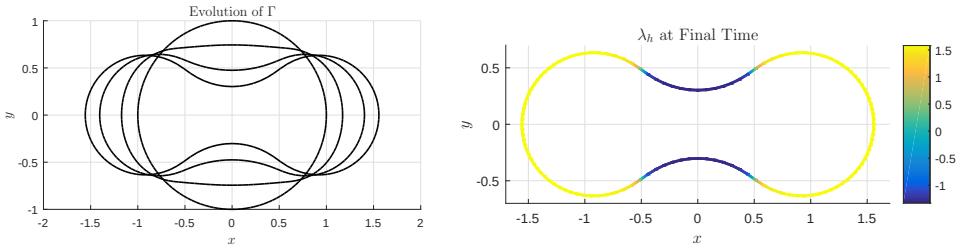

FIG. 3. *Illustration of the evolution of the droplet boundary* Γ^i*, from solving (*SM6*), at time index* $i = 0, 1, 4, 30$*. Solution* λ_h^{30} *is also shown.*

After solving one time-step, we define the new domain Γ^{i+1} via the parameterization $\mathbf{x}^{i+1}$; the domain Ω^{i+1} is defined so that $\partial\Omega^{i+1} \equiv \Gamma^{i+1}$. Note that $\mathbf{u}_h^{i+1}$ is implicitly mapped from Ω^i to Ω^{i+1} for use at the next time-step. Again, the formulation (14) is an example of how one can have variables defined on interfaces (e.g., Γ) interacting with variables defined in the bulk (e.g., Ω).

The implementation of this example is given in supplementary section SM1.2; the output for this example is shown in Figure 3.

5. Conclusions. We have described the FELICITY MATLAB/C++ toolbox for modeling and simulating coupled PDE systems and for developing FEMs in general. It provides a convenient platform to rapidly develop an FE code, experiment with an algorithm, and solve complicated multiphysics problems with multiple interacting subdomains. The code generation aspect of FELICITY is central to its flexibility and ease of use. In this regard, we were inspired by the FEniCS project [37, 39].

FELICITY allows for building a stand-alone application where *parts* of the implementation are automated. Complete automation hides details that must be accessed if the user goes outside the main "interface" of a package. On the other hand, doing everything oneself is tedious, especially for sections of code that almost never need modification. Our philosophy is to strike a balance between these two extremes.

5.1. Successes. The biggest success of FELICITY is the code generation, which enables the toolbox to tackle diverse problems while preserving a user-friendly interface. Complicated expressions in a bilinear form are easy to implement at a high level in the abstract definition file; the user does not need to look at the special purpose matrix assembly code that is generated, although the generated C++ code *is* commented and readable. The code generation capability for DoF allocation, interpolation, and point searching have analogous advantages. Thus, a user can easily build an infrastructure to solve a variety of problems by the FEM. Other successes include the following:

- Handling of multiple FE spaces on subdomains of varying co-dimension; FE forms can have contributions from multiple subdomains.
- Mesh geometry can be higher order, and users can access explicit higher order geometry (e.g., curvature) when defining FE forms.
- Lagrange, $H(\text{div})$, and $H(\text{curl})$ elements are implemented.
- Mesh classes with useful methods, including adaptive refinement in one and two dimensions. Some mesh generation utilities are included.
- "Manager" classes, such as `FiniteElementSpace`, provide fundamental tools for managing DoFs, boundary conditions, and other FEM details.
- Efficient C++ implementations of search trees (bitree, quadtree, octree) are

available for interpolation and closest point searching on higher order meshes.
- Users can couple their FELICITY code directly with other MATLAB code or toolboxes, such as the optimization toolbox.
- Several demos, help documentation, a manual, and online wikis are available.

No other FE toolbox automatically handles subdomains of varying co-dimension embedded in higher order meshes (to the best of our knowledge). Moreover, the available elements in FELICITY are stored in a "flat" m-file that is directly readable by the user. The mesh classes are easy to use and take advantage of the MATLAB `triangulation` class. The "manager" classes are object oriented, flexible, and simple to use. The search tree implementations are independent of any FEM so they can be used in many applications; closest point searching of meshes is also useful outside of FEMs [47]. Furthermore, the ability to leverage other MATLAB functionality cannot be overstated.

5.2. Areas of improvement. One drawback of FELICITY is that MATLAB is proprietary software. However, it is professionally maintained and an industry standard. Another limitation is that FELICITY is mainly meant for solving elliptic and parabolic PDEs by the FEM. Other limitations are not fundamental, but due to limited development resources. Indeed, the functionality in FELICITY is a consequence of the author's interests, a fact for most open-source software. Some future improvements include the following:

- Put in automated parallelism. In principle, one can use the MATLAB parallel toolbox, but this requires a significant amount of low-level implementation that has yet to be done.
- Implement more interfaces to external libraries, such as PETSc [10]. Access to robust, fast solvers is limited to packages that have a MATLAB interface.
- Add new classes of FEs; this is not easy for the casual user to do.
- Develop a better mesh class. The current setup uses the MATLAB `triangulation` class, which is convenient but makes the following difficult to implement: Three dimensional adaptive refinement/coarsening, discontinuous Galerkin (DG) methods, and computing multimesh intersections (such as for cutFEM).
- Implement higher order differentiation of basis functions in FE form definitions. This is difficult because the transformation rule, when mapping from the reference element, is complicated and hard to automate when computing higher order derivatives (e.g., third order) on higher order meshes.
- Add more helper classes to lower the barrier for users, such as methods to build higher order meshes from parameterizations.
- Have an option to hide the implementation details when calling FELICITY generated `mex` files.

Another improvement could be to translate FELICITY to use Python or perhaps Julia, which are open-source. This would alleviate some of the limitations listed here, as there are PETSc and parallel tools readily available in Python. Unfortunately, the development time/cost to do this is extremely high.

As for the other improvements, implementing new classes of FEs is difficult for any FE toolbox; it is not clear how to make this user-friendly. Designing a better mesh class, within C++, is a viable option. The author has an initial implementation [51] of the array-based half-facet (AHF) data structure [27, 46, 57] that will be interfaced to MATLAB as a replacement for the current mesh classes. Arbitrary order differentiation is not often needed in applications (currently FELICITY can compute the hessian of Lagrange basis functions, including the surface hessian). More helper

classes can further automate FE implementations, but still allow the user to ignore them and "roll their own code."

Calling the auto-generated `mex` files is somewhat tedious because of all the input arguments that must be supplied. One option to alleviate this would be to auto-generate the script that calls the `mex` file, but the script needs to be interfaced with the rest of the code so it is only a partial solution. Another option is to make the `mex` file input arguments more flexible so that the ordering of the arguments does not matter.

A more powerful option, which would be far-reaching into many other aspects of FELICITY, is to take advantage of the future replacement of the mesh class. Since the mesh class is developed in C++, the auto-generated matrix assembly code could be *directly integrated* into the mesh class. Similarly, other parts could be directly integrated. Indeed, one could envisage auto-generating a special purpose mesh class (built on the standard mesh class) that handles the FE spaces (e.g., allocating DoFs, getting DoF indices on subdomains), includes the matrix assembly routines, and has the point-searching and interpolation capability built-in. Of course, care must be taken to ensure that the code integration is not so tight that the user cannot get between the modules.

In conclusion, FELICITY has immense potential to expand and solve many kinds of complicated computational problems with the FEM.

REFERENCES

[1] *Feel++: Finite element embedded language and library in c++*, https://github.com/feelpp/feelpp, 2017.

[2] *Getfem++*, http://getfem.org, 2017.

[3] *Lifev: The parallel finite element library for the solution of pdes*, https://cmcsforge.epfl.ch/projects/lifev/, 2017.

[4] *Mfem*, http://mfem.org/publications/, 2017.

[5] *Vega fem library* http://run.usc.edu/vega/index.html, 2016.

[6] J. Alberty, C. Carstensen, and S. A. Funken, *Remarks around* 50 *lines of matlab: Short finite element implementation*, Numer. Algorithms, 20 (1999), pp. 117–137.

[7] A. Allievi and R. Bermejo, *A generalized particle search-locate algorithm for arbitrary grids*, J. Comput. Phys., 132 (1997), pp. 157–166, https://doi.org/10.1006/jcph.1996.5604.

[8] M. Alns, J. Blechta, J. Hake, A. Johansson, B. Kehlet, A. Logg, C. Richardson, J. Ring, M. Rognes, and G. Wells, *The FEniCS project version* 1.5, Archive of Numerical Software, 3 (2015), https://doi.org/10.11588/ans.2015.100.20553.

[9] P. Amestoy, I. Duff, and J.-Y. L'Excellent, *Multifrontal parallel distributed symmetric and unsymmetric solvers*, Comput. Methods Appl. Mech. Engrg., 184 (2000), pp. 501–520, https://doi.org/10.1016/S0045-7825(99)00242-X.

[10] S. Balay, S. Abhyankar, M. F. Adams, J. Brown, P. Brune, K. Buschelman, L. Dalcin, V. Eijkhout, W. D. Gropp, D. Kaushik, M. G. Knepley, L. C. McInnes, K. Rupp, B. F. Smith, S. Zampini, H. Zhang, and H. Zhang, *PETSc Web page*, http://www.mcs.anl.gov/petsc, 2016.

[11] W. Bangerth, R. Hartmann, and G. Kanschat, *deal.*II *– a general purpose object oriented finite element library*, ACM Trans. Math. Softw., 33 (2007), 24.

[12] E. Bänsch, *Finite element discretization of the Navier-Stokes equations with a free capillary surface*, Numer. Math., 88 (2001), pp. 203–235.

[13] S. Bartels, C. Carstensen, and A. Hecht, *Isoparametric FEM in matlab*, J. Comput. Appl. Math., 192 (2006), pp. 219–250, https://doi.org/10.1016/j.cam.2005.04.032.

[14] P. T. Bauman and R. H. Stogner, *Grins: A multiphysics framework based on the libMesh finite element library*, SIAM J. Sci. Comput., 38 (2016), pp. S78–S100, https://doi.org/10.1137/15M1026110.

[15] D. Bindel, *femat_sparse: Fast finite element matrix assembly in matlab*, http://www.cs.cornell.edu/~bindel//blurbs/femat_sparse.html, 2017.

[16] D. Boffi, F. Brezzi, and M. Fortin, *Mixed Finite Element Methods and Applications*, Springer Ser. Comput. Math. 44, Springer, Heidelberg, 2013.
[17] A. Bonito, R. H. Nochetto, and M. Sebastian Pauletti, *Parametric FEM for geometric biomembranes*, J. Comput. Phys., 229 (2010), pp. 3171–3188, https://doi.org/10.1016/j.jcp.2009.12.036.
[18] D. Braess, *Finite Elements: Theory, Fast Solvers, and Applications in Solid Mechanics*, 2nd ed., Cambridge University Press, 2001.
[19] S. C. Brenner, A. E. Diegel, and L. yeng Sung, *An efficient solver for a mixed finite element method for the Cahn-Hilliard equation*, submitted.
[20] S. C. Brenner and L. R. Scott, *The Mathematical Theory of Finite Element Methods*, 3rd ed., Texts Appl. Math. 15, Springer, New York, 2008.
[21] F. Brezzi and M. Fortin, *Mixed and Hybrid Finite Element Methods*, Springer-Verlag, New York, 1991.
[22] S. K. Cho, H. Moon, and C.-J. Kim, *Creating, transporting, cutting, and merging liquid droplets by electrowetting-based actuation for digital microfluidic circuits*, Journal of Microelectromechanical Systems, 12 (2003), pp. 70–80.
[23] P. G. Ciarlet, *The Finite Element Method for Elliptic Problems*, Classics Appl. Math. 40, 2nd ed., SIAM, Philadelphia, 2002, https://doi.org/10.1137/1.9780898719208.
[24] R. Cimrman, *SfePy - write your own FE application*, in Proceedings of the 6th European Conference on Python in Science (EuroSciPy 2013), P. de Buyl and N. Varoquaux, eds., 2014, pp. 65–70, https://arxiv.org/abs/1404.6391.
[25] M. Dabrowski, M. Krotkiewski, and D. W. Schmid, *Milamin: Matlab-based finite element method solver for large problems*, Geochemistry, Geophysics, Geosystems, 9 (2008), Q04030.
[26] A. Dedner, R. Klöfkorn, M. Nolte, and M. Ohlberger, *A generic interface for parallel and adaptive discretization schemes: Abstraction principles and the Dune-Fem module*, Computing, 90 (2010), pp. 165–196, https://doi.org/10.1007/s00607-010-0110-3.
[27] V. Dyedov, N. Ray, D. Einstein, X. Jiao, and T. Tautges, *AHF: Array-based half-facet data structure for mixed-dimensional and non-manifold meshes*, Engineering with Computers, 31 (2015), pp. 389–404, https://doi.org/10.1007/s00366-014-0378-6.
[28] G. Dziuk, *An algorithm for evolutionary surfaces*, Numer. Math., 58 (1990), pp. 603–611.
[29] G. Dziuk and C. M. Elliott, *Finite element methods for surface PDEs*, Acta Numer., 22 (2013), pp. 289–396, https://doi.org/10.1017/S0962492913000056.
[30] R. S. Falk and S. W. Walker, *A mixed finite element method for EWOD that directly computes the position of the moving interface*, SIAM J. Numer. Anal., 51 (2013), pp. 1016–1040, https://doi.org/10.1137/12088567X.
[31] G. N. Gatica, *A Simple Introduction to the Mixed Finite Element Method: Theory and Applications*, SpringerBriefs in Mathematics, Springer, 2014.
[32] F. Hecht, *New development in freefem++*, J. Numer. Math., 20 (2012), pp. 251–265.
[33] M. Heil and A. Hazel, *oomph-lib*, http://oomph-lib.maths.man.ac.uk/doc/html/index.html, 2017.
[34] T. J. R. Hughes, *The Finite Element Method: Linear Static and Dynamic Finite Element Analysis*, Dover Publications, Mineola, NY, 2000.
[35] A. Hunt and D. Thomas, *The Pragmatic Programmer: From Journeyman to Master*, Addison-Wesley, 2000.
[36] X. S. Li, *An overview of SuperLU: Algorithms, implementation, and user interface*, ACM Trans. Math. Softw., 31 (2005), pp. 302–325.
[37] A. Logg, *Automating the finite element method*, Arch. Comput. Methods Eng., 14 (2007), pp. 93–138.
[38] A. Logg, K.-A. Mardal, and G. Wells, *Automated Solution of Differential Equations by the Finite Element Method*, Springer, 2012.
[39] A. Logg and G. N. Wells, *DOLFIN: Automated finite element computing*, ACM Trans. Math. Softw., 37 (2010), pp. 1–28, https://doi.org/10.1145/1731022.1731030.
[40] H. Moon, A. R. Wheeler, R. L. Garrell, J. A. Loo, and C.-J. Kim, *An integrated digital microfluidic chip for multiplexed proteomic sample preparation and analysis by MALDI-MS*, Lab on a Chip, 6 (2006), pp. 1213–1219, https://doi.org/10.1039/b601954d.
[41] A. Napov and Y. Notay, *Algebraic analysis of aggregation-based multigrid*, Numer. Linear Algebra Appl., 18 (2011), pp. 539–564, https://doi.org/10.1002/nla.741.
[42] A. Napov and Y. Notay, *An algebraic multigrid method with guaranteed convergence rate*, SIAM J. Sci. Comput., 34 (2012), pp. A1079–A1109, https://doi.org/10.1137/100818509.

[43] R. H. Nochetto and S. W. Walker, *A hybrid variational front tracking-level set mesh generator for problems exhibiting large deformations and topological changes*, J. Comput. Phys., 229 (2010), pp. 6243–6269.

[44] C. G. Petra, O. Schenk, and M. Anitescu, *Real-time stochastic optimization of complex energy systems on high-performance computers*, IEEE Computing in Science & Engineering, 16 (2014), pp. 32–42.

[45] C. G. Petra, O. Schenk, M. Lubin, and K. Gärtner, *An augmented incomplete factorization approach for computing the Schur complement in stochastic optimization*, SIAM J. Sci. Comput., 36 (2014), pp. C139–C162, https://doi.org/10.1137/130908737.

[46] N. Ray, I. Grindeanu, X. Zhao, V. Mahadevan, and X. Jiao, *Array-based hierarchical mesh generation in parallel*, Procedia Engineering, 124 (2015), pp. 291–303, https://doi.org/10.1016/j.proeng.2015.10.140.

[47] S. J. Ruuth and B. Merriman, *A simple embedding method for solving partial differential equations on surfaces*, J. Comput. Phys., 227 (2008), pp. 1943–1961, https://doi.org/10.1016/j.jcp.2007.10.009.

[48] C. Sanderson and R. Curtin, *Armadillo: C++ linear algebra library*, http://arma.sourceforge.net/, 2017.

[49] S. W. Walker, *Tetrahedralization of isosurfaces with guaranteed-quality by edge rearrangement (TIGER)*, SIAM J. Sci. Comput., 35 (2013), pp. A294–A326, https://doi.org/10.1137/120866075.

[50] S. W. Walker, *The Shapes of Things: A Practical Guide to Differential Geometry and the Shape Derivative*, Adv. Des. Control 28, SIAM, Philadelphia, 2015, https://doi.org/10.1137/1.9781611973969.

[51] S. W. Walker, *C++ mesh class implementing the ahf*, https://github.com/walkersw/AHF, 2017.

[52] S. W. Walker, *FELICITY: Manual*, https://www.mathworks.com/matlabcentral/fileexchange/31141-felicity/, 2017.

[53] S. W. Walker, *Felicity wiki documentation*, https://github.com/walkersw/felicity-finite-element-toolbox/wiki, 2017.

[54] S. W. Walker, A. Bonito, and R. H. Nochetto, *Mixed finite element method for electrowetting on dielectric with contact line pinning*, Interfaces Free Bound., 12 (2010), pp. 85–119.

[55] S. W. Walker and B. Shapiro, *Modeling the fluid dynamics of electrowetting on dielectric (EWOD)*, Journal of Microelectromechanical Systems, 15 (2006), pp. 986–1000.

[56] S. W. Walker, B. Shapiro, and R. H. Nochetto, *Electrowetting with contact line pinning: Computational modeling and comparisons with experiments*, Phys. Fluids, 21 (2009), 102103, https://doi.org/10.1063/1.3254022.

[57] X. Zhao, R. Conley, N. Ray, V. S. Mahadevan, and X. Jiao, *Conformal and non-conformal adaptive mesh refinement with hierarchical array-based half-facet data structures*, Procedia Engineering, 124 (2015), pp. 304–316, https://doi.org/10.1016/j.proeng.2015.10.141.