Proceedings of the ASME 2019
International Design Engineering Technical Conferences and
Computers and Information in Engineering Conference
IDETC/CIE 2019
August 18-21, 2019, Anaheim, CA, USA

IDETC2019-98000

CHECKING THE AUTOMATED CONSTRUCTION OF FINITE ELEMENT SIMULATIONS FROM DIRICHLET BOUNDARY CONDITIONS

Kevin N. Chiu
Dept. of Mechanical Engineering
University of Maryland
College Park, Maryland

Mark D. Fuge
Dept. of Mechanical Engineering
University of Maryland
College Park, Maryland

ABSTRACT

From engineering analysis and topology optimization to generative design and machine learning, many modern computational design approaches require either large amounts of data or a method to generate that data. This paper addresses key issues with automatically generating such data through automating the construction of Finite Element Method (FEM) simulations from Dirichlet boundary conditions. Most past work on automating FEM assumes prior knowledge of the physics to be run or is limited to a small number of governing equations.

In contrast, we propose three improvements to current methods of automating the FEM: (1) completeness labels that guarantee viability of a simulation under specific conditions, (2) type-based labels for solution fields that robustly generate and identify solution fields, and (3) type-based labels for variational forms of governing equations that map the three components of a simulation set—specifically, boundary conditions, solution fields, and a variational form—to each other to form a viable FEM simulation. We implement these improvements using the FEniCS library as an example case. We show that our improvements increase the percent of viable simulations that are run automatically from a given list of boundary conditions. This paper's procedures ultimately allow for the automatic—i.e.*, fully computer-controlled—construction of FEM multi-physics simulations and data collection required to run data-driven models of physics phenomena or automate the exploration of topology optimization under many physics.*

1 INTRODUCTION

Modern advances in computation provide pathways for data-intensive processes in mechanical design. For example, current state of the art approaches to engineering analysis, topological optimization, generative design, and machine learning can benefit from large amounts of training data or methods to generate that data. Generating this data manually—*e.g.*, by having an employee manually run various test cases—is infeasible due to the large quantity needed, so a method to automate the generation of such data is crucial to these data-driven methods. To acquire this data, we wish to automate the simulation of physical phenomena using the Finite Element Method (FEM) to solve the partial differential equations (PDEs) that govern many physical phenomena. From these simulations, we can calculate, *e.g.*, forces acting on objects from fluid stress fields or differences in electrostatic potential. To build these simulations manually is time-consuming and impractical for the large number of simulations needed in machine learning model training, so this paper proposes a method to check that an automatically-generated simulation will run given Dirichlet boundary conditions that satisfy certain properties.

As an illustrative example, consider a fluid flowing through a pipe in the presence of an electrostatic field (*e.g.*, as is common in biological microfluidic devices) shown in Fig. 1. The fluid in the pipe is governed by the Navier-Stokes equations, whereas the electrostatic field is governed by the Poisson equation. If the fluid and electric field are not coupled, then two simulations can be run separately. However, if they are coupled, then both

fluid and electrostatic boundary conditions are needed to run a fully coupled simulation; if not enough boundary conditions are provided, the coupled simulation cannot run, but some subset of uncoupled simulations may. If an algorithm wanted to automatically construct each of the possible subsets of simulations in this example, it would need to overcome several crucial problems:

1. Choosing a variational equation that models the problem, defining appropriate solution fields to solve the simulation, and applying the correct boundary conditions to ensure convergence of the solution.
2. Guaranteeing the viability of a simulation given the boundary conditions, solution fields, and variational form.
3. Ensuring robustness of simulations to permutations in variational equations and boundary conditions—*i.e.*, so long as there is sufficient information to construct the simulation, the order of that information should not matter.

Specifically, the key contributions of this paper are:

1. We discuss properties of simulation sets necessary to run FEM simulations, differentiating them into what we call *complete*, *partially complete*, and *incomplete* simulations. Under certain conditions, these labels guarantee simulation viability.
2. We propose the use of type-based labeling to generate appropriate solution fields from Dirichlet boundary conditions without knowing ahead of time the types (scalar, vector, or tensor) or dimensions of the boundary conditions. This contrasts previous implementations where we have prior knowledge of the boundary condition types and manually build solution fields based on that knowledge.
3. We apply type-based labeling in variational forms to improve robustness of simulations to ordering permutations of solution fields, boundary conditions, and variational form terms. This gives us the ability to construct simulations more robustly than previous implementations in which, *e.g.*, the order of boundary conditions affects whether a simulation runs or not.

To demonstrate our above contributions, we specifically use the FEniCS library [1] to implement the FEM; however, these contributions are applicable to other implementations of the FEM as well.

2 RELATED WORK

Past approaches to automating the construction of FEM models falls roughly into three camps: (1) Automating Mesh Generation, (2) Knowledge-Based Engineering, and (3) Languages for general purpose, multi-physics solvers.

Much of the previous work in automating FEM focuses on the mesh generation process. For example, [2] and [3] describe basic methods of automating the meshing process, leading to a variety of applications, e.g., from bio-medical [4] to environmental [5]. Such approaches have led to major advances in how to discretize FEM problems, assuming that the variational form of the governing equation and boundary conditions are known. In contrast, our paper focuses on how to assemble all parts of the FEM solution including the solution fields (for a given mesh), the variational form, and the boundary conditions. As such, this past work in automated mesh generation is complementary to this paper, and we assume an appropriate quality mesh is provided (or can be computed using past methods) because our concerns lie with guaranteeing simulation viability and setting up the FEM problem definition.

Above the mesh-level, approaches from knowledge-based engineering (KBE) abstract the mid- and high-level components of a design by capturing rules and hierarchies between levels of different components [6]. By implementing high-level primitives, such as airplane wings or fuselages, KBE allows a designer to consider more detailed improvements by providing a shortcut to methods of analysis early in the design process [7]. When applied to geometry as in [8], KBE allows a simplified method of generating designs by adjusting input parameters to the program. Additionally, KBE can be extended to multi-disciplinary optimization [9] and, with machine learning, to predicting, *e.g.*, manufacturing cycle times [10]. While KBE promotes the use of high-level abstractions in design, such abstractions are largely manually constructed and cannot automatically generate new multi-physics FEM simulations needed for generating data or optimizations in data-driven design.

The last area of related work lies in programmatic languages for writing and simulating PDEs using FEM. Specifically, our work extends the work of [11] and [12], which laid the framework for the FEniCS library [1]—a programmatic language for writing and simulating FEM solutions to PDEs, such as the Unified Form Language [13]. [14], a similar package, also provides a language for implementing complex FEM simulations, acting either as a black-box solver or a framework for custom implementations. While such approaches have helped standardize how one can describe and implement various types of single and multi-physics analysis (*e.g.*, [15]) they still require researchers to write customized code for implementing specific physics, particularly in coupled models. The closest work to this paper is the work of Xia [16], who implements a multi-physics solver that can generate FEniCS code automatically by assuming that boundary conditions exist for every portion of the domain. Our approach differs from [16] in that we make no assumptions regarding the types or number of boundary conditions given, and can automatically generate and check for multiple subsets of physics models that are consistent with the boundary conditions.

3 EXPERIMENTAL SETUP

In this paper, we assume that only *physically-realizable* boundary conditions are given as inputs. For example, fluid flowing through a pipe can have one inlet velocity Boundary Condition (*BC*) with the other end as a pressure *BC*.[1]

For clarity throughout the rest of this paper, we use our previous example of a fluid flowing through a pipe in an electrostatic potential field to demonstrate the core contributions. This example provides us with two different types of physics, namely fluid flows and electrostatic potential fields, governed individually by the Navier-Stokes and Poisson equations, respectively, and by a coupled form in [17]. For the purposes of this paper, we do not consider the fully-coupled form of the equations; instead, we choose a "synthetic" variational form that requires the same types of *BC*s and solution fields as the fully-coupled form to make computation simpler and faster to replicate.

Specifically, we use the following three variational forms:

1. Quasi-static Incompressible Navier-Stokes (Pressure, Velocity)

$$F = \langle u, v\rangle + \langle \nabla u, \nabla v\rangle + \langle \nabla p, v\rangle + \langle \nabla \cdot u, q\rangle \tag{1}$$

2. Poisson

$$F = \langle \nabla u, \nabla v\rangle - f \cdot v - g \cdot v \tag{2}$$

3. "Synthetic" (Pressure, Velocity, Poisson)

$$F = \langle u, v\rangle + \langle p, q\rangle + \langle s, t\rangle + f \cdot t + g \cdot t \tag{3}$$

In the Navier-Stokes equations, u and p are the velocity and pressure, respectively, with v and q the respective test functions. In the Poisson equation, u is the variable of interest with v as the test function. In the Synthetic variational form, u, p, and s are velocity, pressure, and Poisson terms, respectively, with v, q, and t the respective test functions. In both the Poisson and Synthetic variational forms, f and g are internal and boundary source terms, respectively. dx signifies integration over the domain's interior, while ds signifies integration over the boundary.

While the Navier-Stokes and Poisson equations are fairly standard, the "Synthetic" variational form we use is constructed for the purposes of this paper. This form has no physical meaning; rather, it is used merely as a test equation that requires multiple and different *BC*s and combines a pressure field, a velocity field, and a Poisson field. The only criteria of the Synthetic variational form are that it requires different *BC*s (and subsequently solution fields) and that it converges to a solution, regardless of the physical interpretation of that solution.

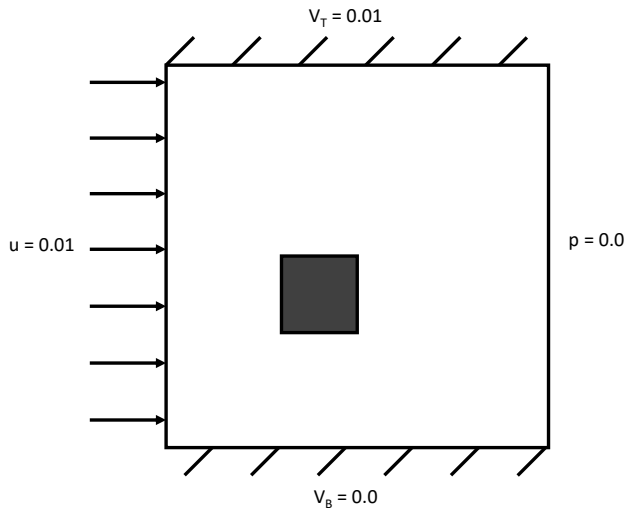

FIGURE 1. Example problem of fluid in a pipe in an electrostatic field. All boundary conditions are non-dimensionalized Dirichlet boundary conditions. The black square is an "obstacle" on which we want to calculate the forces from the fluid flow and the electrostatic potential field.

We also use three types of Dirichlet boundary conditions: fluid pressure, fluid velocity, and Poisson. We separate them into the following sets (denoted $\{BCs\}$) for our experiments:

0. {}
1. {Velocity L}
2. {Pressure R}
3. {Poisson T}
4. {Poisson T, Poisson B}
5. {Poisson T, Pressure R}
6. {Velocity L, Pressure R, Poisson T}
7. {Velocity T, Velocity L, Poisson B}
8. {Velocity B, Velocity T, Velocity L, Pressure R}
9. {Velocity L, Pressure R, Poisson T, Poisson B}
10. {Velocity B, Velocity T, Velocity L, Pressure R, Poisson T}
11. {Velocity B, Velocity T, Velocity L, Poisson T, Poisson B}
12. {Velocity B, Velocity T, Velocity L, Pressure R, Poisson T, Poisson B}

[1] In contrast, if both ends mandate velocities leaving the domain with no inputs, then fluid must be generated somewhere within the pipe to conserve mass (*i.e.*, maintain a divergence of 0). If there is no source of fluid, then the simulation will break due to non-physicality, not necessarily because the FEM solution to the PDE is incorrect. (That is, solutions might be mathematically possible but not physically possible.) Assuring that a given set of *BC*s is physically-realizable, while important, is not within the scope of this paper.

Here, "B", "T", "L", and "R" represent "Bottom," "Top," "Left," and "Right" respectively, referring to the edge on the example domain on which the *BC* is applied. As seen in Figure 1, Velocity L is a small, positive, uniform, horizontal flow, while Velocity B and Velocity T are both 0-magnitude flows (*i.e.*, the "no-slip" condition). Pressure R is 0 pressure. Poisson T and Poisson B are uniform positive and zero values, respectively.

These sets of *BC*s are heuristically chosen by the authors to maximize diversity in the following traits in the aforementioned sample problem:

1. Number of boundary conditions —how many boundary conditions are applied
2. Types of boundary conditions —fluid velocity, fluid pressure, and Poisson
3. Combinations of types —*e.g.*, fluid velocity with Poisson, fluid pressure by itself, *etc.*

Although many other possible test problems are possible, we combinatorically choose among these sets to simplify the method of testing our approaches. While a more exhaustive test set would include all 2^6 possible combinations of 6 *BC*s, many of these tests would be redundant for testing our approach (*e.g.*, two sets of *BC*s where the only difference between them is removing one no-slip condition and adding in the other).

This paper refers only to simulations with Dirichlet boundary conditions. While Neumann and Robin boundary conditions are important, they are not considered in this work due to the differences in their implementation in FEM, specifically because they are added as terms in the variational form. In this work, we assume the variational form is already given. It is expected that this paper is compatible with Neumann and Robin boundary conditions with minimal conceptual changes, though the specific implementation may differ if there are no Dirichlet boundary conditions acting on the same field(s).

Our experiment in §4.3 explores the accuracy with which we can automatically label a simulation set given the Variational Form of the governing equation (*VF*) and set of *BC*s while assuming the associated Solution Field (*SF*) is correct. We use the same sets of *BC*s in §5.3, testing different methods of generating these *SF*s. With the now fully-defined and complete simulation sets, we use the *VF*s from above and the same sets of *BC*s to build and run viable simulations to test the implementation of our approach compared to standard methods in §6.4.

4 COMPLETENESS OF VIABLE SIMULATION SETS

Each simulation requires what we call a *simulation set* of $\{BCs\}$, *SF*s, and a *VF*. To solve a PDE, boundary conditions are applied on *SF*s, whose values are manipulated to satisfy the *VF*. Because of these interwoven relations, a viable simulation has a simulation set with two important properties between its different members.

First, we define a *viable* simulation set as a simulation set whose members fully define a simulation in FEniCS (or any other FEM package) that: (1) runs to completion (*e.g.*, no shape mismatches or undefined terms) and (2) converges to a solution (*e.g.*, not a singular matrix, no infinite or NAN values).

In this section, we assume that, for a given *VF*, the *SF*s in the simulation set match the *VF* exactly; we discuss generating these *SF*s in §5. We define a simulation set with at least one *BC* for every variable type in the *VF* without extra as *complete*; if there are unused/extra *BC*s, that simulation set is called *partially complete*, and extra *BC*s can be trimmed off to make a complete simulation set. In both of these cases, the simulation set can define a viable simulation.

If the *VF* contains variables whose types do not have corresponding *BC*s, then that *VF* is considered *incomplete*, leading to a nonviable simulation set.

With this definition , we can decide whether a simulation set is viable or not by making several observations about its members and their relationships to each other.

4.1 Claim 1: A viable simulation must have a bijective function between *SF* and *VF*.

Define $S_f(v_{type})$ as a function that maps a type of variable field in a *VF* to its corresponding *SF*. To prove that $f = S_f(v_{type})$ is bijective, we want to show that f is both surjective and injective.

4.1.1 Claim 1.1: f is surjective.

Assume f is not surjective. According to the definition of "surjective," there must be some *SF* to which f does not map, *i.e.*, *SF* is unused in the *VF*. Because *SF* is unused, then the coefficients of the corresponding row(s) in the stiffness matrix are 0. Because at least one row of the stiffness matrix is all 0s, the stiffness matrix is singular and cannot be inverted, thus leading to no solution to the equation $\mathbf{A}x = b$ where $\mathbf{A}$ is the stiffness matrix. This contradicts our assumption that a simulation converges to a solution, so our initial assumption that f is not surjective must be false. Therefore, f is surjective.

4.1.2 Claim 1.2: f is injective.

Assume f is not injective. According to the definition of "injective," there must be two types v_1 and v_2 where

$$f(v_1) = S_f(v_1) = S_f(v_2) = f(v_2)$$

but $v_1 \neq v_2$. Because

$$S_f(v_1) = S_f(v_2)$$

v_1 and v_2 map to the same variable and are manipulated in exactly the same way in the *VF* (*e.g.*, they have the same $\{BCs\}$ applied). This implies that the same values satisfy the *VF*, leading to the same output values in the variable field to which they map. Because the outputs of the simulation are the same, then $v_1 = v_2$, which contradicts our assumption that f is not injective. Therefore, f is injective.

Since we have shown that f is both surjective and injective, f is bijective by definition. Thus, the function that maps a type of variable field to its corresponding *SF* is a bijective function, so we can treat a variable type and its *SF* as one unit. For the rest of this section, we assume a *VF* has its corresponding *SF*s and refer to them interchangeably unless otherwise specified. In §5, we show how to derive the appropriate *SF*s for a given *VF*.

4.2 Claim 2: A viable simulation set must have a surjective function from *BC* to *SF*.

Define $apply(bc)$ as a function that maps a *BC* to its corresponding *SF*. Thus, we impose a condition such that $g = apply(bc)$ exists and we want to prove g is a surjective function from *BC* to *SF*.

4.2.1 Condition 2.1: A viable simulation set does not contain unused *BC*s.

Assume we have a viable simulation set. An unused *BC* can be removed without changing the viability of the simulation set, so we remove that *BC* and redefine the simulation set without changing its solution.

4.2.2 Claim 2.2: g is surjective.

Assume g is not surjective. By definition, there must be some *SF* on which no *BC*s are applied. By [18], the stiffness matrix being solved in this simulation is singular, rendering the equation unsolvable. This contradicts our assumption that the simulation set is viable, so our assumption that g is not surjective is incorrect. Thus, g must be surjective.

4.3 Experiments

To ensure completeness labels can be generated automatically, we compare automatically-generated completeness labels with the ground truth labels manually assigned by the author. For each run, we take a single $\{BCs\}$ from §3 and consider its completeness with the three *VF*s also from §3. Table 1 shows the results of our experiments.

As can be seen from Table 1, our approach correctly labels the simulation sets for every test case we attempted, regardless of the type of *VF* or its completeness. That there are no partially complete Synthetic tests is due to the Synthetic *VF* requiring all 3 possible types of *BC*s. The total number of tests matches the number of unique $\{BCs\}$.

TABLE 1. Number of labels based on *VF* and ground truth labels. Our approach, row "# Correct," correctly labels 13 out of 13 cases for each *VF*.

	Poisson	Navier-Stokes	Synthetic
Incomplete	4	8	9
Part. Complete	7	4	0
Complete	2	1	4
# Correct	**13**	**13**	**13**
Total	13	13	13

5 TYPE-BASED INDEXING FOR SOLUTION FIELDS

We previously mentioned that we must define *SF*s that match the requirements of the *VF*. In most FEM implementations, one knows the type of physics or equation being solved ahead of time and, consequently, can build *SF*s that match the *VF* exactly. However, this approach is limited in that each code is specific to a single type of physics; to simulate a different type of physics, an entirely new simulation must be coded from scratch. The goal of this paper is to develop a generalizable system that can check that multiple single- and multi-physics FEM simulations can run without the need of checking each simulation manually, eventually to generate large datasets of simulations completely automatically.

To this end, we propose the use of type-based labels to derive the corresponding *SF*s for a simulation set from a list of *BC*s. We discuss baseline methods of deriving *SF*s before describing our implementation and experimental comparison of different approaches.

5.1 Baseline Methods for Solution Field Generation

Arbitrary The most naïve approach to generate *SF*s is to initialize some arbitrarily large number of *SF*s of multiple dimensions and types (scalar, vector, and tensor). This approach assumes $\{BCs\}$ provides no information about the required *SF*s. However, this is computationally expensive and makes no guarantees on the viability of the simulation (if there are too many *SF*s, or alternatively too few). In essence, this can produce too few or too many *SF*s, and we cannot know which without more information.

Minimal-Maximum As a small improvement on this naïve approach, we can use the number of *BC*s in $\{BCs\}$ as an upper bound on the number of *SF*s. In this case, if n is the number of *BC*s, n scalar fields, n vector fields, and n tensor fields would be initialized. This would, of course, be many more *SF*s than are needed.

Unique BC Rather than gathering no information from $\{BC\}$, we can assign each *BC* in $\{BCs\}$ to its own *SF*. This imposes an upper limit on the number of *SF*s that can possibly be required in a specified *VF* because, as stated in §4.1.1, each *SF* requires at least one *BC* for convergence. However, this approach does not link variables of the same type together, *e.g.*, if two fluid velocity *BC*s are given for a two-inlet pipe system, then two unrelated *SF*s would be generated, leading to ambiguity in which fluid velocity field the *VF* "should" use and resulting in an inaccurate simulation.

Unique Dimension A third approach could be to build a *SF* for each *BC* of a different *dimension*. For example, this method would work for the Navier-Stokes equations as pressure *BC*s, applied to a scalar *SF*, and velocity *BC*s, applied to a 2-D vector *SF*, would have separate fields. Unfortunately, this approach breaks down when different *BC*s require *SF*s of the same dimension, *e.g.*, an electrostatics-coupled Navier-Stokes simulation [17]. Specifically in this example, electrostatic potential is a scalar quantity and requires a scalar *SF*, and while pressure is also a scalar quantity and requires a scalar *SF*, electrostatic potential and pressure should not be treated as the same variable. This approach gives a lower bound of the number of *SF*s required because *BC*s of different dimensions must have unique *SF*s.

Unique Names Instead of extracting information from $\{BCs\}$, we can assume that a given $\{BCs\}$ completes the *VF* of interest and try to derive necessary *SF*s from the unique variable names in the *VF*. In fact, this approach correctly derives the number of *SF*s needed for a specific *VF*. However, naming each variable in a variational equation is somewhat arbitrary, and a simple difference in convention (*e.g.*, ϕ, electrostatic potential, vs. u, temperature in the heat equation, though both are applications of the Poisson equation) should not necessarily imply a different simulation type entirely (*e.g.*, a general Poisson simulation vs. electrostatic potential or heat simulations), only a different physical interpretation of the solution. Additionally, while there is some sort of convention for naming variables, ordering the *SF*s that correspond to those variables is loosely alphabetical at best and entirely random at worst. Finally, augmenting a single-physics model to multi-physics, *e.g.*, from Navier-Stokes to electrostatic-coupled Navier-Stokes, requires manually naming the additional *SF*(s). This ambiguity with both naming and ordering leads to uncertainties in which *BC*s and which *SF*s correspond.

Indexed Names A slight modification to the name-based labels stated above replaces names with numerical indices. For example, an electrostatic-coupled Navier-Stokes equation, which requires three *SF*s, could assign, *e.g.*, velocity to SF_0, pressure to SF_1, and electrostatic potential to SF_2. Numerical indices are easily extensible to an (almost) arbitrarily large number of *SF*s, but the same issue arises with ordering the *SF*s in some unambiguous manner, *e.g.*, the indices of the velocity and electrostatic potential *SF*s should not change the viability of the simulation set. This is portrayed in more detail in Table 3.

5.2 Proposed Implementation of Type-Based Labels for Solution Fields

We propose type-based labeling to mitigate the disadvantages of name-based or index-based labels. While this approach has some overhead in encoding more information into the *VF*, it provides an unambiguous label for each unique type of *BC* in $\{BCs\}$, which provides more robust checks for completeness, and removes the issue of ordering the *SF*s as they are referred to only by their type, rather than an arbitrary name or index.

To build the *SF*s, we use the *builder* class, described in Algorithm 1 and below.

Algorithm 1 Builder

```
1:  Initialize dictionary sf
2:  for BC in {BCs} do
3:      Add BC to sf with the key type(BC)
4:  end for
5:  Initialize empty mixed element mix
6:  for key_type in sf do
7:      Add dimensioned element to mix
8:  end for
9:  Build the function space from mix
10: for key_type in sf do
11:     for BC in key_type do
12:         fieldToUse = key_type
13:         value = BC.value
14:         location = BC.location
15:         Create FDBC and append to list of FDBCs
16:     end for
17: end for
18: Return mixed function space and FDBCs
```

A list of BCObjects is the input to this algorithm. We start with a dictionary (line 1) to store each unique *type* (line 3) since it only matters that each *type* has a *SF* rather than each *BC*. Once we have extracted the types, we add appropriate elements to a mixed element placeholder (line 7). Each of the added elements has dimensions according to the *type* it represents (*e.g.*, fluid pressure fields are scalars and use scalar elements, whereas fluid velocity fields in 2-D have 2-D vector elements). These dimensions can be derived from the *value* property of a *BC* object. With this mixed element *mix*, FEniCS can build a function space (line 9) that contains *SF*s corresponding to the different elements in the mixed element.

Finally, the algorithm creates FEniCS Dirichlet Boundary Condition objects (*FDBCs*), which require information about the solution field to be used, value, and location of each *BC* (lines 10-17). The mixed function space and list of *FDBCs* are returned as outputs. The *FDBCs* are specific to the FEniCS library, but any implementation-specific object or function to apply Dirichlet boundary conditions (*DBCs*) on the stiffness matrix can be used instead of *FDBCs*.

We would like to draw special attention to lines 3 and 12. In line 3, we use a key that, rather than a number or a name, is the *type* of *BC* being applied. This ensures that the function mapping from the *BC*s to the *SF*s is surjective, thus meeting one of the criteria for a complete simulation set from section §4.

In line 12, we again refer to key_{type}. Referring to the solution field by its type is fairly straightforward: we find the *SF* whose key matches the *BC*'s type[2] and apply the *BC* onto that *SF*.

Also of note are lines 6 and 10, where we iterate over the keys matching each type of *BC*, thus ensuring *BC*s of the same types are kept on the same *SF* and those of different types are separated.

5.3 Experiments

We compare our approach of type-based indices with the baseline approaches. We use the sets of *BC*s mentioned in §3. For every ordering permutation of that $\{BCs\}$, we use each baseline and the proposed approach to generate *SF*s and compare the generated fields to those we generate manually. The ratio of correctly generated[3] *SF*s to total generated is the accuracy of the method. Table 2 shows the results of our experiment.

From Table 2, we note that as we move to more sophisticated methods, we see an increase in the number of $\{BCs\}$ that may have correctly generated *SF*s. We also note that every method is generalizable[4] except method "Unique Names," which uses names of variables to differentiate fields. Even so, methods "Unique Names," "Indexed Names," and "Types" all generate at least some correct *SF*s. With the switch to indexed names, method "Indexed Names" seems to generalize method "Unique Names." However, these two methods perform poorly as the possible number of orderings increases.

To test robustness to *BC* ordering, we run this experiment with every order permutation of the *BC*s in each $\{BCs\}$. Table 2 includes the results of these permuted ("shuffled") boundary conditions.

From Table 2, the "Unique BC" and "Unique Dimension" approaches correctly generate *SF*s regardless of *BC* ordering, but in general, these approaches are not robust to the combinations of *BC*s that may be encountered. While the "Unique Names" and "Indexed Names" approaches generated the correct *SF*s at least sometimes in all of our test cases, Table 2 shows they lack of robustness when the *BC*s are ordered differently, especially as the number and types of *BC*s increase. Our type-indexed approach "Types" provides the correct references to the corresponding *SF*s regardless of the order of the *BC*s in all of our test cases.

6 TYPE-BASED INDEXING FOR VARIATIONAL FORMS

Similar to the previous section, we use type-based indexing to differentiate between variables in the *VF*s. This consistent labeling allows easy mapping of *SF*s to the corresponding variables in the *VF*. This also facilitates robustness to the order of terms in the *VF*.

6.1 Baseline Methods for Variational Form Encoding

Named Variables As previously mentioned, in many implementations of the Navier-Stokes (e.g., [19]) and Poisson (e.g., [19]) solvers, *SF*s are named with relatively intuitive or conventional letters. For example, in the Navier-Stokes *VF*, the velocity *SF* is often denoted u, and the pressure *SF* as p, as seen in Table 4, with the test functions v and q (not shown), respectively. Similarly, if somewhat less interestingly, the Poisson equation's *VF* in Table 4 uses u for its single *SF* with a test function of v.

Such naming conventions are common in existing state-of-the-art programmatic FEM solvers (such as when using the Unified Form Language [13]), and it assumes knowledge of the *VF* to be used beforehand. In contrast, this paper explores methods for constructing such FEM simulations in FEniCS automatically. Thus, hard-coding the names of *SF*s into the *VF* is not useful.

Indexed Variables Rather than hard-coding variable names, we again consider indexing *SF*s numerically. Table 4 gives an example of this implementation. A simple substitution of $vars[0]$ and $vars[1]$ for u and p, respectively, and their counterparts of $testVars[0]$ and $testVars[1]$ (not shown), allows these *SF*s to be referenced without hard-coded names, if somewhat less compactly. As an added bonus, any number of *SF*s can be referenced merely by increasing the index number. However, we run into the same issues of ambiguous ordering and references as stated in Section 5.

Consider the example in Table 3. In this 2-D Navier-Stokes fluid example, we apply two types of *BC*s, fluid velocity ("V") and fluid pressure ("P"). A velocity *SF* is a 2-D vector field, whereas a pressure *SF* is a scalar field, as seen in columns 2 and

[2]Our implementation actually uses an intermediate index-type dictionary, but because this dictionary is bijectively defined, we refer to the index and the type interchangeably.

[3]"Correctly generated," here, means that the algorithm builds the correct number of *SF*s of the correct size (scalar, vector, or tensor) and dimensionality. Additionally, all order permutations are compared to the default sets in §3, *i.e.*, permuted lists must generate the same *SF*s and refer to them in the same manner to be considered "correct."

[4]Generalizable, meaning extendable to different *VF*s without additional adjustments.

TABLE 2. Results of order-permuted *SF* generation using baseline and the proposed type-based approaches. Numbers are ratios of successfully-generated *SF*s to total generated *SF*s. Dashes indicate 0 successes and are used to make the table more readable.

BC Set	Arbitrary	Min-Max	Unique BC	Unique Dimension	Unique Names	Indexed Names	Types
0	-	1.0	1.0	1.0	1.0	1.0	1.0
1	-	-	1.0	1.0	1.0	1.0	1.0
2	-	-	1.0	1.0	1.0	1.0	1.0
3	-	-	1.0	1.0	1.0	1.0	1.0
4	-	-	-	1.0	1.0	1.0	1.0
5	-	-	1.0	-	0.5	0.5	1.0
6	-	-	1.0	-	0.167	0.167	1.0
7	-	-	-	1.0	0.333	0.333	1.0
8	-	-	-	1.0	0.75	0.75	1.0
9	-	-	-	-	0.083	0.083	1.0
10	-	-	-	-	0.3	0.3	1.0
11	-	-	-	1.0	0.3	0.3	1.0
12	-	-	-	-	0.0083	0.0083	1.0
Average	0.0	0.077	0.462	0.615	0.572	0.572	1.0
Generalizable	✓	✓	✓	✓	-	✓	✓

TABLE 3. Different orderings of *BC*s can result in inconsistently-referenced solution fields, even if the same type of simulation is run

BCs Applied	vars[0] Expected	vars[0] Actual	vars[1] Expected	vars[1] Actual
{V, P}	2-D Vector	2-D Vector	Scalar	Scalar
{V, V, P}	2-D Vector	2-D Vector	Scalar	Scalar
{V, P, V}	2-D Vector	2-D Vector	Scalar	Scalar
{P, V}	2-D Vector	Scalar	Scalar	2-D Vector

4 of Table 3. In the first three cases, where V is first, a 2-D vector field is labeled as vars[0], and a scalar field is vars[1]; when P comes before V as in the last row of Table 3, vars[0] now becomes a scalar field, and attempting to perform vector operations on scalar values results in a failed simulation. This required ordering severely limits the number of simulations that can be generated and run automatically as the order of *BC*s should not affect the solution to a viable simulation.

6.2 Proposed Implementation of Type-Based Labels for Variational Forms

We convert the numerical indices in the baseline approach into type-based indices[5]. These types are encoded into the *VF* as *needed types* to make completeness labeling easier. Table 4 provides an example of the (standard) name-based approach, indexed names, and our typed indices. Section 3 gives the full *VF*s we test.

[5]Our implementation actually uses an additional dictionary-type object to map indices to types of boundary conditions; however, because mapping between indices and *BC*s is bijectively defined, we can treat them as referring to the same object.

TABLE 4. Several possible references to *VF* variables

	Poisson	Navier-Stokes
Unique Names	**u**	**u**, **p**
Indexed Names	vars[0]	vars[0], vars[1]
Typed Index	vars["Poisson"]	vars["FluidVelocity"], vars["FluidPressure"]

6.3 Selection of Applicable Boundary Conditions for Viable Simulation Set Generation

With these labeled simulation sets and robust *SF*s, we can choose the sets which are complete and know that the simulation will run. However, partially complete simulation sets still contain extra *BC*s that need to be removed. For this, we can apply our type-based approach to select only the *BC*s that are applicable to our current *VF*. We implement this in Algorithm 2, which returns the set of *BC*s that are applicable to the simulation set. Lines 2 and 9 use our completeness labeling from Section 4 to check whether the there is a need to strip out the applicable *BC*s. Line 4 uses type-checking to determine if a given *BC* is required for a given *VF*.

Algorithm 2 Selector

```
1:  Inputs: VF, BCList
2:  if isPartiallyComplete(VF, BCList) then
3:      for BC in {BCList} do
4:          if type(BC) is in the required types of VF then
5:              Add BC to the list of applicable BCs
6:          end if
7:      end for
8:  else
9:      if isComplete(VF, BCList) then
10:         All BCs are applicable
11:     else
12:         Simulation set is incomplete
13:     end if
14: end if
15: Return applicable BCs
```

6.4 Experiments

We input into our Algorithm 3 objects: a mesh, on which we want to solve the simulation; a set of *BC*s, BCList, which we want to apply; and a *VF*. We take the BCList and VF and label these two, assuming the *SF* corresponding to the *VF* is appropriate. If the result is incomplete, we stop the simulation; if it is partially complete, we use Algorithm 2 (the "Selector") to turn this into a complete simulation. Once the (partial) simulation set is complete, we send it into Algorithm 1 (the "Builder"). This generates the actual *SF*s for the simulation set, with which the FEM can be solved. Figure 2 illustrates a flowchart of the process.

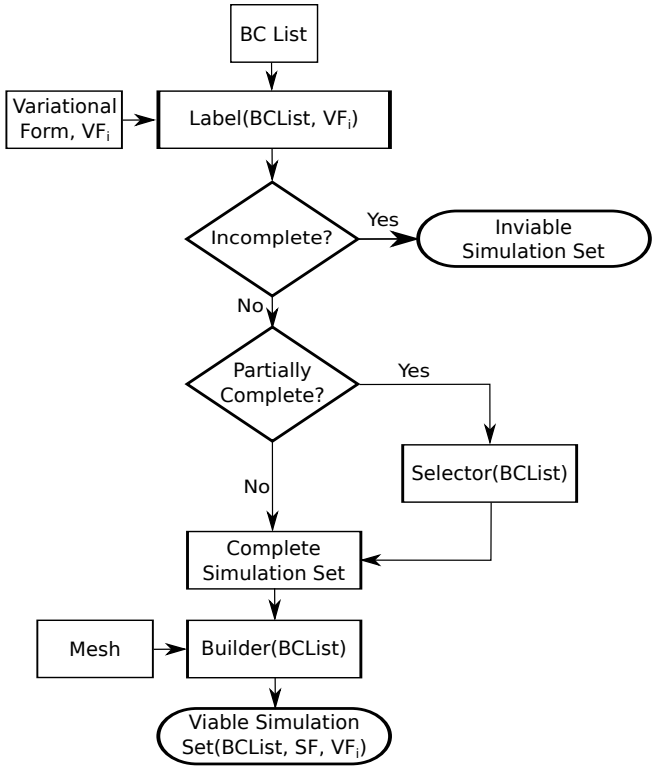

FIGURE 2. Flowchart of Experiments in Section 6.4

Following this process, we run two related experiments, the first to test the viability of a given order of *BC*s, and the second to test robustness to permutations of *BC* order.

6.4.1 Use of Completeness Labels to Determine Simulation Viability

With our fully defined and viable simulation sets, we now want to determine whether knowing these simulation sets' labels is useful in deciding which simulation sets to run. We run three different groups of simulation sets: "Complete," "Viable" (*i.e.*, both Complete and Partially Complete), and "All" sets. We tabulate the number of attempted simulations

TABLE 5. Results of non-shuffled simulation testing with and without completeness labels. Here, "Complete" refers to a too-restrictive class of simulation sets that are exactly complete; "Viable" refers to the class of simulation sets that are both complete and partially complete; and "All" refers to all of the simulation sets being tested, regardless of completeness.

	Attempted	Success	% Success
Complete	7	7	100
Viable	18	18	100
All	39	8	20.5

TABLE 6. Results of shuffled simulation testing with and without completeness labels. This is an extension of the previous experiment to include different *BC* orderings.

	# Attempted	# Succeeded	% Succeeded
Complete	897	897	100
Viable	**2765**	**2765**	**100**
All	3084	921	29.9

and the number of successful simulations to calculate the success percent of simulations each group attempts. With 3 variational forms and 13 sets of *bc*s, we expect $3 \times 13 = 39$ total simulations for each group. Table 5 shows the results of this experiment.

We expect every Complete or Partially Complete simulation to be viable, as is confirmed by the 100% running success rate of those simulation. From Table 5, we also see that, as "Viable" is a superset of "Complete," there are 11 more sets in "Viable" than there are in "Complete." Similarly, "All" is a superset of "Viable," resulting in 21 more attempted simulations.

6.4.2 Robustness of Type-Based Indices to *BC* Order Permutations We run a similar setup as Section 6.4.1, but we permute the order of *BC*s in every combination possible. Again, we calculate the number of simulations attempted by each method and compare it to the number of simulations that were run to completion. These results are in Table 6.

From Table 6, we can see that both "Complete" and "Viable" simulations run with 100% success in our tested cases. However, fewer simulations (in this case, less than 1/3) are run when the "Complete" criterion is used as compared to the "Viable" criterion. "All" simulations encompass 3084 possible cases, of which only 921 (29.9%) are able to run to completion. In this case, "Viable" covers 89.7% of "All" simulations while running to completion more than triple the number of simulations. From this, we can see that the "Viable" set attempts fewer simulations but runs more successful simulations than previous implementations, thus saving in computational time wasted on failed simulation attempts.

7 DISCUSSION

From the results of our experiment shown in Table 5, we note there are 10 simulations that "Viable" runs successfully but "All" does not. While at first glance, this seems odd, this behavior is expected. Because some of the sets in "All" are partially complete, there are extra *BC*s that violate condition 4.2.1 from §4. "Viable" sets, on the other hand, contain this completeness label, which informs whether extraneous *BC*s should be stripped before constructing a simulation. Thus, when an "All" simulation set is used to construct a simulation, extraneous *BC*s cause the construction to fail, whereas "Viable" simulation sets have extra *BC*s stripped before construction.

Even taking out Partially Complete simulation sets, one should expect that "All" runs the same number of successful simulations as "Complete" does since both should run only Complete sets. However, Table 5 shows this is not the case: "All" contains 8 successfully-run sets, whereas "Complete" only contains 7 sets. This extra viable simulation set actually comes from *BC* set 8 (*BC8*), which contains 3 fluid velocity and 1 fluid pressure *BC*. Specifically, in the "Complete" pass through the *BC*s, our approach checks *BC8* against the requirements of the Poisson *VF*. Finding that there is no Poisson *BC*, this simulation set (of fluid velocity/pressure *BC*s and Poisson *VF*) does not make a viable simulation. However, the "All" pass does not check the *BC* against the *VF*; thus, when the "All" algorithm encounters a set of fluid velocity and pressure conditions, a Navier-Stokes simulation is performed, regardless of the intended Poisson simulation. Because the Navier-Stokes *VF* is technically viable for *BC8*, the simulation completes without any problems, despite the fact that an entirely different set of equations is solved. A similar reasoning applies to the 897 "Complete" vs. 921 "All" simulations in Table 6.

Our results in Table 6 suggest that our approach allows both a larger number of simulations to be run without additional manual processing and that more of the attempted simulations run successfully, especially when combined with completeness labels. While it may seem that many of the simulations being run are redundant or repetitions of other simulations (*e.g.*, *BC3* and *BC5* differ in the addition of a Poisson *BC*), many of these cases include *BC*s whose addition does not fully-define other types of simulation sets, so running simulations with each of these *BC*s *should* result in the same exact simulations. In addition, our type-based indices allow the simulations to run without regard to the order of the *BC*s, providing the framework for automated simulation construction from *BC*s. Because our approach allows more successful runs with fewer attempts, we claim that implementing the completeness labels and type-based indices successfully

allows for robust automatic simulation checking.

However, our work is limited in several aspects. We previously mentioned that each set of *BC*s is assumed to be physically realizable; however, we do not check for whether a given set of *BC*s is physically realizable in this paper. For example, given two velocity inlets to a pipe, our implemented approach will consider that set as viable, even though it is physically impossible for a pipe to have only inlets and no outlets assuming that mass is conserved.

Secondly, we assume that *VF*s are given. The derivation of these equations can be done manually, but the process is difficult to automate while keeping the scope of this paper reasonably limited. We are investigating this in our future work.

Third, we restricted our tests to only 3 types of *VF*s and 3 types of *DBC*s to make experimentally testing our claims more straight-forward. Implementing more types of physics and *VF*s would extend the practical functionality of this approach, though that is separate from the intellectual contributions of the paper.

Fourth, our work only discusses Dirichlet *BC*s (DBCs) without Neumann (NBCs) or Robin *BC*s (RBCs). NBCs and RBCs are implemented differently than DBCs in that they are additional terms in the *VF*, while DBCs are enforced by manipulating values in the stiffness matrix. However, simulation sets would still need to be defined appropriately to create viable simulations, even in cases with pure NBCs or simulations that do not require DBCs, *e.g.*, those using inertial relief methods.

Finally, we have used only boundary value problems with no time dependence in this work. Initial value problems can be solved with similar setups of simulation sets, *e.g.*, assuming each time step in the simulation is quasi-static but depends on the previous state to determine the next state (*e.g.*, the Runge-Kutta method).

Although our initial goal in this work was the autonomous generation of meaningful FEM simulations, we realized that several critical issues arose. This paper is in response to one of those issues, namely that ensuring the viability of a simulation is critical to generating such simulations autonomously.

8 CONCLUSION

In this paper, we proposed three improvements to the algorithmic construction of FEM simulations. First, we discussed conditions between different aspects of a simulation to ensure viability. Second, we proposed a type-based indexing method to generate the correct solution fields for a simulation automatically. Third, we used a similar type-based indexing method to connect boundary conditions and solution fields to a variational form robustly. We showed that our implementation of completeness labels guarantees simulation viability, and when combined with type-based indices for solution fields and variational forms, provides the framework needed for constructing FEM simulations automatically from boundary conditions.

With our contributions, we are able to generate and check for viable physics simulations automatically from collections of physically-realizable boundary conditions. With extension to a larger number of PDEs, we lay the foundations of automatically constructing single- and multi-physics FEM models, allowing future implementations to build and run simulations autonomously with a higher degree of certainty that the simulation will run than current, manually-constructed methods. These automatic simulations can then lead to physics- and data-driven design optimization, machine learning, analysis, and topology optimization, allowing us to apply new computational methods to mechanical design and to provide a path forward for future data-driven design methods.

ACKNOWLEDGMENT

We acknowledge the funding and support provided by DARPA through their Fundamentals of Design program (#HR0011-18-9-0009). The views, opinions, and/or findings contained in this article are those of the author and should not be interpreted as representing the official views or policies, either expressed or implied, of the Defense Advanced Research Projects Agency or the Department of Defense.

REFERENCES

[1] Alnæs, M. S., Blechta, J., Hake, J., Johansson, A., Kehlet, B., Logg, A., Richardson, C., Ring, J., Rognes, M. E., and Wells, G. N., 2015. "The FEniCS project version 1.5". *Archive of Numerical Software,* **3**(100).

[2] Ho-Le, K., 1988. "Finite element mesh generation methods: a review and classification". *Computer-Aided Design,* **20**(1), pp. 27 – 38.

[3] Finnigan, P. M., Kela, A., and Davis, J. E., 1989. "Geometry as a basis for finite element automation". *Engineering with Computers,* **5**(3), Jun, pp. 147–160.

[4] Viceconti, M., Davinelli, M., Taddei, F., and Cappello, A., 2004. "Automatic generation of accurate subject-specific bone finite element models to be used in clinical studies". *Journal of Biomechanics,* **37**(10), Oct, pp. 1597–1605.

[5] Sun, L., Zhao, G., and Yeh, G.-T., 2018. "An automatic quadrilateral mesh generation algorithm applied to 2-d overland flow simulations". *Computational Geosciences,* **22**(5), Oct, pp. 1283–1303.

[6] Rocca, G. L., 2012. "Knowledge based engineering: Between AI and CAD. Review of a language based technology to support engineering design". *Advanced Engineering Informatics,* **26**(2), pp. 159 – 179.

[7] van Tooren, M. J. L., La Rocca, G., Krakers, L., and Beukers, A., 2003. "Design and Technology in Aerospace. Parametric Modelling of Complex Structure Systems Including

Active Components". In 13th International Conference on Composite Materials, San Diego, CA.

[8] van der Laan, A. H., and van Tooren, M. J. L., 2005. "Parametric modeling of movables for structural analysis". *Journal of Aircraft,* **42**(6), November-December, pp. 1605–1613.

[9] La Rocca, G., and van Tooren, M. J. L., 2006. "A modular reconfigurable software modelling tool to support distributed multidisciplinary design and optimisation of complex products". In 16th CIRP International Design Seminar, Kananaskis, Alberta, Canada, 16-19 July.

[10] Quintana-Amate, S., Bermell-Garcia, P., Tiwari, A., and Turner, C., 2017. "A new knowledge sourcing framework for knowledge-based engineering: An aerospace industry case study". *Computers & Industrial Engineering,* **104**, pp. 35 – 50.

[11] Logg, A., 2007. "Automating the finite element method". *Archives of Computational Methods in Engineering,* **14**(2), Jun, pp. 93–138.

[12] Logg, A., and Wells, G. N., 2011. "DOLFIN: automated finite element computing". *CoRR,* **abs/1103.6248**.

[13] Alnæs, M. S., Logg, A., Ølgaard, K. B., Rognes, M. E., and Wells, G. N., 2014. "Unified form language: A domain-specific language for weak formulations of partial differential equations". *ACM Transactions on Mathematical Software (TOMS),* **40**(2), p. 9.

[14] Cimrman, R., 2014. "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., pp. 65–70. http://arxiv.org/abs/1404.6391.

[15] , 2017. Python FEM and Multiphysics Simulations with FEniCS and FEATool. On the WWW, June. URL `www.featool.com/`.

[16] Xia, Q., 2017. "Automated Mechanical Engineering Design using Open Source CAE Software Packages". In FEniCS '18, Oxford, UK, June. URL `github.com/qingfengxia/FenicsSolver`.

[17] Emamy, N., Karcher, M., Mousavi, R., and Oberlack, M., 2015. "A high-order fully coupled electro-fluid-dynamics solver for multiphase flow simulations". In Proceedings of the VI international conference on coupled problems in science and engineering, pp. 753–9.

[18] Logan, D. L., 2014. *A First Course in the Finite Element Method*. Cengage Learning, Boston, MA.

[19] Langtangen, H. P., and Logg, A., 2016. *Extensions: Improving the Poisson Solver*. Springer International Publishing, Cham, pp. 109–141.