Rapid translation of finite-element theory into computer implementation based on a descriptive object-oriented programming approach

Murat YILMAZ∗
Department of Civil Engineering, Faculty of Civil Engineering, İstanbul Technical University, İstanbul, Turkey

Received: 29.12.2017 • Accepted/Published Online: 24.06.2018 • Final Version: 29.11.2018

Abstract: In this study, we present a framework for rapid prototyping of finite element (FE) theory for computer implementations. For this purpose, we propose an object-oriented (OO) application programming interface in the form of a domain-specific modeling language (DSML). In contrast to the traditional OO approach, the proposed framework deliberately avoids the use of subclassing for concrete implementations of node and element classes; it uses external objects, namely descriptors, instead. The descriptive design of the DSML provides developers with generic programming support for the construction and solution of discretization schemes, in the context of partial differential equations, in a self-explanatory syntax. We take advantage of Python’s descriptor protocol to make descriptors behave like natural dependencies of their owner class. We propose several descriptors to account for both theoretical and implementation-specific aspects of FE programming. By using concrete examples, we demonstrate that enhancing these descriptors with both symbolic and numerical computational utilities results in a concise and customizable code base for analysis and pre/postprocessing purposes. We select Python as the base programming language because of its support for essential programming features such as customizable classes, dynamic code, arbitrary arguments, method decoration, and descriptor protocol.

Key words: Automated, descriptive, finite element, programming, object-oriented

1. Introduction

Finite element (FE) programming has received considerable research attention over the years. In addition, among the various programming paradigms, object-oriented (OO) design has been increasingly employed in FE programming [1–8]; these early studies mostly concentrated on the primary concepts of OO programming (OOP), such as classes, objects, encapsulation, inheritance, and polymorphism. They discussed the advantages and disadvantages that OOP has over procedure-oriented programming with respect to FE analysis. Implementing new elements/theories into an existing framework with minimum effort became a significant challenge for code extensibility. The selection of object hierarchies and the design of their interactions are crucial for this purpose. Some researchers have shown that the OOP technique can greatly improve the implementation efficiency, extensibility, and ease of maintenance of engineering software [9–16]. While this is true, researchers are still required to be OOP experts and proficient with certain programming languages to correctly modify the source code to be extended. Unfortunately, most researchers are not coding experts, and learning to use a nontrivial software framework is a difficult and time-consuming activity [17]. A popular way of reducing this burden on FE researchers in a flexible manner is to provide them access to the functionality of the framework via

∗Correspondence: yilmazmura@itu.edu.tr

This work is licensed under a Creative Commons Attribution 4.0 International License.
an interpreter language; the OpenSees [18] and Kratos [15] frameworks offer elegant examples of interpreter bindings to TCL and Python, respectively. This feature provides a high level of extensibility when creating and dealing with different algorithms while using the structure of the framework; however, it does not offer much aside from being an advanced input interface. A reasonable strategy to minimize implementation effort is to offer an automated environment in the derivation of the weak form, its discretization, and the construction of the matrix forms of a given initial boundary-value problem [19–21]. Domain-specific modeling (DSM) [22] offers an alternative and effective way to facilitate developer tasks by detailing the idea of code generation, which automates the creation of executable source code directly from the DSM languages (DSMLs). One of the most striking examples of DSML for FE analysis is presented by Logg in [23]; this was put into practice in the DOLPHIN and FEniCS projects [24]. Several other FE frameworks that mimic the DSM approach also exist, such as GetFEM++ , FreeFem++ [25], and SfePy [26]. Although DSM approaches increase productivity in application-building activities by as much as an order of magnitude, the high-level abstraction provided by a fully equipped DSML inevitably confines developers to a type of restricted predefined domain. The framework presented in this study also shares similar formalism with the studies mentioned above; however, we propose several novel programming techniques while providing a similar level of effectiveness. Furthermore, we aim to maintain the flexibility and functionality of the classical OO style and a sufficiently short and self-expressive syntax. Figure 1 depicts the overall components of the presented framework and briefly indicates the concepts implemented in this study. The structure shown in Figure 1 is an example of a classical OO FE framework in terms of its object composition. The facilitations provided in the creation and use of these objects makes it distinctive. Although it is difficult to distinguish the actual boundaries, four distinctive features are proposed here. The first is easy instantiation and easy access of the node and element objects. For that, we adapted these classes to behave like automatic-instance containers with additional custom-filtering and state-altering support, by exploiting Python’s metaclass and magic-method features. This made the use of instance referencing and loops largely redundant. The second is the class description layer, which automates various standard in-class configurations. For that, we offered several automated utilities in the form of individual objects and allowed the developer to inject them as dependencies into their custom classes by taking advantage of Python’s descriptor protocol. This resulted in a concise DSML as the final product. The third is the variational calculation layer, which offers a matrix-free computation of the discretization schemes by automating the calculations of the element equations. As a distinguishing feature, we expose these schemes as symbolic descriptors and provide ways to perform their evaluations at instance-level, while simultaneously reflecting the results back to the developer in an OO-style. Lastly, the paper briefly addresses atomization in both pre- and postprocessing, which are also important for fulfilling the minimum requirements of an FE framework.

An outline of the rest of the article is described as follows. Section 2 discusses the design of the proposed architecture in detail, along with the derivation of an 8-node 3D structural solid element. In Section 3, the well-known Poisson problem is solved as an example of the complete code base. The paper concludes with a discussion of the presented approach in Section 4.

2. A descriptive FE programming style

2.1. A descriptive node class

An FE node is a specific point in space at which discretization data such as coordinates, degrees of freedom (DOFs), and discontinuities are stored and boundary conditions (BCs) are specified. Listing 1 is the demonstration of a custom Node class with necessary objects included to represent the nodes of a 3D solid element.
Listing 1: A custom node class for a 3D solid element.

```python
# Easy instantiation
with Reader(file="mesh.txt") as mesh:
    Node[1] = mesh.nodes
    Element[1] = mesh.elements

# Multiple object access
Node[lambda X: X==6].fix(U=0)
Node[lambda X: X==10].load(U=1)

# FE descriptors/Helpers
ID, Coordinate, DegreeOfFreedom, BoundaryCondition, Connectivity, Field, Differentiation, Integration, Variation, ShapeFunction.

# Analytical solve(assemble(Element.K) @ Node.fix == Node.load)

# A supplementary third party framework for linear algebra
scipy.linalg

# A supplementary third party framework for symbolic math
sympy

# Custom output
print(sum(Element[1].u))

# Custom graphical display
Draw(Element, Node[1].u)

# Descriptive Node/Element configurations
class Node(Descriptive):
    label = fw.ID()
    X, Y, Z = fw.COORD(3)
    U = fw.DOF()
    fix = fw.Fix()
    load = fw.Load()

class Element(Descriptive):
    nodes = fw.Connectivity.of(Node, by=ID)
    X, Y, Z = Node.nodes.X, Node.nodes.Y
    U = fw.DOF()  # Dirichlet BC collector.
    D = Node.Diff(X, Y)
    du = D @ u

# Variational forms
K = Integ(du @ du.T) @ D(Q))
```

Figure 1. Preliminary outline of the proposed facilitations along with the structure of the framework.

Note that the presented class does not rely on inheritance for customization; it instead relies only on the descriptive attributes defined by lines #2–#7. These attributes are special property-like objects that are externally programmed, which makes them quite useful in describing any particular state or behavior of their owner class (Node). In that regard, they are called descriptors. The processes offered by Python to acquire this utility are called the descriptor protocol (DP), which is a slightly more advanced version of the canonical property protocol (PP) of OOP. The difference is that in the DP, the owner class trusts the descriptor to change its state, while exposing itself as an argument in the property calls; thus, any intended functionality for the owner can be implemented externally by the descriptor. As an example, we take the “label,” defined with the “ID” descriptor-type (line #2), which is specifically programed to serve as an automatic reverse-mapper for the
owner instances, as shown with the pseudoimplementation in Figure 2. One should note that the developer
does not need to code anything, and only has to declare the ID attribute in order to acquire the promised
functionality.

Figure 2. A demonstration of the descriptor protocol (pseudocode for the ID).

Another aspect that makes descriptors so useful is their portability, especially when they are designed
to operate on a custom utility. For example, an Element class with a similar ID attribute will automatically
benefit from the exact same reverse-map functionality.

Other descriptors with Node are also designed in a similar fashion. COORD and DOF (lines #3 and #4)
are both implemented as symbolic math variables, which makes them quite useful in creating new descriptions
through mathematical operations in the form of “x = X + U” (line #5). A proof of concept implementation is
given in Figure 3.

Figure 3. Proof of concept demonstration of how DP can assist in the natural evaluation of symbolic expressions (note:
DOF and COORD are initiators for the Symbol objects).
Lastly, the BC descriptors, `Fix` and `Load` (lines #6 and #7), are designed exclusively as data collectors whose sole job is to record the match between the caller instance and the corresponding keyword arguments (kwargs) in a typical DP attribute call with the form “instance.attribute(kwargs).” Thus, they are utilized to identify any direct BC information in a self-expressive syntax with the forms “node.fix(U = 0)” and “node.load(W = 10).” The interpretation of this information is to be conducted later in the framework (or with the developer code) by checking the actual class types or names of these descriptors, in order to properly consume the BC data.

Another important facilitation offered by the proposed framework is the introduction of the `Descriptive` class (line #1), which is specifically designed to provide its child class with easy object instantiation and state-altering support, as demonstrated in Figure 4. Briefly, the `Descriptive` class declares a `Store` type class-level container to which instances are appended upon their instantiation. With this, the developer is automatically freed from variable referencing. In addition, all instance-states are automatically assigned according to the provided values of the keyword argument dictionary (see the magic method with **kwargs). Any action performed on the subclass is then captured by its metaclass, which is referred to as `Meta`. In the proposed case, calls in the form of “ClassName[filter]” are delegated to the `Store`, in which a decision is made to either return the call as a single instance or to return a multiple instance handler object, called a `Subscription`, by checking the custom filter provided by the developer. In the case of complex filters, such as slices or predicate methods, the `Subscription` is utilized to allow filtering and state-altering operations to be performed on multiple instances. This activity makes the use of loops largely redundant and is demonstrated in Section 3.

2.2. A custom master-element domain

The most essential part of any FE theory is the discretization of the primary variables of the original governing equation on a subdomain, which is called the element domain (ED). It is also common to map the element domain to a fixed and topologically compatible domain, which is known as the master domain (MD), given by

\[
X_i(r) = q^T_{X_i} \cdot \psi(r),
\]

where \( X \) and \( r \) represent the element and master coordinates, respectively. Here, \( q_{X_i} \) is the vector that constitutes the nodal coordinates of the corresponding ED, and \( \psi(r) \) is the vector that constitutes the shape functions. The latter will be referred to as the shape-function vector (SFV) in the remainder of the text. The Jacobian matrix is defined as

\[
J = \frac{\partial X}{\partial r} = q^T_{X} \cdot [\partial_r \psi], \quad [\partial_r \psi]_{ij} = \frac{\partial \psi_i}{\partial r_j},
\]

which includes the derivatives of SFV components with respect to \( r \). A single shape function (SF) can be expressed in terms of its bases in the following form:

\[
\psi_i(r) = \alpha_{ij} b_j(r),
\]

where \( \alpha \) is the coefficient matrix and \( b \) is the vector of the bases that span the shape function space in the MD. From Eq. (1), the SF should satisfy

\[
\psi_i(p^k) = \alpha_{ij} b_j(p^k) = \delta_{ik},
\]
where $p^k$ represents the master coordinates of the $k$th node and $\delta$ is the Kronecker Delta. From Eq. (4), the coefficient matrix $\alpha_{ij}$ has a solution in the following form:

$$[\alpha] = \begin{bmatrix} b(p^1) & b(p^2) & \cdots & b(p^{N-1}) & b(p^{N}) \end{bmatrix}^{-1},$$

(5)

where $N$ is the total point count (or node count) of the mapping. Once coefficients are obtained, the SFV of Eq. (1) and its derivatives can be calculated as

$$\frac{\partial^{n+m+p} \psi}{\partial r_1^n \partial r_2^m \partial r_3^p} = \psi^{nmp}(r) = [\alpha] \cdot b^{nmp}(r).$$

(6)

Here, $n$, $m$, and $p$ symbolize the derivative-orders with respect to the provided master coordinates in sequence.

In order to facilitate the evaluation of Eq. (6) in a customized manner, we offer a solution based on the decorator pattern and symbolic math computations. Figure 5 is a demonstration of the proposed syntax, through which we calculated the SFV of the well-known Trilinear interpolation. Therefore, we define eight base functions that constitute a vector in the form of
where \( r = \begin{bmatrix} r & s & t \end{bmatrix}^T \) represents the master coordinates in the \([-1, +1]\) interval. Note that \( \text{Trilinear} \) has the definition of an ordinary Python function with the return of Eq. \((7)\). As can be understood from its later use, the original function is transformed into a resourceful object that has the same name as the original and is designed to operate in accordance with Eqs. \((2)\) and \((6)\) free of charge.

### 2.3. A descriptive element class

An automated FE framework should offer a short and abstract syntax for basic mathematical operations, such as differentiation and integration. By definition, these operations require an element domain to be specified first. Other FE-related math, such as the discretization of custom field variables and the calculation of variational forms, should also be compacted in a way such that the resulting expressions can be easily used to identify the algebraic element equations. Listing 2 demonstrates the proposed solution for that case along with the derivation of an 8-node hexahedral 3D solid element.

#### Listing 2: Descriptive class definition for an 8-node hexahedral 3D solid element.

```python
#0 class Solid(fw.Descriptive):
    # Connectivity.
#1    nodes = fw.Conn(of=Node, by=fw.ID) @ 8
#2    X, Y, Z = TriLinear @ [nodes.X, nodes.Y, nodes.Z]
#3    D = fw.Diff(X, Y, Z)
#4    u, v, w = TriLinear @ [nodes.U, nodes.V, nodes.W]
#5    du = D @ [u, v, w]
#6    e = (du.T + du)/2
#7    lam, nu = fw.Attribute(2)
#8    s = lam*fw.trace(du)*fw.eye(3) + 2*nu*e
#9    dk = fw.trace(e.T @ s)
#10   K = fw.Integ(dk, D(Q))
```

---

**Figure 5.** Definition of the trilinear SFV using the proposed decorator-pattern.
The Conn decorator (line #1) lies in the heart of the presented abstraction. This object is specifically programmed to serve as a discretization definer, as can be seen in lines #2 and #4, which identifies the custom nodal data to be used in the calculations of both custom domains, given in Eq. (1), and custom field functions (Fields) of the form

\[ u(X(r)) = q^T \cdot \psi(r). \]  

The “@” operator (conventionally used for matrix multiplication in Python) is used to identify the corresponding SFV in the initialization of the symbolic Field variables, which makes the code shorter in comparison with the usual object initialization syntax. As a secondary task, Conn serves as a nodal-data provider. For example, a script in the form of “Solid[0].nodes.U” is programmed to retrieve the corresponding nodal values in an array form, which can be very useful for postprocessing purposes. The overall design of the Conn class is depicted in Figure 6.

![Figure 6. Overall design of the Conn (connectivity) class and example use cases to define interpolated field and nodal data of the element.](image)

The Diff decorator (line #3) is another wrapper designed specifically to help with mathematical abstraction. By inspecting the provided coordinate fields, it automatically calculates the Jacobian matrix according to Eq. (2), which makes it a good utility to use as a gradient operator when applied to multiple Fields (line #5). Diff is also commissioned to work for the integration operator when provided with the relevant quadrature data (line #10). This is because the decorator is also able to calculate the Jacobian determinant.

It is evident that for these expressions to represent the FE formulation of the subject element, all Fields should possess a strong symbolic math functionality as well as an instance-wise numerical evaluation mechanism. Figure 7 depicts a proof of concept implementation that utilized the automatic-code-generation technique as the proposed solution in that regard. It is necessary to specify that we use sympy to determine the custom return values of the “at” methods in the autogenerated classes.

It is important to note that from the perspective of the developer, access to such an automated evaluation
mechanism can be very helpful in transforming the framework into his/her custom FE application. As the next two sections reveal, we extend the evaluation support to all field types, including the field derivatives and variational forms, in order to make this argument even stronger.

2.4. Evaluation of the field derivatives with respect to the element domain

The derivative of Eq. (8) with respect to the \( k \)th coordinate of the domain can be calculated using the chain rule (summation convention applies):

\[
\frac{\partial u}{\partial X_k} = q_u^T \frac{\partial \psi_i}{\partial r_j} \frac{\partial r_j}{\partial X_k} = q_u^T \cdot [\partial \psi_r] \cdot J^{-1}_{column=k}.
\]

Eq. (9) can be written in a similar way to Eq. (8):

\[
\begin{align*}
\frac{\partial u}{\partial X_1} &= q_u^T \cdot \Psi^{100}, \quad \frac{\partial u}{\partial X_2} = q_u^T \cdot \Psi^{010}, \quad \frac{\partial u}{\partial X_3} = q_u^T \cdot \Psi^{001},
\end{align*}
\]

where \( \Psi^{\text{rotation}(k)} = [\partial \psi_r] \cdot J^{-1}_{column=k} \). Although expressions quickly grow for higher-order derivatives, it is important to note that one should preserve the separation between the nodal values, \( q_u \), and the \( \Psi \) vector.

\[\text{Figure 7. Conversion of Fields from descriptive statements into FE evaluators.}\]
along with his/her calculations, as given with the form in Eq. (11).

$$\frac{\partial^{n+m+p}u}{\partial X_1^m \partial X_2^n \partial X_3^p} = q_n^{mp}$$

Figure 8 depicts a brief demonstration of the framework implementation of Eq. (11).

**Figure 8.** Automatic evaluation of the derivatives of the field functions and the separation between nodal values (atNodes) and corresponding bases (BasesAt).

### 2.5. Automatic calculation of the variational forms

An element is a selected subdomain in which the original governing equation of the problem is converted to an algebraic equation system, which is generally represented as

$$k^e(q^e) = p^e,$$  \hspace{1cm} (12)

where $q^e$ is the vector that constitutes the nodal values of all primary variables of an individual element, and $p^e$ represents the natural boundary conditions in their discrete forms. To keep things simple, we consider the linear case of Eq. (12) with the following form:

$$K^e q^e = p^e, \quad K^e = \delta(k^e),$$  \hspace{1cm} (13)

where $K^e$ is called the stiffness matrix. For the construction of Eq. (12), the governing differential equation is first turned into a weighed integral form, and then a weak form is obtained by employing integration by parts as
an application of the divergence theorem. While performing this procedure, one needs to keep track of the row positions of the individual equations constructed by individual weights. The weak formulation is closely related with the variational formulation because the required weights are the variations of the field functions [27]. The following is a practical evaluation of these variations with the introduction of the variation vector (VV), \( \tilde{\Psi} \):

\[
\delta u^{nmp} = \tilde{\Psi}_u^{nmp} \\
\delta v^{nmp} = \tilde{\Psi}_v^{nmp} \\
dk_{INT}^{e} \\
\begin{bmatrix}
\Psi_1^{nmp} \\
0 \\
\Psi_2^{nmp} \\
0 \\
\Psi_3^{nmp} \\
0 \\
\Psi_4^{nmp} \\
0
\end{bmatrix} + \begin{bmatrix}
0 \\
\Psi_1^{nmp} \\
0 \\
\Psi_2^{nmp} \\
0 \\
\Psi_3^{nmp} \\
0 \\
\Psi_4^{nmp}
\end{bmatrix} = \begin{bmatrix}
\Psi_1^{nmp} f \\
\Psi_1^{nmp} g \\
\Psi_2^{nmp} f \\
\Psi_2^{nmp} g \\
\Psi_3^{nmp} f \\
\Psi_3^{nmp} g \\
\Psi_4^{nmp} f \\
\Psi_4^{nmp} g
\end{bmatrix}
\]

Eq. (14) demonstrates the use of the VV in the formation of the internal element vector through a typical 4-node element with two unknown fields. These fields are given by \( u^{nmp} = q_u^T \cdot \Psi_u^{nmp} \) and \( v^{nmp} = q_v^T \cdot \Psi_v^{nmp} \). Here, \( f \) and \( g \) are two arbitrary functions such that Eq. (14) symbolizes the usual weak form operations and \( \delta \) is the variation symbol. It should be noted that the vectors \( \tilde{\Psi}_u^{nmp} \) and \( \tilde{\Psi}_v^{nmp} \) differ from \( \Psi^{nmp} \) in the way that their terms are placed; thus, the weak-form operations automatically render their correct positions in the calculation of \( dk_{INT}^{e} \), which then should be integrated using quadrature rules to obtain \( k_e \). The stiffness matrix, \( K^e \), can be constructed by taking the variation of the weak form as

\[
dK^e = f_{,u}^{nmp} \tilde{\Psi}_u^{nmp} \cdot (\tilde{\Psi}_u^{nmp})^T + f_{,v}^{nmp} \tilde{\Psi}_v^{nmp} \cdot (\tilde{\Psi}_u^{nmp})^T + g_{,u}^{nmp} \tilde{\Psi}_u^{nmp} \cdot (\tilde{\Psi}_v^{nmp})^T + g_{,v}^{nmp} \tilde{\Psi}_v^{nmp} \cdot (\tilde{\Psi}_v^{nmp})^T.
\]

Note that it is quite straightforward to evaluate \( K^e \) without considering the tensor indexes, but only involving the vector operations. In the proposed framework, a call in the form of “Field_...” (see lines #9 and #10 of Listing 2) is suggested to return a FieldVariation object, which wraps all the necessary operations to automatically evaluate Eqs. (14) and (15), as illustrated in Figure 9.

3. Pre/postprocessing and analysis along with example problems
3.1. Solution of the system equations

An FE framework would be useless without the ability to help the developer assemble element matrices and solve the resulting equations. Because both subjects are well studied in the literature, we discuss only the proposed syntax and the solution of an example problem. Figure 10 depicts the details for pre/postprocessing and the analysis of the selected 3D body. Readers should refer to the next example for more details regarding custom graphical display options.
3.2. Poisson’s equation

Figure 11 depicts a triangular domain that consists of a 9-node biquadratic quadrilateral (Quad9) elements. We use it for the solution of the well-known Poisson problem in the twisting of elastic rods, given as follows.

\[-\Delta u = 2 \quad \text{in} \quad \Omega\]
\[u = 0 \quad \text{on} \quad \Gamma\]  \hspace{1cm} (16)

The proper statement of the weak form can be written as

\[
\int_{\Omega} \nabla u \cdot \nabla w \, d\Omega = \int_{\Omega} 2w \, d\Omega,
\]  \hspace{1cm} (17)

where \(w = \delta u\). In Eq. (17), the integrals on the left and right sides are called the bilinear and linear forms, respectively. The element equations can be constructed in the form of
In the twisting of elastic rods, the torsional moment of inertia \( I \) and the shear stress \( (\tau_{\text{max}}) \), along with its components, can be calculated as follows:

\[
I = \int_{\Omega} 2 u \, d\Omega, \quad \tau_x = \mu \beta \frac{\partial u}{\partial Y}, \quad \tau_y = -\mu \beta \frac{\partial u}{\partial Y}, \quad \tau_{\text{max}} = \sqrt{\tau_x^2 + \tau_y^2},
\]

where \( \beta \) and \( \mu \) represent the angle of twist per unit rod length and the shear modulus, respectively. For simplicity, we take \( \mu \beta = 1 \). The complete listing for the derivation, solution, and pre/postprocessing (except...
The complete listing for the solution of the Poisson equation:

```plaintext
# E: Young's modulus p: Poisson's ratio.
Solid[:,:].set(E=100000, p=0.15)

# Lamé parameters.
Solid[:].lam = lambda s: s.p*s.E/((1+s.p)*(1-2*s.p))
Solid[:].nu = lambda s: s.E/(2*(1+s.p))

# Boundary Conditions.
Node[:].lambda X: X == 0 or X==6].fix(U=0, V=0, W=0)
Node[:].lambda X: X == 3].load(V=30)

# Solution |K| u-p.
fw.solve(fw.assemble(Solid.K) @ Node.fix == Node.load)

# Custom Drawing.
draw = Drawer(Solid, coord=['x', 'y', 'z'])
draw[:,:].SX()
draw[lambda SX: abs(mean(SX))>3000].SX()
```

**Figure 11.** The complete listing for the solution of the Poisson equation.

**Figure 12.** Contour plots from Figure 11: (a) solution field u, (b) $\tau_{x}$, (c) $\tau_{\text{max}} = 0.215$.

...mesh-reader definition) of the example is given in Figure 11 and Figure 12. It is clear from the presented examples that, for the solution of partial differential equations with the FE method, the proposed framework offers a sufficiently short and self-expressive syntax while providing a few good customization facilitations.

### 4. Conclusions

In this study, a novel descriptive programming approach for FE development is discussed in detail. Instead of offering a super-compact FE framework, we aimed at implementing specific tasks to improve the maneuverability of the users in their custom designs. By wrapping these tasks in a symbolic DSML, we attempted to provide a sufficiently short and self-expressive code base. In addition, reinforcing the proposed symbols with natural numerical evaluation mechanisms led to the development of a concise framework that can intuitively be used in the conversion of variational forms of theory into matrix equations. The framework was demonstrated using two linear examples.

Clearly, this framework does not constitute a final and complete solution package for all FE-related theories. Its approaches should be reconsidered frequently for possible improvements with respect to both...
theoretical and implementation-specific issues. However, developers should have the freedom to adapt their own functionality for it to be considered a framework; otherwise, the resulting structure would be an application. For this reason, we spent tremendous effort to provide natural evaluation routines for the symbolic descriptors and not simply provide an application to solve a particular group of differential equations. Although implemented in Python, the main ideas presented here are mostly applicable to other programming languages, with potential compromises for syntax. It is important to note that by using our framework, developers can easily create their own custom objects. These objects may include shape functions, quadrature rules, mesh-readers, graphical displays, and theoretical elements, which can be used to build their own mini-FE applications. In conclusion, leveraging the presented descriptive approach can change the way custom FE software is built and used.

Although the approach can be generalized equally well for nonlinear analysis, its demonstration will be presented in future studies. The proposed approach is also well suited for parallel and distributed computation purposes because we use descriptions rather than direct implementations for the code. Moreover, certain speed improvements can also be relatively easily implemented in a single CPU by using caching techniques and automatic element-wise vector/matrix operations in the background framework; these tasks will be a part of future work.

References


