Proceedings of the Seventh International Conference on
Parallel, Distributed, GPU and Cloud Computing for Engineering
Edited by: P. Iványi, F. Magoulès and B.H.V. Topping
Civil-Comp Conferences, Volume 4, Paper 3.4
Civil-Comp Press, Edinburgh, United Kingdom, 2023
doi: 10.4203/ccc.4.3.4
©Civil-Comp Ltd, Edinburgh, UK, 2023

A Python Interface for Symbolic and Vectorized Computation of Finite Element Matrices

M. Yilmaz

Department of Civil Engineering
Istanbul Technical University
Istanbul, Turkey

Abstract

This paper presents a simple framework for rapid development and fast calculation of Finite Element (FE) matrices in vectorized forms using various symbolic utilities written in Python. For the end-user code, three goals are being pursued: the code should be easily perceptible, explicitly include the necessary FE formulations, and run reasonably fast. For these purposes, an Object Oriented (OO) architecture with direct vectorized processing has been proposed. Unlike classical OO programming, the management of Node and Element data is carried out through single objects. In this way, the number of method-calls, which is one of the weakest points of the Python language in terms of speed, has been greatly reduced.

Keywords: finite elements, object oriented programming, python, stiffness matrix, symbolic, vectorized, descriptors

1 Introduction

Finite Element (FE) programming has received a lot of attention and is used in many commercial and non-commercial software/frameworks providing different levels of customizability for FE researchers/developers. In [1, 2], end-user is assisted with an interpreter language to customize the functionality of the framework. Domain-specific modelling (DSM) [3] also offered an alternative way of customizability by introducing the code generation from the user scripts, successfully implemented in [4

and 5]. Many frameworks, including the aforementioned, either support custom uses of pre-programmed theoretical elements, or offer compact formulations by handling the problem at the differential equation level [6, 7]. In both cases, developing/adding extra theory beyond the existing functionality offered in the framework turns into a difficult or even impossible effort for the end-users.

At [8 and 9] the author presented a Python FE framework that allows the end-user to develop his/her own custom code by combining only some specific helper objects, namely Descriptors. In this study, a similar pattern for the descriptors will be provided with the ability to work with vectorized data, so that the end user's need for a fast and customizable framework will be tried to be met at the same time.

An outline of the study is described as follows. Section 2 discusses the overall design of the proposed utilities with an example Frame2D element implementation. In Section 3, the initialization of the elements is discussed and run-time performance of the framework is investigated. The paper concludes with a discussion of the presented approach in Section 4.

2 Methods

Listing 1. demonstrates the implementation of the well-known 2D frame finite element with symbolic utilities programmed in Python programming language [10].

Listing 1: An example implementation of 2D Frame Element.

```
00: from FRAMEWORK import *
:
01: @SHAPE_FUNCTION_1D(points=lambda p1, p2: [p1, p1, p2, p2],
:                      derivatives=          [0,  1,  0,  1 ])
02: def cubic_hermite(x): return [1, x, x**2, x**3]
:
03: class FrameNode(VECTORIZED):
04:     id = LABEL()
05:     X, Y = COORD(2)
06:     u, v, t = DOF(3)
07:     fix = FIX()         # Dirichlet Boundary Conditions.
08:     load = LOAD()       # Neumann Boundary Contions.
:
09: class FrameElement(VECTORIZED):
10:     id = LABEL()
11:     conn = CONNECTIVITY(2)
12:     n1, n2 = conn
:
13:     Lx = n2.X - n1.X
14:     Ly = n2.Y - n1.Y
15:     L = (Lx ** 2 + Ly ** 2) ** 0.5
```

```
:
16:   E, b, h = FLOAT(3)              # Young's modulus, cross section width and height.
17:   Area, Iz = b*h, b*h**3/12     # Cross section properties
18:   EA, EI = E * Area, E * Iz
:
19:   x, sfv = cubic_hermite(0, L)  # Shape Function Vector (Hermite polynomials).
20:   K = EI*INTEGRATE(sfv.diff(x, 2) * sfv.T.diff(x, 2), (x, 0, L))  # Beam Stiffness Matrix.
:
21:   KL = MATRIX([[EA/L,   0,       0,       -EA/L,  0,       0      ],
:                  [0,      K[0, 0], K[0, 1],  0,     K[0, 2], K[0, 3]],
:                  [0,      K[1, 0], K[1, 1],  0,     K[1, 2], K[1, 3]],
:                  [-EA/L,  0,       0,        EA/L,  0,       0      ],
:                  [0,      K[2, 0], K[2, 1],  0,     K[2, 2], K[2, 3]],
:                  [0,      K[3, 0], K[3, 1],  0,     K[3, 2], K[3, 3]]])
:
22:   c, s = Lx/L, Ly/L
:
23:   T = MATRIX([[ c, s, 0, 0, 0, 0],
:                 [-s, c, 0, 0, 0, 0],
:                 [ 0, 0, 1, 0, 0, 0],
:                 [ 0, 0, 0, c, s, 0],
:                 [ 0, 0, 0,-s, c, 0],
:                 [ 0, 0, 0, 0, 0, 1]])  # Stiffness Transformation Matrix.
:
24:   sm = STIFFNESS_MATRIX(matrix=T.T @ KL @ T,
:                           dofs=[n1.u, n1.v, n1.t, n2.u, n2.v, n2.t])
:
```

As it can be understood from the code, all FE aspects are structured as independent class-level objects. In Python programming terminology, these structures are called *Descriptors*. In short, descriptors are externally programmed objects that can receive event calls from and change the behaviour of their owner classes (the class in which they are initialized) [10].

The magic behind the code lies in the design of the proposed descriptors. Figure 1. demonstrates a simplified interface for the numerical evaluation of the *symbolic descriptors*.

As Figure 1. suggests, a symbolic framework, namely "*sympy*" [11] is used to be able to perform math operations in symbolic forms. For the Descriptor base-class, *sympy* Symbol is extended to behave like a descriptor by introducing the __get__ method of the descriptor protocol [10]. Inheriting from the base Descriptor; *SYMBOLS* are intended to be responsible for hosting the relevant data, while *EXPRESSIONS* are intended to be responsible for generating functions that match their expressions and executing these functions using the corresponding symbols as their data sources. As can be seen from the code, the "lambdify" function is able to compile expressions in alternative ways. In this study, two modules, namely, "numpy"

[12] and "numexpr" [13] are utilized for different compilations of the “func” function. A performance comparison between the two modules can be found in the Results Section.

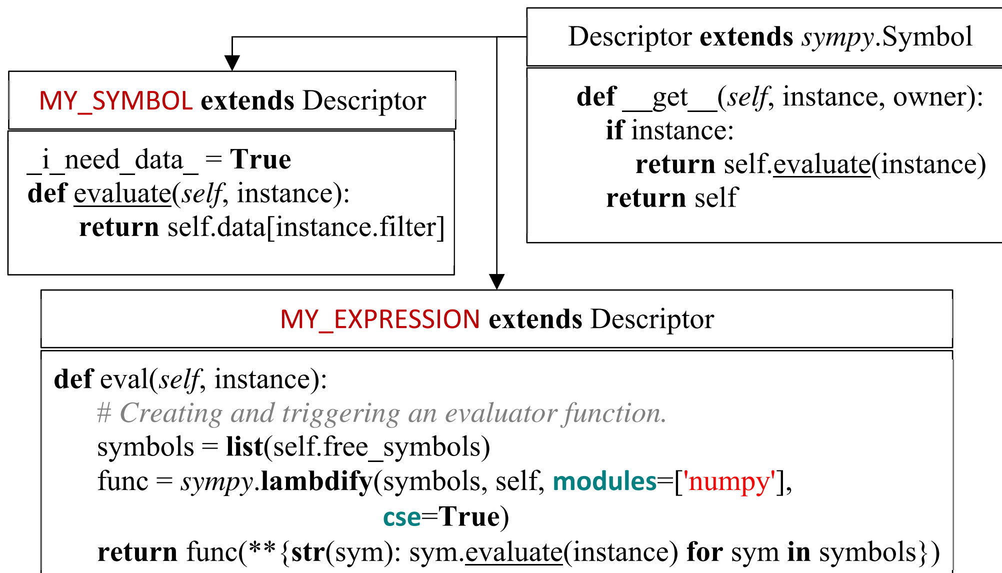

Figure 1: Essential parts of a symbolic descriptor interface.

The descriptors need to be fed with data to be functional. To address this issue, the VECTORIZED class (see Listing 1) is introduced to both identify the descriptors and transmit data to them, as shown in Figure 2.

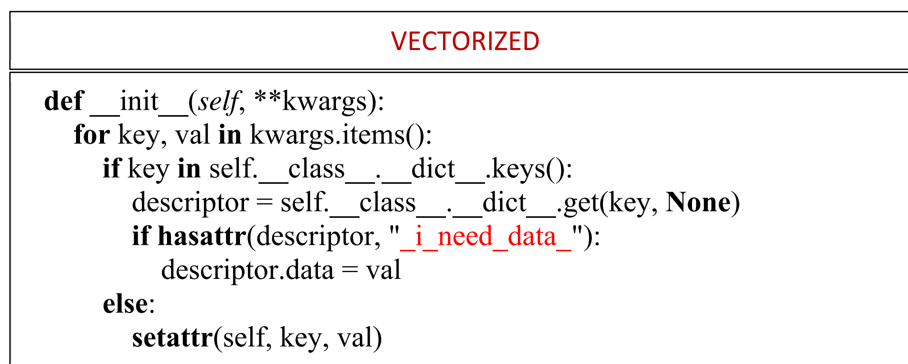

Figure 2: Constructor of the VECTORIZED class: Descriptor identification and data transmission to descriptors.

In a nutshell, the code assigns the data of the keyworded arguments supplied to the constructor to matching descriptors with the same parameter keyword (see Listing 2. of the Results Section for an example use case).

It is also worth noting that the VECTORIZED class does more than merely transport data. It also performs other functions that will not be explained here as they are not relevant for the purposes of this paper.

3 Results

Listing 2. demonstrates the initialization and analyses of a frame structure with 3xN elements. Since only the runtime performance of element stiffness matrix formations will be tested, N is used as a multiplication factor to increase the matrix dimensions of the structure.

Listing 2: An example run of the Frame 2D system for N*3 = 3 million elements.

```
:    …
:
25:  N=1_000_000   # Factor for increasing the number of elements.
:
26:  nodes = FrameNode(id=[0, 1, 2, 3], X=[0, 6, 0, 6], Y=[0, 0, 3, 3])
:
27:  elements = FrameElement(id=list(range(3*N)),
:                            E=[28e6/N]*(3*N),
:                            b=[0.3]*(3*N),
:                            h=[0.6]*(3*N),
:                            conn=CONNECTIVITY.of(nodes=nodes, data=[[0, 2],
:                                                                   [2, 3],
:                                                                   [1, 3]]*N))
28:  nodes[0:2].fix(u=0, v=0, t=0)
29:  nodes[2].load(u=1000)
:
30:  U, P, ok = SOLVE(elements.sm @ nodes.fix == nodes.load)
```

Table 1. shows the run-time performance of the example in Listing 2. for different number of elements. In the table, the Stiffness Matrix Calculations include all mathematical operations. The Sparse Matrix Assemblage, on the other hand, involves creating a Coordinate Format Matrix known as the COO form.

When the results are evaluated, it is clear that a good performance for an average computer configuration has been attained. It is worth noting that, even with 375000 elements, the identical computations take around 84.59 and 10.23 seconds, respectively, in a Python program built in traditional Python with non-vectorized individual method calls.

It should also be mentioned that 16 GB of RAM was insufficient for more than 18 million elements. Vectorized processes must be separated into chunks to handle this issue, which will be the subject of future studies.

Number of Elements x [Million]	Stiffness Matrix Calculations (numexpr [13] / numpy [12]) (10 runs average) [sec]	Sparse Matrix Assemblage (10 runs average) [sec]
0.375	0.12 / 0.21	0.39
0.75	0.26 / 0.44	0.76
1.5	0.44 / 0.84	1.54
3	0.93 / 1.68	3.08
6	1.46 / 3.39	6.56
9	2.33 / 6.27	10.29
12	4.05 / 11.07	16.36
15	8.57 / 15.75	26.95
18	10.95 / 18.73	35.35

Table 1: Average wall-times for Stiffness Matrix Operations and Sparse Matrix Formation (Laptop: ACER INC. Model PREDATOR HELIOS 300, Intel Core i7 10750H @ 2.60 GHz, 6 Cores, 12 Logic Processors, 16 GB Memory (Single) DDR4-3200, Operating System: Windows 11 Pro 64-bit: 22H2: OS Build 22621.1555).

4 Conclusions and Contributions

This paper presented a brief overview of a customizable Python framework design that may be utilized for quick programming and computation of basic FE analyses. It should be underlined that this framework was created with minimal effort, thanks to the flexibility and rich ecosystem of the Python programming language.

References

[1] F.T. McKenna, "Object-oriented finite element programming: frameworks for analysis, algorithms and parallel computing", Ph.D. Thesis, Department of Civil Engineering, University of California, Berkeley, 1997.

[2] P. Dadvand, R. Rossi, E. Oñate, "An object-oriented environment for developing finite element codes for multi-disciplinary applications", Archives of Computational Methods in Engineering, 17, 253-97, 2010.

[3] S. Kelly, J.P. Tolvanen, "Domain-specific modeling", Wiley-IEEE Computer Society Press, 2008.

[4] A. Logg. "Automating the finite element method", Archives of Computational Methods in Engineering, 14, 93-138, 2007.

[5] A. Logg, G.N. Wells, "DOLFIN: automated finite element computing", ACM Transactions on Mathematical Software, 37(2), 20, 2010.

[6] F. Hecht, "New development in FreeFem++", Journal of Numerical Mathematics, 20, 251-65, 2012.

[7] R. Cimrman, "SfePy—write your own FE application", In: de Buyl P, Varoquaux N,editors. "Proceedings of the 6th European conference on Python in science (EuroSciPy) ". p. 65-70, 2013.

[8] M. Yilmaz, "Rapid translation of finite-element theory into computer implementation based on a descriptive object-oriented programming approach", Turkish Journal of Electrical Engineering and Computer Sciences, 26, 3367-82, 2018.

[9] M. Yilmaz, "Easy pre/post-processing of finite elements with custom symbolic-objects: A self-expressive Python interface", Computers & Structures, 222, 82-97, 2019.

[10] Python Software Foundation. Python Language Reference, version 3.10.6. Available at <http://www.python.org> and <https://docs.python.org/3/>.

[11] SymPy, a Python library for symbolic mathematics, version 1.11.1. Available at <https://www.sympy.org>

[12] NumPy, the fundamental package for scientific computing with Python, version 1.23.4. Available at <https://www.numpy.org/>.

[13] Numexpr, Fast numerical expression evaluator for NumPy, version 2.8.4. Available at <https://pypi.org/project/numexpr/>.