Open∇FOAM®
Journal

Volume 2 [Full Papers], Pages 166–182
ISSN: 2753-8168

A GENERAL APPROACH FOR RUNNING PYTHON CODES IN OPENFOAM USING AN EMBEDDED PYBIND11 PYTHON INTERPRETER

S. RODRIGUEZ[1,*] AND P. CARDIFF[1]

[1]School of Mechanical and Materials Engineering, University College Dublin, Ireland.
Email address: `simon.rodriguezluzardo@ucdconnect.ie`

DOI: 10.51560/ofj.v2.79
Version(s): OpenFOAM® v2012, OpenFOAM® 9 and foam-extend 4.1
Repo: https://github.com/solids4foam/pybind11foam.git

Abstract. As the overlap between traditional computational mechanics and machine learning grows, there is an increasing demand for straightforward approaches to interface Python-based procedures with C++-based OpenFOAM. This article introduces one such general methodology, allowing the execution of Python code directly within an OpenFOAM solver without requiring Python code translation. The proposed approach is based on the lightweight library pybind11, where OpenFOAM data is transferred to an embedded Python interpreter for manipulation, and results are returned as needed. Following a review of related approaches, the article describes the proposed method, focusing on data transfer between Python and OpenFOAM, executing Python scripts and functions, and practical details about the implementation in OpenFOAM. Three complementary test cases are presented to demonstrate the approach: a Python-based velocity profile boundary condition, a Python-based solver for prototyping, and a machine learning mechanical constitutive law class for solids4foam, which performs field calculations.

1. Introduction

Appearing in 1985, C++ was motivated by the limitations of the structured programming paradigm: code reusability, code sharing, and code extensibility [1]. Built on code abstraction and pioneering object-oriented programming (OOP), C++ quickly became the number one choice for creating large, complex, and extendable software. In fact, before C++, OOP was mostly unknown in industry as its techniques were believed to be excessively expensive for real-world applications and too complex for non-expert programmers [2]. OpenFOAM was one of the first prominent computational mechanics software to embrace C++ and OOP, tracing its origin to its predecessor 'FOAM' with its roots in the late 80s. Using OOP techniques, the authors of FOAM represented computational entities using mathematical language without sacrificing computational efficiency [3].

Over the subsequent three decades, there has been an explosion in the availability of computational resources, mainly via parallel computing and C++ software to exploit it. However, as acknowledged by Bjarne Stroustrup when creating C++, code elegance was not its primary goal: *"Even I knew how to design a prettier language than C++"* [2]. The aim for C++ was utility, not abstract beauty. In contrast, publicly released in 1991, Python aimed to emphasise readability and usability through simpler syntax and fewer rules. As a result, when compared to C++, Python's learning curve is less steep and requires considerably less effort for prototyping code and performing auxiliary tasks such as handling files; however, these benefits are typically achieved at a higher computational cost [4].

Over time, Python has become one of the most widely used programming languages, with the 'Popularity of Programming Languages Index' [5] placing it as the most popular programming language in the world. This momentum has resonated with the OpenFOAM community, which has tried to merge the advantages of Python with OpenFOAM, as demonstrated by projects such as:

- PyFoam: A widely used library for manipulating OpenFOAM cases and controlling OpenFOAM runs [6].
- PythonFlu: A Python-based OpenFOAM wrapper which aims to make the calculation environment more interactive and automated [7]; however, in recent years, this project has become inactive.

[*] Corresponding author
Received: 9 March 2022, Accepted: 6 December 2022, Published: 15 December 2022

- fluidfoam: Python classes for plotting OpenFOAM data [8];
- Owls: Python tools for data analysis and plotting of OpenFOAM cases [9].
- swak4Foam: A collection of parsers for OpenFOAM-types, designed to minimise the use of C++ in OpenFOAM [10], including a function object that can execute Python code using an integrated Python interpreter.
- *Like OpenFOAM, but Python*: Anderluh and Jasak [11] proposed to re-write the core OpenFOAM functionality entirely in Python.

In general, these projects have used Python to provide OpenFOAM functionality in a "pythonic" way via wrappers or by extending OpenFOAM pre/post-processing utilities. Projects like PyFoam, fluidfoam and Owls all interact with the text-based files within an OpenFOAM case, whereas swak4Foam extends the functionality of OpenFOAM solvers through additional function objects and boundary conditions. In contrast, PythonFlu and *Like OpenFOAM, but Python* are aimed at OpenFOAM developers, allowing them to create OpenFOAM solvers directly in Python; the fundamental difference between these two projects is that PythonFlu uses the underlying OpenFOAM C++ libraries, whereas *Like OpenFOAM, but Python* re-writes them in Python.

In addition to these projects, Maulik et al. [12] and Weiner et al. [13, 14] have aimed to provide capabilities to use data analysis tools from Python within OpenFOAM; the former constructed OpenFOAM applications that have bindings to data libraries in Python, and the latter transfers data and neural network models from Python to OpenFOAM via the PyTorch library.

In contrast to the previous projects, this article proposes a general, minimally invasive approach for extending OpenFOAM capabilities with Python support, inspired by the philosophy of the swak4Foam "pythonIntegration" function object. This is achieved by integrating an embedded Python interpreter in OpenFOAM using the pybind11 library, where special attention is given to the use of machine learning techniques.

In particular, the approach is motivated by two opportunities:

(1) The ability to rapidly prototype numerical algorithms using Python while taking advantage of the OpenFOAM framework;
(2) The capacity to exploit the vast ecosystem of scientific programming tools that have been developed for Python, particularly those in machine learning and data science.

Related to the second point, the recent machine learning revolution has led to the developing area of hybrid machine-learning-computational-mechanics procedures, including numerous research avenues: machine learning-acceleration of solvers [15, 16]; machine learning-constitutive laws in solid mechanics [17–19]; machine learning-turbulence models in fluid dynamics [20]; and targeting both acceleration and accuracy gains [21]. In these approaches, the first challenge involves the development of a machine learning procedure - typically based on a neural network - capable of solving the required task; this is commonly done with Python or similar languages such as R, Matlab and Julia. The next immediate challenge is coupling the machine learning model with a computational mechanics solver, whether OpenFOAM or another; currently, there is no standard way of doing this in OpenFOAM. Possible approaches include:

(1) Using the Tensorflow [22] or PyTorch [14] C++ APIs, where the machine learning model is exported to an intermediate file format, which can be read in C++. In TensorFlow, the frozen graph may be exported/imported as a *Protocol Buffer* (protobuf) file, or more recently using the *SavedModel* functionality. In PyTorch, the intermediate format is called a *TorchScript*.
(2) Using the Python C API combined with the NumPy package C API [12];
(3) Translating the Python-based neural network to C++ code via specialised conversion tools, such as frugally-deep [23].

These solutions require the Python code and models to be translated or converted to allow their use with C++ in OpenFOAM. The primary conceptual disadvantage with this is that it conflicts with the main motivation of many programmers who want to use Python: coding simplicity; for example, this is particularly evident when attempting to compile and link the TensorFlow C++ API with OpenFOAM, where both must be built from source with consistent settings, even though this challenge can be mitigated using containerisation, e.g. Docker, Singularity. These challenges can limit the adoption of Python as a development tool for applications beyond traditional machine learning for which Python might be well-suited; individual components of OpenFOAM solvers that are particularly suitable are field calculations, boundary conditions, function objects, and pre/post-processing utilities.

Motivated by this, the current article presents a general approach to executing Python code from within OpenFOAM without converting the Python code to C++. This approach uses an embedded Python interpreter based on the lightweight library pybind11 [24]. The approach is similar to the approaches adopted in swak4Foam [10] and by Maulik et al. [12], and enables the use of Python classes/functions/libraries without resorting to their C APIs.

The remainder of this article is structured as follows: Section 2 presents an introduction to the required pybind11 support, including a general description of pybind11 and how to include it in OpenFOAM; this is followed by a description of the interaction between OpenFOAM and Python. In Section 3, three test cases illustrate how the proposed approach can be used in OpenFOAM. The first case demonstrates implementing a general velocity profile boundary condition via Python. The second case shows how the Python interpreter can be used to quickly test "proof of concept" Python numerical implementations via an OpenFOAM solver. The final case gives an example of how to perform field calculations via analytical calculations in Python as well as via TensorFlow/Keras machine learning models in Python; in this case, an elastic mechanical constitutive law for a solid mechanics simulation is chosen for demonstration, but the approach is equally applicable to any field calculation. Section 4 summarises the article's main findings and learning points and briefly discusses future directions.

2. Approach and implementation

This section briefly describes pybind11, followed by steps on how to use it with OpenFOAM. After explaining the high-level interaction between OpenFOAM and Python via pybind11, the core technical steps are outlined:

- Initialising the Python interpreter;
- Transferring data between OpenFOAM and the Python interpreter;
- Executing Python code from OpenFOAM.

2.1. The pybind11 library.

pybind11 is a lightweight header-only library primarily used to allow C++ code to be called from Python [24]; however, as used in this article, pybind11 also offers the possibility of embedding a Python interpreter in a C++ application.

Installing pybind11 with Conda Conda [25] is an open-source package and environment management system that can manage Python environments. Using Conda, pybind11 can be installed in a *nix terminal with:

```
$> conda install -c conda-forge pybind11
```

Including pybind11 in OpenFOAM Once pybind11 is installed on the system, it can be used in an OpenFOAM application or library:

- In the shell, declare two environment variables, `PYBIND11_INC_DIR` and `PYBIND11_LIB_DIR`, to store the location of the required pybind11 header files and the Python libraries:

```
export PYBIND11_INC_DIR=$(python3 -m pybind11 --includes)
export PYBIND11_LIB_DIR=$(python3 -c 'from distutils import sysconfig;
print(sysconfig.get_config_var("LIBDIR"))')
```

Where the shell commands here use the `python3` command to look up the location of the include and library directories; alternatively, the user can set these directories manually.

Then, in the `Make/options` file:

- Include `PYBIND11_INC_DIR` in the `EXE_INC` field, for example:

```
EXE_INC = \
    $(PYBIND11_INC_DIR)
```

- Include `PYBIND11_LIB_DIR` and the Python C dynamic library in the `EXE_LIBS` field (or in the `LIB_LIBS` field if compiling a library), for example:

```
EXE_LIBS = \
    -L$(PYBIND11_LIB_DIR) \
    -lpython3.8
```

Python version 3.8 is used in this case, but the programmer should change this to whichever version they use. Further details about pybind11 can be found online in the pybind11 manual [26].

Including the required header files The header files that will be required in the OpenFOAM/Python communication depend on the specific interaction scheme used by the programmer; however, in general, the following header files should be included in the OpenFOAM source code:

```
#include <pybind11/embed.h>
#include <pybind11/eval.h>
```

These two header files provide the following functionality:

- `embed.h`: Allows the Python interpreter to be embedded in a C++ program;
- `eval.h`: Allows Python code to be executed/evaluated from C++ in the embedded Python interpreter.

Assuming pybind11 was installed correctly, these header files should all be available within the `PYBIND11_INC_DIR` directly defined in the `Make/options` file.

Note on `setRootCase.H` OpenFOAM applications typically include a header file named `setRootCase.H` which reads and checks command-line arguments; however, the default arguments which are used when constructing the `Foam::argList` object within `setRootCase.H` can cause issues with certain Python modules, such as TensorFlow, as noted in online forums [27]. This can be solved by modifying the default `setRootCase.H` file in the following way:

```
// Foam::argList args(argc, argv); // previous line
Foam::argList args(argc, argv, true, true, false);
if (!args.checkRootCase())
{
    Foam::FatalError.exit();
}
```

2.2. **General workflow for using pybind11 with OpenFOAM.** The central idea revolves around using pybind11 to create a Python interpreter that exists alongside the OpenFOAM application. Then, pybind11 allows data to be transferred to and from the Python interpreter and the ability to execute Python functions and scripts within OpenFOAM.

The entire process is conceptually depicted in Figure 1. OpenFOAM execution begins as usual, with common tasks involving creating and initialising fields and other auxiliary data structures (Step 1). At a convenient point, a Python interpreter is created and held in the background (Step 2). The OpenFOAM execution continues, and when required, it sends data to the Python interpreter (Step 3). An OpenFOAM command is then called to execute a piece of Python code (Step 4) and invokes the Python interpreter (Step 5). While the Python interpreter executes, the OpenFOAM code waits. Finally, the data is sent back to OpenFOAM, where it can be used in the rest of the OpenFOAM workflow (Step 6). Details of the different steps shown in Figure 1 are given in the subsequent sub-sections.

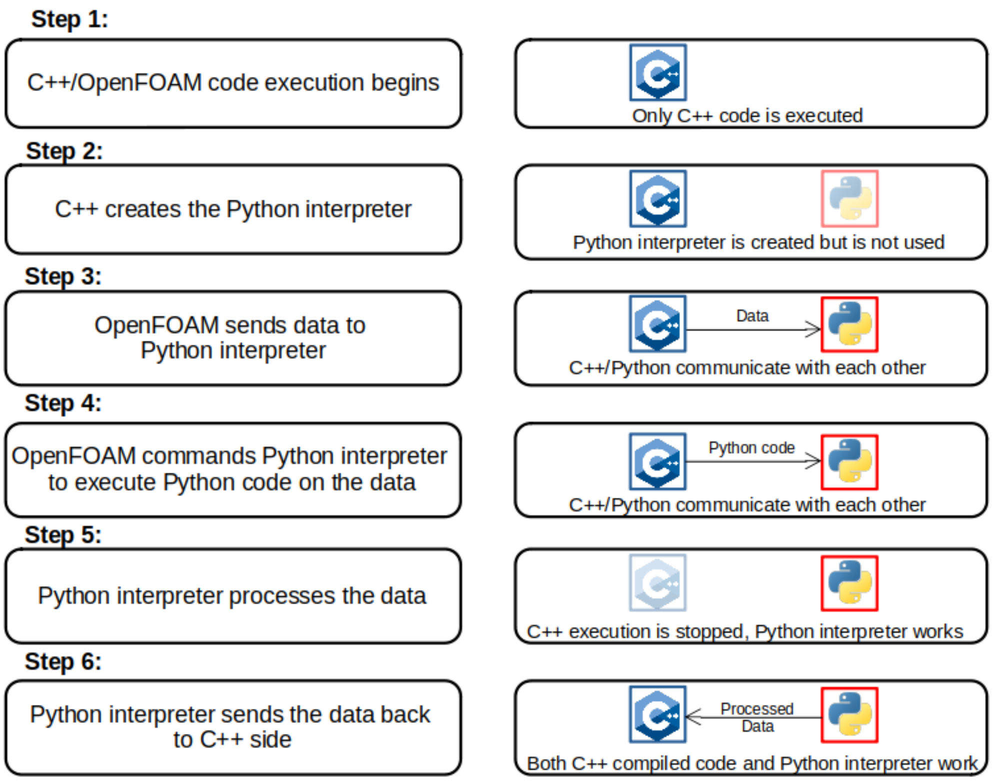

Step 1:
C++/OpenFOAM code execution begins
Only C++ code is executed
Step 2:
C++ creates the Python interpreter
Python interpreter is created but is not used
Step 3:
OpenFOAM sends data to Python interpreter
Data
C++/Python communicate with each other
Step 4:
OpenFOAM commands Python interpreter to execute Python code on the data
Python code
C++/Python communicate with each other
Step 5:
Python interpreter processes the data
C++ execution is stopped, Python interpreter works
Step 6:
Python interpreter sends the data back to C++ side
Processed Data
Both C++ compiled code and Python interpreter work

FIGURE 1. Overview of the interaction between OpenFOAM and Python, via pybind11.

2.3. **Initialising the Python interpreter.** To improve readability, it is assumed that all subsequent OpenFOAM code has invoked the C++ namespace `pybind11` as shown:

```
namespace py = pybind11;
```

To initialise, embed and expose the Python interpreter within OpenFOAM, the following line is used:

```
py::initialize_interpreter();
```

This can be considered a higher-level version of the `Py_Initialise()` method from the Python C API, and its location in the code must ensure that it is run only once (or once per process when running in parallel). After the interpreter is created, it can be interacted with at any time. The line can be placed anywhere convenient within the OpenFOAM code, for example, in the `createFields.H` header file of a solver or in the constructor of a class.

2.4. **Transferring data between OpenFOAM and Python.** To facilitate data transfer between OpenFOAM and Python, a `scope` object must be created and stored in OpenFOAM:

```
py::object scope = py::module_::import("__main__").attr("__dict__");
```

A dissection of this line requires analysing both `__main__` and `__dict__` in a Python context. According to the official Python documentation [28], `__main__` is the name of the environment where top-level code is run. It is top-level because it imports all other modules that the program needs. `__dict__`, on the other hand, is a dictionary or other mapping object used to store an object's (writable) attributes [29]. Therefore, this line imports the dictionary that contains all the variables defined in the Python interpreter and assigns it to the object `scope` on the C++ side, which can be used to declare variables in the Python interpreter from C++ or to retrieve data in the opposite direction. For example, the OpenFOAM scalar variable `'a'` can be copied to a new variable in the Python scope `'x'` as follows:

```
const scalar a = 2;
scope["x"] = a;
```

For simplicity, the previous command left it to Python to infer the variable type at run-time via Python's dynamic typing. However, the resulting Python data type can be explicitly specified, as well.

Similarly, the `scope` object can be used to copy variables from the Python scope to OpenFOAM:

```
const scalar b = scope["y"].cast<scalar>();
```

It is assumed here that a scalar variable `'y'` exists in the Python scope before executing this line. The command declares a `scalar` variable named `'b'` and assigns the value of the Python variable `'y'` to it. At compilation time OpenFOAM does not know whether the result from `scope["y"]` will be a `scalar`. For that reason, the command invokes the pybind11's `py::cast()` method [24] which takes the output from `scope["y"]` and converts it to a `scalar`, providing their data types are compatible. In general, `py::cast()` performs conversions between C++ and Python by internally copying the data from one variable to the other and, as stated in the pybind11 documentation, *"all major Python types are available as thin C++ wrapper classes"* [30], including `object`, `bool`, `int`, `float`, `str`, `bytes`, `tuple`, and `list`.

To pass an OpenFOAM `List/Field` to Python we take advantage of three functionalities:

- The pybind11's `py::cast()` method, previously presented;
- `ctypes`, which *"provides C compatible data types"* [31];
- The built-in NumPy support for `ctypes`.

The first step is to retrieve the low-level pointer to the underlying data of the OpenFOAM `List/Field`, using the non-const `data()` function and the const `cdata()` functions of the `UList` base class. This address is stored as a `long int` and sent to the Python interpreter via pybind11 along with the dimensions of the array ([number of points, number of dimensions of the field]). For example:

```
// Create an OpenFOAM Field (and optionally populate it)
symmTensorField myField(100, symmTensor::zero);
// Access the pointer to the underlying data
symmTensor* myFieldPointer = myField.data();
// Store the address to the data
const long myFieldAddress = reinterpret_cast<long>(myFieldPointer);
// Pass the field address to Python
scope_["myFieldAddress"] =  myFieldAddress;
// Pass the field size to Python
scope_["myFieldSize"] = myField.size();
```

In the Python interpreter, the address is converted to a Python-compatible C-pointer using the function `ctypes.cast`. Finally, this pointer and the size of the array are used to create the NumPy array using the function `numpy.ctypeslib.as_array`. For example:

```
import numpy
import ctypes

data_pointer = ctypes.cast(myFieldAddress, ctypes.POINTER(ctypes.c_double))
my_data = numpy.ctypeslib.as_array(data_pointer, shape = (myFieldSize, 6))
```

where `ctypes` is the imported ctypes module (i.e., `import ctypes`), `myFieldAddress` is the integer address from C++/OpenFOAM and `myFieldSize` is the size/length of the array. Since the arrays created on the Python side are directly manipulating the underlying data of the OpenFOAM fields, all the modifications done on the Python side are immediately propagated to the OpenFOAM side without copying the data; this is in contrast to the approach of Maulik et al. [12] where data is copied. In other words, Step 6 in Figure 1 is implicit. Care must be taken when using this approach as memory errors may occur if the address changes on the C++ side without informing Python.

Note that `py::cast` also supports conversions between C++ `std::vector` and NumPy arrays. For example, it is possible to create a `py::array` object, which can be directly transferred to the Python interpreter as a NumPy array, starting from a previously declared and initialised OpenFOAM/C++ `std::vector<double>`. The following example demonstrates this behaviour:

```
#include <pybind11/stl.h>
#include <pybind11/numpy.h>

std::vector<scalar> x{10, 20, 30};
py::array pyX = py::cast(x);
scope_["x"] = pyX;
```

where the `std::vector` is copied to the C++ `py::array`. The same conversions are allowed when the arrays are multi-dimensional, for example, 2-D arrays; the only difference in that case is that the `x` variable must be declared as a vector of vectors, i.e. `std::vector<std::vector<type>>`. However, care should be taken when using the `py::cast` approach as data is copied to Python, which may be computationally inefficient.

Based on pybind11's support for transferring arrays, as introduced in the previous paragraph, it is possible to conceive an alternative approach to exchange data between OpenFOAM and Python where the information from an OpenFOAM `Field` is extracted to an intermediate `std::vector` which is then copied via `py::cast` to a `py::array` that is sent to the Python interpreter. In general, this *pass-by-copy* approach should be discouraged, given the memory and computing overhead related to copying the data and saving the additional copies.

Therefore, it is not used for fields/lists in this article but is mentioned here for completeness.

2.5. **Executing Python code from OpenFOAM.**

The command `py::eval_file(string, scope)` where `string` is the name of a Python file/script, loads all the entities defined in the Python file, which can then be consumed using the pybind11 `py::exec` command. For example, assume the following trivial Python function exists in the loaded Python script (`python_code.py`):

```
def double_value(number):
    return 2.0*number
```

This function receives a number as an argument and returns the number × 2. The following C++ code demonstrates how to use this Python function:

```
py::eval_file("python_code.py", scope);

// Define scalar 'a' in OpenFOAM
const scalar a = 2.0;
// Transfer OpenFOAM variable 'a' to a variable 'x' in the Python interpreter
scope["x"] = a;
// Execute the Python function 'double_value'
py::exec("y = double_value(x)", scope);
// Transfer the value of the Python variable 'x' to the OpenFOAM scalar 'b'
```

```
const scalar b = scope["y"].cast<double>();
```

- Line 4 declares a `scalar` variable `a` and initialises its value to 2.0;
- Line 6 declares a variable `x` in the Python interpreter and assigns it the value of the OpenFOAM variable `a`;
- Line 8 calls the Python function `double_value` with the Python variable `x` as an argument, and assigns the result to the Python variable `y`;
- Finally, line 10 copies the value of the Python `y` variable to a new OpenFOAM variable `b`.

Table 1 shows the variables in the OpenFOAM/C++ and Python scopes and how their values change as each line of code is executed in sequence.

Executed OpenFOAM/C++ code	**OpenFOAM/C++ variables**	**Python variables**
const scalar a = 2.0;	a = 2.0	
scope["x"] = a;	a = 2.0	x = 2.0
py::exec("y = double_value(x)", scope)	a = 2.0	x = 2.0, y = 4.0
const scalar b = scope["y"];	a = 2.0, b = 4.0	x = 2.0, y = 4.0

Table 1. Evolution of the variables values as each line in the C++ code is executed.

As stated in the pybind11 documentation [32], when an exception occurs on the Python side, pybind11 raises an exception of type `pybind11::error_already_set`, which is propagated back to the C++ side and contains a C++ string textual summary and the actual Python exception.

3. Test Cases

This section presents three complementary test cases that demonstrate the methodology proposed in Section 2:

(1) **Python velocity profile boundary condition**: An OpenFOAM wrapper boundary condition is presented, which uses a Python script - supplied at run-time - to define a temporally-spatially varying velocity profile;
(2) **Python heat transfer prototyping solver**: A wrapper OpenFOAM heat transfer solver is presented, which invokes a run-time Python script to solve the governing equations at each time step; this case shows how new solvers could be more quickly prototyped using a combination of OpenFOAM and Python;
(3) **Field calculations using Python** A wrapper solids4foam [33] mechanical constitutive law is presented to demonstrate how a `volSymmTensorField` stress field can be calculated using Python as a function of a `volSymmTensorField` strain field. Analytical expressions are compared with a TensorFlow Keras [34] neural network. The approach is expected to be easily adapted to other field calculations.

These cases, the code to run them, and the files to set up a suitable conda environment are provided with the paper.

3.1. Python velocity profile boundary condition.

This is an introductory case that demonstrates how the general methodology presented in Section 2 can be used to produce a Python-based boundary condition in OpenFOAM. In this case, a velocity profile boundary condition is created, where a Python script calculates the patch velocities as a function of spatial coordinates and time. The boundary condition, named `pythonVelocity`, is a Dirichlet condition and hence derives from the `fixedValueFvPatchField` boundary condition class. The central concepts of the `pythonVelocity` boundary condition - related to the private data and `updateCoeffs` function - are described here.

Boundary condition private data Two private data objects are stored in the boundary condition class:

- The name of the Python script (`fileName pythonScript_`) which is read at run-time;
- The Python interpreter scope object (`py::object scope_`) is used for transferring data from OpenFOAM to Python and evaluating Python functions.

Boundary condition `updateCoeffs` function For each finite volume boundary condition in OpenFOAM (those derived from `fvPatchField`), the `updateCoeffs` function is called as a preliminary step to the linear solver call. Consequently, the `updateCoeffs` function is an appropriate location to update patch velocities as a function of spatial position and time. As per Section 2, three essential operations need to be included in the custom boundary condition:

- Within the boundary condition constructor, initialise the Python interpreter and load the Python file;
- Within `updateCoeffs`, transfer the face-centre coordinates and time value to Python;
- Within `updateCoeffs`, use Python to calculate the new face velocities.

In the boundary condition constructor, the Python interpreter is initialised, and the Python script is read:

```
// Initialise the Python interpreter
py::initialize_interpreter();
scope_ = py::module_::import("__main__").attr("__dict__");

// Evaluate the Python script to import modules
py::eval_file(pythonScript_, scope_);
```

Next, in the `updateCoeffs` function, the patch face-centre coordinate vectors and current time are transferred to the Python scope:

```
// Take a const reference to the face—centre position vectors
const vectorField& C = patch().Cf();

// Get a pointer to the C array
const Foam::Vector<scalar>* CData = C.cdata();

// Get a pointer to the velocities array
Foam::Vector<scalar>* velocitiesData = this;

// Pass the C array pointer address to Python
const long CAddress = reinterpret_cast<long>(CData);
scope_["CAddress"] = CAddress;

// Pass the velocities array pointer address to Python
const long velocitiesAddress = reinterpret_cast<long>(velocitiesData);
scope_["velocitiesAddress"] = velocitiesAddress;

// Pass the size of C and velocities fields to Python
scope_["SIZE"] = C.size();

// Pass the time to Python
scope_["time"] = db().time().value();
```

The run-time Python script is then called to calculate the new velocities as a function of the face-centre coordinate vectors (`face_centres`) and the current time (`time`):

```
py::exec("calculate()", scope_);
```

Note that the only requirement of the run-time loadable Python script is that it contains the definition of a function called `calculate` that:

- Creates two NumPy arrays: `face-centres` and `velocities`, respectively, from the addresses of their original OpenFOAM counterparts;
- Sets the values of the velocities field.

A suitable example Python function, included in the accompanying `setInletVelocity.py` file, is:

```
import numpy as np
import ctypes as C

def calculate():
   data_pointer_face_centres = C.cast(CAddress, C.POINTER(C.c_double))
   face_centres = np.ctypeslib.as_array(data_pointer_face_centres, shape = (SIZE, 3))
   data_pointer_velocities = C.cast(velocitiesAddress, C.POINTER(C.c_double))
   velocities = np.ctypeslib.as_array(data_pointer_velocities, shape = (SIZE, 3))

   if face_centres.shape[0] != 0:
      # Initialise result
      result = np.zeros(shape = face_centres.shape)

      # Calculate values using the x coordinates and time
      x = face_centres[:, 0]

      # Update the velocity field
      velocities[:, 0] = np.sin(time * np.pi) * np.sin(x * 40 * np.pi)
```

The expression, in this case, varies the X component of velocity as a function of the X component of the face-centre coordinate vector (x) and time (t), according to:

$$u(x,t) = \sin(\pi t)\sin(40\pi x) \tag{1}$$

Finally, the boundary condition is enforced on the patch:

```
fixedValueFvPatchVectorField::updateCoeffs();
```

Verification of the proposed boundary condition The proposed `pythonVelocity` boundary condition is verified on the classic `cavity` tutorial case [35], where it is used to set the velocity on the upper moving wall patch. The only modifications made to the tutorial were to change the velocity boundary condition for the moving wall patch in the initial time (in `0/U`):

```
movingWall
{
    type            pythonVelocity;
    pythonScript    "$FOAM_CASE/setInletVelocity.py";
    value           uniform (0 0 0);
}
```

where the `setInletVelocity.py` script is placed in the case root directory.

Running the case produces the velocity profile on the `movingWall` patch at 0.5 s as shown in Figure 2, where the expected analytical expression is given for comparison. The corresponding velocity streamlines and pressure distribution are shown in Figure 3.

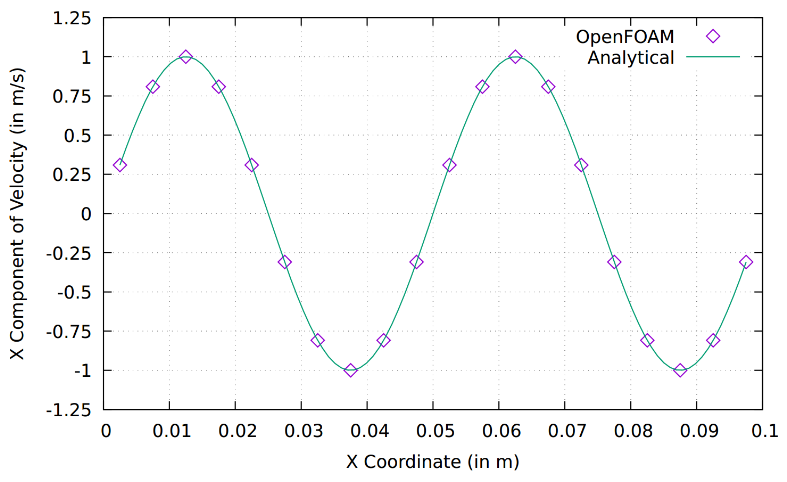


FIGURE 2. Velocity along the top boundary at 0.5 s

3.2. **Python heat transfer prototyping solver.** This case demonstrates one of Python's primary advantages over C++: fast prototyping. To do this, as an example, the classic `laplacianFoam` solver is modified such that a run-time selectable Python script is passed mesh, material and time information and is expected to calculate the temperature field. In this case, a simple explicit finite difference method is implemented in Python; however, the approach could be modified to accommodate the prototyping of any particular solution strategy or component.

There are two main components added to the OpenFOAM code:

- Data transfer from OpenFOAM to Python;
- Execution of the Python code;

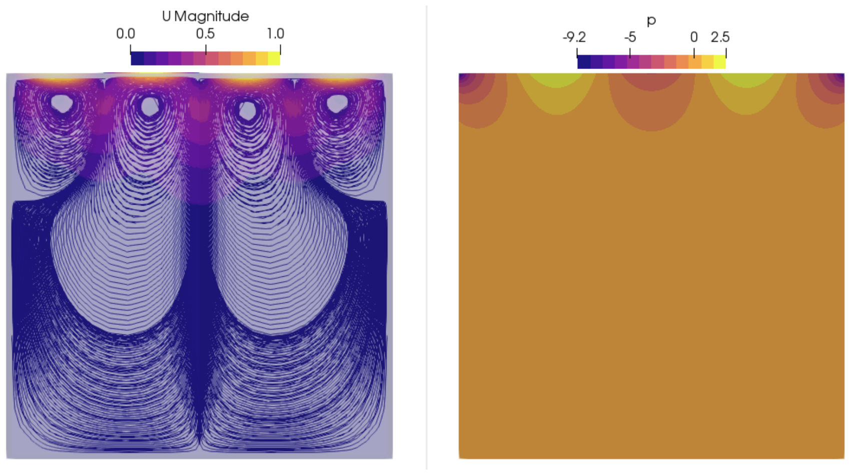


FIGURE 3. Velocity streamlines in m s^{-1} (left) and pressure field in m^2 s^{-2} (right) for the cavity case, where the velocity on the upper wall is prescribed via a Python script.

Data transfer from OpenFOAM to Python Following the approach in Section 2, OpenFOAM data is mapped, where relevant, to a corresponding Python NumPy array data. In this case, two pieces of data are passed to the Python function:

- A parameter `gamma`, which encompasses diffusivity, mesh spacing and pseudo-time-step data:

```
scope["gamma"] =
(
    DT*runTime.deltaT()*Foam::pow(max(mesh.deltaCoeffs()), 2)
).value();
```

- The temperature field, including boundary condition values, which are assumed here to be `fixedValue`-type (Dirichlet) conditions:

```
// Set boundary cell values to be equal to the boundary values as we
// will use a finite difference method in the Python script
// These boundary cells will be the boundary nodes in the finite
// difference code
scalarField& TI = T.primitiveFieldRef();
forAll(T.boundaryField(), patchI)
{
    const labelUList& faceCells =
        mesh.boundary()[patchI].faceCells();
    forAll(faceCells, faceI)
    {
        const label cellID = faceCells[faceI];
        TI[cellID] = T.boundaryField()[patchI][faceI];
    }
}

// Get a pointer to the T array
scalar* TData = TI.data();

// Pass the array pointer address to Python
const long TAddress = reinterpret_cast<long>(TData);
scope["T_address"] = TAddress;

// Pass the size of the T field to Python
scope["SIZE"] = TI.size();
```

Perform Python calculations As in the first test case, the Python script is selected at run-time and is expected to contain a `calculate` function. This is called in the OpenFOAM solver as:

```
py::exec("calculate()", scope);
```

The Python function, in this case, employs a simple steady-state explicit finite difference approach to solve the Laplace equation, implemented using NumPy built-in capabilities [36] :

```python
import numpy as np
import ctypes as C

def calculate():

    data_pointer_T = C.cast(T_address, C.POINTER(C.c_double))
    T = np.ctypeslib.as_array(data_pointer_T, shape=(SIZE, 1))

    # Get the number of cells in x and y directions
    Nx = np.sqrt(SIZE).astype(int)
    Ny = Nx

    # Reshape T to a 2-D array
    T2d = np.reshape(T, (Nx, Ny)).copy()
    newT2d = T2d.copy()

    # Use the explicit finite difference method to update the non-boundary cells
    newT2d[1:-1, 1:-1] = (gamma*(T2d[2:, 1:-1] + T2d[:-2, 1:-1] + T2d[1:-1, 2:]
                              + T2d[1:-1, :-2] - 4*T2d[1:-1, 1:-1])
                          + T2d[1:-1, 1:-1])

    T[:, :] = np.reshape(newT2d, (SIZE, 1))
```

Note that under the current design, this function assumes the mesh comprises a single structured square mesh with an equal number of cells in the x and y directions, where the cell numbers increase row by row from one side to the other. Modifications to the numbering scheme, such as applying the OpenFOAM `renumberMesh` utility, would require the Python script to be modified.

Results The proposed prototyping solver is demonstrated on a steady-state heat conduction problem with a 2-D square geometry (0.1×0.1 m) and 20×20 cells - the geometry and mesh from the `cavity` tutorial. The diffusivity is 4×10^{-5} m^2 s^{-1}. The left, bottom, and right boundaries are set to 273 K. In contrast, the top boundary is set to 373 K. With our modified implementation of `laplacianFoam` called `pythonLaplacianFoam`, we chose to specify the Python script in the system/controlDict as:

```
pythonScript    "$FOAM_CASE/calculateT.py";
```

where `calculateT.py` is the Python explicit finite difference code shown above. Here, the python script was specified in the controlDict for convenience; however, it could be given in any dictionary or even as a command-line argument.

Figure 4(left) shows the final converged temperature field. For comparison, the temperatures along a vertical line from the centre of the bottom patch to the centre of the top patch are compared with the results as calculated by `laplacianFoam` (Figure 4(right)). As expected, the predictions are similar, with the differences primarily attributed to the Python finite difference code enforcing boundary conditions in the centres of cells adjacent to the boundaries.

3.3. **Field calculations using Python.** The purpose of this final test case is to demonstrate how to perform general field calculations using the embedded Python interpreter; specifically, this case shows how this can be done using machine learning models implemented in TensorFlow/Keras, scikit-learn, and Python in general. A solid mechanics problem is chosen, where the stress tensor in each cell is calculated as a function of the displacement gradient via a run-time selectable constitutive mechanical law. This approach could be readily applied to similar field calculations, such as the calculation of turbulent quantities, thermo-physical property fields, or even discrete differential operators. The case (Figure 5) consists of a plate, with a circular hole in the centre, loaded by uniform tension $t_x = 1$ MPa on the right boundary, while symmetry conditions are applied to the left and bottom edges.

Three quadrilateral meshes are considered:

- Coarse: 100 000 cells;
- Medium: 400 000 cells;
- Fine: 1 600 000 cells.

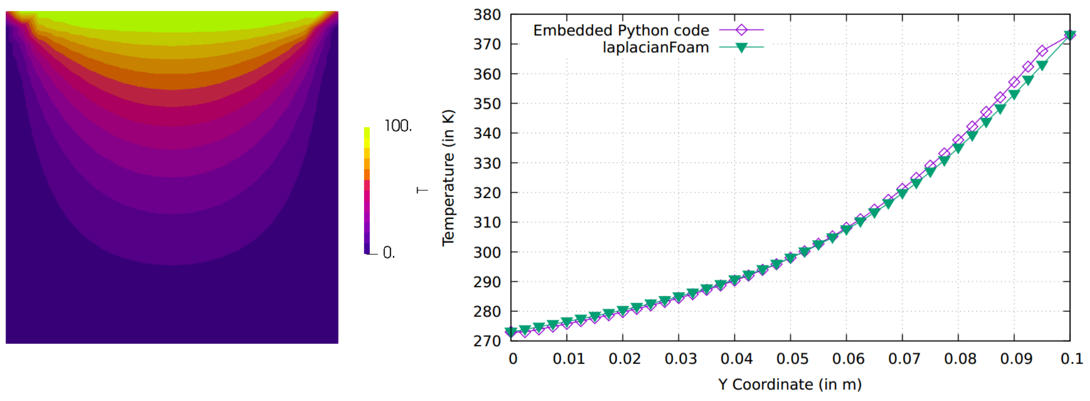


FIGURE 4. Steady-state temperature distribution as calculated by a Python finite difference code embedded in an OpenFOAM solver (left) and temperature profile along a vertical line from the centre of the bottom patch to the centre of the top patch, compared with the predictions from `laplacianFoam` (right).

All the simulations were solved for one time step, small strains were assumed, and the inertia and gravity terms were neglected.

In an incompressible Newtonian fluid, the mechanical resistance to deformation is defined solely by viscosity. In contrast, a linear elastic solid is defined in terms of two independent parameters: resistance to shearing (shear modulus, μ) and resistance to volume change (bulk modulus, κ). Several other pairs of independent parameters can also be used, which are defined through simple expressions in terms of the shear and bulk moduli [37]; more commonly, linear elastic behaviour is reported in terms of Youngs modulus (resistance to uni-axial deformation) and Poissons ratio (ratio of axial-to-lateral contraction during uni-axial deformation). However, Hookes law [38] is more concisely expressed mathematically in terms of Lames first parameter μ (synonymous with the shear modulus) and Lame's second parameter λ. The λ parameter is related to the bulk modulus but does not have a direct physical interpretation. In the current test case, the parameters of a typical steel are assumed: Youngs modulus $E = 200$ GPa and Poissons ratio $\nu = 0.3$, corresponding to $\lambda = 115.4$ GPa and $\mu = 76.9$ GPa.

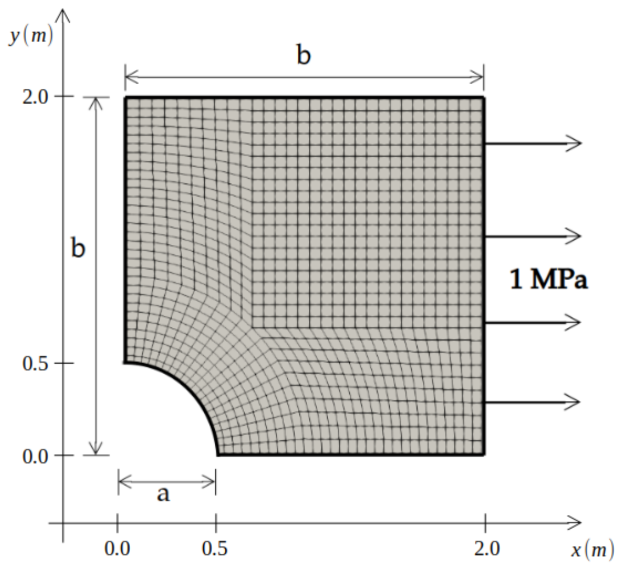


FIGURE 5. Geometry of the spatial computational domain for the flat plate with a circular hole (a = 0.5 m, b = 2 m, $E = 200$ GPa, $\nu = 0.3$, $t_x = 1$MPa).

At each outer iteration within a time-step, the stress tensor $\boldsymbol{\sigma}$ in each cell is calculated as a function of the strain tensor $\boldsymbol{\epsilon}$, according to Hooke's law (presented here in several equivalent forms)

$$\begin{aligned}\boldsymbol{\sigma} &= 2\mu\boldsymbol{\epsilon} + \lambda \operatorname{tr}(\boldsymbol{\epsilon})\,\mathbf{I}\\ &= 2\mu \operatorname{dev}[\boldsymbol{\epsilon}] + \kappa \operatorname{tr}(\boldsymbol{\epsilon})\,\mathbf{I}\\ &= \frac{E}{1+\nu}\boldsymbol{\epsilon} + \frac{E\nu}{(1+\nu)(1-2\nu)}\operatorname{tr}(\boldsymbol{\epsilon})\,\mathbf{I}\end{aligned} \tag{2}$$

where dev[·] is the deviatoric operator, and the strain tensor $\boldsymbol{\epsilon}$ is the symmetric component of the displacement gradient $\boldsymbol{\nabla d}$:

$$\boldsymbol{\epsilon} = \frac{1}{2}\left[\boldsymbol{\nabla d} + (\boldsymbol{\nabla d})^T\right] \tag{3}$$

For demonstration purposes, the calculation of the stress tensor in each cell (Equation 2) is performed using three alternative approaches:

(1) **`solids4foam` reference case**: this simulation uses the native C++ `solids4foam`/OpenFOAM implementation and does *not* use Python.
(2) **Analytical expression in Python** (Analytical-Python): the strain tensor is passed to the Python interpreter; the stress is then calculated in Python as per Equation (2) and returned to OpenFOAM; in this case, no machine learning methods are used.
(3) **Neural network base case** (NN-Base): as per approach 2, the strain is passed to Python, but in this case, the stress is called using an artificial neural network created in Keras, which has been trained on data from Equation (2). The prediction method uses the built-in `.predict` Keras function.

The constitutive mechanical law is specified in the case file `constant/mechanicalProperties` and loaded at run-time. Approaches (2) and (3) are implemented in a Python script and are called by OpenFOAM via the user-defined library `libpythonLinearElastic`, which calls the necessary pybind11 APIs and the Python script.

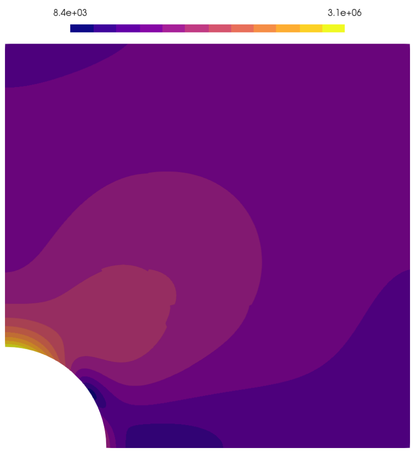
Performance comparison In terms of accuracy, the predictions from approaches (2) and (3) closely agree with the expected behaviour given by approach (1). Figure 6(a) shows the von Mises stress distribution for the reference `solids4foam` approach on the medium mesh case, while the predicted xx component of the stress tensor $\boldsymbol{\sigma}_{xx}$ along the vertical line $x = 0$ is shown for all three approaches in Figure 6(b). Table 2 lists the L_∞ norm and average L_2 norm of the differences with the reference case for the displacement magnitude and von Mises stress fields; the maximum errors in the von Mises stress ($<$ 2.65 kPa) and displacement magnitude ($<$ 8 nm) are small in comparison to the maximum von Mises stress (3.12 MPa) and maximum displacement magnitudes (6.02 μm) in the reference solution.

Table 3 shows the execution times for the different approaches. The analytical Python method (approach 2) exhibits essentially the same execution time as the reference solids4foam case, whereas the neural network approach introduced a small overhead of less than 4%. This additional time was expected because the neural network involves more floating point operations than the analytical expression.

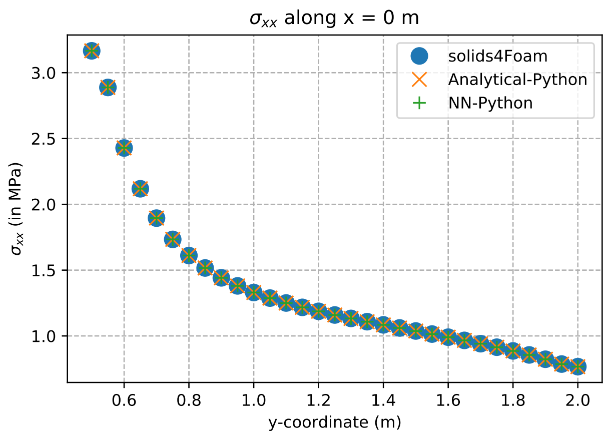
	Von Mises Stress (in Pa)		**Displacement (in m)**	
	L_2	L_∞	L_2	L_∞
solids4foam	-	-	-	-
Analytical-Python	11.40	1380	1.14e-10	3.82e-9
NN-Base	21.93	2620	2.38e-10	7.73e-9

Table 2. L_∞ and mean L_2 norm differences for the medium resolution mesh between the predictions from approaches 2-3 and the reference approach, where the maximum von Mises stress in the reference case is 3.12 MPa, and the maximum displacement magnitude is 6.02 μm.

Running in parallel Test cases 1 and 3 in this paper can also be run in parallel without any changes via the standard OpenFOAM MPI approach. When using pybind11 in parallel, each processor (process) creates an independent copy of the Python interpreter. Figure 7 shows the performance in parallel on the



(a) Von Mises stress distribution for the solids4foam reference case (in Pa).



(b) σ_{xx} stress from the top of the hole to the top boundary.

FIGURE 6. Stress predictions for the hole-in-a-plate case

	Coarse mesh (100 000 cells)	**Medium mesh (400 000 cells)**	**Fine mesh (1 600 000 cells)**
solids4foam	72	491	3,663
Analytical-Python	73	497	3,670
NN-Base	90	517	3,805

TABLE 3. Execution times (in s) for the three approaches and the three mesh resolutions.

medium mesh (400 000 cells) for both the pure OpenFOAM and mixed OpenFOAM/Python approaches, where the simple domain decomposition was used with the same number of subdivisions in the x and y directions; the overhead introduced by the combined Python/OpenFOAM code is almost negligible.

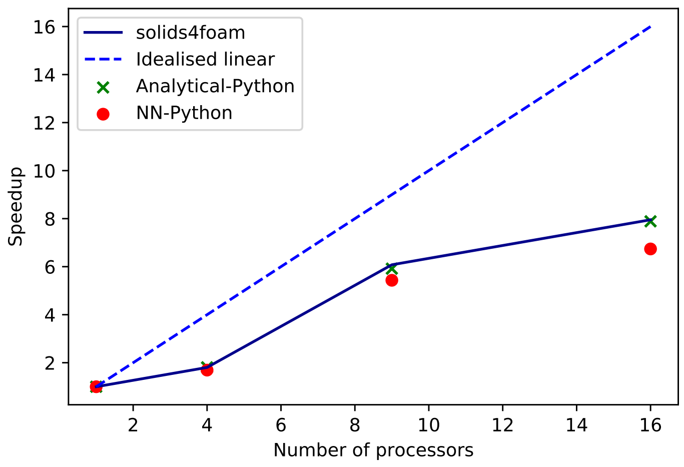


FIGURE 7. Parallel performance of the three approaches on the *plate with a hole* case with the medium mesh (400,000 cells).

Although not shown in the current article, if inter-process communication was required at the Python level, this could be achieved using Python MPI libraries, for example, as shown by Maulik et al. [12].

4. The PythonPal.

Finally, this section introduces `pythonPal4foam`, or simply `pythonPal`, a header-only library that provides high-level methods for the approach presented in this article, removing the need to know how to combine `ctypes`, NumPy and OpenFOAM to communicate with Python. The `pythonPal.H` file defines the header-only `pythonPal` class with three fundamental methods:

- `pythonPal (const fileName& pythonScript, const bool& debug = true)`. It is the class constructor. The python filename's path (`pythonScript`) is a required argument. An optional debug switch can be given to control how much information is printed to the standard output;
- `void passToPython (List<T>& myList, const std::string& fieldNameInPython) const`. This is a public member function that creates a NumPy array in the Python interpreter with the name `fieldNameInPython` which references the data from the OpenFOAM `List/Field myList`. After this, any transformation of the `fieldNameInPython` or `myList` will be propagated to the other side, as explained in Section 2. An important remark here is that if an OpenFOAM `GeometricField` is passed to `passToPython`, it will only transfer the internal field; if required, the boundary patches must be independently transferred;
- `void execute(const word& command) const`. This is a public member function that executes the Python command given by `command`.

To demonstrate its use, we create a modified version of `icoFoam` called `pythonPalIcoFoam`, where we use `pythonPal` to calculate the specific kinetic energy, $k = \frac{U^2}{2}$, via Python.
Provided that `pybind11` is installed, the steps to go from `icoFoam` to `pythonPalIcoFoam` are:

- Include `pythonPal.H` at the top of `pythonPalIcoFoam.C`;
- In `createFields.H`, construct a `volScalarField` called k, initialised to zero;
- In `pythonPalIcoFoam.C`, instantiate `pythonPal` and load the Python file `python_script.py`:

```
pythonPal myPythonPal("python_script.py", true);
```

The `python_script.py`, which should be available in the case, defines a function `calculatek`.

```
def calculatek(field):
    return np.sum(field * field, axis = 1)[:, np.newaxis] / 2
```

- At the end of the time-loop, pass the U and k fields to Python via `pythonPal`:

```
myPythonPal.passToPython(U, "U");
myPythonPal.passToPython(k, "k");
```

- Calculate k via `pythonPal`:

```
myPythonPal.execute("k[:, :] = calculatek(U)");
```

- Finally, write the k field to disk, e.g. for viewing in ParaView.

```
k.write();
```

Figure 8 shows the cavity case's velocity magnitude and specific kinetic energy field results.

5. Conclusion

This article introduces a general approach for running Python code in OpenFOAM. This is achieved with the pybind11 header library, which is used to create an instance of the Python interpreter and for exchanging data between OpenFOAM and Python. The three examples demonstrate the feasibility and efficiency of the presented approach, where the use of machine learning models is shown in the final case.
The key findings of the article:

- It is straight-forward to implement functionality such as boundary conditions, solution algorithms and material models in Python via the pybind11 interpreter approach;
- The approach is computationally and numerically equivalent to the native C++ OpenFOAM code, with an overhead of less than a few percent for the feed-forward neural network with 266 weights, on a mesh with 400 000 cells.

The paper also introduced `pythonPal`. This high-level header-only library allows the presented pybind11 functionality to be used simply, eliminating the need to know the details of how the Python/OpenFOAM communication is performed.

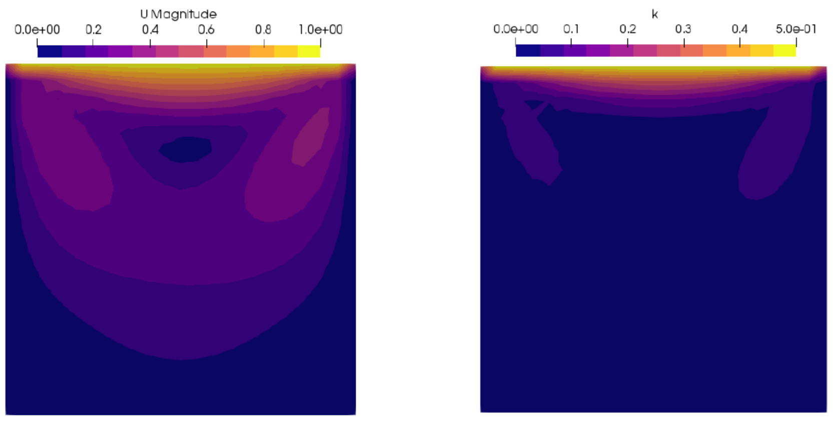


FIGURE 8. pythonPalIcoFoam results. Velocity magnitude field (left) and specific kinetic energy field (right).

Hopefully, the methods presented in this article will expand the universe of Python-based solutions applicable to OpenFOAM, especially those coming from the rapidly evolving machine learning field. Future work will explore more complex problems related to elastoplasticity, requiring machine learning models such as recurrent neural networks.

Acknowledgements

Financial support is gratefully acknowledged from the Irish Research Council through the Laureate programme, grant number IRCLA/2017/45.

Additionally, the authors want to acknowledge project affiliates, Bekaert, through the Bekaert University Technology Centre (UTC) at UCD (www.ucd.ie/bekaert), and I-Form, funded by Science Foundation Ireland (SFI) Grant Number 16/RC/3872, co-funded under European Regional Development Fund and by I-Form industry partners.

The authors wish to acknowledge the DJEI/DES/SFI/HEA Irish Centre for High-End Computing (ICHEC) for the provision of computational facilities and support (www.ichec.ie).

Author Contributions: Conceptualization, SR and PC; methodology, SR and PC; software, SR and PC; validation, SR; formal analysis, SR; data curation, SR and PC; writing original draft, SR; writing review and editing, SR and PC; visualization, SR; supervision, PC; project administration, PC; funding acquisition, PC. All authors have read and agreed to the published version of the manuscript.

References

[1] A. Sharma, *Object-oriented Programming with C++*. Pearson Education India, 2014.
[2] B. Stroustrup, *Programming: principles and practice using C++*. Pearson Education, 2014.
[3] H. G. Weller, G. Tabor, H. Jasak, and C. Fureby, "A tensorial approach to computational continuum mechanics using object orientated techniques," *Computers in Physics*, vol. 12, pp. 620–631, 1998.
[4] W. McKinney, *Python for data analysis: data wrangling with pandas, NumPy, and IPython*, 2nd ed. Sebastopol, California: O'Reilly Media, Inc, 2018, oCLC: ocn959595088.
[5] "PYPL PopularitY of Programming Language index." [Online]. Available: https://pypl.github.io/PYPL.html
[6] "PyFoam." [Online]. Available: https://openfoamwiki.net/index.php/Contrib/PyFoam#Short_description
[7] "pythonFlu." [Online]. Available: https://openfoamwiki.net/index.php/Contrib/pythonFlu
[8] CyrilleBonamy, "fluidfoam." [Online]. Available: https://github.com/fluiddyn/fluidfoam
[9] "Owls." [Online]. Available: https://github.com/greole/owls
[10] "swak4Foam." [Online]. Available: https://openfoamwiki.net/index.php/Contrib/swak4Foam#Contents
[11] Anderluh and Jasak, "Like OpenFOAM, but Python," in *16th OpenFOAM Workshop, University College Dublin (Online)*, Dublin, Ireland, Jun. 2021.
[12] R. Maulik, D. Fytanidis, B. Lusch, V. Vishwanath, and S. Patel, "Pythonfoam: In-situ data analyses with openfoam and python," *arXiv preprint arXiv:2103.09389*, 2021.
[13] A. Weiner, D. Hillenbrand, H. Marschall, and D. Bothe, "DataDriven SubgridScale Modeling for ConvectionDominated Concentration Boundary Layers," *Chemical Engineering & Technology*, vol. 42, no. 7, pp. 1349–1356, Jul. 2019. [Online]. Available: https://onlinelibrary.wiley.com/doi/10.1002/ceat.201900044
[14] A. Weiner, "Running PyTorch models in OpenFOAM basic setup and examples." [Online]. Available: https://ml-cfd.com/openfoam/pytorch/docker/2020/12/29/running-pytorch-models-in-openfoam.html
[15] D. Kochkov, J. A. Smith, A. Alieva, Q. Wang, M. P. Brenner, and S. Hoyer, "Machine learning–accelerated computational fluid dynamics," *Proceedings of the National Academy of Sciences*, vol. 118, no. 21, 2021.

[16] O. Obiols-Sales, A. Vishnu, N. Malaya, and A. Chandramowliswharan, “CFDNet: a deep learning-based accelerator for fluid simulations,” in *Proceedings of the 34th ACM International Conference on Supercomputing.* Barcelona Spain: ACM, Jun. 2020, pp. 1–12. [Online]. Available: https://dl.acm.org/doi/10.1145/3392717.3392772

[17] M. Mozaffar, R. Bostanabad, W. Chen, K. Ehmann, J. Cao, and M. Bessa, “Deep learning predicts path-dependent plasticity,” *Proceedings of the National Academy of Sciences*, vol. 116, no. 52, pp. 26 414–26 420, 2019.

[18] D. W. Abueidda, S. Koric, N. A. Sobh, and H. Sehitoglu, “Deep learning for plasticity and thermo-viscoplasticity,” *International Journal of Plasticity*, vol. 136, p. 102852, Jan. 2021. [Online]. Available: https://linkinghub.elsevier.com/retrieve/pii/S0749641920302096

[19] C. Wang, L.-y. Xu, and J.-s. Fan, “A general deep learning framework for history-dependent response prediction based on UA-Seq2Seq model,” *Computer Methods in Applied Mechanics and Engineering*, vol. 372, p. 113357, Dec. 2020. [Online]. Available: https://linkinghub.elsevier.com/retrieve/pii/S0045782520305429

[20] J. Ling, A. Kurzawski, and J. Templeton, “Reynolds averaged turbulence modelling using deep neural networks with embedded invariance,” *Journal of Fluid Mechanics*, vol. 807, pp. 155–166, 2016.

[21] X. Morales, J. Mill, K. A. Juhl, A. Olivares, G. Jimenez-Perez, R. R. Paulsen, and O. Camara, “Deep Learning Surrogate of Computational Fluid Dynamics for Thrombus Formation Risk in the Left Atrial Appendage,” in *Statistical Atlases and Computational Models of the Heart. Multi-Sequence CMR Segmentation, CRT-EPiggy and LV Full Quantification Challenges*, M. Pop, M. Sermesant, O. Camara, X. Zhuang, S. Li, A. Young, T. Mansi, and A. Suinesiaputra, Eds. Cham: Springer International Publishing, 2020, vol. 12009, pp. 157–166, series Title: Lecture Notes in Computer Science. [Online]. Available: http://link.springer.com/10.1007/978-3-030-39074-7_17

[22] R. Maulik, H. Sharma, S. Patel, B. Lusch, and E. Jennings, “Deploying deep learning in openfoam with tensorflow,” in *AIAA Scitech 2021 Forum*, 2021, p. 1485.

[23] Tobias Hermann, “frugally-deep.” [Online]. Available: https://github.com/Dobiasd/frugally-deep

[24] W. Jakob, J. Rhinelander, and D. Moldovan, “pybind11 – seamless operability between c++11 and python,” 2017, https://github.com/pybind/pybind11.

[25] “Conda documentation.” [Online]. Available: https://docs.conda.io/en/latest/

[26] “pybind11 documentation.” [Online]. Available: https://pybind11.readthedocs.io/en/stable/index.html

[27] “[QUESTION]. Crash when importing NumPy manually Issue #2485 - pybind/pybind11.” [Online]. Available: https://github.com/pybind/pybind11/issues/2485

[28] “__main__ Top-level code environment. Python 3.10.0 documentation.” [Online]. Available: https://docs.python.org/3/library/__main__.html

[29] “Built-in Types. Python 3.10.0 documentation.” [Online]. Available: https://docs.python.org/3/library/stdtypes.html?highlight=__dict__#built-in-types

[30] “Python types - pybind11 documentation.” [Online]. Available: https://pybind11.readthedocs.io/en/stable/advanced/pycpp/object.html

[31] “ctypes A foreign function library for Python. Python 3.10.0 documentation.” [Online]. Available: https://docs.python.org/3/library/ctypes.html

[32] “Exceptions - pybind11 documentation.” [Online]. Available: https://pybind11.readthedocs.io/en/stable/advanced/exceptions.html

[33] P. Cardiff, A. Karač, P. De Jaeger, H. Jasak, J. Nagy, A. Ivanković, and Ž. Tuković, “An open-source finite volume toolbox for solid mechanics and fluid-solid interaction simulations,” *arXiv preprint arXiv:1808.10736*, 2018.

[34] F. Chollet *et al.*, “Keras,” https://keras.io, 2015.

[35] “CFD.Direct.com. OpenFOAM v9 User Guide: 2.1 lid-driven cavity flow.” [Online]. Available: https://cfd.direct/openfoam/user-guide/v9-cavity/#x5-40002.1

[36] “300-times faster resolution of finite-difference method using numpy.” [Online]. Available: https://towardsdatascience.com/300-times-faster-resolution-of-finite-difference-method-using-numpy-de28cdade4e1

[37] “Lamé parameters,” https://en.wikipedia.org/wiki/Lame_parameters, accessed 27 May 2022.

[38] S. Timoshenko and J. N. Goodier, *Theory of Elasticity.* McGraw-Hill, 1951.