OpenFOAM® Journal

Volume 123, Pages 1–999

ISSN: TBD

A SEMI-AUTOMATIC APPROACH BASED ON THE METHOD OF MANUFACTURED SOLUTIONS TO ASSESS THE CONVERGENCE ORDER IN OPENFOAM

DOI: TBD

Version(s): OpenFOAM® v1912

Repo: https://github.com/xxx

Abstract. Code verification is an intricate but crucial part of numerical code development. Due to the complexity of the partial differential equations, an analytical solution might not exist. In those situations and aiming at proving that the code is solving appropriately the governing equations, the method of manufactured solutions (MMS) is a powerful tool. In this method, a source term is derived to enforce the solution to a predetermined function. By performing a mesh refinement study, one can verify if the code is correctly solving the desired equations. In this work, a methodology that allows the automation of the MMS within the OpenFOAM® framework is proposed. The developed computational framework comprises a set of tools prepared, in an open-source environment, for the symbolic computation of the associated source term, and to generate the code required for its implementation as well as appropriate boundary conditions and functions to calculate the error norms.

1. Introduction

Code verification is an intricate but crucial part of numerical code development [1, 2]. In computational modelling, systems of partial differential equations (PDE) equipped with appropriate boundary conditions are used to model several physical phenomena, such as fluid dynamics, structural mechanics, conjugated heat transfer, combustion, transport in porous media, etc. In that context, verification procedures aim at demonstrating that the continuum governing equations are solved correctly, and that the computed solution tends asymptotically to the exact solution as finer meshes and smaller time-step sizes, are applied. In that regard, the convergence order of the method measures the rate at which the error decreases with mesh and time-step refinement.

Several methodologies of verification can be employed to demonstrate that the code correctly solves the physical model. The classical approach consists in deriving an analytical (exact) solution for the governing equations, which is usually only feasible for problems with reduced dimension (1D or 2D) and simple geometries. For some physical phenomena, exact solutions might not be available even for simplified setups. This is usually known as the method of exact solutions and has been used extensively

* Corresponding author

Received: 14 Juntember 2022, Accepted: 29 Januaril 2022, Published: 17 Decemby 2022

©2021 OpenFOAM® Journal

in the context of OpenFOAM® for solver verification [3]. However, for real-world problems, the models to solve often contain non-linear terms, couplings, variable coefficients, and complex geometries [4], which hinders or makes it impossible to derive analytical solutions. Moreover, depending on the mathematical techniques employed for deriving the exact solution, the use of infinite sums, special functions or non-trivial integrals will raise issues, such as where to truncate the summation, what integration scheme should be used, and the possible existence of singularities in the solution [4].

The method of manufactured solutions (MMS) is an alternative methodology that is more practical and interesting for code verification in complex problems [4, 5, 6]. The method consists in providing an analytical solution for the unknowns of the problem as a function of space and time. Then, specific source terms for each governing equation are determined by applying the corresponding differential operators to the predefined solution. Plugging the computed source terms into the original PDE(s) will result in a new set of equations whose exact solution is the manufactured solution. The convergence order of the method can then be assessed by performing a series of tests, through either a mesh or time-step refinement, or a combination of both.

In most situations, the manufactured solution should be smooth and regular in space and time (for unsteady problems), non-trivial, and allow the calculation of all derivatives comprised in the PDE(s) (e.g. cross-derivatives). Additionally, the best type of manufactured solutions should be described with simple functions, such as trigonometric, exponential or high degree polynomials. Detailed guidelines for creating manufactured solutions can be found in Salari and Knupp [4].

In the context of OpenFOAM® , the MMS has been employed to assess different solvers, covering several areas, such as computational fluid dynamics [7, 8, 9, 10], solid mechanics [11], fluid-structure interaction [12], heat transfer [13], nuclear engineering [14], reacting flows [15], among others. However, in all the referred works, the MMS is used to assess the developed solvers, but no details are provided to guide users interested in extending the approach to other cases. Moreover, the manipulation of the complex analytical expressions associated even with simple cases, and the implementation of required source terms and boundary conditions is often a complex, labour-intensive and error-prone task.

In the present work, a semi-automatic approach to undertake code verification in OpenFOAM® through the MMS is proposed and illustrated. The approach allows a straightforward extension to different case studies and solvers. It resorts to *Sympy* [16] v1.8, an open-source symbolic computation module from *Python* programming language, and OpenFOAM® 'v1912 `fvOptions` and `coded` functionality, which allows a high degree of flexibility for a swift, and automatic, computation of source terms and boundary conditions.

The remaining paper is organized as follows. Section 2 introduces the MMS and the computation of associated error norms and convergence orders. In Section 3, the semi-automatic approach for implementing the MMS in OpenFOAM® is presented. The subsequent section, Section 4, a number of case studies

involving the *laplacianFoam* [17] and *simpleFoam* [18] solvers are carried out, using structured meshes, to demonstrate the application of the proposed semi-automatic approach. In Section 5, the results obtained for the mentioned case studies are presented and discussed. Finally, in Section 6, the main conclusions of the present work are drawn.

2. Methodology

2.1. **Unsteady heat transfer problem.** In the present section, the methodology of the MMS is described in more detail and illustrated for the two-dimensional unsteady heat equation, considering only conduction. The governing equation is solved for a spatial domain $(x, y) \in \Omega$ with boundary $\Gamma = \Gamma^{\mathrm{D}} \cup \Gamma^{\mathrm{N}}$, and time range $t \in \left(0, t^{\mathrm{F}}\right]$, equipped with appropriate boundary and initial conditions, and reads:

$$\frac{\partial T}{\partial t} - \nabla \cdot (D_T \nabla T) = S_{\mathrm{T}}, \qquad \text{in } \Omega \times \left(0, t^{\mathrm{F}}\right], \tag{1}$$

$$T = g^{\mathrm{D}}, \qquad \text{in } \Gamma^{\mathrm{D}} \times \left(0, t^{\mathrm{F}}\right], \tag{2}$$

$$\nabla T \cdot \boldsymbol{n} = g^{\mathrm{N}}, \qquad \text{in } \Gamma^{\mathrm{N}} \times \left(0, t^{\mathrm{F}}\right], \tag{3}$$

$$T(x, y, t = 0) = T^0, \qquad \text{in } \Omega, \tag{4}$$

where:

- $T \equiv T(x, y, t)$ is the unknown temperature distribution,
- $D_{\mathrm{T}} \equiv D_{\mathrm{T}}(x, y, t)$ is the thermal diffusivity coefficient,
- $S_{\mathrm{T}} \equiv S_{\mathrm{T}}(x, y, t)$ is the heat source term (sinks have a negative value),
- $g^{\mathrm{D}} \equiv g^{\mathrm{D}}(x, y, t)$ is a prescribed boundary temperature value,
- $g^{\mathrm{N}} \equiv g^{\mathrm{N}}(x, y, t)$ is a prescribed boundary temperature gradient normal to the boundary,
- $T^0 \equiv T^0(x, y)$ is the temperature distribution at time $t = 0$, and
- $\boldsymbol{n} \equiv \boldsymbol{n}(x, y) \equiv (n_x(x, y), n_y(x, y))$ is the unit outward vector normal to the boundary.

In the above notation, the superscripts `F`, `D` and `N` stand for "Final", "Dirichlet" and "Neumann", respectively.

2.2. **Method of manufactured solutions.** Consider that some function $T_{\mathrm{ex}} \equiv T_{\mathrm{ex}}(x, y, z, t)$ is the manufactured solution of the unsteady heat transfer problem, which must be regular and smooth in space and time. For instance, trigonometric functions, such as the sine and cosine, tend to be good choices, since they are continuous and infinitely differentiable. Following the methodology proposed in [4], a source term function $S_{\mathrm{T}}(x, y, z, t)$ for the unsteady heat equation is determined as $S_{\mathrm{T}} = \partial T_{\mathrm{ex}}/\partial t - \nabla \cdot (D_{\mathrm{T}} \nabla T_{\mathrm{ex}})$. Then, the unsteady heat equation, Equation (1), is solved, considering appropriate discretization method(s) for the unknown variable, $T(x, y, z, t)$, and including the source term and boundary conditions determined with the manufactured solution, $T_{\mathrm{ex}}(x, y, z, t)$. Finally, having an approximate discrete solution for the

unknown temperature function, the corresponding errors can be computed, since the exact solution for this specific problem was previously manufactured and, therefore, is known.

Example 1. *The application of the MMS to the unsteady heat transfer problem is hereafter demonstrated with a simple example. Assume a constant value for* D_T *and a manufactured solution for a 2D case study given as:*

$$T_{ex}(x,y,t) = 150\left(\cos\left(x^2+y^2+\omega t\right)+1.5\right). \tag{5}$$

As previously presented, the corresponding heat source term, $S_T(x,y,t)$*, is obtained by replacing, in the governing PDE* (1)*, the unknown temperature distribution,* $T(x,y,t)$*, with the manufactured counterpart,* $T_{ex}(x,y,t)$*, which yields:*

$$\begin{aligned} S_T(x,y,t) &= \frac{\partial T_{ex}(x,y,t)}{\partial t} - \nabla\cdot\left(D_T\nabla T_{ex}(x,y,t)\right) \\ &= 600D_Tx^2\cos\left(\omega t+x^2+y^2\right)+600D_Ty^2\cos\left(\omega t+x^2+y^2\right) \\ &\quad +600D_T\sin\left(\omega t+x^2+y^2\right)-150\omega\sin\left(\omega t+x^2+y^2\right). \end{aligned} \tag{6}$$

The problem is equipped with the appropriate boundary conditions, assuming either a prescribed boundary temperature (Dirichlet boundary condition) or a prescribed temperature normal gradient (Neumann boundary condition). In any case, the corresponding functions to be imposed in Equations (2) *and* (3)*, are computed based on the manufactured solution,* $T_{ex}(x,y,t)$*. For this example, the Dirichlet boundary condition function is determined as:*

$$g^D(x,y,t) = T_{ex}(x,y,t) = 150\left(\cos\left(x^2+y^2+\omega t\right)+1.5\right), \tag{7}$$

while the Neumann boundary condition function is given by:

$$\begin{aligned} g^N(x,y,t) &= \nabla T_{ex}(x,y,t)\cdot\boldsymbol{n} \\ &= \begin{bmatrix} -300x\sin\left(\omega t+x^2+y^2\right) \\ -300y\sin\left(\omega t+x^2+y^2\right)\end{bmatrix}\cdot\begin{bmatrix} n_x \\ n_y\end{bmatrix} \\ &= -300\sin\left(\omega t+x^2+y^2\right)\left(n_xx+n_yy\right). \end{aligned} \tag{8}$$

Other boundary conditions, such as the Robin boundary condition, can be prescribed following the same procedure. The initial condition is simply set as $T^0(x,y) = T_{ex}(x,y,t=0)$*. In practice, different combinations of boundary conditions can be selected for the boundaries of the domain.*

Finally, solving the governing PDE, Equation (1)*, using the previously determined source term (Equation* (6)*), boundary (Equation* (7) *and/or* (8)*), and initial values, the exact solution of the unsteady heat transfer problem is the manufactured solution, Equation* (5)*.*

2.3. **Errors and convergence orders.** Having a manufactured solution for the problem under study, the employed numerical techniques (discretization method, solution algorithm, etc.) can be verified by determining the accuracy of the computed approximate solution. For that purpose, consider that vector $\overline{\boldsymbol{T}}^n$ gathers the exact piece-wise solution determined from the manufactured solution, Equation (5), with a given mesh and at a specific time $t = t^n$. On the other side, vector $\boldsymbol{T}^{n,*}$ gathers the approximate values obtained with the numerical code in the same mesh and at the same instant of time, considering the appropriate source term, boundary and initial conditions. In the context of the finite volume method, the exact value for a given cell C_i at time $t = t^n$, denoted as $\overline{T}_i^n$, corresponds to the cell mean-value of the manufactured solution, that is:

$$\overline{T}_i^n = \iint_{C_i} T_{\text{ex}}\left(x, y, t^n\right) \mathrm{d}x \, \mathrm{d}y, \tag{9}$$

whereas the corresponding approximate value, denoted as $T_i^{n,*}$, is defined at the cell mass center. Then, a global measure of the error can be calculated, where the L^1-, L^2- and L^∞-norms are usually chosen, for which the associated errors at time t^n are denoted as E^1, E^2 and E^∞, and are defined as:

$$E^1 = \frac{\displaystyle\sum_{i=1}^{N_\text{C}} \left|\overline{T}_i^n - T_i^{n,*}\right| V_i}{\displaystyle\sum_{i=1}^{N_\text{C}} V_i}, \tag{10}$$

$$E^2 = \sqrt{\frac{\displaystyle\sum_{i=1}^{N_\text{C}} \left(\overline{T}_i^n - T_i^{n,*}\right)^2 V_i}{\displaystyle\sum_{i=1}^{N_\text{C}} V_i}}, \tag{11}$$

$$E^\infty = \max_{i=1}^{N_\text{C}} \left|\overline{T}_i^n - T_i^{n,*}\right|, \tag{12}$$

where:

- N_C is the total number of cells in the mesh, and
- V_i is the volume of cell C_i.

In the L^1-norm, the absolute individual errors are scaled with the corresponding cell volumes and, therefore, all the cells have a comparable contribution to the error norm for reasonably uniform meshes. On the other side, in the L^2-norm, also known as the Euclidean norm, the weight of the cells with larger errors is more pronounced when compared to the L^1-norm. Finally, the L^∞-norm represents the largest error magnitude in the whole mesh. The comparison of the errors in the L^1- or L^2-norm with the corresponding error in the L^∞-norm is also interesting as it provides an insight into the error distribution without visualizing it. That is, if the errors in the L^1- or L^2-norm is comparable to the corresponding error in the L^∞-norm, then the individual errors have a relatively even distribution. On the contrary,

when the ratio between these error norms becomes more significant, larger errors do exist for specific cells in the mesh, which indicates that a careful analysis might be necessary.

The errors of the computed approximate solution obtained with just one mesh (or time-step) are not sufficient to conclude whether the method is correctly implemented. Indeed, provided that the method is consistent, the approximate (numerical) solution should converge to the exact counterpart, while the mesh characteristic or time-step sizes decrease. In that case, the rate at which the approximate solution error decreases under mesh or time-step refinement is called convergence order. More specifically, for steady-state problems or unsteady problems at a certain instant of time, the error in some norm is given as:

$$E = Ch^P + H.O.T., \tag{13}$$

where:

- C is a constant independent of the mesh size,
- h is the mesh characteristic size, a representative measure of the size of the elements in the mesh, such that for the particular case of structured uniform meshes it corresponds to the length of the edges,
- P is a constant that determines the rate at which the error changes with h, the so-called convergence order, and
- $H.O.T.$ represents the higher order terms that are not explicitly defined.

The contribution of $H.O.T.$ can be usually neglected and, therefore, the error in some norm can be given as $E = Ch^P$, that is, proportional to h^P. In that case, as the mesh characteristic size decreases, the error is also expected to decrease with h at a rate corresponding to P. For unsteady problems, the same premises hold and, therefore, the error in some norm is also a function of the employed time-step size, Δt.

In practice, for steady-state problems, the convergence order of the method can be determined by solving the same test case under the same conditions with successively finer meshes. In that case the ratio between the characteristic size of two consecutive meshes, r_h (referred to as coarser and finer), is given as:

$$r_h = \frac{h_{\text{coarser}}}{h_{\text{finer}}} \approx \left(\frac{N_{\text{finer}}}{N_{\text{coarser}}} \right)^{\frac{1}{D}}, \tag{14}$$

where:

- h_{coarser} is the characteristic size of the coarser mesh,
- h_{finer} is the characteristic size of the finer mesh,
- N_{coarser} is the number of cells in the coarser mesh,
- N_{finer} is the number of cells in the finer mesh, and
- D is the problem dimension ($D = 1, 2, 3$ for a 1D, 2D and 3D problems, respectively).

Finally, the convergence orders for the errors in the L^1-, L^2- and L^∞-norms, denoted as O^1, O^2 and O^∞, respectively, are given as:

$$O^1 = \frac{\ln\left(E^1_{\text{coarser}}/E^1_{\text{finer}}\right)}{\ln\left(r_h\right)}, \tag{15}$$

$$O^2 = \frac{\ln\left(E^2_{\text{coarser}}/E^2_{\text{finer}}\right)}{\ln\left(r_h\right)}, \tag{16}$$

$$O^\infty = \frac{\ln\left(E^\infty_{\text{coarser}}/E^\infty_{\text{finer}}\right)}{\ln\left(r_h\right)}, \tag{17}$$

where E^1_{coarser}, E^2_{coarser} and $E^\infty_{\text{coarser}}$ are the the L^1-, L^2- and L^∞-norms errors for the coarser mesh and E^1_{finer}, E^2_{finer} and E^∞_{finer} are the same error norms for the finer mesh.

The same analysis can be performed to assess the convergence order of the method under mesh refinement for unsteady problems, employing the previous procedure for a fixed time-step size. However, in that case, a sufficiently small time-step size must be carefully chosen, such that the time discretization error does not exceed the space discretization counterpart. To determine if the chosen time-step size is appropriate, the same simulations can be replicated with a slightly smaller time-step size, but with the same successively finer meshes. If the computed solution error for any of the meshes is significantly different than previously for the same mesh, the time discretization error is still predominant. In that case, the verification procedure must be repeated until no significant changes are observed in the computed solution errors from employing a smaller time-step size. This is accomplished by selecting the time-step below which the reported significant digits for the error remain unchanged. It is important to notice that since the time and space discretization errors decrease under mesh and time refinement, respectively, larger time-step sizes can be used for coarser meshes, without affecting the obtained conclusions. Accordingly, as smaller mesh characteristic sizes are considered, decreasing progressively the time-step size might be required to avoid the influence of the time discretization error on the results obtained. Consequently, identifying an appropriate time-step size for each mesh refinement level might be more difficult to employ but can provide significant computational savings in computationally demanding test cases.

On the other side, to assess the convergence order under time-step refinement, the same procedure can be employed, considering successively smaller time-step sizes for a fixed mesh characteristic size. In that case, Equations (15) to (17) for the convergence order in time must consider the ratio between two successive time-step sizes, $r_{\Delta T}$. As before, a sufficiently small mesh characteristic size must be carefully chosen such that the space discretization error does not exceed the time discretization error. To determine the appropriate mesh characteristic size, the same simulations can be replicated with a finer mesh until no significant changes are observed in the computed solution errors. Alternatively, decreasing simultaneously both the mesh characteristic and the time-step sizes can be computationally more efficient, although it is generally more difficult to employ.

3. Semi-automatic approach

Based on a selected manufactured solution, the implementation of the MMS approach in OpenFOAM® requires the calculation of: (i) the associated distributed source term, (ii) the associated Dirichlet and/or Neumann boundary condition functions, and (iii) the `coded functionObject` for the computation of the error norms. Given the arbitrary complexity of choosing a non-trivial manufactured solution, the programming of the associated source term and boundary condition functions can become a cumbersome and error-prone task. In that regard, a *Python* package (referred to as *pyMMSFoam*) was created to perform the necessary mathematical differentiation based on the *Sympy* v1.8 module [16], a rich-featured library for symbolic computation written in *Python*. The package converts the generated mathematical expressions to *C* syntax so that they can be copied directly into an OpenFOAM® case setup files. In addition, the *Sympy*'s built-in common sub-expression elimination (CSE) routine [19] is used to simplify the mathematical expressions and, therefore, optimize the generated code.

The semi-automatic approach consists, firstly, in generating the required inputs for the implementation of the MMS approach in OpenFOAM® , which can be easily performed with the developed package. Then, the test case setup is prepared in OpenFOAM® and consecutive simulations are performed to determine the computed solution errors and convergence orders. The flowchart of the proposed semi-automatic approach based on the MMS in OpenFOAM® is illustrated in Figure 1 and is described in more detail hereafter. For the sake of simplicity, consider a 2D unsteady heat transfer equation, given by Equation (1), with $\omega = 0.1$ and $D_{\mathrm{T}} = 10^{-3}$ m^2/s. This will also be used for the case study of the *laplacianFoam* solver in Section 4.1.

In **Step 1** (see Figure 1), the manufactured solution has to be defined, and the user can rely on the *Sympy*'s built-in functionality, such as trigonometric and exponential functions. Listing 1 illustrates the code used for the 2D unsteady heat transfer case (Equation (1)), with the above referred parameters.

From lines 1 to 3, the *pyMMSFoam* package is being loaded with the alias `mms`, the Cartesian space symbolic variables `x`, `y` and `z`, the temporal symbolic variable `t`, and the `cos` function from *Sympy* are being imported from their respective modules. Notice that the symbolic variable `z` is not required for this specific manufactured solution, but it is shown for the sake of completeness and easy adaptation to other cases. In lines 5 and 7, the value of diffusion coefficient and the manufactured solution, Equation (5), are declared.

The differential operators available in *pyMMSFoam* are the following: time derivative (for scalar and vector quantities), divergence (for vector and tensor quantities), gradient (for scalar and vector quantities) and Laplacian (for scalar quantities). Moreover, *pyMMSFoam* interprets a symbolic expression as a scalar quantity, a matrix of shape (3 x 1) as a vector and a matrix of shape (3 x 3) as a tensor.

For **Steps 2** to **5** (see Figure 1), the generation of the input codes for the the MMS implementation in OpenFOAM® is performed. In the second part of Listing 1, the source term, Dirichlet and Neumann

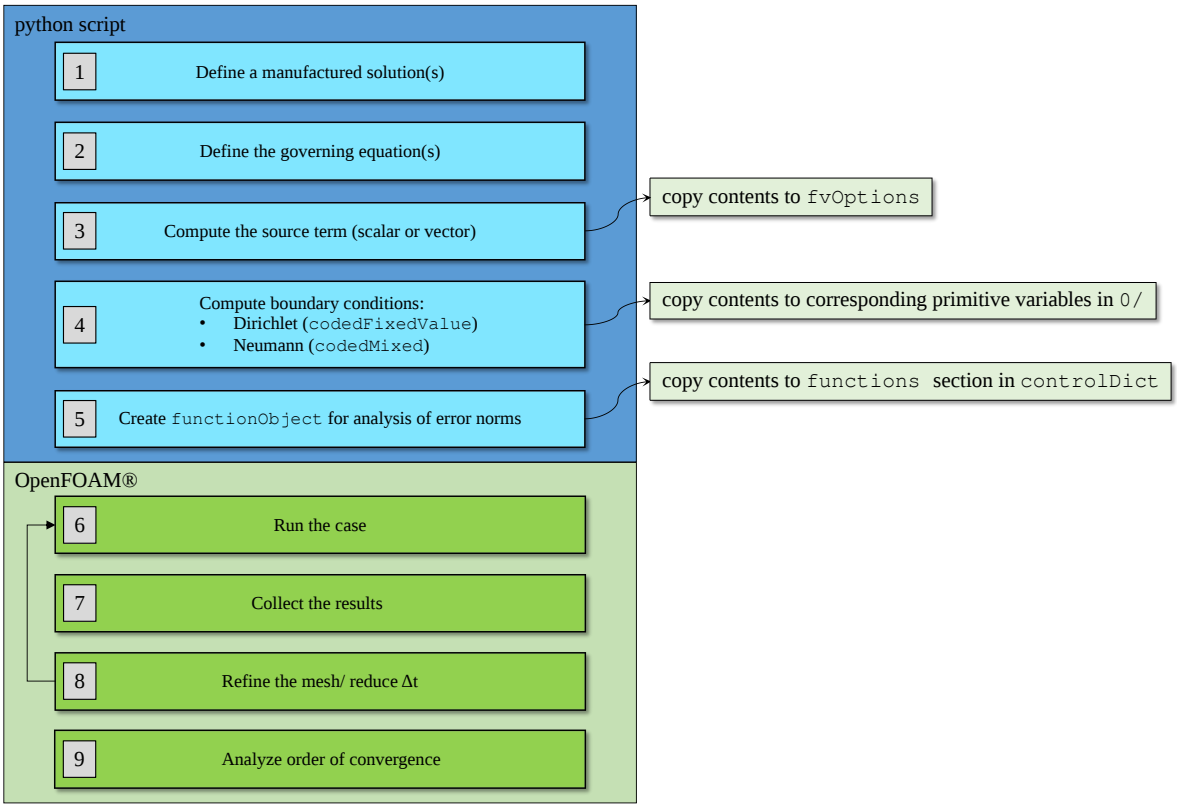


FIGURE 1. Flow chart of the semi-automatic approach based on the MMS in OpenFOAM® .

```
1  import pyMMSFoam as mms
2  from pyMMSFoam import x,y,z,t
3  from sympy import cos
4
5  DT = 1e-3
6
7  T = 150*(cos(x*x + y*y + 0.1*t) + 1.5)
8
9  S_T = mms.ddt(T) - mms.laplacian(T, DT)
10
11 mms.generateFvOptions(S_T, "sourceTerm", "T")
12 mms.generateDirichletBoundaries(T, "T")
13 mms.generateNeumannBoundaries(T, "T")
14 mms.generateFunctionObject(T, "T")
```

LISTING 1. *laplacianFoam* MMS script.

boundary conditions are symbolically created. Moreover, a `coded functionObject` is also generated to compute the approximate solution error using different norms (Equations 10, 11 and 12) .

In line 9 of Listing 1, the associated source term is computed from the governing equation for the unsteady heat transfer problem, Equation (1), that includes the time derivative and Laplacian differential operators. The command presented in line 11 generates a `fvOptions` dictionary, by running the function `generateFvOptions`, whose arguments are the computed source term and two strings, the names of the source term and the field where the source term should be applied (`T` for the present problem). The implemented function will check whether the source term is a scalar or a vector quantity and write a

`scalarCodedSource` or a `vectorCodedSource` `fvOptions` dictionary, respectively. The output for this example is presented in Listing 2.

```
sourceTerm
{
    type            scalarCodedSource;
    selectionMode   all;
    fields          (T);
    name            sourceTerm_2;

    codeInclude
    #{
    #};
    codeCorrect
    #{
    #};
    codeConstrain
    #{
    #};
    codeAddSup
    #{
                const scalarField& V = mesh_.V();
                const volVectorField& C = mesh().C();
                scalarField& TSource= eqn.source();

                const Time& time = mesh().time();
                const scalar t = time.value();

                forAll(TSource, cellI)
                {
                        const scalar x = C[cellI].x();
                        const scalar y = C[cellI].y();

                        const scalar tmp0 = x*x;
                        const scalar tmp1 = y*y;
                        const scalar tmp2 = 0.10*t + tmp0 + tmp1;
                        const scalar tmp3 = 0.6*Foam::cos(tmp2);
                        const scalar solution_T = tmp0*tmp3 + tmp1*tmp3 -
                        14.4*Foam::sin(tmp2);

                        TSource[cellI] -= V[cellI]*(solution_T);
                };
    #};
}
```

LISTING 2. `fvOptions` dictionary generated for the 2D unsteady heat transfer equation (*laplacianFoam* solver).

In lines 12 and 13 of Listing 1 Dirichlet and Neumann boundary conditions are generated using a `codedFixedValue` and `codedMixed` boundary condition, respectively. The arguments of these functions are the symbolic variable that represents the manufactured solution and a string with the field name. Analogously to what was described for function `generateFvOptions`, the tool will check whether the input is a scalar or vector quantity, and generate the output accordingly. For this example, the output for each type of boundary condition is illustrated in Listings 3 and 4. Notice that the name of the patches should be changed afterwards from `"(patch1|patch2|patch3)"` to the ones defined in the test case setup.

```
"(patch1|patch2|patch3)"
{
    type        codedFixedValue;
    value       uniform 0;

    name        T_dirichlet;

    code
    #{
        const fvPatch& boundaryPatch = patch();

        const vectorField& Cf = boundaryPatch.Cf();

        scalarField& field = *this;

        const scalar t = this->db().time().value();

        forAll(Cf, faceI)
        {
            const scalar x = Cf[faceI].x();
            const scalar y = Cf[faceI].y();
            const scalar T = 225.0 + 150*Foam::cos(0.10*t + x*x + y*y);

            field[faceI] = T ;
        }
    #};
}
```

LISTING 3. Dirichlet boundary condition generated for the 2D unsteady heat transfer equation (*laplacianFoam* solver).

Finally, in line 14 of Listing 1, a `coded functionObject` is created to calculate the computed solution error using the norms previously defined (Equations 10, 11 and 12). The function arguments are the variable that represents the manufactured solution and a string with the field variable name. The output of this function for this illustrative example is provided in Listing 5.

Following the procedure flow chart, Figure 1, the outputs of performing **Steps 3-5** in *pyMMSFoam* are illustrated in Listings 3, 4, 5 and 6, respectively. The generated codes must be copied into the `fvOptions` file, the corresponding variable file in the "0" folder and the function section of the `controlDict` file, respectively.

After defining the manufactured solution and computing the source term, the appropriate boundary condition functions and the `coded functionObject` to monitor the computed solution error, a set of OpenFOAM® test cases must be prepared to determine the solver convergence order. For this purpose, successively finer meshes or time-steps should be used for each of those test cases, following the procedure described in Section 2.3. For each of the test cases, the output errors as well and the associated mesh characteristic sizes (or the number of cells) or time-steps should be collected. After computing the ratio between the characteristic sizes of two successively finer meshes (or time-steps) according to Equation (14), the solver convergence order can be determined as in Equations (15) to (17).

```
"(patch1|patch2|patch3)"
{
        type            codedMixed;
        refValue        uniform 0;
        refGradient     uniform 0;
        valueFraction   uniform 0;

        name        T_Neumann;

        code
        #{
                const fvPatch& boundaryPatch = patch();
                const vectorField& Cf = boundaryPatch.Cf();
                const vectorField nf = patch().nf();
                const scalar t = this->db().time().value();

                forAll(this->patch(), faceI)
                {
                        const scalar x = Cf[faceI].x();
                        const scalar y = Cf[faceI].y();

                        const scalar tmp0 = 300*Foam::sin(0.10*t + x*x + y*y);
                        const scalar dT_dx = -tmp0*x;
                        const scalar dT_dy = -tmp0*y;
                        const scalar dT_dz = 0;

                        const vector gradT (dT_dx, dT_dy, dT_dz);

                        const scalar normalGradient = gradT & nf[faceI] ;
                        this->refGrad()[faceI] = normalGradient;
                        this->valueFraction()[faceI] = scalar(0);
                }
        #};
}
```

LISTING 4. Neumann boundary condition generated for the 2D unsteady heat transfer equation (*laplacianFoam* solver).

4. CASE STUDIES

In this section, some test case studies are presented to illustrate the application of the proposed semi-automatic approach in OpenFOAM® for the verification of the *laplacianFoam* and *simpleFoam* solvers. For all the cases, a two-dimensional unitary square domain $\Omega = [0,1]^2$ m, on the $x-y$ plane, is considered, as shown in Figure 2.

On the `top`, `bottom`, `left` and `right` patches of the domain, Dirichlet and/or Neumann boundary conditions are prescribed. On the patches normal to the z-axis, referred to as `front` and `back`, a boundary condition of type `empty` is prescribed to reduce the dimensionallity of the problem to 2D. In that regard, as is standard practice in OpenFOAM® , the generated meshes have a single cells layer along the z-direction.

The solver spatial convergence order is assessed with 5 grids, the coarser one having 32 elements along the x- and y-directions, and doubling the number of the cells in each direction for each successively finer grid.

```
functions
{
    errorNorm_T
    {
        type coded;
        libs (utilityFunctionObjects);
        writeControl writeTime;
        name analyticalSolution_T;
        codeWrite
        #{
            const volScalarField& T_find = mesh().lookupObject<volScalarField>("T");
            const volVectorField& C = mesh().C();
            const scalarField& V = mesh().V();

            volScalarField MMS_diff_T
            (
                IOobject
                (
                    "MMS_diff_T",
                    mesh().time().timeName(),
                    mesh(),
                    IOobject::NO_READ,
                    IOobject::AUTO_WRITE
                ),
                mesh(),
                dimensionedScalar ("MMS_diff_T_", dimless, 0.0)
            );

            const Time& time = mesh().time();
            const scalar t = time.value();

            forAll(MMS_diff_T, cellI)
            {
                const scalar x = C[cellI].x();
                const scalar y = C[cellI].y();
                const scalar solution = 225.0 + 150*Foam::cos(0.10*t + x*x + y*y);

                MMS_diff_T[cellI] = mag(solution - T_find[cellI]);
            }

            Info<< "L1 norm is: " << gSum(MMS_diff_T*V)/gSum(V)  << endl;

            Info<< "L2 norm is: "
                << sqrt(gSum(MMS_diff_T*MMS_diff_T*V)/gSum(V))<< endl;

            Info<< "Linf norm is: " << gMax(MMS_diff_T) << endl;

            MMS_diff_T.write();
        #};
    }
}
```

LISTING 5. `coded functionObject` generated for the 2D unsteady heat transfer equation (*laplacianFoam* solver).

4.1. ***laplacianFoam* test case studies .** The *laplacianFoam* solver implements an unsteady/steady-state heat transfer model governed by Equation (1). Three test case studies are presented to verify the solver: a 2D steady-state test case and a 2D and a 1D unsteady test cases. For all, the manufactured

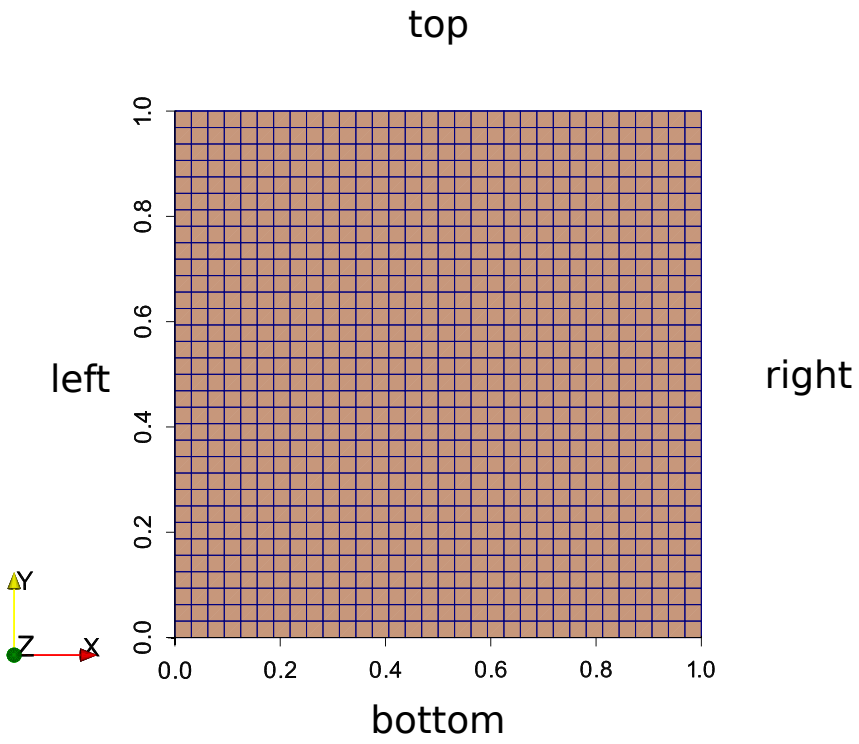


FIGURE 2. 2D domain with representation of the corresponding patches (dimensions in meters).

solution for the temperature corresponds to Equation (5) and the associated source term is given in Equation (6). For the sake of simplicity only Dirichlet boundary conditions, Equation (7), are prescribed.

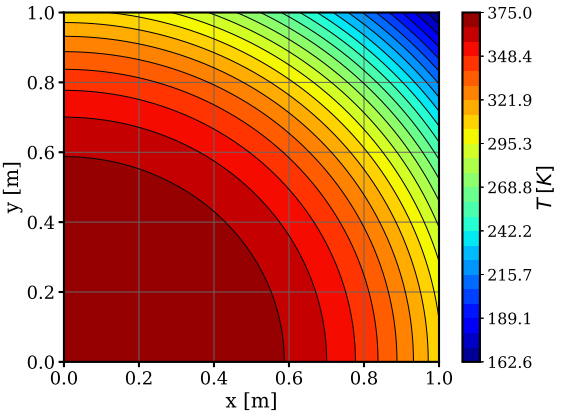
The `fvOptions` for the source term, the `codedFixedValue` for the boundary conditions, and the `coded functionObject` for the computation of the error norms, all generated with *pyMMSFoam*, are provided in Listings 2 to 5. Notice that $\omega = 0$ is set for the 2D steady-state test case study. Moreover, for the 1D unsteady test case study, only cells along the x-direction are generated and the y-coordinate dependence is removed from the manufactured solution, Equation (5). Finally, a thermal diffusivity coefficient of $D_T = 10^{-3}$ m^{2}s^{-1} is considered for all case studies. The resulting temperature field for the three test case studies is depicted in Figure 3.

For the three test case studies, the temporal and spatial discretization schemes employed for the *laplacianFoam* solver are provided in Table 1. Moreover, the resulting system of linear equations is solved with a preconditioned conjugate gradient (PCG) method using a diagonal-based incomplete Cholesky (DIC) preconditioner. The associated parameters are given in Table 2.

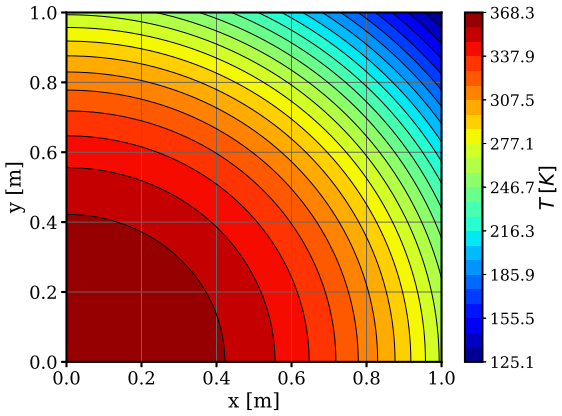
4.2. ***simpleFoam* test case study.** The governing equations solved in the *simpleFoam* solver correspond to the Navier-Stokes equations for a steady-state and incompressible flow [18], given as:

$$\nabla \cdot \boldsymbol{U} = 0, \tag{18}$$

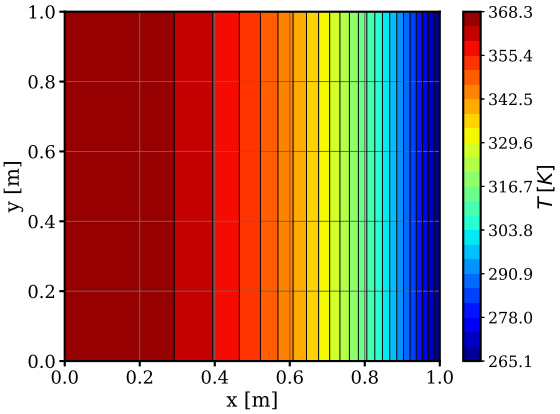
$$\nabla \cdot (\boldsymbol{U} \otimes \boldsymbol{U}) - \nabla \cdot \boldsymbol{R} = -\nabla p + S_{\mathrm{U}}. \tag{19}$$



(a) 2D steady-state test case study.



(b) 2D unsteady test case study at $t = 3$s.



(c) 1D unsteady test case study at $t = 3$s.

FIGURE 3. Manufactured solutions for the verification of the *laplacianFoam* solver.

TABLE 1. Temporal and spatial discretization schemes for the verification of the *laplacianFoam* solver (*fvSchemes* dictionary).

Parameter	Steady-state	Unsteady
ddtSchemes	steadyState	Euler
gradSchemes	Gauss linear	Gauss linear
divSchemes	none	none
laplacianSchemes	Gauss linear uncorrected	Gauss linear uncorrected
interpolationSchemes	linear	linear
snGradSchemes	none	none

TABLE 2. Solution methods for the verification of the *laplacianFoam* solver (*fvSolution* dictionary).

Parameter	
solver	PCG
preconditioner	DIC
tolerance	10^{-12}
relTol	0.0

Equations (18) and (19) represent the mass conservation and the momentum balance, respectively, where for a 2D case:

- $\boldsymbol{U} \equiv (u, v) \equiv (u(x, y), v(x, y))$ is the velocity vector,
- $\boldsymbol{R} \equiv \boldsymbol{R}(x, y)$ is the deviatoric stress tensor divided by the density,
- $p \equiv p(x, y)$ is the kinematic pressure (pressure divided by the density), and
- $S_U \equiv S_U(x, y)$ is the source term.

In the scope of the present study, the deviatoric stress tensor is defined as:

$$\boldsymbol{R} = \nu \left[\nabla \boldsymbol{U} + (\nabla \boldsymbol{U})^{\mathrm{T}} \right], \tag{20}$$

where ν is the kinematic viscosity (dynamic viscosity divided by the density).

The manufactured solution chosen for this case study is known in the literature as the Kovasznay Flow [20], given by:

$$\begin{aligned} u(x, y) &= 1 - \mathrm{e}^{\lambda x} \cos(2\pi y), \\ v(x, y) &= \frac{\lambda}{2\pi} \mathrm{e}^{\lambda x} \sin(2\pi y), \\ p(x, y) &= \frac{1}{2} \left(1 - \mathrm{e}^{2\lambda x}\right), \\ \lambda &= \frac{Re}{2} - \sqrt{\frac{Re^2}{4} + 4\pi^2}, \end{aligned} \tag{21}$$

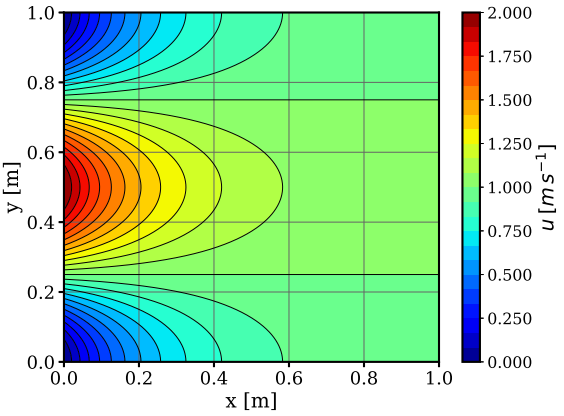
where Re is the flow Reynolds number.

For this test case study, kinematic viscosity of $\nu = 0.01$ $\mathrm{m^2 s^{-1}}$ is assumed. In this flow, the velocity satisfies the null divergence condition and, for $Re = 5$, the resulting velocity and kinematic pressure fields are depicted in Figure 4.

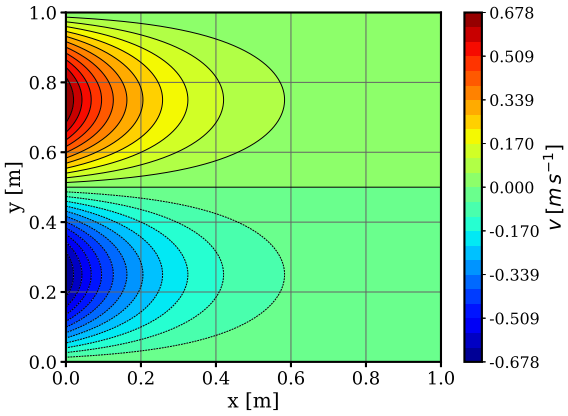
The code used to generate the source term, the boundary conditions and the `coded functionObjects` for the computation of the error norms, is shown in Listing 6.

From lines 6 to 11, the Reynolds number and the manufactured solution for the velocity components and pressure fields are defined. In lines 13 and 18, the velocity vector and stress tensor are calculated, respectively, and, in line 20, the momentum source term is computed. The advantages of using the *pyMMSFoam* package are clearly evidenced in such a convoluted manufactured solution, since it allows to avoid lengthy handmade and error-prone calculations. Finally, from lines 23 to 36, the `fvOptions` dictionary, boundary condition functions and `coded functionObject` are generated the OpenFOAM® test case study.

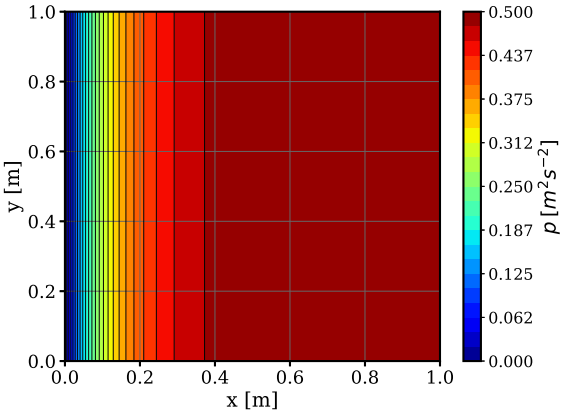
For this test case study, Dirichlet boundary conditions are prescribed for the velocity on the `left`, `top` and `bottom` patches of the domain, and for the pressure on the `right` patch. Moreover, Neumann boundary conditions are prescribed for the velocity on the `right` patch of the domain and for the pressure



(a) x-component of velocity.



(b) y-component of velocity.



(c) Kinematic pressure.

FIGURE 4. Manufactured solutions for the verification of the *simpleFoam* solver.

on `left`, `top` and `bottom` patches. The code required to setup these features in this case study are automatically generated by *pyMMSFoam*.

The temporal and spatial discretization schemes considered in this case study are presented in Table 3.

TABLE 3. Temporal and spatial discretization schemes for the verification of the *simpleFoam* solver (*fvSchemes* dictionary).

Parameter	Discretization scheme
ddtSchemes	steadyState
gradSchemes	Gauss linear
divSchemes	Gauss linear
laplacianSchemes	Gauss linear uncorrected
interpolationSchemes	linear
snGradSchemes	uncorrected

Is this test case the system of PDEs (18) and (19) is solved with the consistent version of the SIMPLE algorithm (SIMPLEC) [21]. For the resulting system of linear equations, a geometric algebraic multigrid method (GAMG) and a smooth solver were used to compute the approximate pressure and velocity fields, respectively. Table 4 reports the solution methods considered and their respective parameters, and Table 5

```
import sympy as sym
from sympy import sin, cos, exp, pi, sqrt
import pyMMSFoam as mms
from pyMMSFoam import x,y,z,t

Re = 5
Lambda = (Re/2) - sqrt( (Re**2/4) + 4*pi**2 )
u = 1 - exp(Lambda*x)*cos(2*pi*y)
v = (Lambda/(2*pi))*exp(Lambda*x)*sin(2*pi*y)
w = 0
p = 0.5*(1-exp(2*Lambda*x))

U = sym.Matrix([u,v,w])

# Momentum balance equation
nu = 0.01

R = nu*(mms.grad(U) + mms.grad(U).T)

S = mms.div(U*U.T) - mms.div(R) + mms.grad(p)

# Generate fvOptions
mms.generateFvOptions(S, "momentumSource", "U")

# Generate boundary conditions
# Velocity
mms.generateDirichletBoundaries(U, "U")
mms.generateNeumannBoundaries(U, "U")

# Pressure
mms.generateDirichletBoundaries(p, "p")
mms.generateNeumannBoundaries(p, "p")

# Generate functionObjects
mms.generateFunctionObject(U, "U")
mms.generateFunctionObject(p, "p")
```

LISTING 6. *simpleFoam* MMS script.

provides the `SIMPLE` sub-dictionary parameters. Additionally, a relaxation factor of 0.9 was employed for the momentum balance equation, to assure appropriate calculation stability. For the solution stopping criterion, a tolerance of 10^{-9} for the initial residual of both the velocity and pressure is considered. Notice that such tolerance must be sufficiently small to guarantee that the obtained computed solution errors correspond only to discretization errors.

TABLE 4. Solution methods for the verification of the *simpleFoam* solver (*fvSolution* dictionary).

Parameter	`p`	`U`
solver	GAMG	smoothSolver
smoother	DICGaussSeidel	symGaussSeidel
tolerance	10^{-11}	10^{-11}
relTol	0.01	0.1

TABLE 5. Parameters for the SIMPLEC calculation procedure of the *simpleFoam* solver (`SIMPLE` sub-dictionary from *fvSolution* dictionary).

Parameter	Value	
momentumPredictor	yes	
consistent	yes	
residualControl	`p`	10^{-9}
	`U`	10^{-9}

5. Results and Discussion

The results obtained from the test case studies presented in Section 4 to obtain the convergence order of the *laplacianFoam* and *simpleFoam* solvers are reported and discussed in the following subsections.

5.1. *laplacianFoam* test case studies.

5.1.1. *2D steady-state test case study.* For this 2D steady-state case study, the errors in the L^1-, L^2- and L^∞-norms, obtained for successively finer meshes are reported in Table 6. The associated convergence orders between two consecutive finer meshes are given in Table 7. An overall second-order of spatial convergence is obtained for the *laplacianFoam* solver, regardless of the error norm. Moreover, these results are in accordance with the theoretical convergence orders for the spatial discretization schemes employed, which supports that the numerical code is correctly implemented for the meshes type employed.

TABLE 6. Error norms obtained in the 2D steady-state test case study for the `laplacianFoam` solver.

Mesh	N_C	h	E^1	E^2	E^∞
Mesh 1	1,024	3.13×10^{-2}	3.33×10^{-2}	3.84×10^{-2}	6.66×10^{-2}
Mesh 2	4,096	1.56×10^{-2}	8.33×10^{-3}	9.59×10^{-3}	1.70×10^{-2}
Mesh 3	16,384	7.81×10^{-3}	2.08×10^{-3}	2.40×10^{-3}	4.31×10^{-3}
Mesh 4	65,536	3.91×10^{-3}	5.21×10^{-4}	6.00×10^{-4}	1.09×10^{-3}
Mesh 5	262,144	1.95×10^{-3}	1.30×10^{-4}	1.50×10^{-4}	2.73×10^{-4}

TABLE 7. Convergence orders obtained in the 2D steady-state test case study for the `laplacianFoam` solver.

Mesh	O^1	O^2	O^∞
Mesh 1 $\rightarrow$ *Mesh 2*	2.00	2.00	1.97
Mesh 2 $\rightarrow$ *Mesh 3*	2.00	2.00	1.98
Mesh 3 $\rightarrow$ *Mesh 4*	2.00	2.00	1.99
Mesh 4 $\rightarrow$ *Mesh 5*	2.00	2.00	1.99

5.1.2. *2D unsteady test case study.* For the 2D unsteady test case study, the computed solution error increases at every time-step due to the discretization in time, but the same theoretical convergence orders are expected regardless of the last simulated time. Therefore, for the solver verification and to assess the convergence order under mesh refinement, an appropriate time-step size should be determined

to avoid the computed solution error stagnating due to the temporal discretization error, as described in Section 2. Moreover, notice that the Euler discretization scheme [22] employed to discretize the time derivative is first-order accurate, whereas the spatial discretization schemes considered have a theoretical second-order of accuracy. Therefore, without a careful choice of the time-step size, the verification of the spatial convergence order in this unsteady test case might be inappropriate. Accordingly, for this test case, the value of the time-step size was iteratively decreased until observing that the computed solution error (considering only the reported significant digits) remain unchanged for all the employed meshes. In that regard, a time-step size of $\Delta t = 10^{-7}$ s was chosen and a final time of $t^{\mathrm{F}} = 3$ s was considered for the simulations.

The errors in the L^1-, L^2- and L^∞-norms obtained for successively finer meshes, just for the last calculated instant of time, are reported in Table 8, while the associated convergence orders between two consecutive finer meshes are given in Table 9. As observed, the *laplacianFoam* solver provides an overall second-order spatial convergence, regardless of the error norm. As in the previous case study, these results are in accordance with the theoretical convergence orders for the considered spatial discretization schemes. Moreover, the procedure employed for the choice of the time-step size was shown to be effective for the correct assessment of the spatial convergence order of the *laplacianFoam* solver.

TABLE 8. Error norms obtained in the 2D unsteady test case study for the `laplacianFoam` solver.

Mesh	N_{C}	h	E^1	E^2	E^∞
Mesh 1	1,024	3.13×10^{-2}	6.98×10^{-3}	1.26×10^{-2}	4.90×10^{-2}
Mesh 2	4,096	1.56×10^{-2}	1.76×10^{-3}	3.18×10^{-3}	1.30×10^{-2}
Mesh 3	16,384	7.81×10^{-3}	4.42×10^{-4}	7.97×10^{-4}	3.34×10^{-3}
Mesh 4	65,536	3.91×10^{-3}	1.11×10^{-4}	1.99×10^{-4}	8.44×10^{-4}
Mesh 5	262,144	1.95×10^{-3}	3.09×10^{-5}	5.04×10^{-5}	2.12×10^{-4}

TABLE 9. Convergence orders obtained in the 2D unsteady test case study for the `laplacianFoam` solver.

Mesh	O^1	O^2	O^∞
Mesh 1 $\rightarrow$ *Mesh 2*	1.98	1.99	1.91
Mesh 2 $\rightarrow$ *Mesh 3*	2.00	2.00	1.96
Mesh 3 $\rightarrow$ *Mesh 4*	2.00	2.00	1.98
Mesh 4 $\rightarrow$ *Mesh 5*	1.84	1.98	1.99

5.1.3. *1D unsteady test case study.* Contrarily to the previous, successively smaller time-step sizes, and a fixed mesh characteristic size, are considered in this test case study, to assess the temporal convergence order of the *laplacianFoam* solver. For that purpose, an appropriate mesh characteristic size was iteratively determined following the procedure described in Section 2, to avoid the computed solution error stagnating due to the spatial discretization error. The chosen mesh consists of $8,192$ cells along the x-direction.

The errors in the L^1-, L^2- and L^∞-norms obtained for successively smaller time-step sizes, for the last calculated instant of time, are reported in Table 10, while the associated convergence orders between two consecutive smaller time-step sizes are given in Table 11. As expected, the first-order of convergence is generally achieved, for all error norms. This emphasizes the simplicity and practicality of the proposed semi-automatic approach to assess the convergence order of a solver in OpenFOAM® and to detect possible numerical issues or implementation inconsistencies.

TABLE 10. Errors obtained in the 1D unsteady test case study for the `laplacianFoam` solver.

N_C	Δt [s]	E^1	E^2	E^∞
8,192	5×10^{-3}	8.82×10^{-3}	9.17×10^{-3}	1.10×10^{-2}
	5×10^{-4}	8.82×10^{-4}	9.17×10^{-4}	1.10×10^{-3}
	5×10^{-5}	8.81×10^{-5}	9.17×10^{-5}	1.10×10^{-4}
	5×10^{-6}	8.76×10^{-6}	9.15×10^{-6}	1.10×10^{-5}

TABLE 11. Convergence orders obtained in the 1D unsteady test case study for the `laplacianFoam` solver.

Δt [s]	O^1	O^2	O^∞
$5 \times 10^{-3} \rightarrow 5 \times 10^{-4}$	1.00	1.00	1.00
$5 \times 10^{-4} \rightarrow 5 \times 10^{-5}$	1.00	1.00	1.00
$5 \times 10^{-5} \rightarrow 5 \times 10^{-6}$	1.00	1.00	1.00

5.2. ***simpleFoam* test case study.** For the sake of simplicity, only the errors in the L^2-norm are presented in this test case study, to verify the convergence order of the *simpleFoam* solver for both velocity vector components and pressure fields. In that regard, the errors in the L^2-norm obtained for successively finer meshes are reported in Table 12, while the associated convergence orders between two consecutive finer meshes are given in Table 13. For the velocity, the second-order of convergence is achieved for both components, although more consistently for v, whereas for u the convergence order is slightly below with coarser grids. For the pressure, the second-order of convergence is obtained but it seems to slightly deteriorate as finer grids are considered. Indeed, due to the pressure-velocity coupling in the incompressible Navier-Stokes equations, discretization techniques might provide convergence orders below the theoretical values, which is in accordance with the information provided in the literature [4, 23, 24]. Therefore, it is important to identify the theoretically expected convergence orders, not only according to the employed discretization schemes but also considering the equations being solved. However, such a detailed analysis is out of the scope of the present work. Nevertheless, it can be easily analyzed with the support of the proposed semi-automatic approach.

TABLE 12. Error norms obtained for u, v, and p in the 2D steady-state test case study for the `simpleFoam` solver.

Mesh	N_C	h	u	v	p
Mesh 1	1,024	3.13×10^{-2}	6.04×10^{-2}	5.43×10^{-3}	2.29×10^{-2}
Mesh 2	4,096	1.56×10^{-2}	1.68×10^{-2}	1.34×10^{-3}	5.81×10^{-3}
Mesh 3	16,384	7.81×10^{-3}	4.48×10^{-3}	3.37×10^{-4}	1.55×10^{-3}
Mesh 4	65,536	3.91×10^{-3}	1.16×10^{-3}	8.43×10^{-5}	4.38×10^{-4}
Mesh 5	262,144	1.95×10^{-3}	2.97×10^{-4}	2.11×10^{-5}	1.31×10^{-4}

TABLE 13. Convergence orders obtained for u, v, and p in the 2D steady-state test case study for the `simpleFoam` solver.

Mesh	u	v	p
Mesh 1 $\rightarrow$ *Mesh 2*	1.85	2.01	1.98
Mesh 2 $\rightarrow$ *Mesh 3*	1.90	2.00	1.90
Mesh 3 $\rightarrow$ *Mesh 4*	1.95	2.00	1.82
Mesh 4 $\rightarrow$ *Mesh 5*	1.97	2.00	1.74

6. CONCLUSIONS

The verification of a numerical method and the associated implementation is an essential step in any solver development, to ensure that the implemented code is correctly solving the prescribed mathematical model. In that regard, assessing that the computed solution error converges with some order to the underlying exact solution, as the mesh characteristic size and/or time-step size decreases, is a necessary condition. In the general case, finding analytic solutions for such a verification process can be a cumbersome task, especially for complex geometries and/or mathematical models. For the more demanding cases, the method of manufactured solutions provides a practical, simple and versatile framework for the verification procedure of numerical codes (discretization schemes, solution methods, boundary conditions, material models, etc.). In the present work, a semi-automatic approach was proposed based on the method of manufactured solutions for the verification of solvers in OpenFOAM® .

For this aim, a *Python* package, *pyMMSFoam*, was developed to support the calculation of source terms, boundary conditions and functions to calculate the errors associated with the numerical method implemented to solve a prescribed mathematical model. Moreover, *pyMMSFoam* automatically generates the associated C code and case setup *dictionaries*, as well as `coded functionObjects` to compute the errors in different norms, allowing a less invasive, less error-prone and more flexible methodology for code verification. The application of the proposed approach was illustrated for the verification of the *laplacianFoam* and *simpleFoam* solvers of OpenFOAM® , with different test case studies, consisting of 1D and 2D, steady-state and unsteady problems, employing structured meshes. The approach proved to be very practical to apply and provides simple and versatile means of assessing the convergence orders of a numerical code. It is important to notice that consecutive lower residual tolerances for the solution

method should be used in the calculations until confirming that the computed solution is influenced only by the discretization schemes and not by the algorithm that solves the system of linear equations.

Finally, *pyMMSFoam* is open to the community and the authors welcome suggestions and improvements.

Acknowledgments

This work is funded by FEDER funds through the COMPETE 2020 Programme and National Funds through FCT (Portuguese Foundation for Science and Technology) under the PhD Grant 2020.07424.BD and projects UID-B/05256/ 2020, UID-P/05256/2020.

References

[1] P. J. Roache, *Verification and validation in computational science and engineering.* Hermosa Albuquerque, NM, 1998, vol. 895.

[2] W. L. Oberkampf and C. J. Roy, *Verification and validation in scientific computing.* Cambridge University Press, 2010.

[3] "Verification and validation," https://www.openfoam.com/documentation/guides/latest/doc/guide-verification-validation.html, accessed: 2021-08-20.

[4] K. Salari and P. Knupp, *Code Verification by the Method of Manufactured Solutions*, Sandia National Laboratories, Albuquerque, NM 87 185-0825, 2000.

[5] P. J. Roache, "Code verification by the method of manufactured solutions," *Journal of Fluids Engineering, Transactions of the ASME*, vol. 124, no. 1, pp. 4–10, 2002.

[6] C. J. Roy, C. C. Nelson, T. M. Smith, and C. C. Ober, "Verification of Euler/Navier-Stokes codes using the method of manufactured solutions," *International Journal for Numerical Methods in Fluids*, vol. 44, no. 6, pp. 599–620, 2004.

[7] B. Blais and F. Bertrand, "On the use of the method of manufactured solutions for the verification of CFD codes for the volume-averaged Navier-Stokes equations," *Computers and Fluids*, vol. 114, pp. 121–129, 2015.

[8] H. Noriega, F. Guibault, M. Reggio, and R. Magnan, "A case-study in open-source CFD code verification, Part I: Convergence rate loss diagnosis," *Mathematics and Computers in Simulation*, vol. 147, pp. 152–171, 2018.

[9] H. Noriega, F. Guibault, M. Reggio, and R. Magnan, "A case-study in open-source CFD code verification. Part II: Boundary condition non-orthogonal correction," *Mathematics and Computers in Simulation*, vol. 147, pp. 172–193, 2018.

[10] A. Choudhary, C. J. Roy, E. A. Luke, and S. P. Veluri, "Code verification of boundary conditions for compressible and incompressible computational fluid dynamics codes," *Computers & Fluids*, vol. 126, pp. 153–169, mar 2016.

[11] R. Fisch, J. Franke, R. Wüchner, and K.-U. Bletzinger, "Code verification examples of a fully geometrical nonlinear membrane element using the method of manufactured solutions," in *Textile Composites and Inflatable Structures VI - Proceedings of the 6th International Conference on Textile Composites and Inflatable Structures, Structures Membranes*, 2013, pp. 102–113.

[12] R. Fisch, J. Franke, R. Wüchner, and K. Bletzinger, "Code verification of a partitioned FSI environment for wind engineering applications using the method of manufactured solutions," *11th World Congress on Computational Mechanics, WCCM 2014, 5th European Conference on Computational Mechanics, ECCM 2014 and 6th European Conference on Computational Fluid Dynamics, ECFD 2014*, no. Wccm Xi, pp. 2186–2197, 2014.

[13] F. Habla, C. Fernandes, M. Maier, L. Densky, L. Ferrás, A. Rajkumar, O. Carneiro, O. Hinrichsen, and J. M. Nóbrega, "Development and validation of a model for the temperature distribution in the extrusion calibration stage," *Applied Thermal Engineering*, vol. 100, pp. 538–552, 2016.

[14] S. Radman, C. Fiorina, and A. Pautz, "Development of a novel two-phase flow solver for nuclear reactor analysis: algorithms, verification and implementation in OpenFOAM," *Nuclear Engineering and Design*, vol. 379, p. 111178, 2021.

[15] M. Hassanaly, H. Koo, C. F. Lietz, S. T. Chong, and V. Raman, "A minimally-dissipative low-Mach number solver for complex reacting flows in OpenFOAM," *Computers and Fluids*, vol. 162, pp. 11–25, 2018.

[16] "*Sympy*," https://www.sympy.org/en/index.html, accessed: 2021-08-20.

[17] "*laplacianFoam*," https://www.openfoam.com/documentation/guides/latest/doc/guide-applications-solvers-basic-laplacianFoam.html, accessed: 2021-08-20.

[18] "*simpleFoam*," https://www.openfoam.com/documentation/guides/latest/doc/guide-applications-solvers-incompressible-simpleFoam.html, accessed: 2021-08-20.

[19] "Sympy's common subexpression elimination," https://docs.sympy.org/latest/modules/simplify/simplify.html?highlight=cse#module-sympy.simplify.cse_main, accessed: 2021-08-20.

[20] L. Kovasznay, "Laminar flow behind a two-dimensional grid," in *Mathematical Proceedings of the Cambridge Philosophical Society*, vol. 44, no. 1. Cambridge University Press, 1948, pp. 58–62.

[21] "*Solution and algorithm control*," https://www.openfoam.com/documentation/user-guide/6-solving/6.3-solution-and-algorithm-control, accessed: 2021-08-20.

[22] "Euler scheme," https://www.openfoam.com/documentation/guides/latest/doc/guide-schemes-time-euler.html, accessed: 2021-08-20.

[23] R. Costa, S. Clain, G. J. Machado, and R. Loubère, "A very high-order accurate staggered finite volume scheme for the stationary incompressible Navier–Stokes and Euler equations on unstructured meshes," *Journal of Scientific Computing*, vol. 71, no. 3, pp. 1375–1411, 2017.

[24] R. Costa, S. Clain, and G. J. Machado, "A sixth-order finite volume scheme for the steady-state incompressible Stokes equations on staggered unstructured meshes," *Journal of Computational Physics*, vol. 349, pp. 501–527, 2017.