OpenFOAM® Journal

Volume 3 [Technical Notes], Pages 120–145
ISSN: 2753-8168

Modular Framework for the Solution of Boundary-Coupled Multiphysics Problems

Gabriel St-Onge and Mathieu Olivier*

Département de génie mécanique, Université Laval, Québec, QC, Canada, G1V 0A6
Email address: mathieu.olivier@gmc.ulaval.ca

DOI: https://doi.org/10.51560/ofj.v3.64
Results with version(s): OpenFOAM® v2006
Repository: https://github.com/molivier19/BC-multiphysics

Abstract. This paper presents a modular multiphysics framework developed for OpenFOAM®. The framework is built around an iterative implicit coupling scheme based on a multi-region partitioned approach. This scheme allows the implementation of formal implicit time-marching schemes, which improves the stability of strongly interacting coupled problems. This methodology allows physical interactions to be handled through specifically designed interface boundary conditions. It also allows region-specific solvers to be implemented as modular class solvers. The coupling methodology is handled with a main program that manages solver-specific actions. This framework aims to facilitate the implementation and testing of new multiphysics coupling problems in an integrated code structure. To show the capabilities of the framework to integrate new physics, solvers and boundary conditions requirements are discussed. Also, three validated examples involving fluid-structure interactions, conjugate heat transfer, and fluid-structure-thermal interactions are presented. Although all these problems are boundary-coupled multiphysics problems, the framework is conceptually not limited to this kind of problem. The benefit of this work to the OpenFOAM® community is a general and modular framework that facilitates the setup and solution of diversified multiphysics problems, and that illustrates the implementation of modular interface boundary conditions between physics regions.

1. Introduction

Multiphysics problems are complex and varied. To solve these kinds of coupled problems, multiple equations need to be solved together. For example, Fluid-Structure Interactions (FSI) are a particular multiphysics type of problem where a flexible solid structure is interacting with the fluid surrounding it. Thus, to solve FSI problems, governing equations of elastic solids need to be coupled with those governing the fluid flow. To solve these two sets of equations, a coupling scheme is used to make good predictions of the physical phenomena involved. In the literature, several algorithms have been designed to solve such coupled problems. These approaches are generally classified into two categories: the monolithic approach and the partitioned approach. Also, to get accurate and numerically stable predictions of multiphysics problems, special attention needs to be given to the physical strength of the multiphysics interaction (weak or strong), but also to numerical aspects such as the chosen temporal integration schemes (explicit or implicit) and the coupling scheme itself (explicit or implicit).

In the partitioned approach, the set of equations of each physical region is solved with its own numerical solver. The interaction between multiple regions is handled either through boundary conditions or through source terms implemented within the corresponding solvers. Thus, with this approach, state-of-the-art solvers can be used mostly as black boxes to handle different physics. Though this approach is relatively easy to implement, it can lead to stability issues such as those experienced in FSI problems [1]. For weak physical interactions, the use of simple non-iterative partitioned coupling schemes can lead to stable simulations. However, for strong physical interactions, where all physical regions have a strong influence on each other, stability issues will appear. In those cases, an outer iterative loop should be used to allow the implementation of implicit schemes and, if needed, stabilization techniques such as solution under-relaxation.

* Corresponding author

Received: 6 October 2021, Accepted: 2 June 2023, Published: 31 July 2023

In the monolithic approach, the multiphysics simulation is solved as a single problem. The discretized equations are assembled in a single linear system which also includes the interface equilibrium conditions. As opposed to the partitioned approach, this method exhibits fewer stability issues with stronger physical interactions and the computational time required to obtain a solution may, in some cases, be better than with the partitioned approach. However, developing a general-purpose monolithic multiphysics code is much more involved because existing solvers cannot be used without being extensively modified or rewritten. Moreover, while the stability problems encountered with the partitioned approach can be mitigated with the monolithic approach, this comes at the price of matrix structure complexity. Devising efficient and general preconditioners in this context may be quite challenging as well [2].

When solving unsteady physics, the temporal scheme usually refers to a time-marching procedure in which the temporal derivatives are discretized with either implicit or explicit schemes. On the other hand, the coupling scheme refers to the way multiple physical regions are numerically coupled together. This includes the interpolation of physical data between regions, but also, and most importantly, the sequence with which the data is transferred from one region to another. Therefore, in the partitioned approach, there is a distinction between the time-marching scheme used by individual solvers and the sequence of the whole coupling scheme itself.

To better illustrate this, consider a fluid-structure interaction problem involving the coupling of a structural solver with a flow solver. Both solvers have their respective time-marching schemes and need to exchange information on the fluid-solid interface through the boundary conditions. Typically, the flow solver needs the structural displacement field, and the structural solver needs the load (pressure and shear) coming from the flow. If implicit time-marching schemes are used in both solvers to obtain the solution at time level $n + 1$, implicit coupling is achieved if the data transferred at the interface is taken at the same time level, hence $n + 1$. In a partitioned method, this necessarily requires an outer iterative procedure. On the other hand, even though implicit time schemes are used in the physics solvers, it is possible to use interface coupling data that lags one step behind (i.e., at time level n). In this case, the coupling scheme is explicit and no outer iteration is needed. It can also be argued that the time-marching schemes used by individual solvers are not formally implicit anymore since at least some boundary terms are then explicit. Moreover, in such a strategy, special attention is needed to ensure proper precision of the time-marching schemes [3]. Lastly, explicit time-marching schemes or different explicit-implicit combinations thereof can be used when coupling different physics regions. In these cases, the whole coupling scheme cannot be fully implicit since some parts of the scheme are inherently explicit. However, these scheme combinations do not require an outer loop to be consistent.

Multiphysics simulation capabilities are now quite common in many CFD codes and the open-source software OpenFOAM® makes no exception. In version v2012, a new multi-application coupling procedure was proposed [4]. This approach uses a coupling scheme based on a parallel partitioned approach where application-based solvers are coupled through mapped boundary conditions. Also, the coupling between regions is managed through an MPI communicator such that application solvers are being kept mostly unchanged from previous OpenFOAM® versions. At the moment, this feature is still being developed and only a few simple examples are provided [4].

In addition to the capabilities of the current OpenFOAM® release, some external tools were also developed by the community to solve complex multiphysics problems. For example, the solids4foam [5] framework was developed with a focus on solid mechanics and FSI problems. In this framework, the coupling scheme used in FSI problems is based on a partitioned approach that implements both explicit and implicit coupling schemes. The framework includes multiple incompressible flow and structural models implemented in an object-oriented manner where physical models resolving similar physics are inherited from the same base class. The implementation of new incompressible flow and structural models is thus fairly easy.

For more versatile multiphysics capabilities, third-party software like preCICE may be considered. The preCICE library creates a multiphysics simulation environment where existing single-physics solvers can be coupled together to solve complex multiphysics problems [6,7]. By using third-party software, one can take advantage of the strength of different simulation tools to improve multiphysics simulations. The coupling of the different physics is done with specially designed adapters, and the software implements explicit and implicit coupling schemes. Time interpolation and data mapping methods are also implemented to ensure compatibility between the different physics involved. Furthermore, preCICE adapters are already compatible with the OpenFOAM® framework. Thus, FSI problems and Conjugate Heat Transfer (CHT) problems can be solved readily with the current release of the software.

In this technical note, a modular multiphysics framework based on a multi-region partitioned approach is presented. The proposed framework uses an iterative procedure in which solver objects are used to

solve multi-region problems. As such, the implementation is conceptually similar to that of solids4foam, but it goes further in terms of modularity: the physics solver base class has been generalized to be independent of the physics such that there are no restrictions on the physics implemented within the inherited classes. Moreover, the coupling between different physical regions is handled through dedicated interface boundary conditions classes. The purpose of this paper is thus to provide implementation guidelines and working examples of this multiphysics simulation framework implemented within OpenFOAM®. Different coupling capabilities such as FSI, Fluid-Structure-Thermal Interaction (FSTI), and CHT have been integrated. The implementation of these capabilities is presented in Section 2, and validated examples illustrating these capabilities are presented in Section 3. Among these examples, we demonstrate the use of artificial compressibility stabilization for FSI through a dedicated interface boundary condition, the implementation of a mixed interface boundary condition for CHT applications, and the integration of the HiSA solver [8] to solve FSTI problems involving supersonic flows. Although not presented in the paper, the framework has also been used for problems involving FSI in which an in-house finite-element structure code was used [9, 10]. Solutions to problems with strong fluid-body interactions were also obtained with the framework by implementing quasi-Newton methods (variants of Broyden's method) [11, 12]. Hence, this paper contributes to providing new multiphysics coupling capabilities and guidelines to the community. Some ideas provided herein could also be considered along with recent developments (v2012) regarding multi-application coupling.

2. Methodology

As mentioned in the introduction, the modular multiphysics framework is developed around a partitioned coupling approach. That is, even if all physics solvers are implemented within OpenFOAM®, they are treated independently as black boxes in an algorithmic perspective. Since this framework is intended to support implicit temporal discretization schemes, all solvers involved in the solution need to achieve iterative convergence at each time step by considering the effects of the neighbor regions. Hence, the coupling algorithm is built around a main multi-region loop, which is implemented in the main application program. As for the multi-region interactions, they are implemented through the boundary conditions since this work focuses on boundary-coupled problems. However, the proposed framework could also be used for problems involving multiphysics coupling within a single physical region. In such cases, virtual regions (sharing the same physical space) could interact through source terms in the governing equations of physics solvers.

Algorithm 1 illustrates how the coupling process can be interpreted as a fixed-point iterative procedure for a typical case where multiple regions interact. In this pseudo-algorithm, d_I could be the boundary field that is transferred by the interface boundary condition (e.g.: displacement, heat flux, traction force). The solution process of each region is represented by R_0, R_1, and R_N. The order in which these solution processes are used also corresponds to the solution sequence of the multiphysics problem that is being solved. For example, in FSI problems the region R_0 could correspond to the solid region and the region R_1 could correspond to the fluid region. In this specific example, the fluid solver R_1 would return the traction on the interface boundary to the solid solver R_0. The field information at the interface would then be updated by the solid solver for the next iteration. In this algorithm, the parameter w is an example of an under-relaxation coefficient that can be implemented in cases involving strong coupling between regions. Its implementation can be done directly within specific solvers, or through boundary conditions, if needed.

Algorithm 1: Partitioned implicit coupling scheme combined as a fixed point iterative procedure.

d_I: boundary field;
R_i: solution process in region i;
N: number of time steps;
r: residual;
w: under-relaxation coefficient;
for n=0, n<N **do**
  while r>*tolerance* **do**
    $d_I^* = R_0(R_1(...R_N(d_{I,i}^{n+1})))$;
    $d_{I,i+1}^{n+1} = d_{I,i}^{n+1} + w(d_I^* - d_{I,i}^{n+1})$;
    $r = \left\| d_{I,i+1}^{n+1} - d_{I,i}^{n+1} \right\|$;

The multiphysics algorithm consists of three main components. These components are respectively the region-specific solvers, the interface boundary conditions, and the coupling algorithm. These three components and their implementation in the framework are discussed independently in the next subsections.

2.1. **Solvers.** The coupled partitioned scheme in this framework is taking advantage of standard solvers to solve each region. However, these solvers need to be compatible with the coupling procedure. Thus, to standardize the coupling process, an abstract base class was developed. With this base class, any solver can be organized into key components, and then be implemented as an inherited class. The base class is called `physicsSolver` and is presented in Fig. 1. This class diagram presents all the virtual and pure

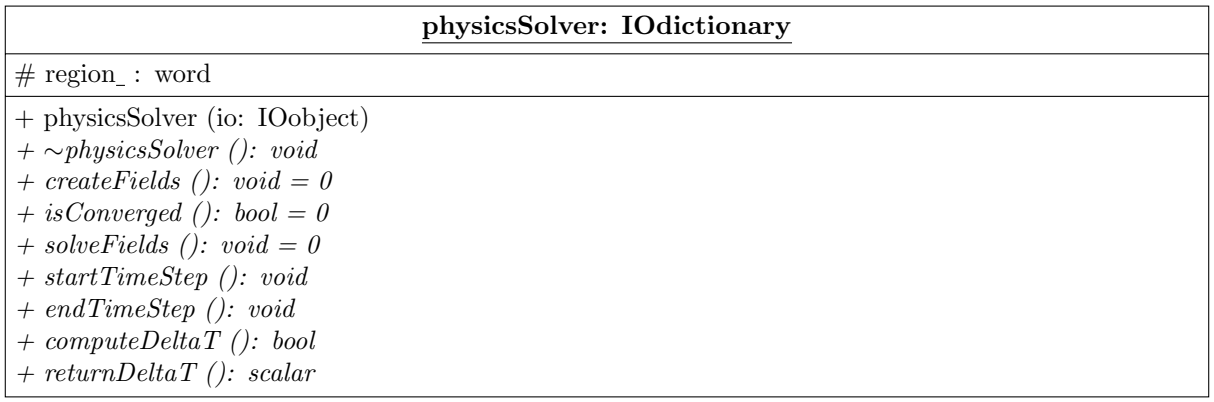


Figure 1. `physicsSolver` abstract class diagram.

virtual functions that need to be implemented in each solver to comply with the coupling procedure. Also, these virtual functions correspond to specific steps of the simulation process. It is then possible to adapt standard and custom OpenFOAM® solvers to fit these requirements. It is also important to mention that the abstract class is independent of any physical parameter. Thus, the same base class can be used to implement fluid flow solvers as well as other types of solvers like structural solvers, thermal conduction solvers, etc. To implement a physics solver in the framework, a new class inherited from the `physicsSolver` base class needs to be created. The implementation of new physics solvers can be achieved by following a few guidelines. The specific implementation steps are presented in Appendix A.1.

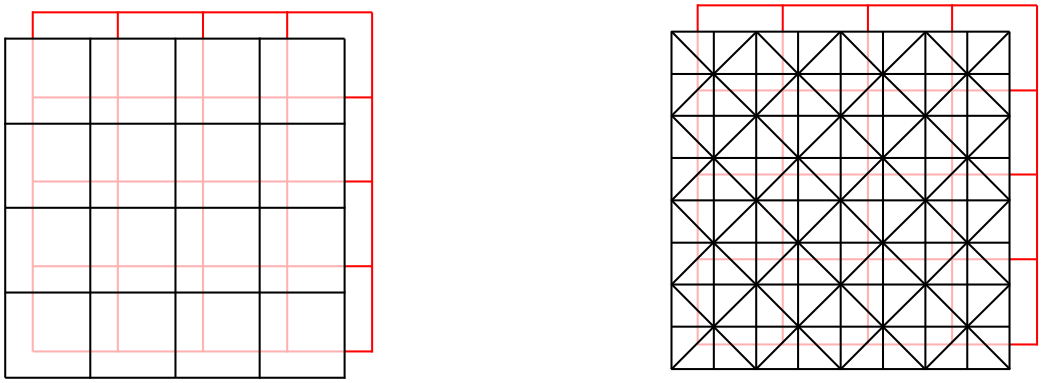
Five physics solvers are provided with this paper. A brief description of these solvers' capabilities is listed below:

- `fluidphysicsSolver` is an incompressible flow solver based on the PIMPLE algorithm.
- `HiSAphysicsSolver` is a compressible flow solver that uses the AUSM (Advective Upstream Splitting Method) or the HLLC (Harten-Lax-van Leer-Contact) scheme family to solve supersonic and hypersonic flow problems. This solver is an adaptation of the HiSA solver released by Heyns *et al.* [8].
- `rhoCentralDyMFluidphysicsSolver` is a compressible flow solver that uses the Kurganov and Tadmor central scheme [13, 14] to solve supersonic and hypersonic flow problems. This solver is based on the `rhoCentralDyMFoam` solver, which is part of the standard OpenFOAM® release.
- `solidElasticThermalphysicsSolver` is a structural thermo-elastic solver based on a Lagrangian formulation. The material is described using the Duhamel-Neumann model presented in the reference book of Malvern [15]. The solver can be configured to solve only the temperature field, the displacement field, or both.
- `solidStVKphysicsSolver` is a geometrically nonlinear structural solver based on a total Lagrangian formulation. The material is described using the Saint Venant-Kirchhoff constitutive law [15]. This solver can be used for the solution of elastic solids undergoing large displacements.
- `solidThermalStVKphysicsSolver` is a geometrically nonlinear thermoelastic solver based on a total Lagrangian formulation. The material is described using the Saint Venant-Kirchhoff constitutive law [15] with the addition of thermal dilatation in the model. This solver can be used for the solution of thermoelastic solids undergoing large displacements.

2.2. **Interface boundary conditions.** Boundary conditions are an important aspect of the coupling procedure used in multi-region problems. These conditions allow the transfer of information between regions and thus the interaction phenomena to be considered. Since most multi-region problems constitute

Mesh.pdf Mesh.pdf Mesh.pdf Mesh.pdf
Mesh.pdf

Conformal Mesh.pdf Conformal Mesh.pdf
Conformal Mesh.pdf Conformal Mesh.pdf
Conformal Mesh.pdf



(a) (b)

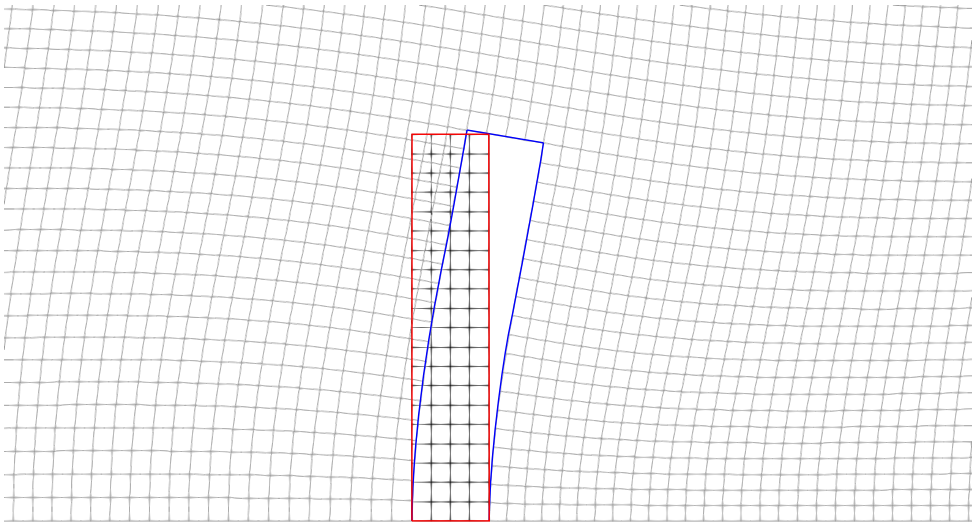
Figure 2. Illustration of different types of interfaces that can be used for multi-region problems: (a) conformal interface; and (b) non-conformal interface. The black and red meshes represent the interface mesh of their respective regions.

a continuum that combines all regions, some conservation conditions need to be satisfied at the interfaces between the different regions. In a partitioned approach, these conservation conditions can be imposed with a combination of Dirichlet and Neumann boundary conditions at the interface. In such combination, the interface is treated as a Dirichlet condition in one region, and as a Neumann condition in the other. Alternatively, a combination of mixed (Robin) conditions on both sides of the interface can also be used.

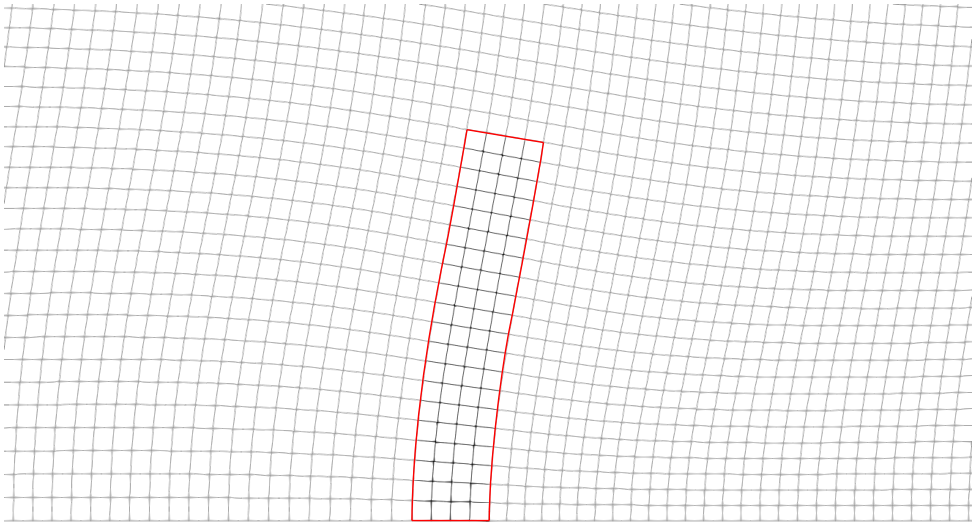
The OpenFOAM® software already provides a strong code base to create these interface boundary conditions. Also, the OpenFOAM® architecture requires the boundary conditions to be linked to their respective fields in their specific regions at run time. Thus, interface boundary conditions do not need to be hardcoded in the main application nor in the physics solvers, which allows them to be used in various scenarios involving different physics without requiring code modifications. This approach makes it easier to set up complex coupled problems with the same code. Also, it is possible to formulate boundary conditions to apply different types of under-relaxation, acceleration, or stabilization methods (an example is presented in Section 3.1). New interface boundary conditions can be implemented by following specific steps provided in Appendix A.2.

To ensure proper interaction between regions, some numerical aspects, such as mesh interpolation and mesh configuration, must be taken into account. Since multiphysics problems involve regions with different physics, each region mesh may differ from one to another in terms of resolution requirements. Hence, the mesh in a region interface can be either conformal or not (see Fig. 2). In the case of conformal meshes, the transfer of information is simplified, and a simple nearest-neighbor interpolation is usually adequate. On the other hand, when non-conformal meshes are involved, mesh nodes are not aligned between regions. Thus, the transfer of information is more complex, and more elaborate interpolation techniques are needed. Integration of mesh interpolation methods in new interface boundary conditions is discussed in Appendix A.2.

Another aspect to take into consideration when implementing multiphysics interface boundary conditions is the mesh configuration of each region. Indeed, different physics solvers may work with different mesh configurations leaving both sides of a given multiphysics interface at different positions. To illustrate this, FSI problems involving large structural displacements can be solved by using a total Lagrangian formulation in the solid solver along with an arbitrary Lagrangian-Eulerian formulation in the fluid solver (this strategy is used in the examples of Section 3.1 and Section 3.3). In such cases, the solid mesh stays in the undeformed configuration while the fluid mesh moves and deforms to account for the solid motion, see Fig. 3. Thus, further attention is needed to ensure that both sides of the interface are interpolated correctly. These aspects are also discussed in Appendix A.2.



(a)



(b)

Figure 3. Examples of meshes for FSI simulation: (a) the fluid (Arbitrary Lagrangian-Eulerian, ALE) and the solid (total Lagrangian) are in different configurations; and (b) the solid mesh has been moved so both regions are in the same configuration before transfer FSI conditions.

At the time of writing this paper, four interface boundary condition combinations are proposed in the multiphysics framework. A short description of these interface boundary conditions is listed below. The first two use custom boundary conditions, while the last two use standard OpenFOAM® boundary conditions.

- `fsiInterfaceDisplacement` and `fsiInterfaceTraction` are used together as FSI interface boundary conditions based on the Dirichlet-Neumann formulation. The first condition ensures the continuity of the displacement field by moving the interface of the fluid region according to the displacement of the corresponding displacement field in the solid region. The second condition ensures that the momentum equation is respected at the interface by transferring the local forces from the fluid region to the solid region. These interface boundary conditions assume a total Lagrangian formulation in the solid region and an Arbitrary Lagrangian-Eulerian (ALE) formulation in the fluid region.

- `fstiTemperature` and `fstiHeatFlux` are used together as CHT interface boundary conditions based on the Dirichlet-Neumann formulation. The first condition ensures the continuity of the temperature field, while the second one ensures the continuity of the heat flux at the interface. These interface boundary conditions were developed assuming a total Lagrangian formulation in both domains.
- `turbulentTemperatureCoupledBaffleMixed` is a CHT interface boundary condition based on the mixed formulation. For this boundary condition, the continuity of the temperature field and the heat flux are enforced by the same class in both regions. For this boundary condition to be consistent, both meshes should be in the same configuration.
- `turbulentTemperatureRadCoupledMixed` is a CHT interface boundary condition based on the mixed formulation. As for the last interface boundary condition, both the continuity of the temperature field and of the CHT are enforced by the same class. However, this condition also considers radiation. Here again, for this boundary condition to be consistent, both meshes should be in the same configuration.

The proposed multiphysics framework builds upon the modularity of OpenFOAM®, which allows end users to easily implement new boundary conditions. In this context, implementing new interface boundary conditions using either the Dirichlet-Neumann formulation or the mixed formulation is fairly straightforward, and it can be done without modifying physics solvers. However, special attention to mesh conformity and configuration (e.g.: total Lagrangian vs. ALE) is needed to ensure proper interpolation between meshes.

2.3. **Coupling scheme.** The whole coupling procedure is implemented within the main program called `multiFieldSolver`, which is the application that is used to run multiphysics simulations. This procedure is shown in Fig. 4 and is divided into five steps. Each block illustrated within these steps corresponds to a member function defined in the solver classes. These functions are discussed in detail in Appendix A.1 along with some implementation guidelines.

All multiphysics simulations begin with the **Initialization** step where solver objects are created, and physical fields are initialized for all regions involved. Each specific solver object is associated with each region at runtime based on user input files, thanks to the selection mechanism implemented in the constructor of the base class `physicsSolver`. Thus, no hard coding of the type of physics is necessary in the main program. After the initialization step, the program enters the **Time loop**. This loop corresponds to the time marching procedure, which is the same for all regions (hence, no time step sub-cycling is implemented at the moment). The time loop is composed of four steps that are sequentially executed. The first step, which is labeled **Start** in Fig. 4 is executed at the beginning of each time step. This step only executes the `startTimeStep()` function for each region. This function is optional, and it allows some operations to be performed before the outer loop. The second step of the time loop, which is called **Adaptive** Δt in Fig. 4, allows adaptive time steps to be used. The execution of this step is also optional. If the adaptive time step option is activated in the *controlDict* file, the minimum time step for a given region is computed at run time with the `computeDeltaT()` function. For supported solvers, relevant quantities used for the computation of a new time step (e.g.: the maximum Courant number) are included in the *fvSolution* file of the corresponding region. The `returnDeltaT()` function is then used to return the requested time step for that region. In the multiphysics context, all solvers can eventually be configured to return a time step based on a given criterion, which means that multiple time step sizes are returned to the main program. Then, the smallest time step size from all regions is used as a target by the main program to adapt the actual time step. The next step is the **Outer loop** shown in Fig. 4, which corresponds to the iterative coupling scheme itself. In this loop, all fields from all regions are solved in sequence using the `solveFields()` function of each solver object. The actual coupling between the different regions occurs when the boundary conditions are updated within each solver, usually before the linear systems solution step. To ensure the formal implicitness of the time-marching scheme, the iterative convergence of the outer loop is monitored using the `isConverged()` function. This function typically monitors the initial residuals of each solver and compares them against stopping criteria. The outer loop stops once all solvers have reached their respective stopping criteria. Lastly, the `endTimeStep()` function allows additional operations to be performed after the outer loop if needed. As with the `startTimeStep()` function, it is optional, and the default function implemented in the base `physicsSolver` class does nothing.

The structure of the main program along with the base solver class provides a general template that allows users to solve different types of multiphysics problems. As illustrated in Fig. 4, the structure of the coupling algorithm is not limited by the number of regions to solve. Since the physical aspects of the simulation are independent of the coupling algorithm, the modular multiphysics framework presented

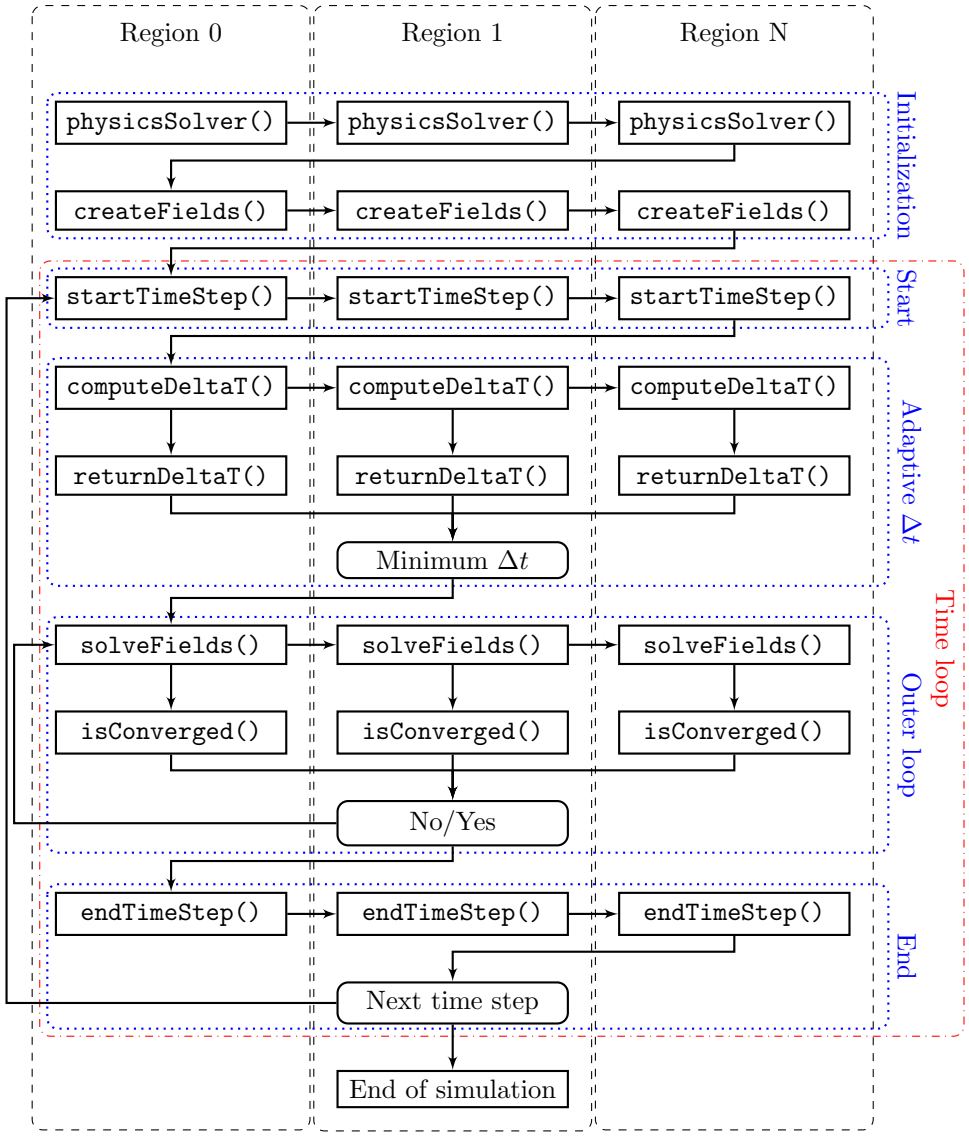


Figure 4. Process diagram of the multiphysics coupling algorithm. Rectangular blocks correspond to member functions that are implemented in each solver object.

in this paper is also not limited by the type of problem that can be solved. As discussed previously, to incorporate new physics in the framework, the implementation of a new solver class is needed as well as the implementation of new interface boundary conditions. These steps are presented in further detail in Appendix A.1 and Appendix A.2 respectively.

The proposed framework assumes that implicit time-marching schemes are used in all physics solvers. A formal implicit coupling scheme formulation is achieved through the outer loop. Using an implicit coupling scheme is key to mitigating constraints associated with stability conditions of explicit coupling schemes that are likely to be restrictive in multiphysics problems. Nevertheless, the use of explicit coupling schemes remains possible for the modular multiphysics framework in case they are suitable for a given problem. In this case, the number of iterations for the outer loop needs to be limited to one. Moreover, relaxation methods or acceleration techniques such as quasi-Newton coupling algorithms can be implemented on a per-solver basis or through interface boundary conditions [12].

3. Examples

To illustrate the capabilities of the proposed multiphysics framework, three examples are presented. These examples were chosen to show the versatility of the framework by illustrating a variety of physics such as FSI, FSTI, and CHT. In some cases, results obtained with the framework are also compared with known reference cases for validation purposes.

3.1. **Flexible tube.** The first example presented is an FSI simulation involving a flexible three-dimensional tube that is deformed through the action of a pressure wave. In this scenario, the flow strongly interacts with the flexible tube since the density ratio is near unity, which results in a strong added-mass effect. This example is a known reference case that has been presented in different papers [16–18]. Also, this is an interesting problem because such scenarios involving incompressible flows are known to be numerically unstable, especially when the mass ratio of the fluid over the solid is significant. To circumvent this problem, an artificial compressibility source term is added to the cell layer next to the fluid-solid interface in the fluid domain, as proposed in [16]. In the proposed implementation, this source term is added through the pressure boundary condition at the fluid-solid interface as discussed further below.

To solve this problem with the proposed framework, the fluid region is solved with the incompressible flow solver `fluidphysicsSolver`, and the solid region is solved with the geometrically nonlinear structural solver `solidStVKphysicsSolver`. The geometry of the flexible tube problem is composed of two regions that consist of concentric cylinders as shown in Fig. 5. The length of the tube is 0.05 m, the inner radius is $R_i = 0.005$ m, and the outer radius is $R_o = 0.006$ m. The density and dynamic viscosity of the fluid are given respectively by:

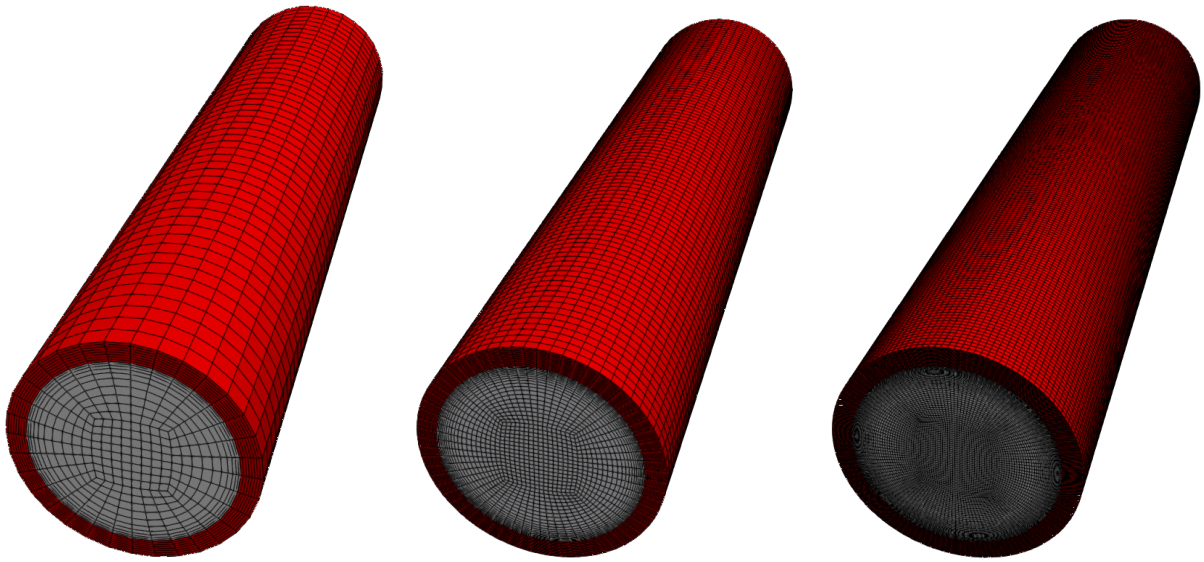
$$\rho_f = 1000 \text{ kg/m}^3, \quad \mu = 3 \times 10^{-3} \text{ kg/(m} \cdot \text{s)}, \tag{1}$$

while the density, Poissons ratio, and Young's modulus of the solid are respectively:

$$\rho_s = 1200 \text{ kg/m}^3, \quad \nu = 0.3, \quad E = 3 \times 10^5 \text{ Pa}, \tag{2}$$

where subscripts f and s are used to distinguish respectively fluid and solid properties when needed.

As for the boundary conditions, the tube is held fixed at both ends, and a zero normal stress condition is applied on the outer wall. A zero static pressure and a zero normal velocity gradient are applied at the outlet of the fluid domain. At the inlet, to initiate the flow, a total pressure pulse of 1333.2 kPa is imposed for the first 0.003 s of simulated time. The total pressure is set at 0 afterward and a zero normal gradient is used for the velocity field. For the interaction between the fluid and the solid to be considered, the velocity and the normal stress must be continuous at the interface. To this end, on the solid side, the `fsiInterfaceTraction` Neumann condition is used to retrieve the local pressure and shear forces from the fluid region and to apply the corresponding load on the solid structure. On the fluid side, the `fsiInterfaceDisplacement` boundary condition is used to retrieve the displacement field from the solid region and move the fluid mesh interface accordingly. The fluid mesh is then adapted with a displacement-based Laplaces equation, and the standard `movingWallVelocity` is used as a Dirichlet boundary condition on the velocity field. For the pressure field, a modified gradient condition called `fsiInterfacePressure` is used. This specific gradient condition is such that the linear



(a) (b) (c)

Figure 5. Meshes used for the grid convergence study of the flexible tube case: (a) coarse mesh; (b) medium mesh; and (c) fine mesh.

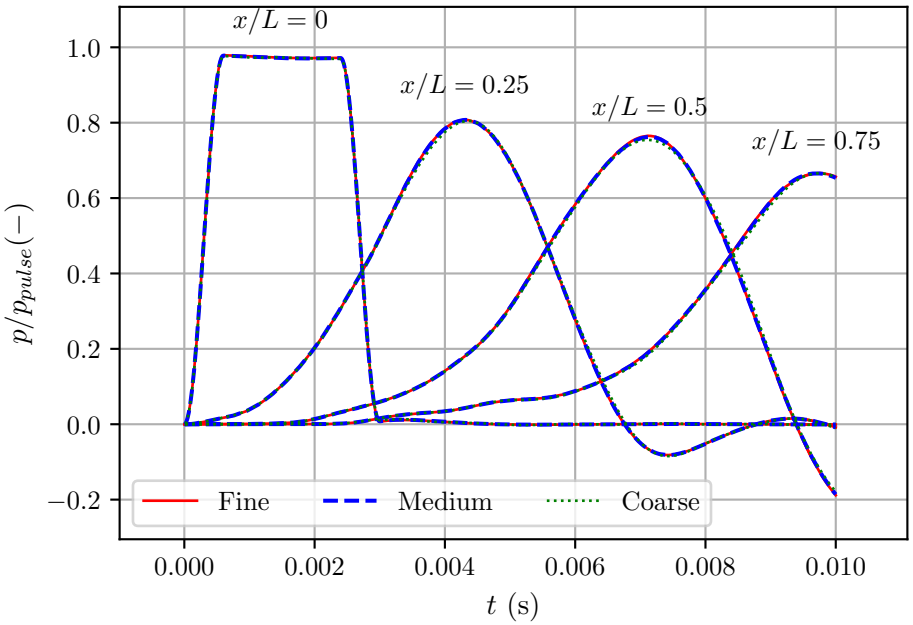


Figure 6. Grid convergence study using the p/p_{pulse} metric at four locations on the centreline of the flexible tube.

system coefficients have been modified to mimic the effect of an implicit artificial compressibility source term in the pressure equation in the cells next to the FSI interface. This is achieved by modifying the `gradientBoundaryCoeffs` and `gradientInternalCoeffs` methods accordingly. This modification ensures the stability of the iterative coupling procedure and tends towards a zero-gradient condition as the iterations converge. Hence, at convergence, the incompressible flow behavior is retrieved. From another perspective, this procedure can be interpreted as implicitly under-relaxing the pressure at the interface between successive iterations of the outer loop.

To validate the flexible tube results, a grid convergence study is presented. Three pairs of meshes of different resolutions are used. These meshes are shown in Fig. 5. The coarse meshes contain 19 968 cells in the fluid region and 13 312 cells in the solid region. The time step for this coarse simulation is set to 1.25×10^{-5} s. Systematic refinement with a factor of 2 is successively applied to obtain the medium (159 744/106 496 cells) and fine meshes (1 277 952/851 968 cells). Similarly, time steps are adjusted according to the mesh refinement factor ($\Delta t = 6.25 \times 10^{-6}$ s for the medium mesh and $\Delta t = 3.125 \times 10^{-6}$ s for the fine mesh). The results of these tests are presented using the dimensionless pressure (p/p_{pulse}) against time. This metric was monitored at four locations on the center line of the fluid region, and the results are presented in Fig. 6.

Moreover, the net volumetric influx is compared with the results from Degroote *et al.* [16]. Results are presented in Fig. 7. It can be observed that the results are quite similar in trends although a slight offset is observed. This offset may be explained by the following hypotheses considering the information provided in [16]:

- Degroote *et al.* [16] uses a first-order temporal scheme, a smaller cell count, and a larger time step.
- Degroote *et al.* [16] use a shell model for the solid region, which may give a slightly different response when compared to the 3D model used here.

3.2. **Heated cylinder.** The second example presented is a CHT simulation. This simulation involves a rigid stainless-steel cylinder that is heated by a hypersonic flow. In this scenario, the compressible flow interacts with the cylinder through conduction and convection; radiation effects are not considered. This example is a known reference case that has been presented in different papers [19–21].

For this problem, the fluid region is solved with the compressible flow solver called `HiSAphysicsSolver` which is an adaptation of the HiSA solver presented by Heyns *et al.* [8]. In addition, since the flow regime is expected to be turbulent, the Reynolds-averaged turbulence model k-ω-SST is used. The solid region is handled with the solver `solidElasticThermalphysicsSolver` which solves the thermoelastic equations.

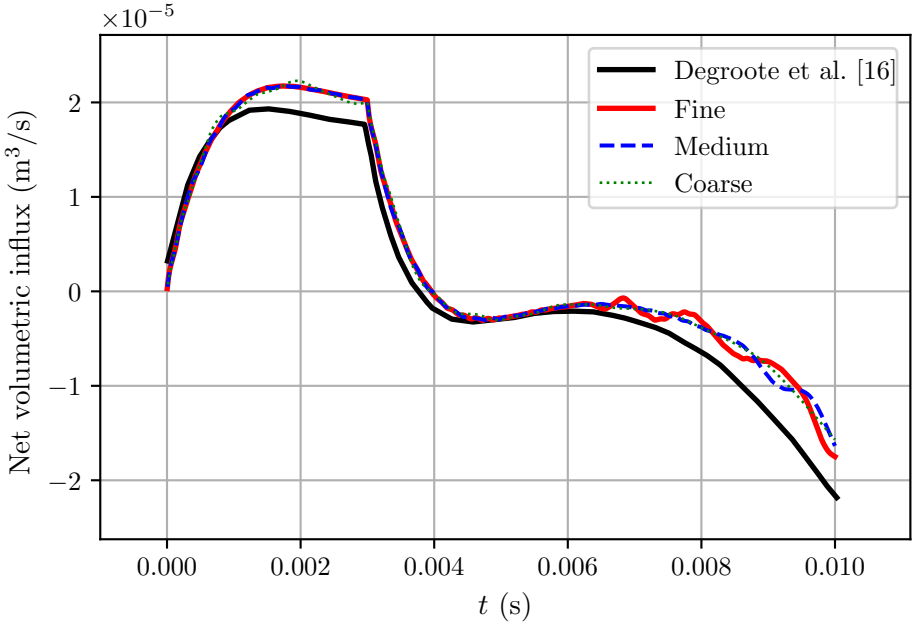


Figure 7. Comparison of the temporal evolution of the net volumetric influx in the flexible tube.

However, for this specific simulation, only the energy equation is used as the solid region is considered a rigid body. The geometry for the heated cylinder problem is composed of two regions. The fluid region is created from a rectangular box of length L and height H while the solid region is composed of an annular region of inner diameter D_i and outer diameter D_o. This geometry is shown in Fig. 8. Values for these geometric parameters are:

$$D_i = 0.0508 \text{ m}, \quad D_o = 0.0762 \text{ m}, \quad L = 15D_o, \quad H = 10D_o. \tag{3}$$

As for the physical properties, they are characterized by the heat capacity C_p, the thermal conductivity κ, the molar mass M, the dynamic viscosity μ, and the density ρ. The physical properties of the fluid are:

$$\begin{gathered} C_{p,f} = 1005 \text{ J}/(\text{kg} \cdot \text{K}), \quad \kappa_f = 9.18 \times 10^{-2} \text{ W}/(\text{kg} \cdot \text{K}), \quad \text{M} = 28.965 \text{ kg/mol}, \\ \mu = 6.58 \times 10^{-5} \text{ kg}/(\text{m} \cdot \text{s}), \end{gathered} \tag{4}$$

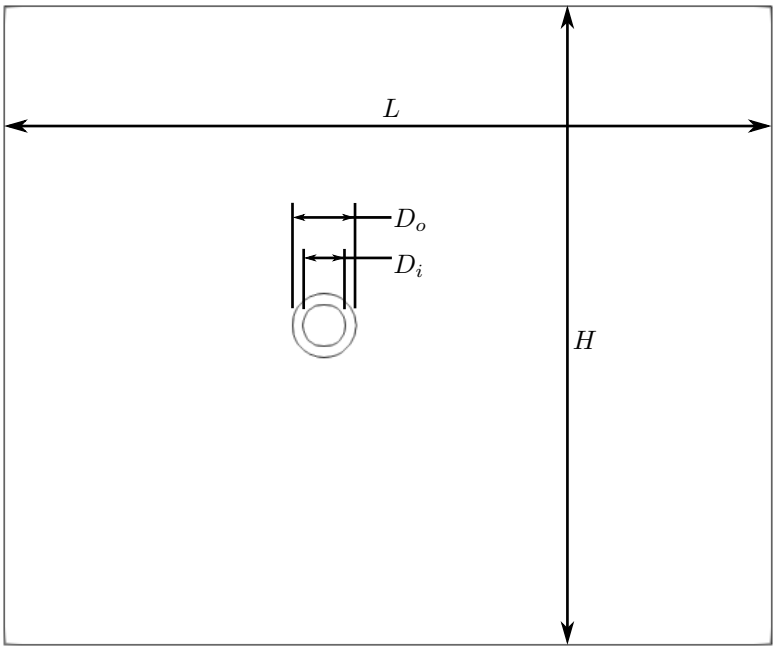
whereas those of the solid are:

$$C_{p,s} = 468 \text{ J}/(\text{kg} \cdot \text{K}), \quad \kappa_s = 13.4 \text{ W}/(\text{m} \cdot \text{K}), \quad \rho_s = 8\,238 \text{ kg/m}^3. \tag{5}$$

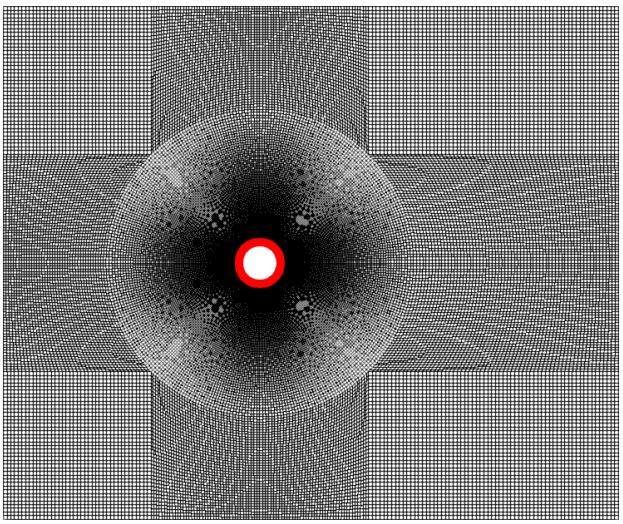
The far-field conditions correspond to a flow with a horizontal velocity of 2 246.54 m/s, a pressure of 101.3 kPa, and a temperature of 300 K. These flow conditions are used as initial conditions as well as on all the domain's outer boundaries by using so-called characteristic boundary conditions provided with the HiSA library. As for the interface boundary conditions, two combinations are tested. The first one is the mixed combination, which uses the `compressible::turbulentTemperatureCoupledBaffleMixed` condition on both sides of the interface, i.e., in the solid and in the fluid as well. The second combination is the Dirichlet-Neumann combination, which uses the `fstiTemperature` condition on the fluid side and the `fstiHeatFlux` condition on the solid side. These two interface conditions combinations should provide the same results although they are formulated differently.

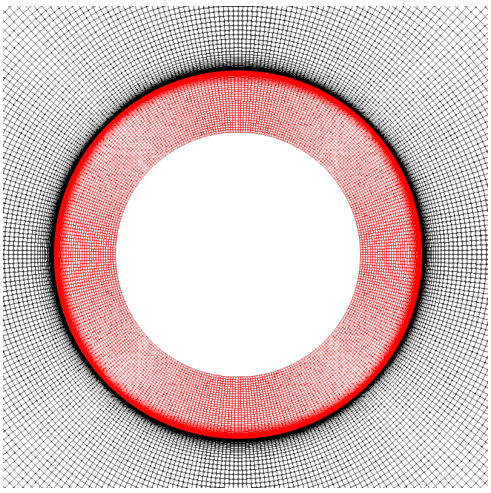
This problem is solved using the mesh presented in Fig. 8. The left figure presents a general picture of the mesh, while the right figure presents a close-up view of the mesh in the cylindrical region. The mesh is composed of hexahedral cells and the interface between the fluid and solid regions is conformal.

To validate the simulations, comparisons with results from the literature are presented. This heated cylinder scenario was first reported in a paper by Wieting [19] where experimental and theoretical results were presented. More recently, Hongpeng *et al.* [21] presented a numerical study where a coupling strategy was used to transfer the heat flux from the fluid region to the solid region. For the sake of this validation, the normalized heat fluxes (q/q_0 where q_0 is the initial heat flux at $\theta = 0$) through the cylinder's wall are compared with the experimental results of Wieting [19] (experimental run 37) and



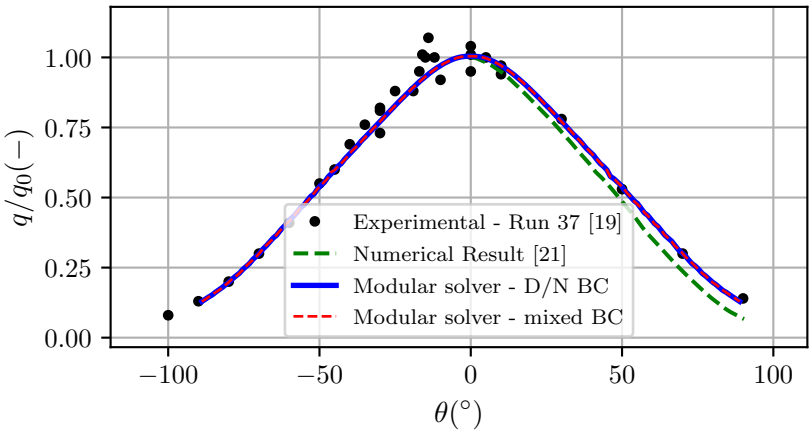
(a)



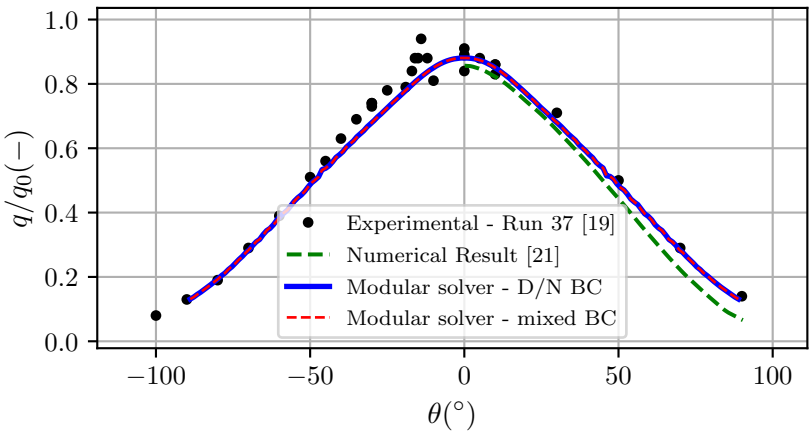


(b) (c)

Figure 8. Geometry and mesh used for the heated cylinder simulation. (a) overview of the geometry; (b) overview of the mesh; and (c) close-up view of the solid region mesh (red) and of the interface. The mesh contains approximately 61 600 cells in the fluid region and 19 200 in the solid region.



(a)



(b)

Figure 9. Heat flux at the interface of the heated cylinder compared with the results of Wieting [19] and Hongpeng *et al.* [21]: (a) heat flux at $t = 0.0002$ s; and (b) heat flux at $t = 0.0012$ s.

the corresponding numerical results of Hongpeng *et al.* [21] for two specific times after the onset of the experiment: $t = 0.0002$ s and $t = 0.0012$ s. Fig. 9 shows that the results obtained with the multiphysics framework are in good agreement with previously published results. Also, no significant differences can be observed between the simulation that uses the Dirichlet-Neumann interface boundary conditions and the mixed interface boundary conditions.

3.3. **Crossflow flexible plate.** The last example presented is an FSTI simulation. This simulation involves a flexible plate that is deformed and heated by a hypersonic flow. The flexible plate is deformed by the aerodynamic forces of the fluid, and it is also heated by the high temperature that is generated close to the solid region. This example is interesting because it involves large deformations of the solid region as well as thermal expansion. Similar scenarios are often used as reference cases [5, 6]. However, these problems usually involve incompressible flows and elastic structures. This example is intended to explore the modularity of the presented framework by coupling multiple physics in a single simulation.

To solve this problem with the multiphysics framework, the fluid region is solved with the compressible flow solver `HiSAphysicsSolver`. In addition, since the flow regime is expected to be turbulent, the Reynolds-averaged turbulence model k-ω-SST is used. The solid region is solved with the solid solver `solidThermalStVKphysicsSolver`. In this simulation, both the momentum and energy equations are coupled at the fluid-solid interface. The geometry of the crossflow flexible plate problem is composed of two regions. The fluid region is created from a rectangular box of length L and height H, and the solid region is composed of a rectangular section of height h and width w located at a distance l from the left

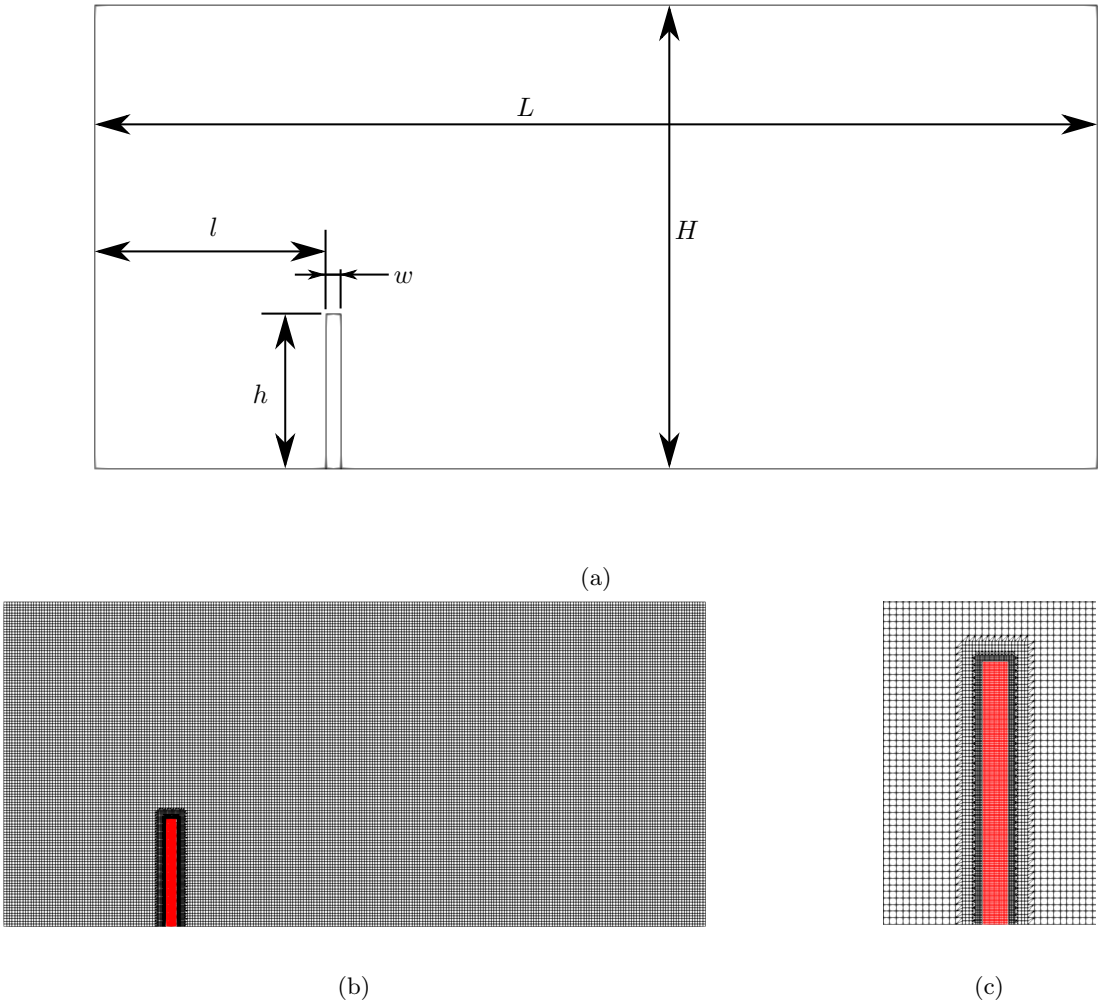


Figure 10. Geometry and mesh used for the crossflow flexible plate simulation: (a) overall view of the geometry; (b) overall view of the mesh; and (c) close-up view of the solid region mesh (red) and the interface.

side of the fluid region. This geometry is shown in Fig. 10, and values for these geometric parameters are:

$$L = 5 \text{ m}, \quad H = 3 \text{ m}, \quad l = 1.5 \text{ m}, \quad h = 1 \text{ m}, \quad w = 0.1 \text{ m} \tag{6}$$

The physical properties of the fluid are:

$$\begin{gathered} C_{p,f} = 1005 \text{ J/(kg} \cdot \text{K)}, \quad \kappa_f = 2.98 \times 10^{-2} \text{ W/(m} \cdot \text{K)}, \quad \text{M} = 28.96 \text{ kg/mol}, \\ \mu = 2.135 \times 10^{-5} \text{ kg/(m} \cdot \text{s)}, \end{gathered} \tag{7}$$

whereas the physical properties of the solid are:

$$\begin{gathered} C_{p,s} = 468 \text{ J/(kg} \cdot \text{K)}, \quad \kappa_s = 13.4 \text{ W/(m} \cdot \text{K)}, \quad \rho_s = 2\,692.8 \text{ kg/m}^3, \\ E = 67.32 \times 10^9 \text{ Pa}, \quad \alpha = 3.48 \times 10^{-6} \text{ 1/K}, \quad \nu = 0.3, \end{gathered} \tag{8}$$

where α is the thermal expansion coefficient, while the other symbols retain the same meaning as in previous cases. Note that, while the physical properties of the fluid correspond to air, those of the solid are arbitrary. They were chosen to illustrate a large deformation case with a relatively thick plate, which prevents the generation of stiff matrices. Far-field conditions for this problem consist of a velocity of 1 736.11 m/s, a temperature of 300 K, and a pressure of 101.3 kPa. These far-field conditions are used as initial conditions, and they are applied at the inlet and the outlet of the domain as well by using characteristic boundary conditions implemented within HiSA. The top and bottom boundaries are symmetry planes.

The following interface boundary conditions are used to solve the interaction between the fluid and the solid. For the momentum equation, the Dirichlet-Neumann combination of `fsiInterfaceDisplacement` for the fluid mesh displacement (with `movingWallVelocity` for the velocity) and `fsiInterfaceTraction` for the solid is used. This interface boundary conditions combination is the same as the one presented in Section 3.1, except that pressure stabilization (provided by the `fsiInterfacePressure` condition) is not required here because the flow is compressible. Hence, the coupling algorithm is not subject to the added-mass instability reported by Causin *et al.* [1]. For the energy equation, the combination of `fstiTemperature` and `fstiHeatFlux` is used. As presented in Section 3.2, these interface boundary conditions are used to predict aerodynamic heating in the solid region.

This problem is solved using the mesh presented in Fig. 10. The mesh is composed of hexahedral cells and the interface between the fluid and solid region is conformal. It also contains approximately 33 090 cells in the fluid region and 2 560 cells in the solid region. Moreover, an adaptive meshing strategy is used in the fluid region to track and refine the mesh in areas with large density gradients. This strategy improves accuracy for phenomena like shock waves and expansion waves while keeping a coarser mesh in other regions. These phenomena also have an impact on the aerodynamic forces and heat fluxes, which are usually the quantities of interest in hypersonic flow problems. Fig. 11 shows the adapted mesh for the current simulation after 0.01 s of simulated time. The dynamic meshing strategy is implemented within a class called `dynamicRefine2DMotionSolverFvMesh` and is based on the work of Kumar *et al.* [22] and Eltard-Larsen *et al.* [23]. The metric that is used for the refinement is a normalized density gradient:

$$||\nabla\rho|| = \left|\frac{\nabla\rho\,\lambda}{\rho_2 - \rho_1}\right|, \tag{9}$$

where the parameter λ is a cell-based reference dimension, while ρ_2 and ρ_1 are typical density values expected in front and behind a normal shock wave that may occur. These last parameters can be estimated with the normal shock theory [24]. In the presented results, the values of these parameters are:

$$\lambda = 3 \times 10^{-3}\ \text{m}, \quad \rho_1 = 1.1765\ \text{kg/m}^3, \quad \rho_2 = 5.8825\ \text{kg/m}^3. \tag{10}$$

Figure 12 presents a visualization of the pressure field for the fluid region and of the displacement field for the solid region at different times. A comparison between the flexible plate and a rigid plate is also presented in the same figure. In the early instants of the simulation, a normal shock is formed in front of the flexible plate, which induces large aerodynamic forces. At this stage, differences between the flexible and the rigid plate are minimal. A few moments later, the flexible plate starts deforming and a detached shock expands in front of it. For large deformation, it can be observed that the shock patterns are effectively different between the flexible plate case and the rigid plate case.

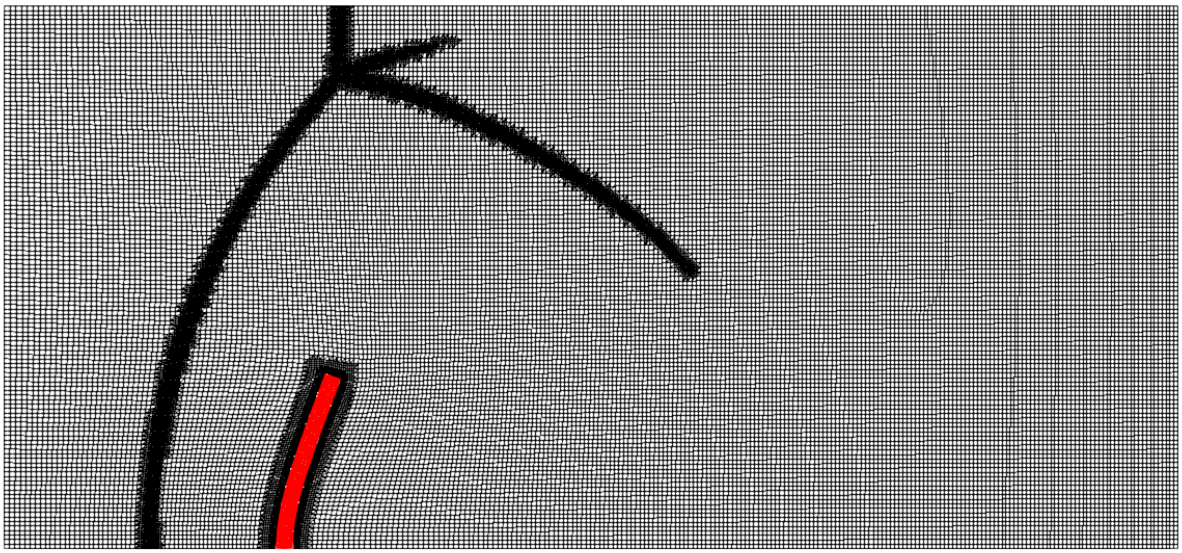


Figure 11. Illustration of the deformed mesh obtained using the adaptive mesh strategy. A region of high-density gradient can be observed with higher cell density. This mesh corresponds to the simulation at 0.01s of simulation.

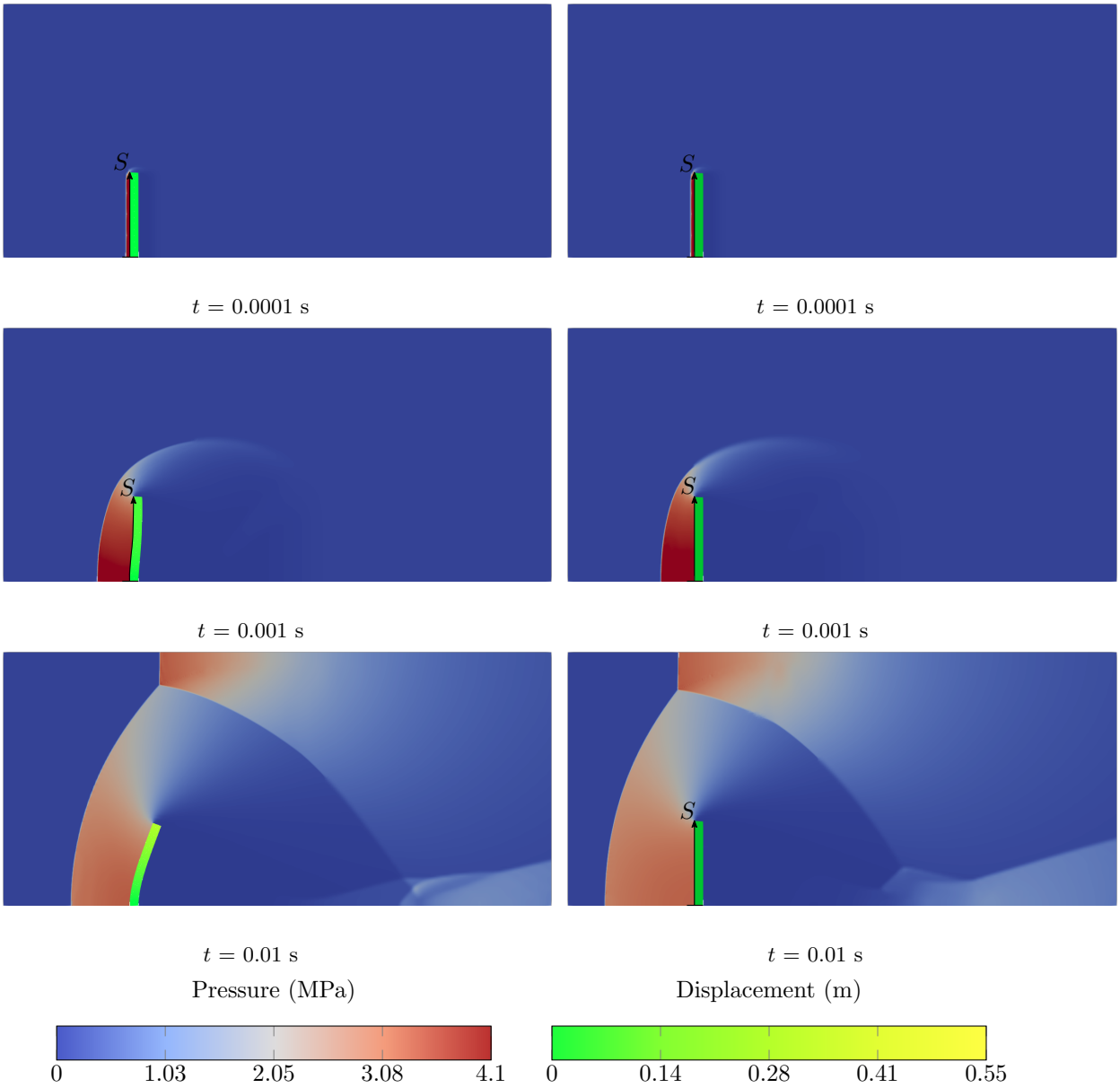


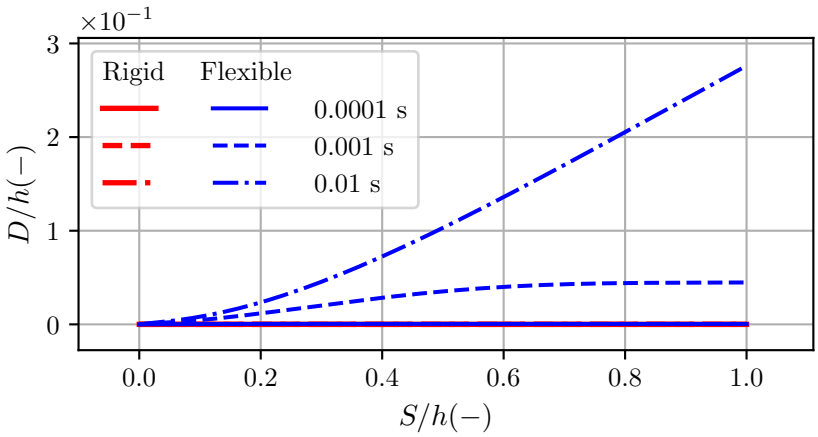
Figure 12. Dynamics of the flexible and rigid plates at different times. The flexible plate is shown on the left, and the rigid plate is shown on the right. The fluid region shows the pressure field, while the solid region shows the displacement field. The S axis corresponds to the location where the heat flux and displacements are sampled.

Large deformations also have an impact on other flow properties. Indeed, the heat exchanged between the fluid and the solid regions is also affected by the deformation of the plate. As a quantitative measure, heat flux and displacements are plotted at different times to compare results for the flexible plate and the rigid plate (see Fig. 13). When the structure is subjected to large deformations, significant heat transfer differences can be observed between the flexible and rigid plate cases.

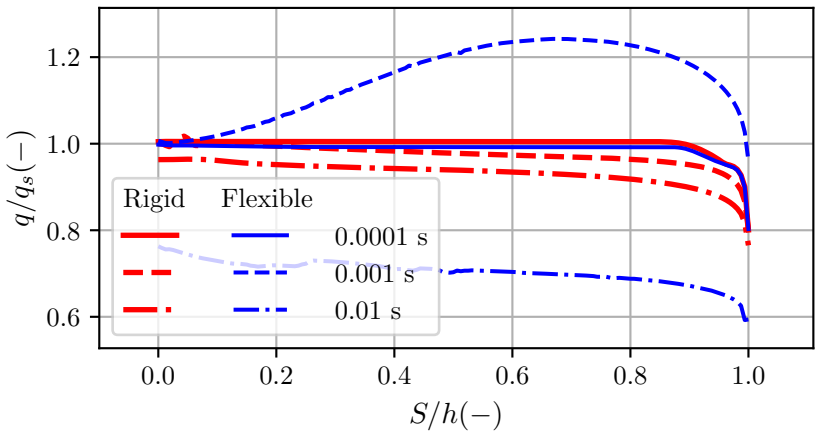
Lastly, this example shows that the modularity of the framework allows the solution of complex interaction for multiphysics problems. Also, the methodology presented in this paper allows interface boundary conditions to be used in different contexts.

4. Conclusion

This paper presented a multiphysics framework implemented within OpenFOAM-v2006. The main application uses a coupling algorithm based on a multi-region partitioned approach. To improve stability for strong physical interactions, an iterative method that manages multiple solvers sequentially was implemented, which allows implicit temporal schemes to be used consistently. The proposed methodology



(a)



(b)

Figure 13. Heat transfer and displacement magnitude for the rigid and flexible plates: (a) displacement magnitude (normalized with the height of the plate); and (b) heat flux (normalized with the stagnation heat flux). Values were monitored along the S-axis at different times.

allows standard solvers to be implemented within C++ classes. This generalizes and facilitates solver interchangeability since solvers can then be used as modular objects within the main application. To ensure standardization of all solvers, solver classes are inherited from an abstract base class.

An important aspect of the proposed approach is the fact that the multiphysics coupling is managed in most parts through interface boundary conditions. These boundary conditions allow physical data to be transferred from one region to another. Moreover, they are implemented as standard OpenFOAM® boundary conditions, which makes the coupling of different physics highly modular. On the other hand, the coupling algorithm itself is managed by the main application. The coupling algorithm consists of a fixed-point iterative loop that allows implicit time schemes to be applied consistently in all solvers involved in a given simulation. The main multiphysics application is independent of any physical model. The application only manages calls to solver objects' specific functions.

In addition to the presentation of the framework, some multiphysics problems involving fluid-structure interactions and conjugate heat transfer were presented to illustrate the capability and versatility of the proposed framework. Results from these examples show that the framework can solve strong interactions with multiple regions involving different physics. Hence, this modular multiphysics framework facilitates the setup and simulation of problems involving interactions between regions with different physics by providing a generalized coupling framework and providing relatively easy-to-use templates to implement new physics solvers and new interface boundary conditions.

Acknowledgements

Financial support from the Natural Sciences and Engineering Research Council of Canada (NSERC Discovery Grant RGPIN-2019-04489) and computational time provided by the Digital Research Alliance of Canada are gratefully acknowledged.

Author Contributions: Conceptualisation, M.O.; methodology, M.O.; software, G.S. and M.O.; validation, G.S. and M.O.; formal analysis, G.S. and M.O.; investigation, G.S.; resources, M.O.; data curation, G.S.; writing—original draft preparation, G.S.; writing—review and editing, G.S. and M.O.; visualisation, G.S.; supervision, M.O.; project administration, M.O.; funding acquisition, M.O. All authors have read and agreed to the published version of the manuscript.

References

[1] P. Causin, J. Gerbeau, and F. Nobile, "Added-mass effect in the design of partitioned algorithms for fluid-structure problems," *Computer Methods in Applied Mechanics and Engineering*, vol. 194, no. 42-44, pp. 4506–4527, 2005.

[2] M. A. Fernndez, J.-F. Gerbeau, and C. Grandmont, "A projection semi-implicit scheme for the coupling of an elastic structure with an incompressible fluid," *International Journal for Numerical Methods in Engineering*, vol. 69, no. 4, pp. 794–821, 2007.

[3] C. Farhat and M. Lesoinne, "Two efficient staggered algorithms for the serial and parallel solution of three-dimensional nonlinear transient aeroelastic problems," *Computer Methods in Applied Mechanics and Engineering*, 2000.

[4] OpenCFD, "New multi-application coupling through boundary conditions," 2020. [Online]. Available: https://www.openfoam.com/news/main-news/openfoam-v20-12/parallel#parallel-multi-world

[5] 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 pre-print server*, 2018.

[6] G. Chourdakis, K. Davis, B. Rodenberg, M. Schulte, F. Simonis, B. Uekermann, G. Abrams, H.-J. Bungartz, L. C. Yau, I. Desai, K. Eder, R. Hertrich, F. Lindner, A. Rusch, D. Sashko, D. Schneider, A. Totounferoush, D. Volland, P. Vollmer, and O. Z. Koseomur, "preCICE v2: A sustainable and user-friendly coupling library [version 2; peer review: 2 approved]," *Open Res Europe 2022*, vol. 2, p. 51, 2022.

[7] G. Chourdakis, D. Schneider, and B. Uekermann, "OpenFOAM-preCICE: Coupling OpenFOAM with external solvers for multi-physics simulations," *OpenFOAM Journal*, vol. 3, p. 125, Feb. 2023.

[8] J. A. Heyns, O. F. Oxtoby, and A. Steenkamp, "Modelling high-speed flow using a matrix-free coupled solver," in *9th OpenFOAM Workshop*, Zagreb, Croatia, 2014, Conference Proceedings.

[9] M. Olivier and G. Dumas, "A parametric investigation of the propulsion of 2D chordwise-flexible flapping wings at low Reynolds number using numerical simulations," *Journal of Fluids and Structures*, vol. 63, pp. 210–237, 2016.

[10] P.-O. Descoteaux and M. Olivier, "Performances of vertical-axis hydrokinetic turbines with chordwise-flexible blades," *Journal of Fluids and Structures*, vol. 102, p. 103235, 2021.

[11] O. Par-Lambert and M. Olivier, "A parametric study of energy extraction from vortex-induced vibrations," *Transactions of the Canadian Society for Mechanical Engineering*, vol. 42, no. 4, pp. 359–369, 2018.

[12] M. Olivier and O. Par-Lambert, "Strong fluid-solid interactions with segregated CFD solvers," *International Journal of Numerical Methods for Heat & Fluid Flow*, vol. 29, no. 7, pp. 2237–2252, 2019.

[13] A. Kurganov and E. Tadmor, "New high-resolution central schemes for nonlinear conservation laws and convection-diffusion equations," *Journal of Computational Physics*, vol. 160, no. 1, pp. 241–282, 2000.

[14] C. J. Greenshields, H. G. Weller, L. Gasparini, and J. M. Reese, "Implementation of semi-discrete, non-staggered central schemes in a colocated, polyhedral, finite volume framework, for high-speed viscous flows," *International Journal for Numerical Methods in Fluids*, pp. 1–21, 2009.

[15] L. E. Malvern, *Introduction to the mechanics of a continuous medium*. New Jersey: Prentice-Hall, inc., 1969.

[16] J. Degroote, A. Swillens, P. Bruggeman, R. Haelterman, P. Segers, and J. Vierendeels, "Simulation of fluid-structure interaction with the interface artificial compressibility method," *International Journal for Numerical Methods in Biomedical Engineering*, vol. 26, pp. 276–289, 2010.

[17] J. Gerbeau and M. Vidrascu, "A Quasi-Newton algorithm based on a reduced model for fluid-structure interaction problems in blood flows," *ESAIM: Mathematical Modelling and Numerical Analysis*, vol. 37, pp. 631–647, 2003.

[18] U. Kttler and W. Wall, "Fixed-point fluid-structure interaction solvers with dynamic relaxation," *Computational Mechanics*, vol. 43, pp. 61–72, 2008.

[19] A. R. Wieting, "Experimental study of shock wave interference heating on a cylindrical leading edge," Thesis, Old Dominion University, 1987.

[20] P. Dechaumphai, E. A. Thornton, and A. R. Wieting, "Fluid-Thermal-Structural Study of aerodynamically heated leading edges," NASA, Report, 1988.

[21] L. Hongpeng and W. Zhenguo, "Fluid-thermal-structural coupling investigations of opposing jet in hypersonic flows," *International Communications in Heat and Mass Transfer*, vol. 120, p. 105017, 2020.

[22] A. Kumar and L. Cornolti, "Adaptive mesh refinement in OpenFOAM-v1812 for 2-dimensional problems," 2022. [Online]. Available: https://github.com/krajit/dynamicRefine2DFvMesh/

[23] B. Eltard-Larsen, H. Nilsson, and R. Antham, "How to make a dynamicMotionRefineFvMesh class," Chalmers University, Report, 2016. [Online]. Available: http://www.tfd.chalmers.se/~hani/kurser/OS_CFD_2015/BjarkeEltard-Larsen/dynamicMotionRefineFvMesh_revised.pdf

[24] J. Anderson, John D., *Modern Compressible Flow: with Historical Perspective*, 3rd ed., ser. Aeronautical and aerospace engineering. New York: McGraw-Hill Compagnies, Inc., 2003.

Appendix A. Numerical implementation and integration

A.1. **Physical solver class template.** To illustrate the creation of a new physics solver, the implementation of a dummy solver called `physicsSolverTemplate` is presented. The presentation of this process aims to give a better understanding of the requirement for the implementation of a new physics solver and to present the way standard solvers can be adapted to fit those requirements.

The first step for creating a new physics solver is to define the solver class in a header file. List. 1 presents the implementation of the *physicsSolverTemplate.H* file. In this code, all parameters that are needed by the solver should be defined between lines 7 and 29. Those parameters include field objects, mesh information, dictionaries, and any other parameters that are needed during the simulation process.

```
1  class physicsSolverTemplate
2  :
3      public physicsSolver
4  {
5  protected:
6      // Protected data
7          // ========================================================
8          // All physical parameters required for the physics
9          // solver should be defined here.
10
11         // Mesh
12         autoPtr<dynamicFvMesh> meshPtr_;
13         dynamicFvMesh& mesh_;
14
15         // Dictionary
16         IOdictionary properties_;
17
18         // Field
19         autoPtr<volVectorField> Uptr_;
20         autoPtr<volScalarField> rhoptr_;
21
22         autoPtr<volVectorField> divPhiU_;
23         autoPtr<volScalarField> divPhiRho_;
24
25         // Other
26         scalar initialResidual_;
27         autoPtr<pimpleControl> pimple_;
28
29         // ========================================================
30
31     // Protected Member Functions
32         // ========================================================
33         // Optional custom functions
34
35         // Functions
36         void computeExplicitFluxes();
37         bool validateIODictionary();
38
39         // ========================================================
40
41         //- Disallow default bitwise copy construct
42         physicsSolverTemplate(const physicsSolverTemplate&);
43         //- Disallow default bitwise assignment
44         void operator=(const physicsSolverTemplate&);
45 public:
46     //- Runtime type information
47         TypeName("physicsSolverTemplate");
48     // Constructors
49         //- Construct from components
50         physicsSolverTemplate
51         (
52           const IOobject& io,
53           const word& regionName
54         );
55     // Destructor
56          virtual ~physicsSolverTemplate();
57     // Member Functions
58          virtual void createFields(); // Required
59          virtual bool isConverged(); // Required
60          virtual void solveFields(); // Required
61          virtual void startTimeStep(); // Optional
62          virtual void endTimeStep(); // Optional
63          virtual bool computeDeltaT(); // Optional
64          virtual scalar returnDeltaT(); // Optional
65 };
```

Listing 1. Implementation of a dummy physics solver called `physicsSolverTemplate` in the multiphysics framework.

Another important aspect of the implementation of a new physics solver is the definition of member functions. These functions are inherited from the base class as it is shown in Fig. 1 of Section 2.1. However,

only `createFields()`, `isConverged()`, and `solveFields()` are required. Other member functions such as `startTimeStep()`, `endTimeStep()`, `computeDeltaT()`, and `returnDeltaT()` are optional. Additional member functions, such as those between lines 32 and 39 in List. 1, can be defined as well if needed. These custom functions can compute or validate data that is needed by other functions, or perform computations that occur multiple times in a given function, for example.

After all components of the solver are fully defined in the *physicsSolverTemplate.H* header file, those components can be implemented in the *physicsSolverTemplate.C* source file. The implementation of this file is discussed in three steps. In the source file, custom class-specific functions are usually implemented first. List. 2 presents the implementation of two custom functions for the physics solver called `physicsSolverTemplate`. Custom functions can be used to compute values for class parameters such as the `computeExplicitFluxes()` member function, or they can be used to return objects like `bool`, `scalar`, etc. An example of a validation function is presented with the function called `validateIODictionary()`.

```
1  void Foam::physicsSolverTemplate::computeExplicitFluxes()
2  {
3    // Field references
4    volVectorField& U = Uptr_();
5    volScalarField& rho = rhoptr_();
6
7    // Compute value
8    volVectorField phiU = ...
9    volScalarField phiRho = ...
10
11   // Assign value
12   divPhiU_() = fvc::div(phiU);
13   divPhiRho_() = fvc::div(phiRho);
14 }
15
16 bool Foam::physicsSolverTemplate::validateIODictionary()
17 {
18   // References
19   IOdictionary& properties = properties_;
20
21   // Local variable
22   bool validData = false;
23
24   // Validate Data
25   if(...)
26   {
27     validData = true;
28   }
29
30   return validData;
31 }
```

Listing 2. Implementation of custom functions for `physicsSolverTemplate` dummy solver.

For the second implementation step, the solver object constructor needs to be implemented. List. 3 presents the implementation of the constructor for the `physicsSolverTemplate` solver. In this specific constructor, different class parameters are initialized. Also, the constructor can be used to perform simple actions that are needed upon the creation of a solver object. If needed, these actions should be placed between lines 46 and 59 in List. 3.

```
1  Foam::physicsSolverTemplate::physicsSolverTemplate
2  (
3      const IOobject& io,
4      const word& regionName
5  )
6  :
7      physicsSolver(io, regionName),
8
9      meshPtr_
10     (
11         dynamicFvMesh::New
12         (
13             IOobject
14             (
15                 region_,
16                 io.time().timeName(),
17                 io.time(),
18                 IOobject::MUST_READ
19             )
20         )
21     ),
22
23     mesh_(meshPtr_()),
24
25     properties_
26     (
```

```
        IOobject
        (
            "properties",
            io.time().constant(),
            mesh_,
            IOobject::MUST_READ,
            IOobject::NO_WRITE
        )
    ),

    Uptr_(nullptr),
    rhoptr_(nullptr),

    divPhiU_(nullptr),
    divPhiRho_(nullptr),

    initialResidual_(0),
    pimple_(nullptr)
{
    // ========================================
    //
    // Actions to do upon the construction
    // of the solver object
    //
    // ========================================

    pimple_.set
    (
      new pimpleControl
      (
        mesh
      )
    );
}
```

Listing 3. Implementation of the constructor for `physicsSolverTemplate` dummy solver.

The last implementation step is dedicated to member functions. The first action that is performed by the algorithm is to initialize pointers to field objects within each solver. This action is performed by the `createFields()` function. The reason pointers are used is to allow the field initialization to be carried out once all region solvers have been initialized. This is required so that multi-field boundary conditions classes can access other regions' information upon creation. Usually, OpenFOAM® solver applications implement the creation of field objects in a file called *createFields.H*. The content of this file can be used as a guide to implement the `createFields()` function. As an example, List. 4 presents the initialization of fields `U`, `rho`, `divPhiU`, and `divPhiRho` with their associated field objects.

```
void Foam::physicsSolverTemplate::createFields()
{
  Uptr_.reset
  (
    new volVectorField
    (
      IOobject
      (
        "U",
        mesh_.time().timeName(),
        mesh_,
        IOobject::MUST_READ,
        IOobject::AUTO_WRITE
      ),
      mesh_
    )
  );

  rhoptr_.reset
  (
    new volScalarField
    (
      IOobject
      (
        "rho",
        mesh_.time().timeName(),
        mesh_,
        IOobject::NO_READ,
        IOobject::AUTO_WRITE
      ),
      thermo.rho()
    )
  );

  divPhiU_.reset
  (
    new volVectorField
```

```
38      (
39          IOobject
40          (
41              "divPhiU",
42              mesh_.time().timeName(),
43              mesh_
44          ),
45          mesh_,
46          dimensionedVector("", dimDensity*dimVelocity/dimTime, vector(0,0,0))
47      )
48  );
49
50  divPhiRho_.reset
51  (
52      new volScalarField
53      (
54          IOobject
55          (
56              "divPhiRho",
57              mesh_.time().timeName(),
58              mesh_
59          ),
60          mesh_,
61          dimensionedScalar("", dimDensity*dimDensity/dimTime, 0.0)
62      )
63  );
64  }
```

Listing 4. Implementation of `createFields()` function for `physicsSolverTemplate` dummy solver.

To solve the physics associated with each region, the function `solveFields()` is used. This function is dedicated to solving governing equations and contains only the solution steps. In standard OpenFOAM® solvers, the solution process is usually implemented in the main application source code. The content of these files can be used as a guide to implement the `solveFields()` function. However, it is important to note that the time loop should not be included in the `solveFields()` function as it is managed by the main multiphysics application (`multiFieldSolver`).

```
1  void Foam::physicsSolverTemplate::solveFields()
2  {
3      // Mesh and fields references
4      dynamicFvMesh& mesh = mesh_;
5      volVectorField& U = Uptr_();
6      volScalarField& rho = rhoptr_();
7      volVectorField& divPhiU = divPhiUptr_();
8      volScalarField& divPhiRho = divPhiRhoptr_();
9
10     scalar& initialResidual = initialResidual_;
11
12     // ===========================================
13     //
14     //          Solve physical fields
15     //
16     // ===========================================
17
18     initialResidual = ...
19 }
```

Listing 5. Implementation of `solveFields()` function for `physicsSolverTemplate` dummy solver.

To ensure that physical interactions between all regions are accurate and consistent with the implicit time-marching schemes, iterative convergence of the main coupling loop should be achieved. Thus, the `isConverged()` function is used to evaluate the convergence state of each region involved in a given simulation. This function is used to check if the equation residuals have reached a certain threshold. The function returns a `bool` object to the main coupling algorithm to indicate if each solver has reached iterative convergence or not. To validate the convergence state, initial residuals are compared to their corresponding criteria, as presented between lines 8 and 12 in List. 6. Also, the initial residual value must be assigned to the corresponding class parameter in the `solveFields()` function, as presented on line 18 in List. 5. However, for physics solvers based on the PIMPLE algorithm, convergence criteria are already checked by the `loop()` function of a `pimpleControl` object. Thus, lines 3 to 14 in List. 6 can be replaced by line 15.

```
1  bool Foam::physicsSolverTemplate::isConverged()
2  {
3      scalar& initialResidual = initialResidual_;
4      bool converged = false;
5
```

```
    scalar Threshold = ... ;

    // Check convergence
    if ( initialResidual < Threshold)
    {
        converged = true;
    }

    return converged;
    // return pimple_.loop();
}
```

Listing 6. Implementation of `isConverged()` function for `physicsSolverTemplate` dummy solver.

As discussed previously, there are four optional member functions that can be implemented in the physics solver. The `startTimeStep()` function can be implemented if some actions are needed before the first iteration. Similarly, the `endTimeStep()` function can be implemented if some actions are needed after the last iteration. Also, the framework allows adaptive time stepping to be implemented within physics solvers. To implement the optional adaptive time-stepping options, functions `computeDeltaT()` and `returnDeltaT()` should be implemented. The `computeDeltaT()` function is used to trigger adaptive time stepping if needed. In such a case, the `returnDeltaT()` function is used to compute the new time step of the corresponding region and return its value to the main program. Adaptive time-stepping algorithms are common among many OpenFOAM® solvers. In most solvers, the adaptive time step is computed within the file *setDeltaT.H*. To compute a new time step, the program uses the maximum Courant number value that is computed by the *CourantNo.H* file. The content of those two files can be used as a guide to implement the `computeDeltaT()` and the `returnDeltaT()` functions.

```
void Foam::physicsSolverTemplate::startTimeStep()
{
    // ========================================
    //
    // Actions to do before solving the fields
    //
    // ========================================
}

void Foam::physicsSolverTemplate::endTimeStep()
{
    // ========================================
    //
    // Actions to do after solving the fields
    //
    // ========================================
}

bool Foam::physicsSolverTemplate::computeDeltaT()
{
    bool computeDeltaT = false;

    // Check if adaptive time step is required
    if (...)
    {
        computeDeltaT = true;
    }

    return computeDeltaT;
}


scalar Foam::physicsSolverTemplate::returnDeltaT()
{
    // Compute Courant number or other metric

    CoNum = ...

    // Compute new time step

    deltaTime = ...

    return deltaTime;
}
```

Listing 7. Implementation of optional member function for `physicsSolverTemplate` dummy solver.

As presented, only two files (*physicsSolverTemplate.C* and *physicsSolverTemplate.H*) are needed to create a new physics solver that works with the multiphysics framework. Also, adapting existing OpenFOAM®

solvers is a matter of moving the corresponding source code to the appropriate functions of a class derived from `physicsSolver`.

A.2. **Multiphysics interface boundary condition template.** This section illustrates the creation of a new interface boundary condition for multiphysics simulations. The integration of interpolation tools and mesh configurations is also discussed.

To implement a new interface boundary condition using the Dirichlet-Neumann formulation, two OpenFOAM® boundary conditions should be created. The Dirichlet condition is inherited from the `fixedValueFvPatchFields` type boundary condition and the Neumann condition is inherited from the `fixedGradientFvPatchFields` type boundary condition. However, to implement a new interface condition using the mixed formulation, only one OpenFOAM® condition is needed. In this case, the mixed condition is inherited from the `mixedFvPatchFields` type boundary condition. All these types of boundary conditions (fixed value, fixed gradients, and mixed) can be created using existing examples from the OpenFOAM® library. Code templates for boundary conditions can also be accessed using the `foamNewBC` command.

As discussed in Section 2.2, mesh interpolation is needed to transfer field information from one region to another. Fortunately, both the standard OpenFOAM® software and the proposed modular multiphysics framework provide mesh interpolation tools that can be used. The mesh interpolation functionality from the standard OpenFOAM® software is called `mappedPatchBase`. This functionality is based on mapping tools from the OpenFOAM® software, which allows the redistribution of mesh information from one region to another. Also, multiple sampling models can be specified, such as the nearest face method for conformal meshes or the nearest face Arbitrary Mesh Interface (AMI) method for non-conformal meshes. To integrate this interpolation tool in a boundary condition, a few steps are needed. Firstly, one would need to include the header file *mappedPatchBase.H* in the source file of the boundary condition, as presented in List. 8. Also, modification to the `updateCoeffs()` method in the boundary condition source file is needed. This mapping tool is used to retrieve the neighbor mesh and neighbor fields, as presented on lines 8 to 20 in List. 8. This information is then mapped on the owner mesh, as presented on line 23 in List. 8. With these mapping steps done, the boundary condition can be updated.

```
1  #include "mappedPatchBase.H"
2
3  ...
4
5  void Foam::...FvPatch...Field::updateCoeffs()
6  {
7    // Access neighbor mesh information
8    const mappedPatchBase& mpp =
9                    refCast<const mappedPatchBase>(patch().patch());
10   const polyMesh& nbrMesh = mpp.sampleMesh();
11   const label samplePatchi = mpp.samplePolyPatch().index();
12   const fvPatch& nbrPatch =
13                   refCast<const fvMesh>(nbrMesh).boundary()[samplePatchi];
14
15   // Access neighbor field information
16   const ...FvPatch...Field& nbrPatchField =
17     refCast < const ...FvPatch...Field >
18     (
19         nbrPatch.lookupPatchField<vol...Field, ...> ( nbrFieldName_ )
20     );
21
22   // Retrieve field to obtain local values
23   tmp<...Field> nbrField( nbrPatchField.patchInternalField() );
24
25   // Distribute field information on owner mesh
26   mpp.distribute(nbrField.ref());
27
28   // ============================================================
29   //  Evaluate boundary condition
30   // ============================================================
31 }
```

Listing 8. Integration of the `mappedPatchBase` mesh interpolation tool in an interface boundary condition.

Alternatively, a mesh interpolation (or extrapolation, as its name suggests) class called `radialExtrapolation` is available in the proposed framework. This class can be used to redistribute specific mesh information from one region to another. Different interpolation (or extrapolation) methods can be specified: nearest-neighbor, Radial Basis Function (RBF), or Inverse Distance Weighting (IDW). Also, a sampling method is needed along with the interpolation methods. The sampling can be done from the source's nearest point, from a source's group of near points, or from all the source's points.

To integrate this mesh interpolation tool in a specific boundary condition, a few steps are needed. Firstly, one would need to include the header file *radialExtrapolation.H* in the source file of the boundary condition, as presented in List. 9. Also, modification to the `updateCoeffs()` method in the boundary condition source file is needed. To interpolate from one region to another, the neighbor mesh and the neighbor fields are needed. To retrieve this information, lines 8 to 23 in List. 9 are used. Also, the `radialExtrapolation` object needs to be created. In List. 9, this is done through lines 28 to 34. When the interpolator object is created, interpolation weights are calculated at the same time. Then, the interpolation object can be used (lines 40 to 43 in List. 9), and the boundary condition can be updated.

```
1  #include "radialExtrapolation.H"
2
3  ...
4
5  void Foam::...FvPatch...Field::updateCoeffs()
6  {
7    // Access neighbor mesh information
8    const fvMesh& nbrMesh
9    (
10     db().time().lookupObject<fvMesh>(nbrRegionName_)
11   );
12
13   // Access neighbor field information
14   label nbrPatchID
15   (
16     nbrMesh.boundaryMesh().findPatchID(nbrPatchName_)
17   );
18
19   const vol...Field& Field =
20     nbrMesh.lookupObject<vol...Field>("Field");
21
22   // Retrieve field to obtain local values
23   ...Field tmpField ( ... );
24
25   if(updateInterpolator_)
26   {
27     // Create interpolator
28     interpolatorNbrOwn_.clear();
29     interpolatorNbrOwn_ = radialExtrapolation::New
30     (
31         nbrMesh.boundary()[nbrPatchID].Cf(),
32         patch().Cf(),
33         dict_
34     );
35
36     updateInterpolator_ = false;
37   }
38
39   // Interpolate field information on owner mesh
40   mapNbrField_ = interpolatorNbrOwn_->pointInterpolate
41   (
42     tmpField
43   );
44
45   // ========================================================================
46   //  Evaluate boundary condition
47   // ========================================================================
48 }
```

Listing 9. Integration of the `radialExtrapolation` mesh interpolation tool in an interface boundary condition.

As discussed in Section 2.2, mesh interpolation tools usually consider all regions to be in the same configuration. However, this is not always the case. For example, as presented in Fig. 3 for FSI simulations, the fluid region is usually solved with an ALE approach where the solid boundary is in the deformed configuration, while the solid is usually solved in the reference configuration. Thus, a few steps are needed in the `updateCoeffs()` method located in the boundary condition source file to perform the interpolation consistently. Firstly, the header file *pointField.H* needs to be included in the source file of the boundary condition, as presented in List. 10. Also, before the creation of the interpolation weights, the mesh in the undeformed or reference configuration is moved to the deformed configuration as presented between lines 10 and 19 in List. 10. After the interpolation step is completed, the mesh is moved back to the original configuration as presented between lines 30 to 39 in List. 10.

```
1  #include "pointField.H"
2
3  ...
4
5  void Foam::...FvPatch...Field::updateCoeffs()
6  {
```

```
7
8     ...
9
10    // Move the static mesh in the deformed configuration.
11    {
12      const pointVectorField& pointD =
13        staticMesh.lookupObject<pointVectorField>("pointD");
14
15      vectorField newPoints = staticMesh.points()
16                              + pointD.primitiveField();
17
18      const_cast<fvMesh&>(staticMesh).movePoints(newPoints);
19    }
20
21    // ============================================================
22    //  Create interpolator step
23    //
24    //  or
25    //
26    //  Distribute field step
27    // ============================================================
28
29
30    // Move the static mesh back in the initial configuration.
31    {
32      const pointVectorField& pointD =
33        staticMesh.lookupObject<pointVectorField>("pointD");
34
35      vectorField newPoints = staticMesh.points()
36                              - pointD.primitiveField();
37
38      const_cast<fvMesh&>(staticMesh).movePoints(newPoints);
39    }
40
41    ...
42
43  }
```

Listing 10. Integration of a moving mesh strategy to ensure interpolation compatibility between regions.