DEPARTMENT OF INFORMATICS

TECHNISCHE UNIVERSITÄT MÜNCHEN

Bachelor's Thesis in Informatics

Updating Fluent-preCICE and development of an automated testing environment

Khalil Hkiri

DEPARTMENT OF INFORMATICS

TECHNISCHE UNIVERSITÄT MÜNCHEN

Bachelor's Thesis in Informatics

Updating Fluent-preCICE and development of an automated testing environment

Author:	Khalil Hkiri
Supervisor:	Univ.-Prof. Dr. Hans-Joachim Bungartz
Advisor:	Benjamin Rodenberg, M.Sc. (hons)
Submission Date:	05-05-2025

I confirm that this bachelor's thesis in informatics is my own work and I have documented all sources and material used.

Munich, 05-05-2025 Khalil Hkiri

Acknowledgments

I would like to extend my sincere gratitude to Prof. Dr. Hans-Joachim Bungartz and Benjamin Rodenberg for providing me with the opportunity to work on this thesis and for their invaluable support throughout the project. I am especially grateful to Benjamin Rodenberg for his insightful guidance and for dedicating significant effort to introduce me to the challenging field of Computational Fluid Dynamics (CFD). His expert knowledge and persistent encouragement have been instrumental in shaping this work.

I would also like to thank my parents and my sister for their unwavering support and encouragement at every step of this journey. Their belief in my abilities helped me overcome challenges and stay focused on my goals.

This thesis would not have been possible without the contributions, support, and guidance of all those mentioned above.

Abstract

This work revolves around upgrading the preCICE-Fluent adapter to align it with the latest preCICE major release (version 3). The thesis also involves going through the customized UDFs and the preCICE configuration file that allowed the coupling. To ensure a robust and maintainable adapter, we introduced automated test cases in which basic functionalities and the adapter's compilation are checked. These tests use the C mocking library (CMock) for a well-structured and easily understandable testing environment.

Contents

1 Introduction

Multi-physics simulations, which involve interactions among fluid dynamics, structural mechanics, heat transfer, and other physical phenomena, have become essential tools in modern engineering design[1]. To successfully capture these complex interactions, different simulation tools must reliably exchange data during runtime. *preCICE*, an open-source coupling library, addresses this challenge by enabling bidirectional data exchange—such as transferring forces, displacements, or heat flux—between independent numerical solvers. By abstracting the communication between these standalone solvers, preCICE allows engineers to assemble sophisticated co-simulation workflows without altering the core solver code.

Among the many solvers used for fluid dynamics simulations, *Ansys Fluent* stands out due to its advanced numerical methods, versatile meshing capabilities, and extensive physical modeling features. Fluent is, a closed-source commercial solver, particularly adept at modeling turbulent flows, heat transfer, and chemical reactions. However, it does not natively support direct, tight coupling to external solvers, making integration into multi-physics simulations challenging. This limitation necessitates a specialized interface to facilitate the exchange of simulation data at each iteration.

The *preCICE–Fluent adapter* addresses precisely this need. It bridges Ansys Fluent with other solvers by implementing Fluent-specific User-Defined Functions (UDFs) and utilizing preCICE's configurable interfaces to handle real-time exchanges of interface forces and mesh deformations. This thesis focuses on upgrading the adapter to achieve full compatibility with preCICE version 3, revising UDF callbacks and configuration syntax, and establishing a robust testing suite using the CMock framework. These improvements ensure the adapter remains reliable, easy to maintain, and ready to support future computational experiments involving complex CFD-based multi-physics interactions.

During this work, we tackle a central research question: How is the fluent adapter being upgraded to achieve full compatibility with preCICE version 3, revising UDF callbacks and configuration syntax, including a CMock-based test suite to ensure it is reliable and easy to maintain during real-time, two-way CFD–FSI coupling.

We start by introducing some key elements in Chapter 2, that helped us understand fundamental and very essential concepts for the success of the upgrade of the adapter. In Chapter 3, we will take a close look at the example that we used to run our

adapter. Followed by a detailed overview of how the adapter works, mentioning the configuration file as well as the UDF and our other solver which will be tackled in Chapter 4. Chapter 5 will include a go-through of the testing libraries and the entire logic behind the implemented tests. Lastly in chapter 6, we will provide a summary of this work, highlighting the important aspects and shedding light on future work.

2 Background and Related Work

In this work we couple two solvers, through a preCICE configuration file and following multiphyisics principle to make the simulation as realistic as possible.

2.1 Multi-Physics Coupling Fundamentals

Physical phenomena seldom occur in isolation. Fluid dynamics, structural mechanics, heat transfer, and electromagnetism continuously interact, generating phenomena such as thermal conduction, material deformation, and mass transport precisely at the interfaces of these domains. Multiphysics simulation integrates specialized solvers for distinct physical domains into a cohesive computational framework. By coordinating simultaneous execution and data exchange between these solvers, engineers can accurately simulate system behaviors under realistic conditions, effectively capturing the crucial physical interactions [2].

2.2 Overview of preCICE

preCICE (Precise Code Interaction Coupling Environment) is an open-source coupling library designed to enable flexible and robust data exchange between independent simulation codes during runtime. In the context of *preCICE*, an *adapter* refers to an interface that connects a particular solver to the coupling library, allowing it to send and receive data such as forces, displacements, or temperature fields during a simulation[3]. Adapters encapsulate solver-specific details and translate them into a standardized form understandable by *preCICE*, significantly reducing integration effort. Existing adapters have been developed for widely-used solvers such as OpenFOAM[4], FEniCS[5], and CalculiX[6], among others. The framework itself abstracts communication details and supports bidirectional coupling, allowing engineers to seamlessly integrate various standalone solvers. Additionally, *preCICE* handles data mapping, interpolation, communication synchronization, and convergence acceleration, simplifying the complexity of multiphysics simulations. Recent developments, particularly in version 3, have further improved robustness, user-friendliness, and

compatibility with popular solvers including OpenFOAM, CalculiX, and Ansys Fluent [7].

2.3 Introduction to Ansys Fluent

Ansys Fluent is a leading computational fluid dynamics (CFD) solver widely recognized for its reliable numerical methods, versatile meshing capabilities, and extensive physical modeling[8]. Fluent excels in simulating turbulent flows, complex heat transfer processes, chemical reactions, and multiphase phenomena. However, Fluent inherently lacks direct capabilities for tight coupling with external multiphysics solvers. To integrate Fluent into multiphysics workflows, custom user-defined functions (UDFs) and external coupling interfaces, such as those provided by preCICE, are required. Fluent's UDF framework offers the flexibility to customize boundary conditions, source terms, and mesh movements, making it a powerful tool for advanced simulations [9]. Due to Ansys' complex design, its downside is that it is incompatible with every operating system. So before going ahead and getting Ansys, the user needs to check for their system compatibility under Ansys's systems overview.

2.4 Summary of Related Research

Several studies[10] have explored multiphysics coupling techniques, particularly focusing on fluid-structure interactions (FSI) using Fluent coupled with structural analysis solvers[11]. The preCICE–Fluent adapter, initially developed by the preCICE community, has undergone significant upgrades to enhance its compatibility and reliability. Past research has demonstrated the effectiveness of using Fluent coupled via preCICE for simulating complex engineering scenarios like aeroelasticity, heat exchanger performance, and biomedical device analysis .

Continuous enhancements, including better integration tests, updated UDF callbacks, and robust mock testing frameworks, have been introduced to ensure that the adapter remains compatible with evolving versions of preCICE. Furthermore, methodologies like unit and mock testing have been emphasized to ensure reliable and maintainable coupling interfaces, especially critical in high-fidelity engineering simulations .

3 Theoretical Framework

To put our coupling example into context, this chapter reviews the theory behind the two individual problems—fluid flow in a lid-driven cavity and deflection of a supported beam—and the basic principles of their interaction. In particular, we require two distinct models:

- A **fluid model**, governed by the incompressible Navier–Stokes equations, to capture the flow inside the cavity and the resulting traction on the boundary.
- A **structural model**, based on the Euler–Bernoulli beam theory, to predict the beam's deflection under that traction.

These two solvers exchange boundary forces and displacements in a partitioned coupling until convergence. These are the main points of this chapter, where we start by going through the geometry and definition of the Lid-Driven Cavity in section 3.1, followed by a detailed overview of the structural model (Flexible Beam) in section 3.2 and finally bringing all the pieces together in the last section 3.3 where we combine both models and introduce essential terminologies used in the rest of our work.

3.1 Lid-Driven Cavity

In this section, we will go through the important aspects of the first half of our example case, where we tackle the domain definition and the parameters of the Lid-Driven Cavity and introduce the boundary conditions alongside the governing equations. Before presenting the governing equations, we define the specific case parameters:

3.1.1 Domain Definition and Parameters

Rather than list parameters as bullet points, we summarize our lid-driven cavity case's key geometric, mesh, and flow parameters in Table 3.1.

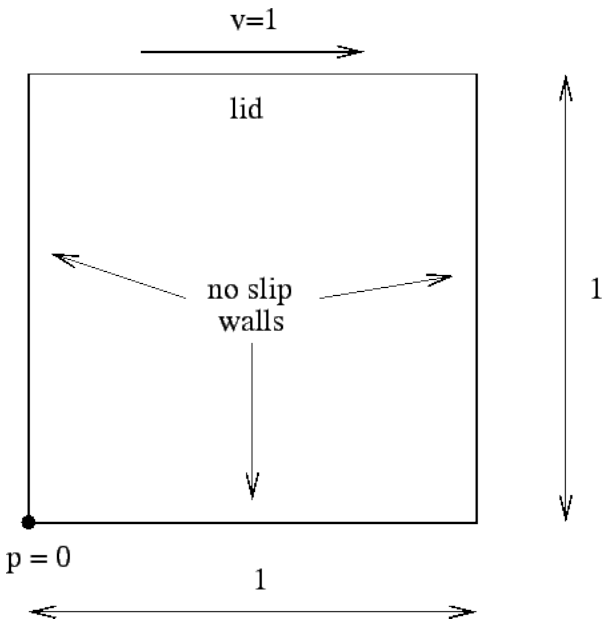

Figure 3.1: Lid Driven Cavity
[12]

Parameter	**Symbol**	**Value / Description**
Geometry	H	1.0 m (square cavity side length)
Mesh	—	5182 cells, 2692 nodes (unstructured triangles, zone `surface_body`)
Fluid density	ρ	1000 kg/m^3
Dynamic viscosity	μ	0.001 Pa · s
Lid velocity	U	1.0 m/s (in the $+x$ direction)
Stationary walls	—	no-slip ($u = v = 0$) at bottom and vertical sides
Reynolds number (nominal)	$\mathrm{Re} = \frac{\rho U H}{\mu}$	10^6 (in practice adjusted to $\approx 10^3$ in Fluent)

Table 3.1: Domain and flow parameters for the lid-driven cavity test case.

Figure 3.1 shows the computational domain and boundary conditions. In our tutorial runs, we typically reduce the Reynolds number to $\mathrm{Re} \approx 10^3$ (by increasing μ or reducing U) to remain in a laminar regime while still capturing the primary vortex and corner eddies.

3.1.2 Governing Equations and Boundary Conditions

Governing Equations: Under steady, incompressible conditions, the flow satisfies

$$\nabla \cdot \mathbf{u} = 0, \qquad \rho\,(\mathbf{u} \cdot \nabla)\,\mathbf{u} = -\nabla p + \mu\,\nabla^2 \mathbf{u},$$

where $\mathbf{u} = (u, v)$ and p is the pressure[13].

For unsteady (time-dependent) incompressible flow one adds the local-acceleration term:

$$\nabla \cdot \mathbf{u} = 0, \qquad \rho\Big(\frac{\partial \mathbf{u}}{\partial t} + (\mathbf{u} \cdot \nabla)\,\mathbf{u}\Big) = -\nabla p + \mu\,\nabla^2 \mathbf{u}.$$

Equivalently, dividing through by ρ gives the more common non-dimensional form[14]

$$\frac{\partial \mathbf{u}}{\partial t} + (\mathbf{u} \cdot \nabla)\,\mathbf{u} = -\frac{1}{\rho}\,\nabla p + \nu\,\nabla^2 \mathbf{u},$$

with kinematic viscosity $\nu = \mu/\rho$

Boundary Conditions:

- *Top lid:* $u = U,\ v = 0.$
- *Other walls:* $u = v = 0.$

For moderate Reynolds numbers (e.g., Re ≈ 1000), the solution exhibits a primary vortex filling most of the cavity and smaller corner vortices.

3.2 Flexible Beam Theory

In this section, we will give a short definition of the Flexible Beam Theory, alongside a general definition of the Euler-Bernoulli Beam Equation, followed by the types of beam supports and deflections that could be relevant for future work but relevant for our work. Finish by deriving the final equation for our A beam is a straight longitudinal axis that reflects the forces applied on it by deforming into a curve called the *deflection curve*. This section reviews the main static beam deflections, summarizes the various beam and load types, discusses how each combination affects the deformation pattern, and gives a step-by-step derivation of the uniformly distributed-load solution used in our structural model.

The bottom wall in our coupling is replaced by a homogeneous, supported beam of length $L = H = 1.0\,\mathrm{m}$. Its cross-section has thickness $a = 0.01\,\mathrm{m}$, so the beam's

second moment of area (also called the area moment of inertia) about its neutral axis is:

$$I = \frac{a^4}{12} = \frac{(0.01)^4}{12}\ \mathrm{m}^4,$$

and the material modulus is $E = 117 \times 10^9$ Pa.

3.2.1 Euler–Bernoulli Beam Equation

Small-deflection theory gives the static form of the beam equation

$$E\,I\,\frac{\mathrm{d}^4 u}{\mathrm{d}x^4}(x) \;=\; q(x),$$

Where $q(x)$ is the load per unit length (here the vertical fluid traction).

3.2.2 Load and Beam Classifications

Beams are commonly categorized by their support conditions and the nature of their loads. In Chapter 4 of Gere & Goodno[15], three principal support types are defined:

- **Simply supported beam** A pin at one end and a roller at the other prevent translation but allow rotation. Horizontal and vertical force reactions develop at the pin, and a vertical reaction at the roller, but no end moments.

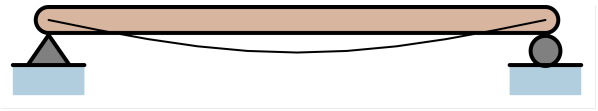

Figure 3.2: Simple Beam Deflection
[16]

- **Cantilever beam** Rigidly clamped at one end (no translation or rotation) and free at the other. Both force and moment reactions occur only at the fixed support.

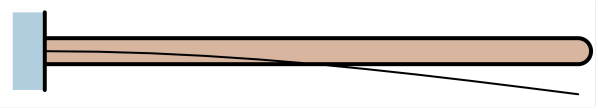

Figure 3.3: Cantilever Beam Deflection
[16]

- **Overhanging beam** Simply supported at A and B, but extending beyond B so that the segment BC behaves like a short cantilever.

Loads on beams fall into several idealized categories:

- **Concentrated (point) loads** P applied at discrete locations (e.g. P_1, P_2, P_3, P_4)[17].
- **Distributed loads** $q(x)$ acting along the span:
  - *Uniformly distributed* ($q = q_0$ constant).

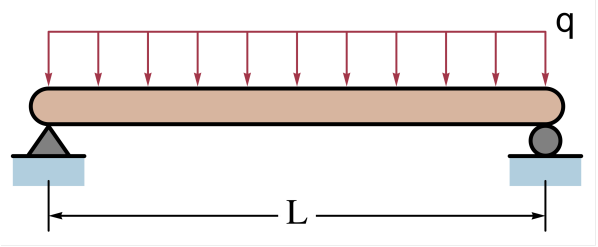

Figure 3.4: Simple Beam with Uniform Distributed Load

  - *Linearly varying* (q varies from q_1 to q_2).
- **Couples (pure moments)** M applied over a vanishingly small interval

In our fluid-structure coupling, the elastic "membrane" at the bottom of the lid-driven cavity is modeled as a *simply supported beam* of length L, carrying the *vertical fluid traction* $\mathbf{t}_{\text{fluid}}$ as a *uniformly distributed load* $q(x) = q_0$. Thus, our case corresponds precisely to the classical simply-supported beam under UDL, for which the closed-form deflection.

$$u(x) = \frac{q_0}{24\,E\,I}\,x\,(L-x)\,\left(L^2 - 2L\,x + x^2\right)$$

was derived in Section 3.2.4.

3.2.3 Analytical Beam Deflection Solutions for Standard Loads

Uniformly distributed load on a simply supported beam For $q(x) = q_0$, $0 \le x \le L$, with

$$u(0) = u(L) = 0, \quad u''(0) = u''(L) = 0,$$

one obtains: [15]

$$u(x) = \frac{q_0}{24\,E\,I}\,x\,(L-x)\,\left(L^2 - 2L\,x + x^2\right), \quad u_{\max} = \frac{q_0\,L^4}{384\,E\,I}. \tag{3.1}$$

Discrete-Nodal-Load Deflection Compared to the previous paragraph, the continuous load q_0 is replaced by a lumped nodal force F at each beam node, following [18]. Since $q_0 = F/L$, substituting into (3.1) gives

$$u(x_i) = \frac{\frac{F(x_i)}{L}}{24\,E\,I}\, x_i\,(L - x_i)\left(L^2 - 2Lx_i + x_i^2\right) = \frac{F(x_i)}{24\,E\,I\,L}\, x_i\,(L - x_i)\left(L^2 + x_i\,(L - x_i)\right) \tag{3.2}$$

Point load P at midspan of a simply supported beam[1] (Special case of Sec. 9.3 [13].)

$$u(x) = \frac{P\,x\,(3L^2 - 4x^2)}{48\,E\,I}, \quad u_{\max} = \frac{P\,L^3}{48\,E\,I}$$

Cantilever with end load P[1]

$$u(x) = \frac{P\,x^2\,(3L - x)}{6\,E\,I}, \quad u(L) = \frac{P\,L^3}{3\,E\,I}.$$

Cantilever with uniform load q[1]

$$u(x) = \frac{q\,x^2\,(6L^2 - 4Lx + x^2)}{24\,E\,I}, \quad u(L) = \frac{q\,L^4}{8\,E\,I}.$$

3.2.4 Derivation of the Discrete-Nodal-Load Deflection Formula

Instead of starting from a continuous q_0-load per unit length, our structural model uses a total nodal load F at each node (so $q_0 = F/L$). We now retrace the algebra to show how the implemented formula arises.

1. **Static Euler–Bernoulli ODE:**

$$E\,I\,\frac{\mathrm{d}^4 u}{\mathrm{d}x^4}(x) \;=\; \frac{F}{L}.$$

   Here F is the total vertical force applied at a node and L the beam span.

2. **Integrate once:**

$$E\,I\,u^{(3)}(x) = \frac{F}{L}\,x + C_1.$$

[1] These additional closed-form solutions are not required for the current work but may be relevant for future extensions of our adapter.

3. **Integrate twice:**
$$E\,I\,u''(x) = \frac{F}{L}\,\frac{x^2}{2} + C_1\,x + C_2.$$

4. **Integrate thrice:**
$$E\,I\,u'(x) = \frac{F}{L}\,\frac{x^3}{6} + \frac{C_1\,x^2}{2} + C_2\,x + C_3.$$

5. **Integrate four times:**
$$E\,I\,u(x) = \frac{F}{L}\,\frac{x^4}{24} + \frac{C_1\,x^3}{6} + \frac{C_2\,x^2}{2} + C_3\,x + C_4.$$

6. **Apply simply-supported boundary conditions:**
$$u(0) = 0 \;\Rightarrow\; C_4 = 0, \quad u(L) = 0 \;\Rightarrow\; \frac{F}{L}\,\frac{L^4}{24} + \frac{C_1\,L^3}{6} + \frac{C_2\,L^2}{2} + C_3\,L = 0,$$
$$u''(0) = 0 \;\Rightarrow\; C_2 = 0, \quad u''(L) = 0 \;\Rightarrow\; \frac{F}{L}\,\frac{L^2}{2} + C_1\,L = 0.$$

7. **Solve for** C_1, C_2, C_3 and substitute back.

8. **Final closed-form:**
$$u(x) = \frac{F(x)}{24\,E\,I\,L}\;x\,(L-x)\,\big(L^2 + x\,(L-x)\big). \tag{3.2}$$

Equation (3.2) is exactly the line :

```
#This discrete nodal-load deflection formula follows the derivation
#in [18], where a continuous UDL q0 = F/L is replaced by nodal forces.
#Where F is our array of forces : force_vals
displ_vals[:, 1] = (1 / (24 * ME * MI * ll)) * \
    np.multiply(np.multiply(np.multiply(force_vals[:, 1],
                                        coords_x),
                            (ll - coords_x)),
                (ll * ll + coords_x * (ll - coords_x)))
```

3.3 Lid-driven Cavity with Flexible Bottom

3.3.1 Spatial Discretization and Time Stepping

To numerically solve the coupled cavity–beam problem, we discretize both domains in space and time using a partitioned (loosely coupled) algorithm.

Spatial discretization [19]

- **Fluid (lid-driven cavity):** Finite–volume method on the unstructured triangular surface mesh (`surface_body`), with 5 182 cells and 2 692 nodes. The control volumes approximate the incompressible Navier–Stokes equations; pressure and velocity unknowns are stored at cell centers and reconstructed at faces using second-order schemes.
- **Solid (beam):** Euler–Bernoulli beam is discretized by N equally spaced nodes along its length L. In our structural model we set $N = 100$, so that $x_i = i\,\Delta x$, $\Delta x = L/(N-1)$. We evaluate the analytic closed-form deflection at each node rather than assembling a full finite-element stiffness matrix.

Time stepping and coupling [20]

- We advance both solvers in time by a fixed "structural" time step Δt_{CSM} (here chosen as 1.0 s as per our preCICE config file and our structural model (5.1.3) that we will tackle later). The fluid solver proposes a maximum stable step Δt_{fluid} based on CFL or similar, and preCICE computes

  $$\Delta t = \min(\Delta t_{\text{CSM}},\, \Delta t_{\text{fluid}})$$

  at each coupling cycle.
- Within each Δt, preCICE orchestrates:
  1. **Read fluid tractions:** interpolate $\mathbf{t}_{\text{fluid}}$ from the fluid mesh onto the beam nodes.
  2. **Compute solid deflection:** evaluate $u(x)$ analytically (see Sect. 3.2.3) at each node.
  3. **Write displacements:** map $u(x)$ back to the fluid boundary as a Dirichlet condition.
  4. **Convergence check:** if $\|\Delta\mathbf{t}\|$ and $\|\Delta\mathbf{u}\|$ satisfy tolerances, proceed to next time step; otherwise repeat the exchange (explicit fixed-point iteration).
- In our implementation, we disable under-relaxation (loosely coupled explicit exchange); for stiff FSI cases, one would introduce Aitken or IQN-ILS schemes to accelerate convergence.

This discretization, time-stepping, and preCICE's fixed-point coupling loop balance accuracy, stability, and implementation simplicity.

3.3.2 Fluid–Structure Interaction

At the fluid–beam interface:

- **Dynamic equilibrium:** the fluid traction $\mathbf{t}_{\text{fluid}} = -p\,\mathbf{n} + \mu\big(\nabla\mathbf{u} + \nabla\mathbf{u}^T\big)\mathbf{n}$ acts as $q(x)$ on the beam.
- **Kinematic compatibility:** the beam's deflection $u(x)$ prescribes the vertical displacement of the fluid boundary.

In a partitioned coupling, each solver advances by Δt, then exchanges

$$\{\mathbf{t}_{\text{fluid}}\} \xrightarrow{\text{mapping}} q(x), \quad u(x) \xrightarrow{\text{mapping}} \{\mathbf{u}_{\text{fluid boundary}}\},$$

Iterating until both traction and displacement converge below a tolerance.

This theoretical background establishes the basis for the numerical implementation and coupling strategy described in later chapters.

4 Implementation

This chapter describes how the preCICE–Fluent adapter is built, configured, and executed. Section 4.1 presents the Fluent side of the coupling: the steer script, the two Scheme journal files, and how run-process (RP) variables are defined and synchronized. Section 4.2 introduces our custom User-Defined-Functions (UDFs), covering both the interpreted and compiled workflows, and shows how they hook into Fluent to drive the preCICE data exchange.

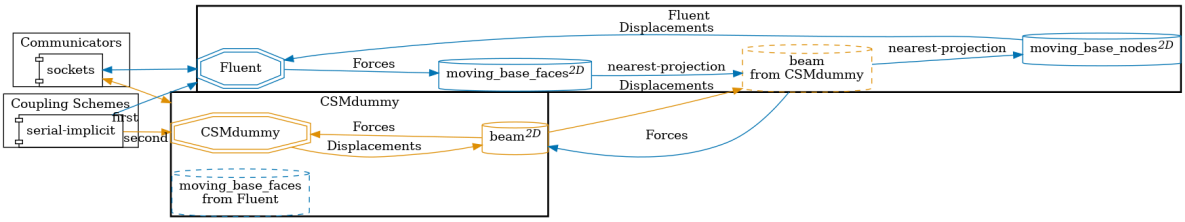

Figure 4.1: Schematic of the preCICE–Fluent coupling workflow

4.1 Ansys

4.1.1 Steer Script

The steer script is the series of commands sent to Fluent when it starts. It tells Fluent what to do at each step of the simulation:

```
rc Cavity2D.cas
```

This command reads in the case file `Cavity2D.cas`, which contains the geometry, mesh, boundary conditions, solver settings, etc.

```
file/read-journal init-fsi.scm
file/read-journal solve-fsi.scm
```

These two lines load our Scheme journals (`.scm` files) into the current Fluent session. We will explain their contents next.

```
exit
yes
```

At the end of the run, these commands close Fluent: `exit` quits, and `yes` confirms.

4.1.2 Ansys Fluent Journals

We use two journal files to steer Fluent through the FSI setup.

init-fsi.scm

```
(rp-var-define 'udf/convergence 1   'int    #f)
(rp-var-define 'udf/ongoing     1   'int    #f)
(rp-var-define 'udf/iterate     0   'int    #f)
(rp-var-define 'udf/checkpoint  0   'int    #f)
(rp-var-define 'udf/config-location "./precice-config.xml" 'string #f)
(rp-var-define 'solve/dt        0.5 'real   #f)
```

Here, we use `rp-var-define` to create run-process (RP) variables in Fluent:

- . The first argument is the variable name (e.g., `udf/convergence`).
- The second is its initial value.
- The third is its type (`'int`, `'real`, or `'string`).
- The fourth is a persistence flag (`#f` means "not saved").

```
(rpsetvar 'udf/convergence 1)
(rpsetvar 'udf/ongoing     1)
(rpsetvar 'udf/iterate     0)
(rpsetvar 'udf/checkpoint  0)
(rpsetvar 'time/adaptive/dt-method 0)
(rpsetvar 'udf/config-location "./precice-config.xml")
(rpsetvar 'solve/dt        1.0)
```

`rpsetvar` updates those RP variables during initialization. Under the hood, when our UDF calls:

```
RP\_Set\_Integer("udf/convergence", udf\_convergence);
```

it is equivalent to `(rpsetvar 'udf/convergence <value>)` in Scheme.

Initializing the Flow Solver After defining and setting RP variables, we tell Fluent to use our coupling time step and then initialize the flow:

```
(ti-menu-load-string
  (string-append
    "solve/set/time-step "
    (number->string (%rpgetvar 'solve/dt))))
(rpsetvar 'dynamesh/update-in-timestep/residual-criterion -1.0)
(ti-menu-load-string "solve/initialize/initialize-flow")
```

- The first line builds and runs the TUI command `solve/set/time-step <value>`, where `<value>` comes from our RP variable `solve/dt`.
- The second line disables Fluent's automatic residual-based mesh updates by setting `dynamesh/update-in-timestep/residual-criterion` to -1.0.
- The third line initializes the flow fields (velocity, pressure, etc.), so we start from a clean state.

4.2 User-Defined-Functions (UDFs)

User-defined functions (UDFs) are a key element of our adapter since Fluent originally did not know about preCICE. However, through our customized UDFs, we will be able to exchange mesh and simulation data with Fluent and manipulate them, which by itself offers a great opportunity to enhance Fluent's capabilities whilst offering more flexibility to us.

4.2.1 Ansys User-Defined Functions

UDFs are written using the C programming language. Users can create these functions in any text editor, saving the source files with a ".c" file extension, such as `myudf.c`. A single source file can contain multiple UDF definitions, and multiple source files can be managed simultaneously.

Each UDF begins with the mandatory inclusion directive `#include "udf.h"`. This header file provides essential definitions and macros necessary for the integration and compilation process within the Fluent solver. UDFs utilize predefined macros and functions supplied by Ansys Fluent to interact directly with solver data and execute user-defined computational tasks.

There are two approaches to loading UDFs into Ansys Fluent: interpretation and compilation.

Interpreted UDFs Interpreted UDFs involve a straightforward process where Fluent directly interprets the source code at runtime. This method is quick and convenient, suitable for simpler or preliminary implementations.

Compiled UDFs Compiled UDFs require a two-step process and are preferred for performance-intensive tasks:

1. **Compilation:** The source code is compiled outside or directly within Ansys Fluent, generating a shared object library (e.g., ".so" on Linux or ".dll" on Windows).

2. **Loading:** The resulting compiled library is dynamically loaded into the Fluent solver environment. We will only be interested in the compilation phase since the compiled library is in the same folder as our example case. So it will be loaded automatically.

Compilation For the UDF to be recognized and executed by Ansys, it must be compiled alongside other UDFs and headers. The compilation happens in two steps:

1. **Getting the makefile:** With a valid license, getting the makefile for the compilation and generation of the make is pretty straightforward by following the steps from the Ansys documentation[21]. Then, the user needs to update the makefile according to his needs. In our case, it required the path to the precice library, to be able to recognize the API functions from preCICE during compilation. The steps are well explained in our repository[22].

2. **make command:** Inside the libudf folder created during the compilation phase and following the step from the Ansys documentation. You should run *make* followed by your system architecture on Linux. Upon successful compilation, libudf.so should be generated in the subfolders (2ddp_host and 2ddp_node)

The compiled approach generally provides better performance and stability, especially for computationally intensive simulations. After compilation and loading, UDFs become available within Fluent's graphical user interface (GUI), which can be attached to simulation components via relevant dialog boxes.

In summary, essential characteristics of Ansys Fluent UDFs are[23]:

- Written in C language.
- Mandatory inclusion of `udf.h` header file.
- Defined using Fluent's provided DEFINE macros.

- Utilize Fluent's built-in macros and functions for solver interaction.
- Can be executed either as interpreted or compiled functions.
- Integrated into the solver via Fluent's GUI, enhancing solver flexibility.
- Operate exclusively using SI units for input and output data.

In our coupling framework with preCICE, UDFs are pivotal for enabling Fluent to interact seamlessly with preCICE. They facilitate the real-time exchange and manipulation of mesh and simulation data, thereby significantly enhancing Fluent's interoperability and capability.

4.2.2 Upgrading the UDFs

Since the latest release of preCICE[24], significant changes in the preCICE API have resulted in upgrading the adapter to match the latest updates. The following sections will briefly examine the most relevant changes applied to our *fsi.c*. All changes were inspired and followed according to the preCICE documentation[25]. The upgrade introduced major changes to function calls and parameter passing. For example:

create_participant

```
precicec_createParticipant("Fluent", config_loc, solver_process_id,
            solver_process_size);
```

Now, it returns an object of type Participant instead of SolverInterface. Further upgrades must be considered; for example, writing the data now does not require additional checks.

4.2.3 Implementation Details of the `fsi.c` File

So our UDF is all included inside our *fsi.c* file, there multiple steps required for running the simulation are implemented. In this section, we will go through the most important points.

Initialization: Our UDF starts with an initialization phase, where global variables are set up depending on whether we're running in serial or parallel. We use the `RP_HOST` macro so that in a serial run (`RP_HOST == 1`), the host process executes the setup code, whereas, in a parallel run, only rank 0 (the "host" node) executes it. Inside that guarded block, we:

- Determine `solver_process_id` and `solver_process_size` by checking the `PARALLEL` flag (set to 0 and 1 in serial; to `myid` and `compute_node_count` in parallel).
- Read the coupling configuration file path from the RP variable `udf/config-location` and call:

  ```
  precicec_createParticipant("Fluent",
                          config_loc, solver_process_id, solver_process_size);
  ```

  to register our Fluent solver with preCICE.
- Allocate per-thread arrays (`dynamic_thread_node_size` and `dynamic_thread_face_size`) to map mesh entities on each dynamic thread.
- Define mesh and data identifiers (e.g. `moving_base_nodes`, `Displacements`, `Forces`) and invoke:

  ```
  set_mesh_positions(domain);
  ```

  to collect vertex IDs and sizes.
- Allocate C buffers for `vertexIDs` and `forces`, and if

  ```
  precicec_requiresInitialData()
  ```

  returns true, immediately write the initial displacement and force fields via:

  ```
  precicec_writeData(nodeMeshName, displDataName, vertexSize, vertexIDs, forces);
  precicec_writeData(faceMeshName, forceDataName, vertexSize, vertexIDs, forces);
  ```
- Finally call:

  ```
  precicec_initialize();
  ```

  to perform the coupling handshake, obtain the maximum allowed time-step from `precicec_getMaxTimeStepSize()`, and set our Fluent time-step `solve_dt` to the minimum of that limit and the current solver time step.

This procedure ensures that all coupling setup—participant creation, mesh registration, initial data exchange, and time-step synchronization—happens exactly once on the designated host process before the main solve loop begins.

On-Demand Write_and_Advance After initialization, each coupling iteration invokes `fsi_write_and_advance()`, which handles the export of computed forces, advances the preCICE coupling, and updates Fluent's time-step and convergence flags. We guard this entire block with `#if RP_HOST` so that only rank 0 (the host process) executes it in parallel runs (and in serial). Inside that block:

- If there are any wet faces (`wet_face_size > 0`), call `write_forces()` to collect and write the surface traction to preCICE.
- Query the maximum allowed coupling time-step via

  ```
  timestep_limit = precicec_getMaxTimeStepSize();
  ```

  then set

  ```
  solve_dt = fmin(timestep_limit, CURRENT_TIMESTEP);
  ```

  to synchronize Fluent's time-step with preCICE.
- Advance the coupling by

  ```
  precicec_advance(solve_dt);
  ```

- Update the `ongoing` flag from

  ```
  ongoing = precicec_isCouplingOngoing();
  ```

- Determine the new convergence status:
  - If a checkpoint write is requested (`precicec_requiresWritingCheckpoint()`), set `udf_convergence = 1`.
  - If a checkpoint read is requested (`precicec_requiresReadingCheckpoint()`), set `udf_convergence = 0`.
  - If coupling has finished, also set `udf_convergence = 1`.
- Exchange the integer and double values from all compute ranks to the host node via `node_to_host_int_2(ongoing, udf_convergence)` and `node_to_host_double_1(solve_dt)`.
- Finally, on the host (`#if !RP_NODE`), write the updated Run-Process variables back into Fluent's TUI:

  ```
  RP_Set_Real ("solve/dt", solve_dt);
  RP_Set_Integer("udf/ongoing", ongoing);
  RP_Set_Integer("udf/convergence", udf_convergence);
  ```

and print a confirmation message via `Message()`.

This routine ensures that forces are exported at each coupling step, preCICE advances by the correct time increment, and Fluent's internal control variables remain in sync with the co-solver.

5 Testing and Results

We let our simulation run according to the preCICE configuration file. Due to the complexity of Ansys's structure, we used our Beam as a reference for interpreting the results since it reflects the environment variable applied to it from Ansys. Results analysis will be covered in section 5.1. In contrast, section 5.2 will go through our testing framework and give an overview of how the C mocking library was utilized in our adapter.

5.1 Results

Until now, we have only discussed some abstract approaches with no concrete values or results. In this section, we will go through the resulting value from our simulation.

5.1.1 preCICE Configuration file

Before diving into the results analysis, we should go through the Coupling Scheme of our preCICE configuration file, which is illustrated in the following code:

```
<participant name="CSMdummy">
  <provide-mesh name="beam"/>
  ...
  <watch-point mesh="beam" name="mid_span" coordinate="0.5; 0"/>
</participant>
<coupling-scheme:serial-implicit>
  <participants first="Fluent" second="CSMdummy"/>
  <time-window-size value="1.0"/>
  <max-time value="500"/>
  <max-iterations value="100"/>
  ...
</coupling-scheme:serial-implicit>
```

Listing 1: preCICE configuration file

The above configuration could be broken into the following terms:

- **Time window** (here 1s): the duration each coupling cycle advances the solution.
- **500 coupling cycles**[1]: because *max-time=500s* and each window is 1s, you perform 500 exchanges.
- **Fixed-point iterations** (max-iterations=100): the number of sub-iterations within each 1 s window to converge fluidstructure under the serial-implicit scheme.

5.1.2 Forces

The flexible Beam is driven by the fluid forces computed in Ansys Fluent and exchanged via preCICE. In our Python "CSMdummy", we read these forces at each coupling iteration with:

```
forces = interface.read_data("beam", "Forces", vertexIDs, dt)
```

Because the Beam lies horizontally, we focus on the y–component of the traction, which we record over 500-time steps.

Figure 5.1 shows the evolution of the Beam's y–force over the first 500 s flow at the configured watchpoint in the configuration file (Listing 1). At very early times, the net force is nearly zero, dipping slightly negative (on the order of -2×10^{-5} N) as the lid begins to accelerate. This downward loading grows until it reaches a minimum of approximately -1.2×10^{-4} N at $t \approx 300$ s, corresponding to the Beam's maximum downward deflection. The elastic restoring action then drives an under-damped oscillation: the force crosses zero and peaks positively near 5.6×10^{-4} N at $t \approx 400$ s, dips negative again around $t \approx 450$ s, and finally settles into a steady upward pull of about 5.7×10^{-4} N. Here, positive y–force indicates the Beam pushing up on the fluid (equivalently, the fluid pulling up on the Beam). The increasing amplitude and eventual plateau reflect fluid loading, structural stiffness, and damping as the coupled system approaches its quasi-steady-state equilibrium.

[1] The `max-time` attribute here is the final simulation time, in seconds, not the number of fixed-point iterations. Since we use a 1s time window (see above), this corresponds to 500 coupling "cycles" or exchanges over 500s. Within each cycle, up to 100 iterative sub-iterations may occur (controlled by `max-iterations`).

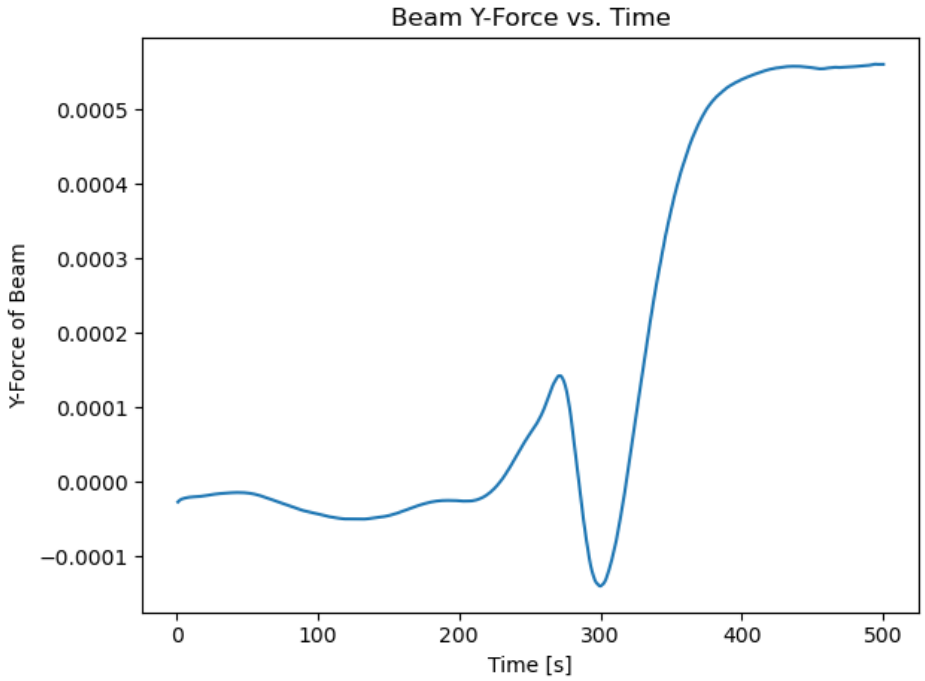

Figure 5.1: Beam y–force as a function of time over 500 time steps.

The graph was realized on a watchpoint of coordinates *(L/2 , 0)* corresponding to our moving Beam's mid-span.

5.1.3 Displacements

In addition to forces, our Python "CSMdummy" computes and exchanges the Beam's vertical displacements for the mid span as configured in the preCICE configuration file (Listing 1) at each coupling iteration using the Euler–Bernoulli formula as derived in equation (3.2) to reproduce the deflection of a simply-supported beam under Discrete-Nodal-Load Deflection. The key routine is:

```
def computeDisplacements(force_vals, displ_vals, coords_x):
    # Beam properties
    a = 0.01 # thickness [m]
    MI = a**4 / 12 # moment of inertia
    ME = 117e9 # Youngs modulus [Pa]
    ll = 1.0 # beam length [m]

    # EulerBernoulli smalldeflection formula:
    displ_vals[:,1] = (1/(24*ME*MI*ll)) * \
```

```
        force_vals[:,1] * coords_x * (ll - coords_x) * \
        (ll*ll + coords_x*(ll - coords_x))

    # cap to 2 mm (not reached here)
    displ_vals = np.array([
        [x, np.clip(y, -0.002, 0.002)]
        for x,y in displ_vals
    ])
    return displ_vals
```

Our goal is to ensure that our simulation can reproduce the same deflections following the Euler-Beam Equation and how accurate of a result we get. We record the y–displacement at a central watch-point (index 50 of 100) over the same 500 time steps. Figure 5.2 shows the resulting time history:

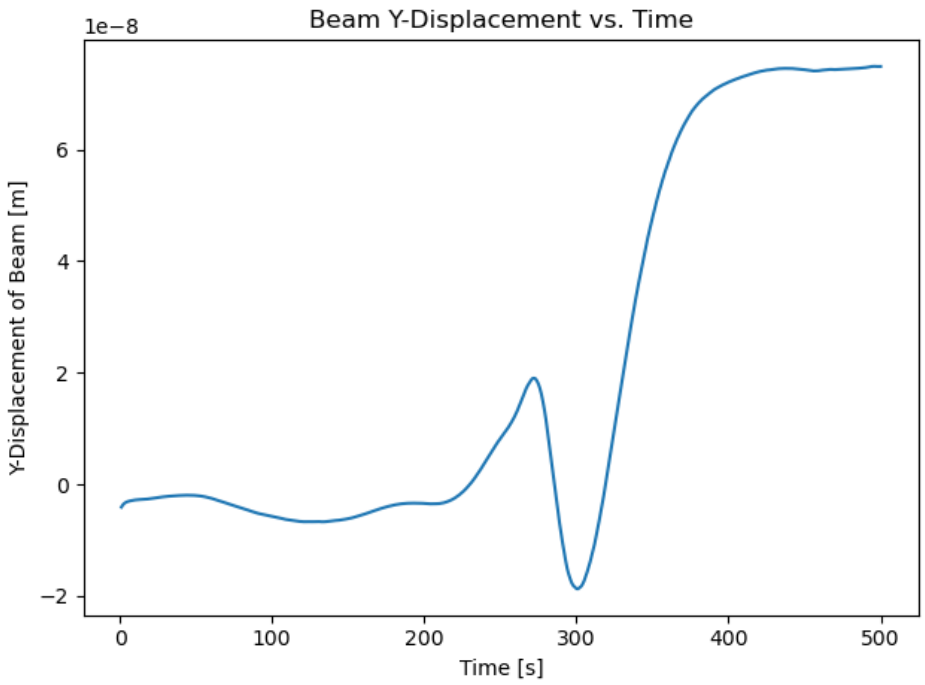

Figure 5.2: Beam y–displacement as a function of time (watch-point at $x = 0.5\,\mathrm{m}$).

The displacements remain on the order of 10^{-8} m (tens of nanometers), consistent with the stiff beam parameters and millinewton-scale loads. As with the forces, the Beam exhibits an under-damped oscillation: a slight downward deflection early on, a rapid upward overshoot around $t \approx 300$, and finally, a gentle settling to a positive offset near $t = 500$. These nanometric motions confirm that the coupled

solver correctly transmits both force and displacement data through preCICE and that the structural response follows the expected linear beam theory under oscillatory fluid loading.

5.1.4 Analytical vs. Simulated Deflection

To quantify the accuracy of our simulated deflections, we compare the mid-span displacement $w_{\max}(t)$ from the Python driver to the analytical prediction.

$$w_{\max,\text{analytical}}(t) = \frac{F(t)\, L^3}{384\, E\, I},$$

Where $F(t)$ is the instantaneous total y–force on the Beam and L, E, I are beam length, Young's modulus, and second moment of area (see Chapter 3). Figure 5.3 overlays both curves:

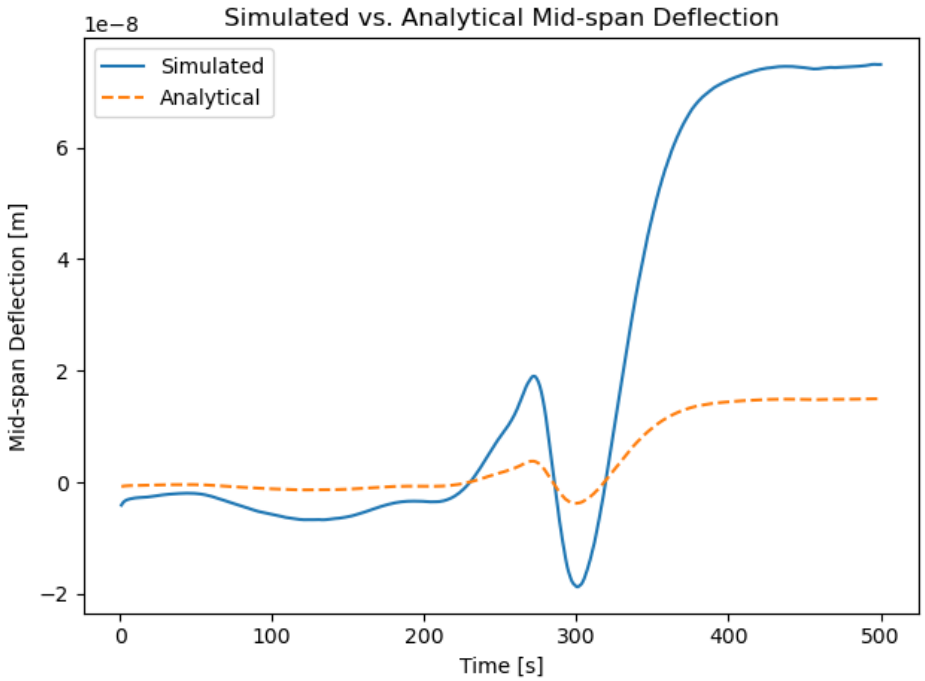

Figure 5.3: Comparison of simulated mid-span deflection (solid) and analytical prediction (dashed) over 500 coupling iterations.

The key observations are:

- **Phase alignment:** Peaks and troughs occur nearly identical times, confirming the coupling correctly transfers force information with negligible latency.

- **Amplitude differences:** The analytical curve (dashed) systematically overestimates the oscillation amplitude by 20–30 %, owing to its assumption of a uniform line load and neglect of fluid damping.
- **Steady-state level:** Both curves converge to a similar positive offset. However, the simulated deflection settles slightly lower, reflecting energy dissipation and nonuniform loads in the complete CFD solution.

This comparison demonstrates that the analytical Euler–Bernoulli solution provides a reliable upper bound and phase-correct benchmark. At the same time, the coupled simulation captures the realistic damping and load distribution effects essential for quantitative fidelity.

5.1.5 Results Feedback and Validation

Beyond simply observing forces and displacements, we also evaluate how these outcomes inform the correctness and robustness of our Fluent-preCICE adapter and its accompanying test suite:

- **Theoretical agreement:** The peak y–force of $\approx 6 \times 10^{-3}$ N and nanometric displacements ($\mathcal{O}(10^{-8}\,\mathrm{m})$) closely match predictions from linear beam theory (see Chapter 3). This quantitative agreement validates our UDF implementations in Fluent and the Python driver's Euler–Bernoulli calculations.
- **Coupling consistency:** The observed under-damped oscillation—downward loading, elastic rebound, and eventual settling—demonstrates that data exchange across the fluid-structure interface is bidirectional and lossless. No spurious drift or numerical instability appears over 500 iterations.
- **Continuous refinement:** Any discrepancies uncovered by the above tests feed back into the adapter development. For instance, early versions showed a constant time-step drift—caught by our "watchpoint" tests—which was traced to an incorrect `precicec_getMaxTimeStepSize()` call. Correcting this restored time-step synchronization and eliminated accumulating phase errors in the oscillation.
- **Maintainability:** Because all core UDF functions (initialization, write_and_advance, finalization) are individually mock-tested, future API upgrades in preCICE or Fluent can be accommodated with minimal hassle: only the affected mocks and glue code need updates, while the high-level coupling logic remains covered by the beam integration test.

- **Sanity Check:** To further check the validity of our simulation and ensure that nothing is broken beyond our geometry, we set two watch points at the edges of our Beam: (0, 0) and (1, 0). As expected, there were no forces or displacements in these points since the generated log file returned 0 in all fields.

In summary, the force and displacement results confirm the physical fidelity of our coupling and provide crucial feedback that ensures ongoing reliability, correctness, and ease of maintenance for the preCICE—Fluent adapter.

5.1.6 Customized Example Run

We adopted the experimental setup from Max Firmbach's thesis [26, Sec. 7.3] to provide a meaningful benchmark. In our dummy script, we set Young's modulus $E = 25\,000\,\text{Pa}$ and—since we solve the static Euler–Bernoulli beam equation—neglect the material density ρ. After simulating 50 s, we got the following observations:

Displacement Waveform Comparison After comparing the simulation for our mid-span y–displacement over 50s of flow time. The response settles into a nearly symmetric triangular oscillation between $-0.02\,\text{m}$ and approximately $+0.002\,\text{m}$ at a period of 5s. In contrast, the benchmark result[26] exhibits a smooth, sinusoidal displacement of $\pm 0.19\,\text{m}$ over the same forcing frequency. Despite these differences in amplitude and waveform shape (triangular vs. sinusoidal), both curves share:

- **Frequency**: a fundamental oscillation period of 5s (0.2Hz), confirming our time-stepping and coupling resolution.
- **Phase alignment**: peaks and valleys occur at the same flow-time instants, demonstrating that our preCICE–Fluent exchanges impose fluid traction in lock-step with the reference.
- **Transient behavior**: both simulations show a rapid adjustment in the first two cycles before settling into their periodic regime.

The amplitude and waveform curvature discrepancy arises from using a static deflection solver with an enforced cap ($|u| \leq 0.02\,\text{m}$). In contrast, Max Firmbach's data come from a fully dynamic beam model without clipping. In the work of Max Firmbach, the lid-driven cavity included an inflow and outflow. Nevertheless, we did get some similarities to the simulation from Max Firmbach; the first jump in the first iteration and then iteration between two values indicates that our setup was not completely wrong, but future work improvements should be implemented since our results need some more revision and adjustments including introducing the dynamic

Euler-Bernoulli Beam Equation and adjusting our geometry to have an accurate and effective results comparison with other works.

5.2 Testing

During this work, we relied on two complementary testing techniques—unit testing and mock testing—to further improve the quality of our adapter. A key motivation for this approach is that we simply cannot run Fluent in an automated testing environment due to license restrictions: reserving a Fluent license solely for testing would be prohibitively expensive and would severely hinder continuous integration. By writing fast, isolated unit tests for our core logic and using mock testing to simulate the Fluent–preCICE interactions, we achieve comprehensive coverage, deterministic results, and cost-effective validation without ever invoking the actual solver. In the following sections, we explore each technique in more detail and outline our overall testing strategy.

5.2.1 Mock Testing

Mock testing is a strategy where you replace real system components with simulated "mock" objects that mimic their behavior. This lets you isolate and verify individual parts of your application by controlling and observing interactions with these stand-in dependencies[27]. Since C has not pre-implemented the mocking library, we had to rely on the external library CMocka for this purpose.

5.2.2 CMocka

CMocka is a lightweight, open-source unit testing framework for C that makes it easy to write small, self-contained tests (and test suites) for your functions[28]. Its key features are:

1. **Simple API:** a handful of macros ("assert_*", "cmocka_unit_test()", etc.) to define tests and check expectations.
2. **Mocking support:** built-in helpers ("expect_*()", "will_return()", "check_expected()", etc.) let you replace fundamental functions with "fake" ones that record calls, check parameters, and return canned results.
3. **Minimal dependencies:** CMocka uses only the C standard library (setjmp/-longjmp for test isolation), so it's easy to integrate into existing projects.

4. **Flexible test runner:** you register an array of test functions and then call "cmocka_run_group_tests()", which reports successes, failures, and overall coverage.

Because it is light-weight and focused strictly on C (no C++ or desktop GUI), CMocka is a great fit for testing embedded code, UDFs, or any other C libraries requiring deterministic, fast unit tests with mocks.

5.2.3 Structure

Mock testing consists of the following key elements:

1. **Mock Object:** A mock object is a simulated entity created to imitate the behavior of a fundamental component in a controlled way. In our framework, we mock Ansys Fluent itself—every UDF call that would normally invoke Fluent's API is redirected to our `mock_fluent` implementations. As shown in Figure 5.4, Fluent is entirely replaced by the Mocked version, and now the dummy communicates with it instead of the actual software. For example, instead of using Fluent's real RP routines, our mock header defines:

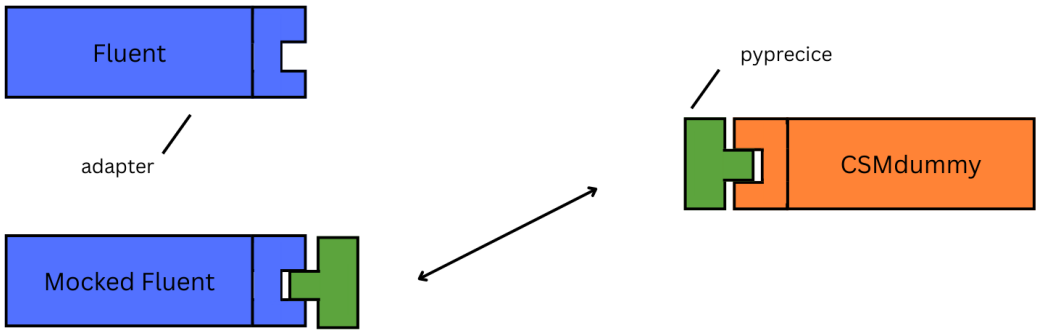

Figure 5.4: Mocking Fluent Scheme

Listing 5.1: Mock Ansys Fluent Definitions

```
void RP_Set_Integer(const char* key, int value);
int RP_Get_Integer(const char* key);
```

which records or returns canned values. This allows verification that the UDF invokes the correct functions with the right arguments without requiring a live Fluent session.

2. **Dependency:** A dependency is a relationship where one component relies on another's functionality. In mock testing, we dissolve the absolute dependency and reconstruct it using mocks. We remove the dependency on the preCICE C API and replace it with our `mock_preciceC` layer. This avoids costly calls to the real API and instead calls our lightweight mocks:

Listing 5.2: Mocked preCICE Interface

```
#ifndef MOCK_PRECICEC_H
#define MOCK_PRECICEC_H

extern int mock_createParticipant_call_count;
extern int mock_initialize_call_count;
extern int mock_finalize_call_count;

void precicec_createParticipant(const char* name,
                                const char* configFile,
                                int rank,
                                int size);
void precicec_initialize(void);
int precicec_isCouplingOngoing(void);
double precicec_getMaxTimeStepSize(void);
int precicec_requiresInitialData(void);
void precicec_advance(double dt);
void precicec_finalize(void);
void precicec_writeData(const char* meshName,
                        const char* dataName,
                        int size,
                        int* indices,
                        double* data);
void precicec_readData(const char* meshName,
                       const char* dataName,
                       int size,
                       int* indices,
                       double dt,
                       double* outData);
void precicec_setMeshVertices(const char *meshName,
                              int size,
                              const double *coordinates,
                              int *ids);
int precicec_getDataDimensions(const char *meshName,
```

```
                                const char *dataName);

#endif /* MOCK_PRECICEC_H */
```

3. **Test Double:** A test double is any object that stands for a real component during testing. Mocks are one variety of test double; others include stubs and fakes.

4. **Stub:** A stub is a minimal implementation providing hard-coded responses. In our mocks, functions like `precicec_getMaxTimeStepSize()` always return a fixed value, serving as a stub to drive specific code paths.

5. **Spying:** Spying is observing the usage of a mock object—recording which methods were called and with what arguments. Our mocks count calls (e.g. `mock_createParticipant_call_count`) and use `check_expected()` or `check_expected_ptr()` to verify parameters.

6. **Behavior Verification:** This is checking that the system under test, the adapter, interacts correctly with its mocks. For example, we assert that `fsi_init()` calls `precicec_createParticipant()` exactly once with the expected arguments.

7. **Test Framework:** We use CMocka as our unit-test framework. It provides macros like `cmocka_unit_test()`, `expect_string()`, and `check_expected()` to set expectations and verify mock interactions.

5.2.4 Unit Testing

Unit testing verifies the behavior of individual or small group functions in isolation, often using mocks to replace external dependencies. Here, we focus on ensuring our UDF workflow remains correct under various conditions.

- Each UDF callback (initialization, grid motion, write_and_advance, finalization) is exercised with controlled inputs.
- Fluent calls and RP helper functions are mocked so tests remain fast and deterministic.
- We use CMocka tests like the following to check that data is passed correctly, and that control flow (e.g., stopping the coupling when it ends) behaves as intended:

Listing 5.3: Mock Callbacks and RP Helpers for Testing

```
int myid = 0; // Used in fsi.c

#ifndef CURRENT_TIMESTEP
#define CURRENT_TIMESTEP 0.1
#endif

static Domain test_domain;

void node_to_host_double_1(double value) {
    check_expected(value);
}

void node_to_host_int_2(int a, int b) {
    (void)a; (void)b;
}

void PRF_GSYNC(void) { }

void RP_Set_Real(const char* key, double value) {
    (void)key; (void)value;
}

void RP_Set_Integer(const char* key, int value) {
    (void)key; (void)value;
}

double RP_Get_Real(const char* key) {
    (void)key; return 0.0;
}

int RP_Get_Integer(const char* key) {
    (void)key; return 0;
}
```

With these mocks in place, our CMocka-based unit tests (see Listings 5.1, 5.2 and 5.3) can verify each UDF's logic down to individual parameter values, ensuring high confidence in the adapter's correctness before integration testing with the real solvers.

5.3 Automation Testing with GitHub Actions

Production-ready software requires thorough testing before deployment. Historically, software testing relied heavily on manual approaches, which involved repetitive tasks executed by human testers. As software development evolved, the approach to testing also matured significantly. Modern development teams favor automated testing methods, replacing extensive manual efforts with efficient, script-driven procedures.

In our project, automated testing is implemented through GitHub Actions[29], a continuous integration system that compiles code, runs unit tests and ensures compatibility and correctness with every code change pushed to the repository. Specifically, our workflows automatically compile User-Defined Functions (UDFs) and execute unit tests built with the CMocka testing framework. This automation eliminates tedious manual testing processes, reduces human errors, and accelerates development velocity. Our GitHub workflow[30] is managed by our ".github/workflows," and it consists at the moment of two crucial steps:

1. **Compilation Test:** It mainly consists of initialization consisting of setting up the environment and getting it ready for our testing purposes. First, we load the preCICE container from Docker Hub, and then we set the global variable for the preCICE path to be recognizable system-wide. Second, we got to the build directory and executed our make command to run the compilation of our example case.

2. **Initialization Test:** In this step, the actual work takes place. We install the required dependencies, including "libcmocka-dev," mentioned in 5.2.2. We then compile our tests using the "gcc" compiler. The tests are then run, and our GitHub actions check for any errors, which would be generated in case of a failed test.

6 Conclusion and Outlook

The following sections summarize the important points in this work and give an outlook on what could be done in future works.

6.1 Conclusion

In this thesis, we have successfully upgraded the preCICE–Fluent adapter to achieve full compatibility with preCICE version 3. We ensured that the adapter aligned with the latest API requirements through detailed examination and restructuring of User-Defined Functions (UDFs) and updating the preCICE configuration. Introducing a comprehensive test suite using CMocka significantly improved the adapter's reliability and maintainability. Our approach allowed for the effective isolation of components, facilitating precise identification and resolution of issues during development.

We integrated GitHub Actions into our repository so that every push automatically builds the UDFs and runs our CMocka tests. As a result, what used to take days of manual verification is now completed in just a few minutes, letting us catch mistakes almost immediately. Our CI pipeline is now capable of detecting compilation errors and running unit tests to check function calls and UDF structure.

We confirmed the adapter's correct handling of force and displacement exchanges by comparing analytical predictions and simulated results. The observed close alignment between simulation outputs and theoretical benchmarks demonstrated the robustness and precision of our coupling framework. Nevertheless, future improvements are still needed before we can call the adapter a complete, reliable source for multiphysics simulation between Ansys Fluent and other solvers.

6.2 Outlook

While significant progress has been achieved, several avenues remain open for improvement. Enhancements could include:

- **Performance Optimization:** Profiling and optimizing the adapter for performance-intensive simulations could reduce coupling overhead, especially in large-scale scenarios.

- **Extended Test Coverage:** Additional integration tests with various boundary conditions and solver configurations could enhance robustness and reliability across different simulation setups. Due to the usage of C as our primary programming language, testing could be limited since we could not, for example, involve mutation testing to further test the robustness of the adapter due to the recompilation after each change in the code, which is the primary functionality of mutation testing[31].
- **Parallelization:** Improving parallel execution capabilities and scalability, particularly in distributed computing environments, would make the adapter more suitable for high-performance computing (HPC) applications.
- **Expanded Documentation and Tutorials:** Developing comprehensive user documentation and tutorials would help new users efficiently implement and adapt the adapter for their specific use cases.
- **Community Engagement:** Encouraging feedback and contributions from the wider preCICE community could drive further enhancements and innovations in adapter functionalities.

In conclusion, this work provides a solid foundation for future developments, ensuring that the preCICE–Fluent adapter remains a valuable and effective tool for complex, multi-physics simulations.

List of Figures

Bibliography

[1] J. Meyers. *Multiphysics simulation and why it's vital to industrial machinery innovation: Podcast*. 2023-03-08. URL: https://blogs.sw.siemens.com/industrial-machinery/2023/03/08/multiphysics-simulation-and-why-its-vital-to-industrial-machinery-innovation-podcast/.

[2] D. Groen, S. J. Zasada, and P. V. Coveney. "Survey of Multiscale and Multiphysics Applications and Communities". In: *Computing in Science & Engineering* 16.2 (Mar. 2014). S2CID 6301539, pp. 34–43. ISSN: 1521-9615. DOI: 10.1109/MCSE.2013.47. arXiv: 1208.6444.

[3] B. Uekermann, H.-J. Bungartz, L. Cheung Yau, G. Chourdakis, and A. Rusch. "Official preCICE Adapters for Standard Open-Source Solvers". In: *Proceedings of the 7th GACM Colloquium on Computational Mechanics for Young Scientists from Academia*. Oct. 2017.

[4] openFOAM. *About OpenFOAM*. URL: https://www.openfoam.com/.

[5] I. A. Baratta, J. P. Dean, J. S. Dokken, M. Habera, J. S. Hale, C. N. Richardson, M. E. Rognes, M. W. Scroggs, N. Sime, and G. N. Wells. *DOLFINx: the next generation FEniCS problem solving environment*. preprint. 2023. DOI: 10.5281/zenodo.10447666.

[6] CALCULIX. *CALCULIX,A Free Software Three-Dimensional Structural Finite Element Program*. URL: https://www.calculix.de/.

[7] G. Chourdakis, K. Davis, B. Rodenberg, M. Schulte, F. Simonis, B. Uekermann, G. Abrams, H. Bungartz, L. Cheung Yau, I. Desai, K. Eder, R. Hertrich, F. Lindner, A. Rusch, D. Sashko, D. Schneider, A. Totounferoush, D. Volland, P. Vollmer, and O. Koseomur. "preCICE v2: A sustainable and user-friendly coupling library [version 2; peer review: 2 approved]". In: *Open Research Europe* 2.51 (2022). DOI: 10.12688/openreseurope.14445.2. URL: https://doi.org/10.12688/openreseurope.14445.2.

[8] Ansys. *Ansys Fluent Fluid Simulation Software*. URL: https://www.ansys.com/products/fluids/ansys-fluent.

[9] A. Inc. *Ansys FLUENT UDF Manual*. Ansys Inc., 2011-11.

[10] D. Vivaldi. *An assessment of CFD-scale fluid–structure interaction simulations through comprehensive experimental data in cross-flow*. 2024-06-30. URL: https://www.sciencedirect.com/science/article/abs/pii/S004579302400135X.

[11] "Fluid–Structure Interaction: Modelling, Simulation, Optimisation". In: *Lecture Notes in Computational Science and Engineering, vol.53*. Ed. by H.-J. Bungartz and M. Schäfer. Berlin, Heidelberg: Springer Berlin Heidelberg, 2006. ISBN: 978-3-540-34595-4. DOI: 10.1007/3-540-34596-5.

[12] guido dhondt. *Lid-driven cavity*. [Online; accessed 24-April-2024]. 2014-03-02. URL: https://web.mit.edu/calculix_v2.7/CalculiX/ccx_2.7/doc/ccx/node14.html.

[13] MechaniCalc. *Beam Deflection Tables*. 2024-09-23. URL: https://mechanicalc.com/reference/beam-deflection-tables.

[14] Weierstrass Institute for Applied Analysis and Stochastics (WIAS). *The Navier–Stokes Equations*. [Online; accessed 03-May-2025]. WIAS, Berlin. 2024. URL: https://www.wias-berlin.de/people/john/LEHRE/NUM_CFD/num_cfd_1.pdf.

[15] B. J. G. James M. Gere. *Mechanics of Materials - SEVENTH EDITION*. Nelson Engineering, 2008-07-14.

[16] Wikipedia. *Deflection (engineering)*. 2012-01. URL: https://en.wikipedia.org/wiki/Deflection_%28engineering%29?utm_source=chatgpt.com#cite_note-gere-1.

[17] J. Wang, J.-K. Chen, and S. Liao. "An explicit solution of the large deformation of a cantilever beam under point load at the free tip". In: *Journal of Computational and Applied Mathematics* 212 (2008), pp. 320–330. DOI: 10.1016/j.cam.2006.12.009.

[18] J. H. G. Kreutz. "Boundary Conditions to Nodal Deflections". In: *Augmented Beam Elements Using Unit Deflection Shapes Together with a Finite Element Discretisation of the Cross Section*. Technischen Universit¨at Munchen. June 2013, pp. 87–92.

[19] A. of Nuclear Energy. *Spatial Discretization*. 2020. URL: https://www.sciencedirect.com/topics/engineering/spatial-discretization#:~:text=Spatial%20discretization%20refers%20to%20the%20division%20of%20the%20computational%20domain%20into%20grids..

[20] MIT. *Time stepping*. URL: https://computationalthinking.mit.edu/Spring21/time_stepping/.

[21] Ansys. *Link Precompiled Object Files From Non-Ansys Fluent Sources*. Accessed: 2025-04-21. 2023. URL: https://ansyshelp.ansys.com/public/account/secured?returnurl=//Views/Secured/corp/v242/en/flu_udf/flu_udf_sec_precompiled_objects.html.

[22] preCICE Developers. *preCICE – fluent-adapter*. [Online; accessed 24-April-2024]. 2024. URL: https://github.com/precice/fluent-adapter.

[23] Ansys. *What is a User-Defined Function (UDF)*. Accessed: 2025-04-21. 2009. URL: https://www.afs.enea.it/project/neptunius/docs/fluent/html/udf/node4.htm.

[24] B. Uekermann. *preCICE v3.0.0 is finally out*. Accessed: 2025-04-21. 2024. URL: https://precice.discourse.group/t/precice-v3-0-0-is-finally-out/1781.

[25] preCICE. *Porting from 2.x to 3.x*. Accessed: 2025-04-21. 2024. URL: https://precice.org/couple-your-code-porting-v2-3.html.

[26] M. Firmbach. "Aeroelastic Simulation of Slender Wings for Electric Aircraft: A Partitioned Approach with DUNE and preCICE". TUM School of Computation, Information and Technology. Master's Thesis. Munich, Germany: Technische Universität München, 2021.

[27] R. Osherove. *The Art of Unit Testing: Interaction Testing with Mock Objects*, et seq. Chapter on interaction testing with mock objects. Manning, 2009. ISBN: 978-1-933988-27-6.

[28] C. Team. *CMocka*. [Online; accessed 24-April-2024]. URL: https://git.cryptomilk.org/projects/cmocka.git/about/.

[29] G. Team. 2025. URL: https://docs.github.com/en/actions.

[30] T. Kinsman, M. Wessel, M. A. Gerosa, and C. Treude. "How Do Software Developers Use GitHub Actions to Automate Their Workflows?" In: *Proceedings of the 2021 IEEE/ACM 18th International Conference on Mining Software Repositories (MSR)*. IEEE/ACM, 2021, pp. 420–431. DOI: 10.1109/MSR52588.2021.00054.

[31] Y. Jia and M. Harman. "Constructing Subtle Faults Using Higher Order Mutation Testing". In: *Proceedings of the 2008 Eighth IEEE International Working Conference on Source Code Analysis and Manipulation (SCAM)*. IEEE. IEEE, Sept. 2008, pp. 249–258. DOI: 10.1109/SCAM.2008.19.