Design and development of an automatized validation framework for the multiphysics solver m-AIA

Marco Franzreb and Luca Mann

Supervisors:
M.Sc. Miro Gondrum
Univ.-Prof. Dr.-Ing. Wolfgang Schröder

Aerodynamisches Institut Aachen
RWTH Aachen

A project thesis submitted for the degree of
Mechanical Engineering B.Sc.

August 2021

Abstract

A framework to validate the simulation data of the multi-physics solver m-AIA has been deployed. Test cases have been created, simulated and the output parameters were compared to analytical or experimental reference data. The first test case consists of a low Reynolds number Taylor-Couette flow representing an example for an analytical test case. The second test case uses experimental reference data in comparison to the simulation data from a flow around a sphere at a low Reynolds number. The framework has been designed in a way to be extendable with more test cases and features. A user friendly input standard has been created to ensure an easy to use workflow with the framework.

Contents

List of Figures

LIST OF FIGURES

List of Tables

LIST OF TABLES

List of Listings

Glossary

Roman Symbols

$\mathbf{D}$	Diameter
$\mathbf{d}$	Distance
$\mathbf{R}$	Radius
$\mathbf{U}$	Contravariant velocity vector
$\mathbf{u}$	Flow velocity vector
$\mathbf{V}^*$	Velocity in circumferential direction
$\mathbf{x}$	Vector of Cartesian coordinates

Greek Symbols

Ω	Angular velocity
ρ	Density
Θ	Angular position

Dimensional numbers

ω_b	De-dimensioned velocity
Ma	Mach number
Re	Reynolds number
Re_c	Critical Reynolds number
$\mathbf{U}^*$	Dimensionless velocity vector

Abbreviations

AIA	Institute of Aerodynamics and Chair of Fluid Mechanics
BC	Boundary condition
CLI	Command line interface
CSV	Comma-separated values
LBM	Lattice Boltzmann method
STL	Standard Triangle Language
$TOML$	Tom's Obvious Minimal Language
ZFS	Zonal Flow Solver

GLOSSARY

1 Introduction

The AIA uses its own multi-physics solver called m-AIA. Ensuring the quality of the solver the regression test case framework Canary is employed. It executes the solver and compares the simulation output to reference data obtained by previous runs. Hence, this kind of quality check relays on the solver itself and does not provide a value of accuracy based on analytical or experimental data.

For this purpose a second framework is developed within this project work. This framework shall simulate different test cases with m-AIA and compare the simulation output to an analytical or experimental reference solution. Hereby, the framework generates a quantitative evaluation of the simulations and determines their accuracy.

The framework is written in the Python programming language. Its structure is designed in such a way that it is possible to be extended in the future. The possibility of adding further test cases is taken into account during development. Creating a flexible and well structured architecture for this framework is the core task of this thesis. The applicability of the new framework is validated by creating two test cases. The first one consist of a low Reynolds number Taylor-Couette flow representing an example for an analytical test case. For the simulation of a second test case experimental data is used for the comparison. For this purpose, a flow around a sphere with a low Reynolds number is being utilised.

The high rate of change of the code requires repeated validation of accuracy. In the future, this framework will be used to validate the correctness of changes to m-AIA supporting the existing framework canary.

The thesis is structured as follows: The next chapter introduces the theory of m-AIA, Canary and the test cases. Then the development and implementation of the framework and its structure are explained. Finally, the results of the framework with the two implemented test cases are explained.

2

Theory

2.1 m-AIA

m-AIA is a multi-physics solver developed by the Institute of Aerodynamics and Chair of Fluid Mechanics (AIA) of the RWTH Aachen University [8]. It was formerly developed as the Zonal Flow Solver (ZFS). The solver is based on various methods, such as the lattice Boltzmann method or the Lagrange particle tracking, which are coupled to investigate multi-physics phenomena. The aim of this solver is to make use of the advancement of computing hardware by improving efficiency when simulating on large-scale systems. The efficiency improvement is achieved through uniting the different solvers into one framework. By running them simultaneously and transferring data in memory the communication effort is reduced. This is a weakness of current co-simulation coupling frameworks.

2.2 Canary

Canary is the testing framework currently in use to preserve the code quality of m-AIA [1, 7]. It is written in Python, and was developed with the aim of building an extensible and modular tool. Canary is divided into a library part and a part for the command line interface (CLI). The functions are nested in the library. The CLI parses command line arguments to invoke the library functions.

The framework simulates a series of test cases with m-AIA and compares the results with a reference solution. The reference solutions stems from a previous run of the solver.

As a result Canary outputs whether each test case passed or failed the test. The test is passed when the comparison per values do not exceed a certain deviation.

Canary's user interface is based on command line input. Execution is performed via a single unified command (`cry`) for all functions, which requires global parameters to specify the test. On top of that, there are also multiple sub-commands which take their own specific arguments to further define the test.

Canary's main limitation is that the comparison of test cases with its previous runs. The framework delivers a good indication on the consistency and quality of its simulation. However, the framework does not provide a value of accuracy based on analytical or experimental data. The test lacks a qualitative evaluation of the simulations and the determination of m-AIA's accuracy.

2.3 Taylor-Couette flow

The flow between two rotating cylinders was described by Taylor [21]. A stable flow between the cylinders in steady motion is described in the following section.

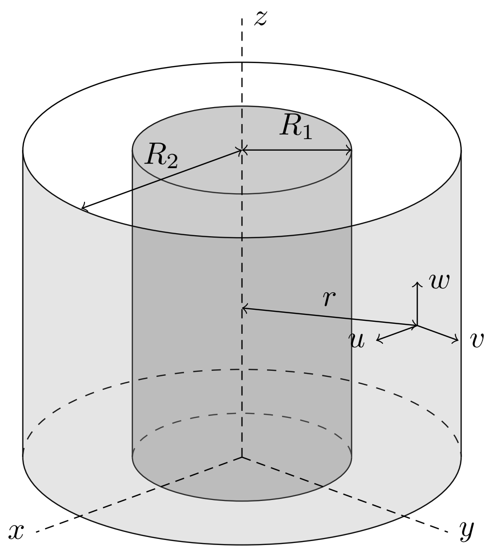


Figure 2.1: Schematic of the Taylor-Couette flow.

The setup consist of two cylinders with the radii R_1 and R_2 ($R_2 > R_1$) as shown in figure 2.1. The cylinders are rotating with their angular velocities Ω_1 and Ω_2. The velocity in circumferential direction V^* is described as

$$V^* = Ar + B/r \,, \tag{2.1}$$

with r being the distance from the axis of the cylinders. The constants A and B are correlated to the angular velocity with

$$\Omega_1 = A + B/R_1^2 \,, \tag{2.2}$$

$$\Omega_2 = A + B/R_2^2 \,, \tag{2.3}$$

so that the solution of the equations equals

$$A = \frac{R_1^2\Omega_1 - R_2^2\Omega_2}{R_1^2 - R_2^2} \,, \tag{2.4}$$

$$B = \frac{R_1^2(\Omega_1 - \Omega_2)}{1 - R_1^2/R_2^2} \,. \tag{2.5}$$

Summarising the equations the velocity in circumferential direction V^* can be calculated as

$$V^* = \frac{(R_1^2\Omega_1 - R_2^2\Omega_2)}{R_1^2 - R_2^2} * r + \frac{R_1^2(\Omega_1 - \Omega_2)}{1 - R_1^2/R_2^2} * \frac{1}{r} \,. \tag{2.6}$$

The flow between two rotating cylinders was experimentally investigated by Mallock [9] and Taylor [21]. Both results show a stable laminar flow for low rotating speeds of the outer cylinder while the inner cylinder is fixed. When a critical Reynolds number is reached, the flow transitions from laminar to turbulent, for which the equations for steady motion are no longer valid. At low rotating speeds the flow is stable, laminar and purely azimuthal.

2.4 Flow past a sphere at low Reynolds numbers

Many investigation concerning the wake behind a sphere in a viscous flow have been carried out by numerical [11, 19, 22] and experimental means [10, 20].

The wake is affected by the Reynolds number Re based on the sphere diameter D and the inflow velocity U_∞. Taneda found in his experiments that for a Reynolds number up to $Re \approx 24$ the wake shows no vortex-ring and is perfectly laminar [20]. For higher Reynolds numbers a vortex-ring can be observed. The ring remains stable and axisymmetric up to $Re \approx 130$ before an oscillating pulsative motion appears at the rear of the vortex. For Reynold numbers in the approximate range $210 < Re < 400$ the wake transitions to a steady, non-axisymmetric, double-thread wake. At even higher Reynolds numbers the wake eventually shredds from the sphere in hairpin vortices [6, 22]. Analytical solutions suggest that a stable ring-vortex is formed for $20 < Re < 130$, an unsteady ring-vortex for $130 < Re < 450$ and an unsteady shredding vortex for $Re < 450$ [3]. The streamlines for Reynolds numbers 50, 100, 150 and 200 are shown in figure 2.2 for display purposes [6].

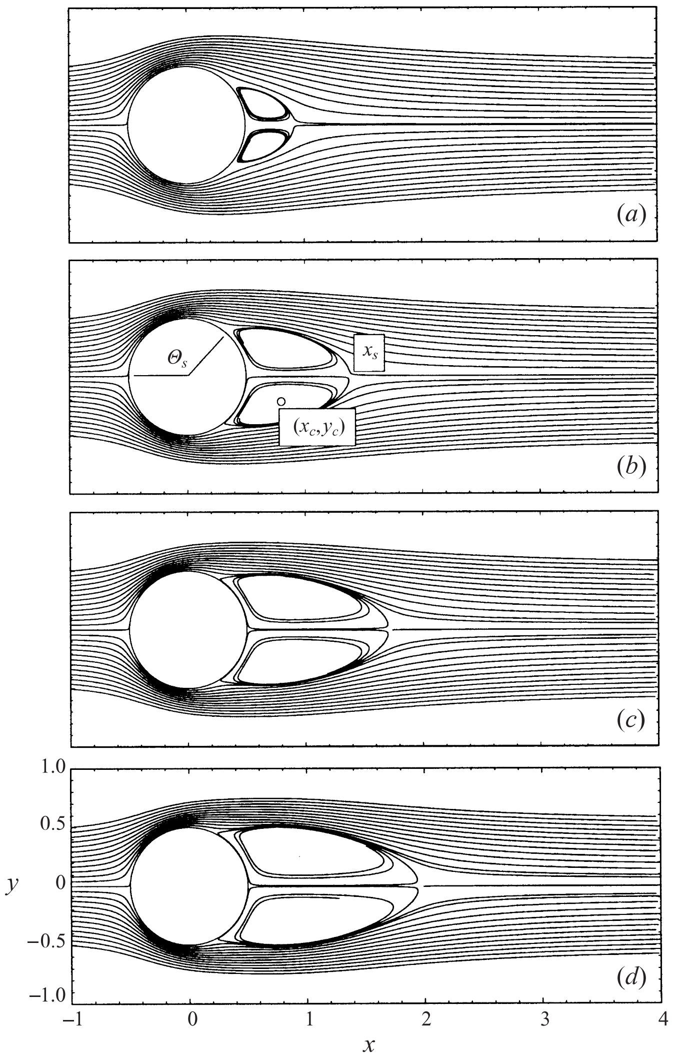


Figure 2.2: Computed streamlines past a sphere based on numerical results by Johnson and Patel [6]: (a) $Re = 50$; (b) $Re = 100$; (c) $Re = 150$; (d) $Re = 200$.

2.5 Lattice Boltzmann method

For both the analytical and experimental solutions, a Lattice-Boltzmann (LB) solver with a hierarchical Cartesian mesh is used. The LB method consists of a set of equations that define simplified kinetic models containing the essential physics at the microscopic level [4]. This simplification avoids the need to solve complicated equations such as the full Boltzmann equation. The method is particularly successful in flow applications with boundary dynamics and complex boundaries. Due to the use of fully parallel algorithms, this method is also ideally suited for use in massively parallel machines.

The hierarchical Cartesian mesh structure is a fully automatic mesh generation method [18]. It consists of cubic cells that are refined by subdividing the parent cell into eight equal cubes. This is repeated until the desired degree of refinement is achieved. By applying suitable interpolation and transformation algorithms, the exchange between cells of different refinement is ensured. This method allows for easy generation of adaptive meshes while maintaining high accuracy through orthogonal mesh lines. It is also suitable for coupling different simulation methods. One limitation is the resolution of thin layers with high gradients [5]. This is based on the use of regular grids with uniform grid spacing. This problem can be circumvented by locally refined grids. Alternatively, a large number of cells is created so that the prevailing physical mechanisms of the flow field are resolved.

2.6 TOML

The TOML format is chosen as the input files format for the framework as explained in detail in chapter 3. TOML (Tom's Obvious Minimal Language) was created by Tom Preston-Werner as a minimalist configuration file format with a focus on readability [13]. A parser for TOML is available in Python as well as many other programming languages which parses the input data into various data structures. With a variety of native types like Arrays, Tables, Integers, Floats and Booleans diverse inputs are possible [14]. The framework uses the parser to create a dictionary which can be accessed later [12]. An example of a TOML input file used for the framework is given in the following code excerpt.

```
[...]
    [solution]
    analytical = true
    experimental = false
    solutionscript = ["nameofsolutionscript.py"]
[...]
```

Listing 2.1: Example of a TOML file.

After parsing the TOML file with the parser, the information is available as a dictionary. A value can be accessed as following:

```
analytical_boolean = parsed_toml["solution"]["analytical"]
print(analytical_boolean)
>>> True
```

Listing 2.2: Accessing the file values of a TOML file.

TOML files are easily extendable as the information is accessed by its specific key after it is parsed. TOML files are easy to read without complex syntax and therefore offer a user friendly input method.

3

Development and implementation

3.1 Requirements

In order to introduce a testing framework for m-AIA that quantitatively assesses the accuracy of the simulation results, this framework is created. Several requirements are defined in advance, which are explained in detail in the following section. A summary of the requirements is depicted in table 3.1. The solution for each requirement is explained in detail in the section 3.2.

Requirement	**Description**	**Solution**
Portability	Must run on various Linux systems.	Programming language Python.
Extensibility	Must be able to add more test cases. Framework should not be restricted by many predefined functional shapes.	The framework is split in different classes and functions for each process. More functions can be added to offer more features.
Usability	Must be easy to use and offer setup choices for user.	TOML input files with setup choices, templates for input files, external analytical solution scripts, command line arguments and help function.
Actuality	Must compare up-to-date data sets.	Runs m-AIA for each execution and does not rely on existing simulation output files.

Table 3.1: Table to summarise requirements.

The main focus of this project work is on developing the structure of the framework. A focus on *extensibility* (table 3.1) is therefore an important factor to consider. Thus, it is necessary to be able to add further test cases in the future as well as adding more functions to analyse the simulation and solution data. Even the ability to add completely new classes and functions should be considered when implementing the framework. The framework should therefore be restricted as little as possible by predefined functional forms. As an additional requirement, the framework should be able to run on different Linux systems. This ensures *portability*.

The framework is to access m-AIA and perform the simulation on each test run for all test cases. This ensures that changes to m-AIA have been taken into account and that current data sets are used for the test. This can be summarised under the requirement *actuality* in table 3.1.

A simple user interface is an essential part of creating an efficient workflow with the framework. The framework must therefore provide easy-to-use input functions and straightforward output. The user should have various setting options to customise the test run. The following requirements can be summarised under the term *usability*.
The input required is divided into two areas. First, a form of general input for the framework to set up the next test run. The user should be able to specify which test cases are to be analysed as well as exclude certain types of solutions. It should be possible to distinguish whether only test cases with an analytical solution or only test cases with experimental reference data are to be used. Second, since all test cases may have unique properties, the user must be able to give the framework a specific input for each test case. This includes the parameters that m-AIA is able to simulate and which parameters can be calculated by the analytical solution or are given in experimental reference data. In addition, the user must be able to specify the coordinates that are relevant for the comparison of this test case. As explained in section 3.2, the framework relies on m-AIA's post-processing feature for Probe-Points, -Lines and -Planes. Therefore, the user must specify the location and name of the output file for each test case. Furthermore, the framework must also distinguish between an analytical solution or experimental reference data for each test case. The user must be able to specify for each test case whether an analytical or experimental solution is available.

The user may only need the maximum, minimum or average deviation of certain parameters for a test run. In addition, the user should be able to decide which outputs should be printed on the console. This enables a situation-specific, concise statement about the result of the test run. The user should therefore be offered various setting options for these functions.

3.2 Foregoing considerations of the requirements

For most of the requirements mentioned, there are different ways to fulfil them. In the following section, alternatives are presented and the chosen solution is explained. A summary of the solution regarding each requirement is depicted in table 3.1. The advantages and disadvantages of each alternative are weighed up in order to achieve a satisfactory solution. Therefore, the functionalities, the input and output, the data structure and the architecture of the framework are considered and investigated.

To ensure the requirement of a user-friendly input method, the TOML file format (section 2.6) is used. TOML files are used as a general input for the framework as well as for each test case in its corresponding directory. The general `test_setup.toml` file contains information about which test to analyse and what output the framework creates. The `testcase_setup.toml` contains information about whether or not an analytical solution or reference data is available for that test case as well as parameters and information about the simulation output files from m-AIA. The easy-to-read configuration files ensure that a first-time user can quickly understand the variables and options. Although the clarity of the associated files is reduced by the need for an additional file, the parameters and input values for a test case can be easily changed. Templates for the TOML files have been created to reduce the initial workload when creating a new test case. Furthermore, comments and default values for various parameters ensure ease of use.

However, the simplicity of the TOML files leads to disadvantages in test cases with an analytical solution. As the analytical solutions for different test cases can be of arbitrary functional type, a predefined functional shape would compromise the flexibility of the framework. Therefore, it should be possible for the user to read in the analytical expression instead of implementing it in the framework. The solutions often consist of multiple equations and unique structures which makes them too complex for a TOML file. TOML does not provide a parser for mathematical functions and writing a parser leaves many failure points. Alternatively Python offers the ability to run a string input as a command with the `eval()` function. This function could be used to directly use a string value from the TOML file as a command without parsing it. However, this method is potentially unsafe as a user could also access and edit system data.

To circumvent the restricted mathematical input via a TOML file, the analytical solution is written in a separate Python script located in the test case folder. The name of the solution script is to be written in the configuration file for the test case. With this information, the framework locates the file, imports the module and passes the coordinates and necessary data to the script. The analytical solution script includes equations for all wanted parameters stated in the test case configuration file. After calculating the analytical solution, the script passes the parameter values back to the framework by getting the attribute value. Therefore, only the input and output of the script are set. A template for the analytical solution script is created to define the structure of the script. This input method for the analytical solution covers various types of test cases as the user is able to create a unique solution. As an additional possibility, this leaves the feature to create multiple solution scripts for a test case. This input method again requires an additional input file and increases complexity. Then again, the solution of a test case for different test cases can be of any functional type and is not restricted by a predefined functional form. It also offers a better readability of the analytical solution and can be easily accessed.

Another decision to be made is which data points the framework compares. The data output by m-AIA is not directly bound to Cartesian coordinates. m-AIA uses its own, very highly compressed grid, which only contains the coordinates of the smallest refinery level. All other cells are assigned based on their parent/child relationship and a cell ID. For this reason, m-AIA must first construct the grid itself, it cannot be read in. Due to this structure, a direct comparison between parameter values for all points is hardly possible.

Moreover, a lot of memory is needed for a direct comparison between all points of the grid for a fine grid refinement level. For a majority of test cases, the relevant data can be compromised to a single line as the test cases are often symmetrical. Often the boundary layers of bodies or surfaces in a flow are of explicit importance. An examination of corresponding points often provides quantitative information about the flow conditions. For a comparison of simulation data and reference data, these explicit points are sufficient.

m-AIA offers the post-processing feature to create ProbePoints, -Lines or -Planes. The user specifies their positions in the test case specific `properties_run.toml` before the simulation. Then, for each point, line or plane, m-AIA returns a file containing the coordinates and solution values of the corresponding locations. In doing so, the coordinates are shifted so that they coincide with the cell centre of the next grid cell. With this feature, the individual points described above and their data can be output. This bypasses the large amounts of data needed to output all points and allows coordinates to be assigned to the points. Despite the reduction in data points, these allow statements to be made about the simulation. However, it is up to the user to set the appropriate ProbePoints, -Lines or -Planes.

With this compromise, test cases with analytical solutions of m-AIA are simulated and the coordinates from the ProbeLine are passed to a script for the analytical reference solution. For test cases with experimental reference data, the coordinates of the data points are passed to m-AIA and matching ProbePoints are created. m-AIA is then able to simulate the test case and return data points for the same coordinates that are present in the experimental data.

It is customary to indicate deviations as relative quantities. Relative values allow for better comparability and can therefore be used as a quantitative statement about the deviations between the simulation results and the reference data. Therefore, the results of the framework should also be output in this form. For relative deviations, the measured deviation is set in relation to a comparative value. However, different values can be used as a reference value, each with its own advantages and disadvantages. One possibility would be to use the analytical reference solution or the experimental data as a reference. This is a very simple option, as the reference data is already used for evaluation and is thus further used. Alternatively, a user-defined conversion factor can be used. This allows a weighting of the results and can be adjusted very specifically. On the other hand, the determination of a suitable conversion factor involves more effort. Therefore, in the context of this project work, the analytical solutions or the experimental reference data were used as reference values. These are sufficient to demonstrate the capabilities of the framework.

3.3 Code-Structure

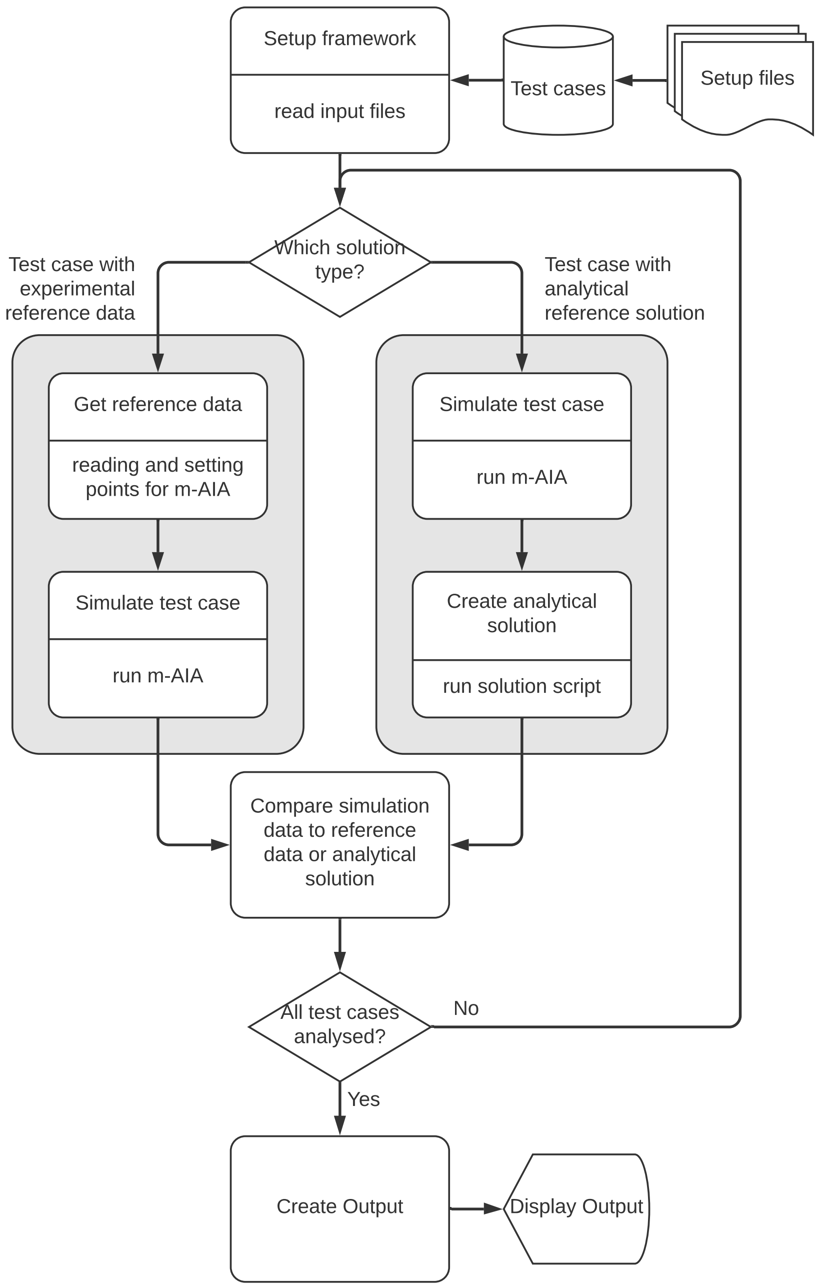


Figure 3.1: Flowchart of the five main processes that the framework goes through.

The code structure is focused in most parts on comparing two data sets. The work flow is subdivided into five general steps as shown in figure 3.1.

1. Setup of the framework and test cases.
2. Simulating each test case and storing the simulation results.
3. Creating the reference data for that test case either from existing experimental reference data or from an analytical reference solution.
4. Comparing simulation data to reference data or an analytical reference solution.
5. Creating a summarised output for the user.

Depending on the type of solution, the order of execution of the simulation and creation of the reference data is reversed.

To apply this process to a Python program, the framework is structured into different classes and functions. The class diagram, depicted in figure 3.2, shows the general structure of the framework with its different classes. The framework is based on the main class `TestExecutor`, which calls and processes other classes and functions.

```
usage: TestExecutor.py [-h] [--testcase T] P

Framework to compare simulation output data from m-AIA with reference data
for different test cases.

positional arguments:
 P               Path of the top level test case directory.

optional arguments:
 -h, --help      show this message and exit
 --testcase T    The name of a single test case to be analysed. If no
                 argument is passed the test_setup.toml file is used to
                 find test cases.
```

Listing 3.1: Displayed information of the help command-line argument `-h`.

First, the framework is configured. This includes reading in the TOML input files for the test run and for each test case. The file path of the test cases is passed as a required command-line argument as shown in listing 3.1 when the framework is called. The class `TestRoot` takes over the task to read the input files. The class `TestConfig` finds and parses the `test_setup.toml` file with its `load_test_config()` function to access the requested test cases. The user is able to request a single test cases for analysis by passing the test case name as a command-line argument when the framework is called. For this, the option `--testcase "testcase_name"` is appended to the execution command. This

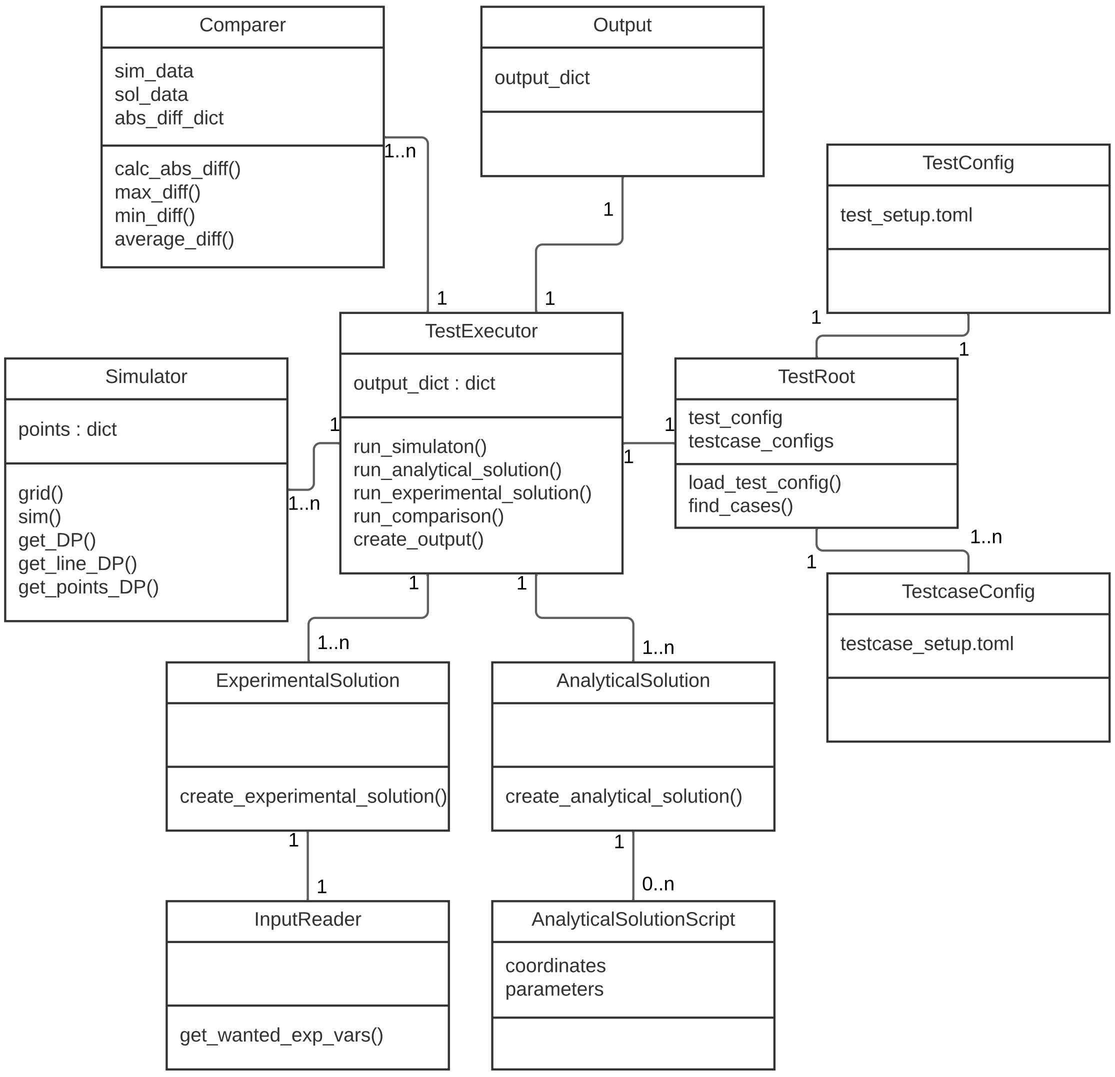


Figure 3.2: Class diagram of the framework.

is possible due to the console argument parser which also features a `-h` help command for further information. The options can be seen in listing 3.1.

The `TestRoot` object analyses the parsed `test_setup.toml` file if no test case is passed via the console. The user has the option to request all test cases in a sub-directory by setting the `all_testcases` Boolean `True`. Alternatively only a list of test cases are requested by filling the `testcases[ ]` list with the names of the test cases in the `test_setup.toml` file. Through these three options, the framework provides a flexible and user-friendly way to set the test cases. In particular, the option to pass only one test case to the framework is very useful for debug runs. After the general configuration of the test run, `TestRoot` creates a `TestcaseConfig` object for each requested test case and adds them to a `testcase_configs` dictionary. This is done by the `find_cases()` function. Each object includes the test case name, path and the parsed `testcase_setup.toml` file with all information regarding the test case.

With the information about each requested test case, the `TestExecutor` loops through all test cases, runs the simulation, creates the reference data, compares the results and inserts its output into an output dictionary. This process requires case discrimination between test cases with different solution types as stated earlier:

1. Test cases with an analytical reference solution.
2. Test cases with experimental reference data.

For test cases with an analytical solution, the simulation must be run first to create the post-processing files and generate the analytical solution for the probe coordinates. However, for test cases with experimental reference data, the correlating coordinates must be passed to the `Simulator` so that m-AIA can create suitable post-processing probes. This case distinction is handled by the `TestExecutor`.

If a test case contains an experimental solution in form of existing reference data, m-AIA needs to output the simulation data correlating to the coordinates of the existing data. Therefore, the input file with the reference data (e.g. stored in a CSV file) is analysed first. The `InputReader` accesses the file as its name is written in the `testcase_setup.toml` file. A dataset with all relevant information such as coordinates and variables is created afterwards. The corresponding coordinates are again read out separately and written as `probeCoordinates` in the `properties_run.toml` file for the test case. In this way, m-AIA can output the values for the coordinates in the post-processing of the simulation. An output file for each point is created. The implemented example of a test case with experimental reference data (flow around a sphere at low Reynolds numbers) can be found in section 3.5.

If a test case contains an analytical solution, a `AnalyticalSolution` object is initialised by calling the `run_analytical_solution()` function in the `TestExecutor`. It accesses the already simulated variables from m-AIA and the individual Python solution script for the test case. The script is shown in figure 3.2 as the `AnalyticalSolutionScript`. The coordinates are passed to the script and processed individually. The framework expects to receive back the calculated solution for all parameters requested in the file `testcase_setup.toml` as an array with all points and their corresponding values. For a detailed explanation of the `AnalyticalSolutionScript`, see section 3.4. If a particular parameter cannot be accessed by the `AnalyticalSolution` an error is thrown and the program is terminated.

For the simulation and post-processing of the test case, the `TestExecutor` creates a `Simulation` object inside the test case loop. During the initialisation the test case grid is created and the simulation is run with m-AIA. If errors are encountered, they are printed to the console and the code is terminated, if not, the post-processing is started. Firstly, it is ensured that all wanted comparison variables are contained in the simulation or post-processing file. If this is not the case, an error record is printed and the code aborted. Otherwise, the solution type is read from the test case file `testcase_setup.toml` and the simulation instance behaves accordingly. If the reference solution is analytical, the probe file to be analysed from the `testcase_parameters.toml` file is read, and a function to extract the data is called. At the moment, only the feature for processing `probeLines` files has been implemented, but expansion for other file types can be easily added. If the solution is experimental, a check ensures that all coordinates have been processed and have a probe file. Afterwards, each probe file is analysed and the wanted variables extracted. The file contains the data from the four nearest cells to the coordinate. If the cell is positioned at a boundary the file contains the data from the two nearest cells to the coordinate. Currently, the data from the nearest cell is selected. A consideration for future development would be to interpolate in order to maximise accuracy.

After these steps, both simulation data and reference data are available either from an analytical solution or experimental reference data. The `TestExecutor` creates a `Comparer` object with the `run_comparison()` function. Both data sets are passed to the object and the `Comparer` processes them. Since the focus of this project work is on the framework structure, the `Comparer` so far only calculates the deviations of the selected parameters.

Three user-defined mathematical functions are implemented to calculate the maximum, minimum and average deviations for each parameter. Relative deviations have not been implemented, but can be easily added. The functions are applied to each test case and an output dictionary of results is created. Each result is assigned a key for its function (e.g. "`max`") to access the values for the user output.

After all test cases are simulated, their solution created and both data sets compared, the output dictionary is finally passed to the `Output` class with the `create_output()` function. Depending on the user input in the `test_setup.toml` file, as shown in the following code extract, the calculated values are output to the console.

```
[...]
    [output]
    max_diff = true
    min_diff = true
    average_diff = true
[...]
```

Listing 3.2: User choices regarding the output of the framework.

The user can choose whether to display certain function results by setting the corresponding Boolean value `true`. The object `Output` loops through all test cases and their solution and simulation dictionaries. For existing reference data, all data points are run through directly. The requested results for each parameter are output to the console. As described further in the sections 3.2 and 5, this is changeable and can be extended.

To make the framework accessible to new users in the future, some debug features are used. In order for new users to better understand the process flow and the individual steps, additional print statements can be output in the console during the execution of the framework. With the help of the logging package, messages about occurring events can be output on the console. Different levels can be used for different events such as `DEBUG`, `INFO`, `WARNING`, `ERROR` or `CRITICAL` [17]. `INFO` is used to confirm an event is occurring as expected whereas `ERROR` or `CRITICAL` inform the user about a problem without raising an exception. The level which is displayed on the console during the execution can be defined in the TestExecutor. The setting option is shown in the next code excerpt.

```python
[...]
   if __name__ == "__main__":
        logging.basicConfig(level=logging.DEBUG)
[...]
```

Listing 3.3: Setting the logging level of the framework.

In this way, little information is displayed for tested and functioning test cases, so that the console output remains very clear. In the case of faulty test cases, very detailed statements can be output for each process. An example of console output using the `logging` feature is shown in listing 3.4.

```
INFO:root:Testsetup requests analytical solution.
INFO:root:Testcase 3D_taylorcouette has analytical solution.
INFO:root:Running simulation for 3D_taylorcouette.
INFO:root:Creating analytical solution for 3D_taylorcouette.
INFO:root:Solution created with file Taylor_Couette_Analytical_Solution.py.
INFO:root:Testsetup does not request experimental solution.
```

Listing 3.4: Example of console output for `logging.INFO` events.

The advantage over normal print statements is therefore the possibility of hiding various logging statements without commenting out the lines in the code. This feature is a user-friendly way to familiarise yourself with the framework. It is also a good way to check the functionality of the framework when implementing new content.

With this structure, the framework follows the process shown in figure 3.1. It provides several settings that can be changed by the user to perform an individual analysis for each test case simulation. Several interfaces have been considered to create an extensible structure. More test cases can be added and different experimental data sets or analytical solution scripts can be used as reference data. The user can add own functions to analyse the created data sets and customise the output in the console. The structural requirements are thus fulfilled. The division of the program into different classes improves clarity. Thus, the main class comprises only a few lines of code and the program flow can be grasped quickly. Due to the strict division of labour between the different classes, individual areas can be quickly localised and changed. Additional features can be thematically assigned to the existing classes and can be implemented there.

To test the integrity and functionality of the framework, two test cases are implemented. A Taylor-Couette flow at a low Reynolds number is chosen to represent a test case with an analytical solution. A flow around a sphere at a low Reynolds number is chosen to represent a test case with experimental reference data. The next sections deal with the specifics of the respective reference data and their implementation in the framework. The implementation of the concrete test cases is dealt with in section 4.1.1 and 4.2.1.

3.4 Analytical reference test case

For test cases with an analytical reference solution, the Python solution script must be written in addition to the required files for m-AIA. The `AnalyticalSolutionTemplate.py` file is used as a template for the script. The class `AnalyticalSolution` accesses the scripts for each test case individually. The template receives the variables created by the simulation run of this test case and the parameters from the file `testcase_parameters.toml` from the framework. During the `__init__()` function, the script loops through each point in the variables. By convention, the first three values in each array of points correlate with its Cartesian coordinates x, y and z, which can be used for solving. It is then up to the user of the template to fill in each equation to create the analytical solution for a test case.

In the last section of the loop, an array with all parameters is created. Using the function `getattr()`, the script loops through all the parameters of the test case and fetches the corresponding value for each item. If the script cannot find a parameter specified in the test case's `testcase_parameters.toml` file, an error is raised and the framework is stopped. The user is expected to set the correct parameters. The array for each item containing the values for the parameters is then added to a list and returned to the `AnalyticalSolution` class. The following code excerpt shows an example for the wanted parameters [u, v] and their fictive solution $u = x^2 - y$ and $v = z$.

```python
[...]
    point_solution = []
    for point in variables:
        self.x = point[0]
        self.y = point[1]
        self.z = point[2]

        # Solution:
        self.u = self.x^2 - self.y
        self.v = self.z

        # array with solution of point
        coords = point[:3]
        parameter_check = []
        parameter_check_bool = False
        for parameter in parameters:
            try:
                coords.append(getattr(self, parameter))
            except AttributeError as e:
                parameter_check_bool = True
                print(f"Parameter {parameter} is not available in the
                        analytical Solutionscript!")
        if parameter_check_bool is True:
            sys.exit("Analytical Solution failed.")
        else:
            point_solution.append(coords)
[...]
```

Listing 3.5: Example of an analytical solution script.

For this example, the following code excerpt represents the parameters listed in the TOML file.

```toml
[...]
    [parameters]
    parameters = [u, v]
[...]
```

Listing 3.6: Example parameters from the `testcase_setup.toml` file.

As an example where the input coordinates are [2, 1, 4] corresponding to [x, y, z] the script would return the array [2, 1, 4, 3, 4] corresponding to [x, y, z, u, v].

With this structure, the analytical solution for different test cases can be of arbitrary functional type. The template provides a user-friendly and simple way to implement an analytical solution for a test case.

3.5 Database driven reference test case

For test cases with given reference data, for example from experiments or simulations, this data must be accessible to the framework. Currently, the data is expected in a `CSV` file. To read in this file, the name of the file is written in the `testcase_setup.toml` file instead of the analytical solution script. This allows the framework to access the file with the `InputReader` and use the variables. By convention, the first three parameters of an array represent the spatial coordinates. These are used to create the `ProbePoints` for the post-processing of m-AIA. An example CSV file is shown in listing 3.7. The shown data originates from numerical simulations by Rimon and Cheng [15] which are used in section 4.2 to verify the frameworks functionality with a test case with experimental reference data.

```
x,y,z,vortZm
-0.545592759,-0.069487706,0,0.001577939
-0.542912687,-0.088010306,0,0.002097375
-0.540313744,-0.102767009,0,0.002491618
-0.536315512,-0.121924861,0,0.003082288
-0.522580989,-0.171490846,0,0.00465279
-0.513783323,-0.196282187,0,0.005178782
[...]
```

Listing 3.7: Example of the CSV file for a flow past a sphere.

Since m-AIA's output values are in de-dimensioned quantities, it is the user's task to de-dimension the reference data in the same way. This is necessary for all database driven reference test cases.

4

Verification

In order to validate the functionality of the framework, two test cases are implemented. A Taylor-Couette test case with an analytical reference solution and a flow around a sphere at low Reynolds numbers as a test case with experimental reference data are conducted. The setup of each test case, the simulation results and the comparison of the simulation data with the corresponding reference data by the framework are investigated in the following section.

4.1 Taylor-Couette Flow

4.1.1 Computational setup

The Taylor-Couette test case is chosen as the test case with an analytical solution and is implemented to be simulated by m-AIA. The geometry is created in ParaView and exported as `.stl` files. The geometry's spatial extensions are defined in a non-dimensionalised manner refering to the gap distance δ.

The inner cylinder has a diameter of 2 and the outer a diameter of 4. The height of both cylinders is 0.2. The cylinders' length is low in order to minimise the number of cells. It is important to note that the cylinders described in section 2.3 are infinite in length, whereas the cylinders in the geometry for the simulation are capped at the top and bottom. The resemblance to cylinders of infinite length is accomplished by having a periodic boundary condition at the caps of the cylinders during the simulation.

For the outer cylinder, a boundary condition (BC) is assigned which allocates a velocity to each cell following the formula:

$$U = -(U^*) * \sin(\arctan(y/x)\,, \tag{4.1}$$

$$V = U^* * \cos(\arctan(y/x)\,, \tag{4.2}$$

$$W = 0.0\,. \tag{4.3}$$

Where U, V, W are the velocities in the cell in x, y and z direction, respectively. U^* is the dimensionless reference velocity, which is modified by the reference Mach number in the `properties_run.toml` file. x and y are the x-coordinate and y-coordinate of the cell, respectively. For the inner cylinder a no-slip wall BC is assigned, prescribed by an interpolated bounce back according to Bouzidi, Firdaouss and Lallemand (BFL-rule) [2]. Both caps have the periodic BC assigned.

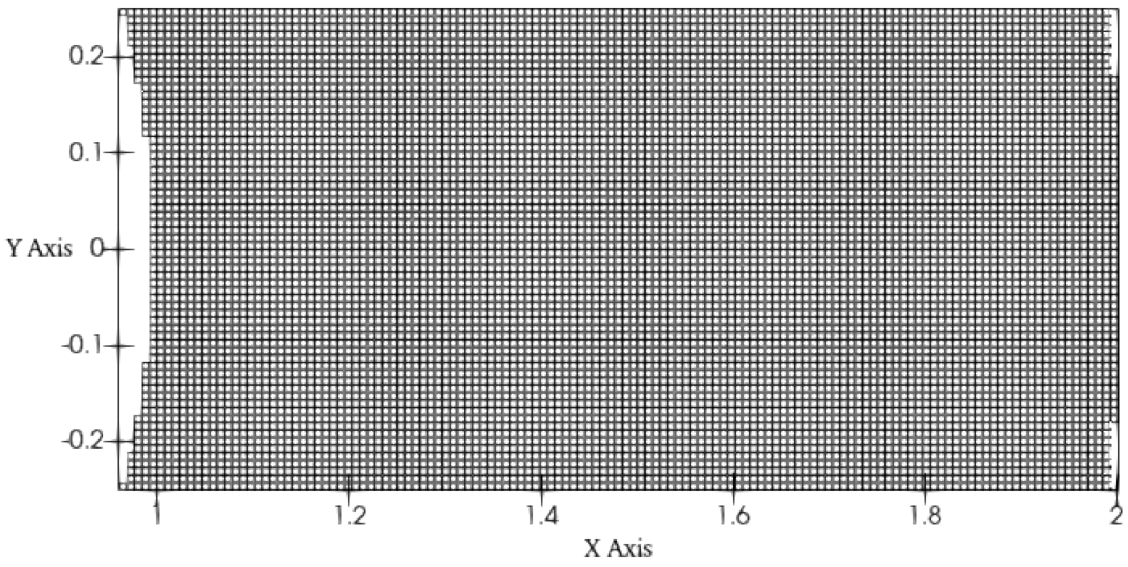


Figure 4.1: Simulation grid of the Taylor-Couette test case.

The simulation grid, shown in figure 4.1, consists of nine evenly refined levels. On the highest level, the length of a cell is 0.0078125*δ. The flow is characterised by a Reynolds number $Re = 25$, with the outer radius as the reference length. The density is $\rho = 1$, and the initial dimensionless Mach number $Ma = 0.1$. This results in the outer cylinder rotating with a dimensionless angular velocity $\omega = 0.0288675$. The simulation is initialised by a quiescent flow field and is performed for around 207 rotations of the cylinder corresponding to 10000 time steps. The parameters of the test case are depicted in the following excerpt of the TOML file.

```
[parameters]
parameters = ['u', 'v', 'w']

probeFile = "probeLines_10000.Netcdf"
simFile = "PV_10000.Netcdf"

[solution]
analytical = true
experimental = false
solutionscript = ["Taylor_Couette_Analytical_Solution.py"]
```

Listing 4.1: testcase_parameters.toml for the Taylor-Couette test case.

In addition to setting up the test case simulation, a Python script was written for the analytical solution. As mentioned in section 3.2 and 3.4, this additional input file is used to define the analytical solution. To use the solution describing the Taylor-Couette flow discussed in section 2.3 for the framework, the analytical solution given in polar coordinates has to be transferred into Cartesian coordinates, where u, v and w describe the velocities correlated with x, y and z, respectively. For Reynolds numbers below a critical Reynolds number Re_c, the flow between two rotating cylinders is stable and the velocity w is zero [23]. The respective critical Reynolds number depends on the radius ratio of the cylinders. The total velocity in circumferential direction V^* can be divided into its two parts u and v. With the Pythagorean theorem $r^2 = x^2 + y^2$, z as the cylinder axis and $\theta = \arctan(x/y)$, the cylindrical coordinate system is transferred to a Cartesian coordinate system. The remaining velocity are calculated accordingly by

$$u = \sin\theta * V^* \,, \tag{4.4}$$

$$v = \cos\theta * V^* \,. \tag{4.5}$$

The solution script therefore contains the equations discussed here in addition to the equations in section 2.3. These were implemented for the Taylor-Couette test case in the `Taylor_Couette_Analytical_Solution.py` solution script. The calculation specification for the analytical solution script of the Taylor-Couette test case is shown in the following code excerpt.

```
[...]
K = 1.7320508075688772935 2 #dimensionless constant
for key, value in variables.items():
    point_solution = []
    for point in value:
        # Order: "x", "y", "z"
        self.x = point[0]
        self.y = point[1]
        self.z = point[2]

        # calc solution for point
        self.R1 = float(1)
        self.R2 = float(2)
        self.Omega1 = float(0)
        self.Omega2 = float(Ma/(self.R2*K))

        self.r = math.sqrt(self.x**2 + self.y**2)
        if self.x == 0.0:
            self.eta = np.sign(self.y)*math.pi/2
        else:
            self.eta = np.arctan2(self.y, self.x)
        self.v_eta = (self.Omega2 * self.R2**2) / (self.R2**2 - self.R1**2)
                     * self.r - (self.Omega2 * self.R1**2 * self.R2**2) /
                     ((self.R2**2 - self.R1**2) * self.r)
        self.u = -math.sin(self.eta) * self.v_eta
        self.v = math.cos(self.eta) * self.v_eta
        self.w = 0
[...]
```

Listing 4.2: Calculation specification for the analytical solution script of the Taylor-Couette test case.

4.1.2 Simulation results

Figures 4.2, 4.3 and 4.4 show the results of the simulation. The tangential velocity at the outer cylinder is correct. However, the speed increases unintentionally afterwards. It is also interesting that the velocity in line direction depicted in the diagrams is not always zero. This should be the case. This may be caused by errors in the interpolation of the outer cylinder's BC, since it is still in development.

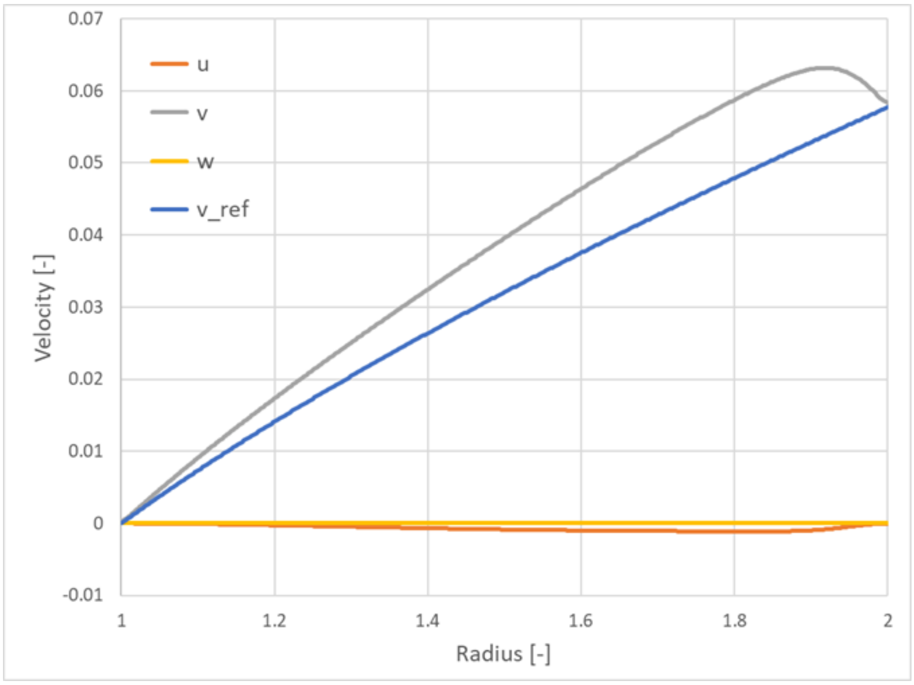


Figure 4.2: Plot of velocity values of the Taylor-Couette simulation over a line from the origin in positive x-axis direction. Radius de-dimensioned with inner cylinder radius.

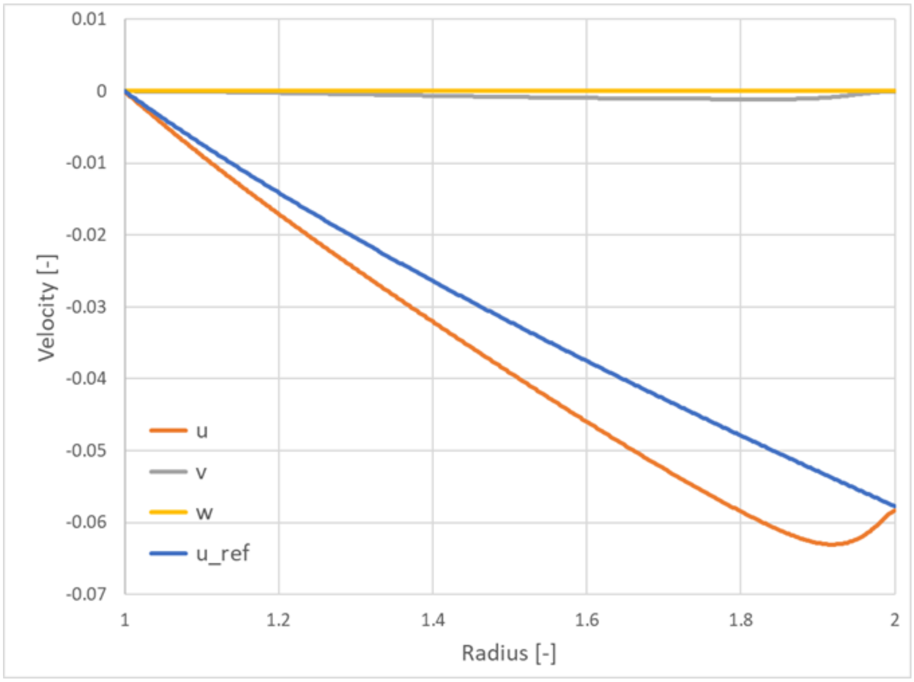


Figure 4.3: Plot of velocity values of the Taylor-Couette simulation over a line from the origin in positive y-axis direction. Radius de-dimensioned with inner cylinder radius.

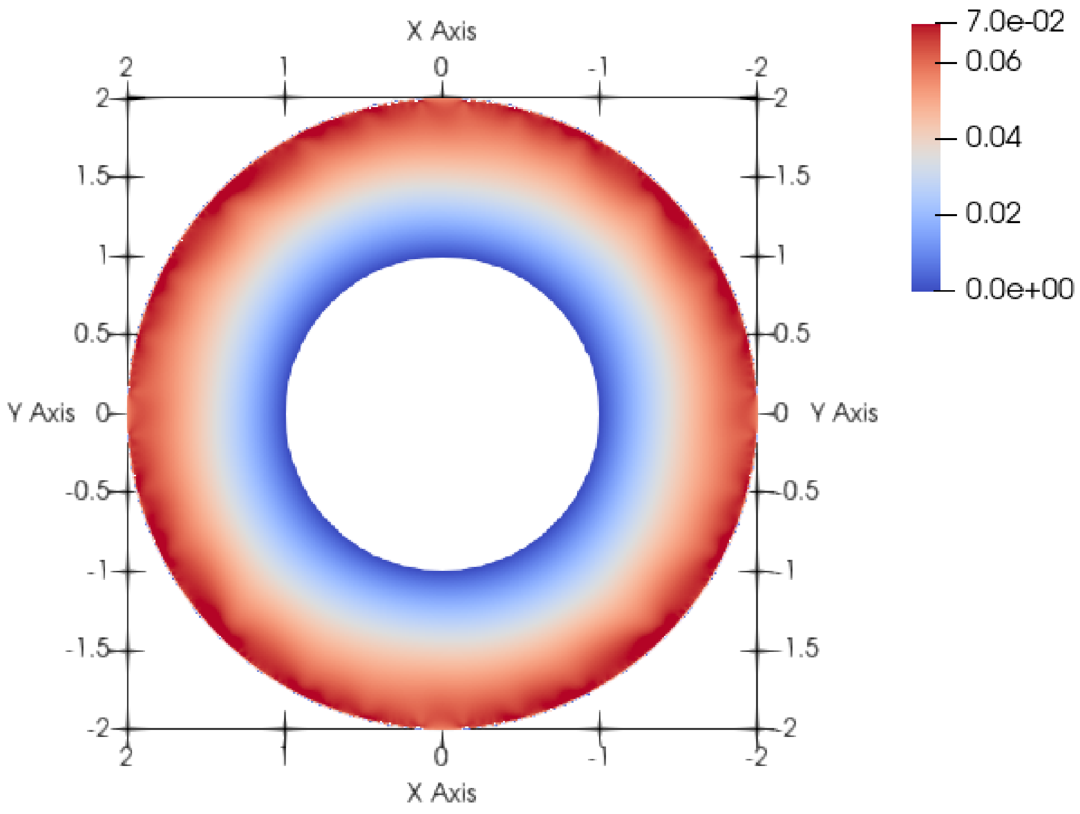


Figure 4.4: Slice through the centre of the cylinders of the Taylor-Couette test case showing a tangential velocity scalar field.

4.1.3 Framework results

After running the framework, the statement depicted in listing 4.3 is written on the terminal.

```
[...]
----------------------------------------------------
OUTPUT

Output for testcase Taylor-Couette, script
    Taylor_Couette_Analytical_Solution.py and key points_0:
Max diff for u is 0.0011787367743356465
Max diff for v is 0.010938399523547984
Max diff for w is 1.0347420157244894e-06
Min diff for u is 0.0
Min diff for v is 9.131906061630654e-05
Min diff for w is 0.0
Average diff for u is 0.0005509640582479376
Average diff for v is 0.005606559424637495
Average diff for w is 8.164372419049707e-08

Output for testcase Taylor-Couette, script
    Taylor_Couette_Analytical_Solution.py and key points_1:
Max diff for u is 0.010937985241638574
Max diff for v is 0.0011792783954642173
Max diff for w is 1.0359074870103756e-06
Min diff for u is 8.328631825618802e-05
Min diff for v is 1.8448863631141243e-20
Min diff for w is 0.0
Average diff for u is 0.0063800315422740846
Average diff for v is 0.0006366289672295614
Average diff for w is 9.811295185577506e-08
```

Listing 4.3: Output of the comparison between simulation and analytical data by the framework for the Taylor-Couette flow.

The different keys (`points_0` and `points_1`) represent the ProbeLines. Line one runs from the centre of the cylinders in x-axis direction. Line two runs from the centre of the cylinders in y-axis direction. The same order of magnitude of the maximum deviation of the velocities is clearly visible. The change of u and v due to the 90° shift shows that the velocity in the outer cylinder is correctly assigned. The not yet optimal simulation of the Taylor-Couette test case shows the functionality of the framework very well. With this statement, the test case can be examined more closely to find the cause of this deviation.

4.2 Flow past a sphere at low Reynolds number

A flow past a sphere at low Reynolds numbers is chosen as the test case with experimental reference data. This decision is based on the fact that the test case has already been implemented in m-AIA for Canary and reference data is widely available. The reference data is obtained from numerical simulations by Rimon, Y. and Cheng, S.I. [15]. The data with a Reynolds number Re=100 is used. In the graph, the de-dimensioned vorticity values are shown over an angle of 180°. The angle being 0° at the front of the sphere and 180° at the back. The data is extracted from figure 4.5 using the tool *WebPlotDigitalizer* [16].

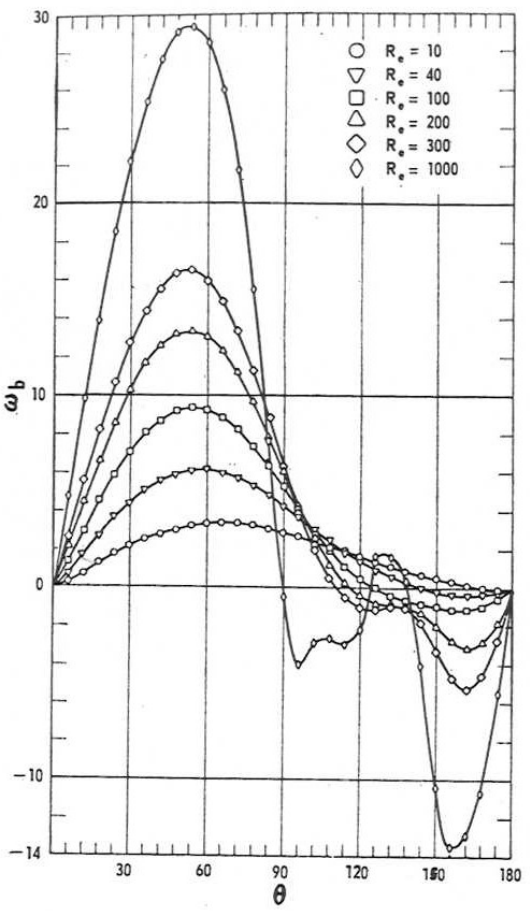


Figure 4.5: Original reference data showing the vorticity distribution used for the flow around a sphere test case [15]. ω_b is the de-dimensioned voricity and Θ the angular position on the sphere.

4.2.1 Computational setup

For the experimental test case, the geometry was constructed to match the geometry of the reference solution. As with the analytical solution, the geometry was created in ParaView and exported as `.stl` files. The geometry's spatial extensions are defined in a non-dimensionalised refering to the sphere's diameter. It consists of a sphere with a diameter of 1 which is located in the radial and axial centre of a pipe. The diameter of the pipe is 8.1662 times larger than the diameter of the sphere. The length of the pipe is 20.1 times the diameter of the sphere [15]. The inflow is prescribed by a velocity BC setting the equilibrium state on one cap. The outflow is prescribed by a pressure BC setting the equilibrium state on the other cap. A non-equilibrium extrapolation BC is applied to the pipe wall. A no-slip wall BC is assigned for the sphere, consisting of an interpolated bounce back with BFL-rule. This corresponds to the BC for the inner cylinder in the Taylor-Couette test case.

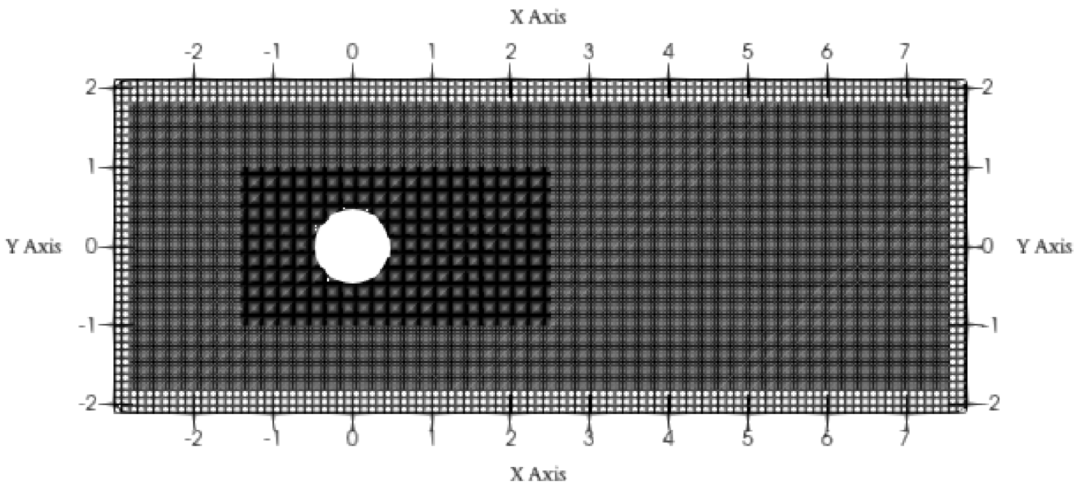
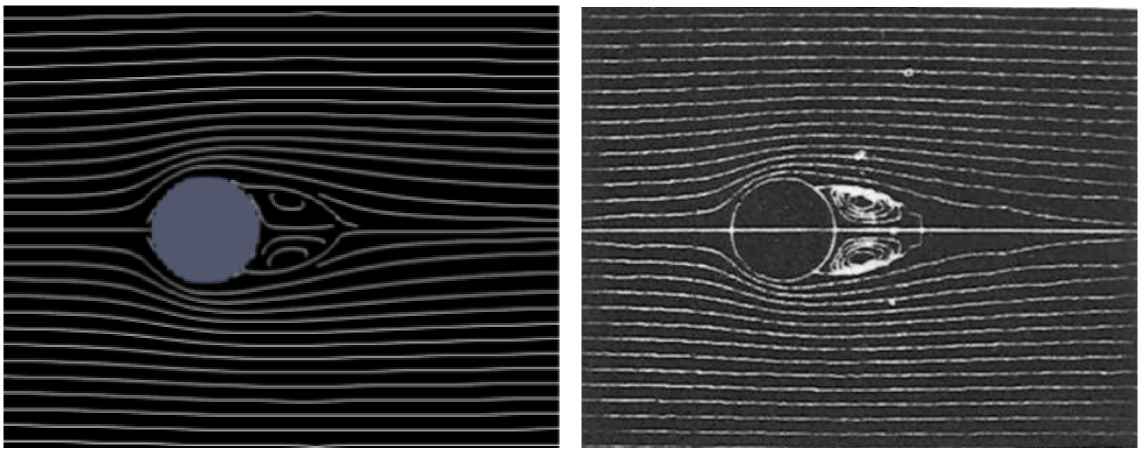


Figure 4.6: Section of a slice through the centre showing the simulation grid of the flow around a sphere.

The simulation grid, shown in figure 4.6, consists of eight evenly refined levels. On the highest level, the length of one cell is 0.07852*D, D being the diameter of the sphere. To ensure a better resolution of the sphere's wake, higher local refinement levels are used. The cells within a cylinder four times the diameter of the sphere, extending 2.5*D in front of and 7.5*D behind the centre of the sphere, are refined to the ninth level. On this level, one cell's length is 0.039258*D. The cells within a cylinder twice the diameter of the sphere, extending 1.5*D in front of and 2.5*D behind the centre of the sphere, are refined to the tenth level. On this level, one cell's length is 0.019629*D. The local refinement levels are clearly visible in figure 4.6. The flow is characterised by a Reynolds number $Re = 100$, density $\rho = 1$, and an initial Mach number $Ma = 0.1$. For the initialisation, the flow is assumed to be laminar, with an axial velocity proportional to the chosen Mach number. The simulation is then run for 20000 iterations, after which the residuals for all variables are, at most, in the order of 10^{-7}.

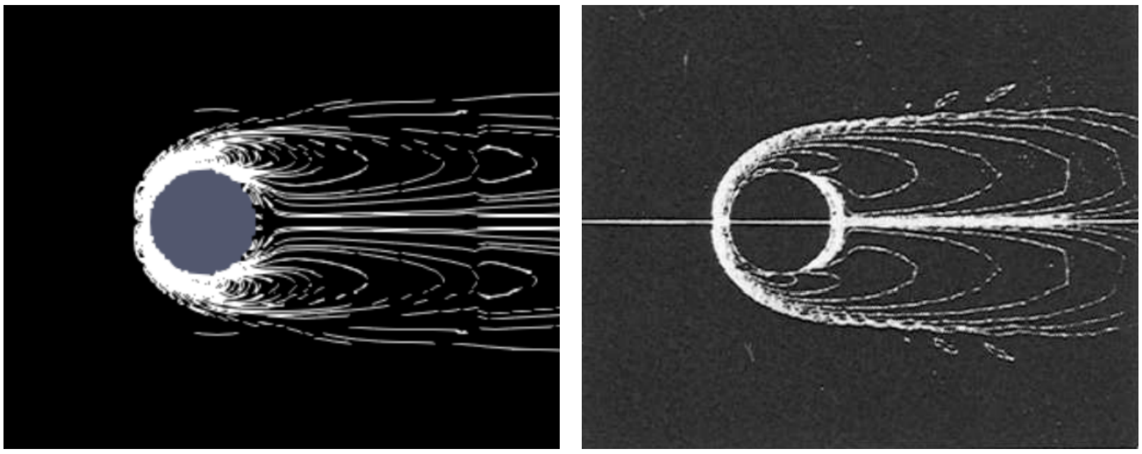
4.2.2 Simulation results



(a) *Simulation.* **(b)** *Reference data.*

Figure 4.7: Velocity streamlines at the centre of the sphere.

The visual differences between the simulation and reference data can be seen by comparing figures 4.7 and 4.8. When looking at the velocity streamlines shown in figure 4.7, there is no noteworthy difference. The vorticity shown in figure 4.8 depict some differences in the contour, especially in the width of the whole. However, comparing the flows through the images does not provide an accurate order of magnitude of the correlation between them.



(a) *Simulation.* **(b)** *Reference data.*

Figure 4.8: Vorticity contour at the centre of the sphere. Each line represents the points where the vorticity is equal.

4.2.3 Framework results

It is important to note that the data used from Rimon and Cheng are de-dimensioned with the uniform velocity U, the radius of the sphere R and R/U as the reference time [15]. In order to obtain a similar flow condition for the simulation the reference data is adjusted according to the similitude.

```
[...]
----------------------------------------------------
OUTPUT

Output for testcase 3D_sphere_Re100_parallel, script 3
    D_sphere_Re100_exp_sol.CSV and key coordinates:
Max diff for vortZm is 0.009747653
Min diff for vortZm is 0.00010863300000000002
Average diff for vortZm is 0.00348929311627907
```

Listing 4.4: Output of the comparison between simulation and reference data by the framework for the flow around a sphere.

After running the code, the output statement shown in listing 4.4 is printed on the terminal. Since the maximum vorticity value in the reference data is $\omega_{\max} = 0.01720922$, the difference between the simulation and reference data is noticeable. Especially notable is the maximum difference, which is more then twice the average. When comparing the values in the plot in figure 4.9, it can be seen that the simulation values differ considerably from the reference data. Also notable is the difference between the lines of best fit. One possible cause is that the sphere's surface is not perfectly smooth, since the cells are cubic, and can therefore not conform to a round surface.

These strong deviations can be seen in the analysis of the framework output data. Especially the difference between the maximum and average deviation is noticeable. Thus, the applicability of the framework is also demonstrated with this test case.

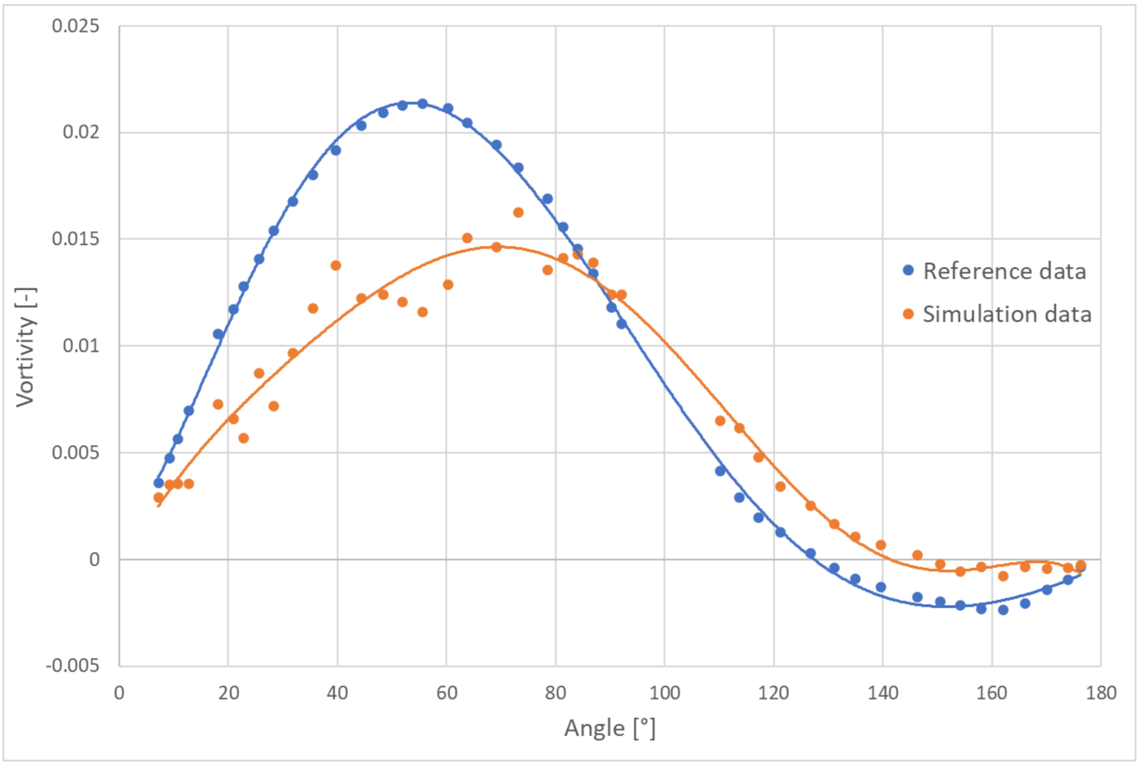


Figure 4.9: Vorticity values of the simulation and the reference data for the flow around a sphere.

5

Conclusion and Outlook

5.1 Conclusion

The new validation framework was created to produce a quantitative evaluation of the simulation data from m-AIA. The existing framework Canary, on the other hand, was set up to produce an evaluation by comparing the simulation output with reference data from previous runs. Therefore, a direct comparison between the two frameworks is hardly possible. Both frameworks aim to evaluate a different aspect of the simulation data and are complementary to each other. To investigate the functionality of m-AIA, both frameworks should be used. A regular or daily evaluation by Canary confirms the consistency of the simulation results. The new framework can additionally provide useful data on the discrepancies between simulation data from different test cases and analytical or experimental reference data.

Through the implementation of two test cases and the general structure of the framework, all requirements defined in advance are fulfilled. In order to test the functionality of the framework, the focus of the test cases was not on correct results. Primarily, the handling of test cases was tested with analytical reference solutions as well as with available reference data. Particular attention was paid to the structure of the framework and simple input options for the user. In order to use the framework for its function, the test cases and the reference data must therefore still be adapted. The current state of the framework offers a good opportunity to expand it and integrate further functionalities.

5.2 Outlook

So far, one test case with an analytical solution (Taylor-Couette flow) and one test case with experimental reference data (flow around a sphere) have been implemented. Further test cases with different parameters can be added to ensure a more accurate validation. With the user-friendly input solution via TOML files, separate Python scripts for the analytical solution and data input files for experimental data, the test case repertoire can be easily extended.

Furthermore, the result of the comparison between the simulation data and either the analytical solution or the reference data is only output on the console. The framework can be extended to create a summary output file to increase usability. Also, an output file for each test case could be created in the test case directory. A visual output in ParaView could be a possible feature to visualise the difference between simulation results and reference data. Furthermore, currently only the maximum, minimum and average difference for each parameter and `ProbePoint` or `ProbeLine` are calculated. More user-defined values can be added to the output in the future.

As already explained, only the post-processing functionalities of ProbePoints and -Lines can currently be used by the framework. An extension of ProbePlanes is possible. As a file format for experimental reference data, only the CSV format is currently implemented. An integration of further data types could be considered if necessary. Furhtermore, the simulation data is set in relation with the reference data. The additional possibility for the user to use his own conversion factor to give more weight to certain data points could be considered.

Due to the many extension possibilities, the framework developed so far can be turned into an even more helpful and versatile tool.

Acknowledgements

We would like to thank Miro Gondrum, who supervised this work. Thank you for the chance of learning about simulations with m-AIA, the advice and the deep insights in scientific work. We also appreciate the opportunity of writing this work at the Institute of Aerodynamics of the RWTH Aachen University.

References

[1] AIA (2021). AIA Git: Canary (Unpublished internal document.). `https://git.rwth-aachen.de/aia/MAIA/canary` [Accessed: 16.07.2021].

[2] Bouzidi, M., Firdaouss, M. & Lallemand, P. (2001). Momentum transfer of a Boltzmann-lattice fluid with boundaries. *Physics of Fluids*, **13**, 3452–3459.

[3] Brauer, H. & Sucker, D. (1976). Umströmung von Platten, Zylindern und Kugeln. *Chemie Ingenieur Technik*, **48**, 665–671.

[4] Chen, S. & Doolen, G.D. (1998). Lattice Boltzmann method for fluid flows. *Annual Review of Fluid Mechanics*, **30**, 329–364.

[5] Eitel-Amor, G., Meinke, M. & Schröder, W. (2013). A lattice-Boltzmann method with hierarchically refined meshes. *Computers Fluids*, **75**, 127–139.

[6] Johnson, T.A. & Patel, V.C. (1999). Flow past a sphere up to a Reynolds number of 300. *Journal of Fluid Mechanics*, **378**, 19–70.

[7] Klemp, F. & Schlottke, M. (2017). AIA Wiki: Canary (Unpublished internal document.). `http://ldap2.aia.rwth-aachen.de/mediawiki-1.22.1/index.php5/ZFS:Canary` [Accessed: 13.07.2021].

[8] Lintermann, A., Meinke, M. & Schröder, W. (2020). Zonal Flow Solver (ZFS): a highly efficient multi-physics simulation framework. *International Journal of Computational Fluid Dynamics*, **34**, 458–485.

[9] Mallock, A. & Thomson, W. (1896). III. Experiments on fluid viscosity. *Philosophical Transactions of the Royal Society of London. Series A, Containing Papers of a Mathematical or Physical Character*, **187**, 41–56.

[10] Nakamura, I. (1976). Steady wake behind a sphere. *The Physics of Fluids*, **19**, 5–8.

[11] Natarajan, R. & Acrivos, A. (1993). The instability of the steady flow past spheres and disks. *Journal of Fluid Mechanics*, **254**, 323–344.

[12] Pearson, W. (2021). Python Library for Tom's Obvious Minimal Language. `https://pypi.org/project/toml/` [Accessed: 07. June 2021].

[13] Preston-Werner, T. (2021). Toms's Obvious Minimal Language. `https://toml.io/en/` [Accessed: 07. June 2021].

[14] Preston-Werner, T., Gedam, P. & et al. (2021). Repository for Toms's Obvious Minimal Language. `https://github.com/toml-lang/toml` [Accessed: 07. June 2021].

[15] Rimon, Y. & Cheng, S.I. (1969). Numerical Solution of a Uniform Flow over a Sphere at Intermediate Reynolds Numbers. *The Physics of Fluids*, **12**, 949–959.

[16] Rohatgi, A. (2020). Webplotdigitizer: Version 4.4.

[17] Sajip, V. (2021). Logging HOWTO. `https://docs.python.org/3/howto/logging.html` [Accessed: 29. June 2021].

[18] Schlottke, M., Lintermann, A., Meinke, M. & Schröder, W. (2012). Parallel Mesh Generation for Hierarchical Cartesian Grid Methods. *Workshop on Octree-based Methods in Computational Physics*.

[19] Shirayama, S. (1992). Flow past a sphere - Topological transitions of the vorticity field. *Aiaa Journal*, **30**, 349–358.

[20] Taneda, S. (1956). Experimental investigation of the wake behind a sphere at low Reynolds numbers. *Journal of the Physical Society of Japan*, **11**, 1104–1108.

[21] Taylor, G.I. (1923). VIII. Stability of a viscous liquid contained between two rotating cylinders. *Philosophical Transactions of the Royal Society of London. Series A, Containing Papers of a Mathematical or Physical Character*, **223**, 289–343.

[22] Tomboulides, A., Orszag, S. & Karniadakis, G. (1993). Direct and large-eddy simulations of axisymmetric wakes. In *31st Aerospace Sciences Meeting*, 546.

[23] Wang, H. (2015). Experimental and numerical study of Taylor-Couette flow. *Graduate Theses and Dissertations*, **14462**.

Declaration

We herewith declare that the present thesis is the result of our own work without the prohibited assistance of third parties and without making use of aids other than those specified; it includes nothing which is the outcome of work done by other authors, unless for commonly understood ideas or where identified to the contrary. This thesis has not previously been presented in identical or similar form to any other institution, German or foreign, for any degree or any other qualification.

Aachen, 22 August 2021

Marco