

Two-Phase Flow Simulation Using a 1D Model

V 1.1.X

Contents

1	Usage	3
1.1	Installation	3
1.2	Stand-Alone Usage	3
1.2.1	User Case Parameters	3
1.3	Library Usage	4
1.3.1	User-Provided Model	4
1.3.2	User-Provided Biological Reaction Model	5
2	System resolution	6
2.1	Finite difference scheme	6
2.1.1	Boundary condition calculation	7
3	Currently implemented model details	7
3.1	General Physics Implementation	7
3.1.1	Mass transfer	7
3.2	Available Models	8
3.2.1	Bubbly Flow	8
3.2.2	IATE Bubbly Flow	11
3.3	Available Reactions	12
3.3.1	Methanation Model	12
3.3.2	COBRA Methanation Model	12
4	Current implementation	13
4.1	Key Package Dependencies	13
4.2	Overview and data layout	13
4.3	Package Overview	13
	List of Figures	16
	List of Tables	16
	References	16

Introduction

Welcome to the documentation for the Two-Phase Flow 1D Model package. This module serves as both a library for integration into other programs and a standalone application for simulating predefined models.

The library is designed to simulate various two-phase flows using a simplified 1D model. This model considers hydrodynamics, phase transfer, and (bio)reactions. The underlying mathematical model is a system of partial differential equations solved through a finite difference scheme, ultimately reduced to a simple ordinary differential equation (ODE). For more details, refer to the paper authored by (Ngu et al., 2022).

The primary goal is to provide a comprehensive physics model for quickly simulating cases without extensive computation time. The simulation covers transient states to steady states, allowing users to explore the impact of parameters on results without relying on physical approximations or overly complex models.

This documentation focuses on the technical implementation of the model, offering insights into maintenance, modification, and understanding of the package's functionality. The library is designed with independence between elements in mind, allowing users to override interfaces and base classes to define their own models and functions. The aim is to facilitate modifications only where necessary, without coupling between elements (e.g., changing reactions without altering transfer rates).

Feel free to explore the provided interfaces and base classes, enabling customization for specific models and applications. The documentation will guide you through the process of adapting the package to suit your simulation needs.

1 Usage

1.1 Installation

Before you begin, ensure that Python is installed on your system. If not, you can download Python from the official website.

Next, install the required dependencies by executing the following commands in your terminal:

```
python -m pip install poetry
# Feel free to add virtualenv
poetry install
```

Please note that the package relies on certain dependencies. For certain operating systems or architectures, a C compiler might be required to compile dependencies such as LAPACK and BLAS. We also advise installing a C compiler to enable Numba for Just-In-Time (JIT) compilation of code segments, potentially enhancing performance. However, it is important to note that this option remains entirely optional, and the package is built without this feature by default.

1.2 Stand-Alone Usage

In this section, we provide details on how to use the stand-alone version of this package. This involves simulating cases with existing models. Refer to Section 4 for a list of available models.

To begin, you need to provide the program with case information. All this information should be stored in a sheet file named "parameter.xlsx". Note that the default file is located in "./config/", and if no path is given to the program, this file will be used.

It is recommended not to use the package directly in the project root directory. Instead, follow these steps:

1. Create a new directory.
2. Copy the file ./config/parameter.xlsx into this new directory.
3. Use the package from this new location.

Once the file is filled with user case information and the case name (see below for the required information), you can execute the following command: A parameter.xlsx file is automatically generated in the executable root folder however, a list of cases can be found in the "examples" folder in the package repository.

Results can be saved in as binary data to be reused later, however this serialisation needs current package to be imported to open result, to collect structure definition.

```
python -m two_phase_flow_1d_model "path to parameter.xlsx" -c "name of sheet"
```

Note that running

```
python -m two_phase_flow_1d_mode --help
```

will provide more information about how to use the command-line interface.

1.2.1 User Case Parameters

For certain scenarios, specific physical constants such as fluid density or viscosity may need adjustment. This can be achieved by modifying the constants in the "./config/constant.cfg" file.

One file can contain several cases, one case by sheet. Specify the sheet name when running the program to identify the case to execute. The case information is stored in an Excel file (.xlsx), where we specify:

1. Simulation details (duration, discretization, inclusion of reactions, etc.).
2. Experimental physical parameters (feed flow rate, geometry, diameter, etc.).
3. The number of species in both phases, their concentrations within the liquid and gas inlet streams, and some thermodynamic constants.

Ensure that the relevant information is accurately provided in the Excel file, allowing for a comprehensive setup of the simulation case.

Please note that it is currently not possible to provide time-varying feed. Therefore, the given feed concentration will be utilized throughout the entire simulation. An alternative parameter file exists in XML format. XML is easier to manipulate compared to XLSX. This option may be beneficial for developers who aim to integrate this tool into a workflow.

1.3 Library Usage

To use this module as an external library, the module should be installed in the current Python environment.

The minimal example to run a case is as follows:

```
# Use the model that you want
bio_model = ModelMethanation(case.molar_mass)
model = BubblyFlow1DModel(case, simconfig, bio_model)

t, y = fdidsolver.solve(simconfig.spatial_discretization_number,
                        model.n_c,
                        simconfig.simulation_duration,
                        model.z,
                        model.model,
                        model.y0,
                        f_boundary=model.boundaries)
```

Solve function

- The **solve** function is needed for discretizing and solving the Ordinary Differential Equation (ODE) system. In this implementation, it is used to supply matrix coefficients A, B, C, and V, as detailed in Section 2.
- Key methods include **y0**, providing the initial value vector, and **model**, returning a function pointer that, when invoked, yields a tuple with coefficients (A, B, C, V).
- The boundaries method is available for applying user-defined boundary conditions (both steady and time varying constant values, constant fluxes are available).

Model API Models used are subclass of **Base1DModel** Users can either use implemented classes or derive their own.

For a biological model, they can either derive a class from **BiologicalModel**, offering abstraction on coefficient calculations or from **CobraModelHandler** offering the same level of abstract but the possibility to get coefficient from a *.sbml* file and use CoBRAPy.

Additionally, users need to provide interface structures: case and simconfig, which serve as interfaces to gather all necessary information. The interfaces can be found in `"/src/two_phase_flow_1d_model/interfaces.py"`. Users can manually initialize structures or use the loader module to load structures from a sheet file.

*Note that that users can opt for plain functions instead of a class for the model. The solve function is designed to be flexible and can accept *args dictionaries (system_args:Any=None,boundary_args:Any=None) for users to provide additional data needed during coefficient calculations. While plain functions are a valid approach, utilizing a class offers advantages. With a class, users can directly access and modify class attributes, avoiding the need for non-typed *args dictionaries, resulting in more readable, understandable, and maintainable code.*

1.3.1 User-Provided Model

To implement a model for a specific reactor, create a child class of **Base1DModel** or a child of any implemented class. In this example, let's use a child class of **BubblyFlow1DModel** for simplicity. Suppose we intend to simulate an additional chemical reaction that subsequently alters the reaction rate.

To modify the reaction rate, we adjust the partial differential equation (PDE) coefficients. In equation 2.2, we aim to change a specific V_i term associated with the reaction coefficient, leaving others unchanged.

The parent class generates two coefficient matrices, one for mass transfer (which we keep) and another for reaction. Our strategy involves preserving mass transfer coefficients in the parent class, independently calculating the desired reaction rate, and substituting it into the reaction coefficient matrix.

However, clear understanding of computations within the `"user_defined_pde_term"` method in the parent class is crucial for this approach. Be cautious when overloading or overriding a method from a derived class, as it may introduce undefined behavior or side effects. In this case, the reaction calculation occurs only if the `"is_reaction"` flag is set to True, and the transfer occurs if `"is_transfer"` is True in the configuration file. These flags determine which computations are executed, emphasizing careful consideration during implementation or modification.

This process maintains the integrity of other elements like advection and diffusion in the equation. The abstraction layer provides easily manipulable coefficients, including concentration, local gas retention, and velocity in our specific case.

```
import two_phase_flow_1d_model as tpfid
import fdidsolver as solver
import numpy as np
from typing import Any

class YourCustomModel(tpfid.BubblyFlow1DModel):
    def __init__(self, case: tpfid.CaseInfo, sim_config: tpfid.SimConfig, biomodel: tpfid.
        BiologicalModel) -> None:
        super().__init__(case, sim_config, biomodel)

    def init_model(self, sim_info: tpfid.SimConfig, case: tpfid.CaseInfo) -> tpfid.
        ModelInit:
        model = tpfid.ModelInit()# use your own parameters
        return model

    def user_defined_pde_term(self, y, concentration_gas, concentration_liquid) -> tuple[
        np.ndarray[Any, np.dtype[np.float64]], ...]:
        self.is_reaction = False #Flag to be false to avoid reaction, if not undefined
        behaviour with non-initialised biological model
        (mat_mass_transfer, _) = super().user_defined_pde_term(y, concentration_gas,
            concentration_liquid) # call the parent function to get transfert
        new_reaction_rate = # Write here your custom implementation

        return (mat_mass_transfer, new_reaction_rate) # use the mat_mass_transfer
            calculated by the parent class
```

We note that biomodel is not used anymore, so a dummy biological model can be provided. To respect typing, we encourage using a base class of BiologicalModel. Still, as Python remains a non-typed language, None can be provided instead.

Finally, we can use:

```
model = YourCustomModel(case,simconfig,None)

t, y = solver.solve(simconfig.spatial_discretization_number,
                    model.n_c,
                    simconfig.simulation_duration,
                    model.z,
                    model.model,
                    model.y0,
                    f_boundary=model.boundaries)
```

1.3.2 User-Provided Biological Reaction Model

User can also derive class from biologicalModel to provided his own model

2 System resolution

The problem to be solved is described in coordinate-free notation as follows, where $(p, q) \in \mathbb{N}^2$ the number of equation and the space discretization step.

$$\frac{\partial y_j}{\partial t} = \nabla(a_j y_j) + \nabla(b_j \nabla y_j) + c y_j + \sum_i^k V_{i,j} \quad \forall j \in \llbracket 1, p \rrbracket \quad (2.1)$$

Here, y_j represents the concentration of the j -th species, which could be either a gas or liquid species depending on the specific value of j . Specifically, $y_j = c_{\text{gas}j} \epsilon_{\text{gas}}$ when j corresponds to a gas species, and $y_j = c_{\text{liq}j} (1 - \epsilon_{\text{gas}})$ when j corresponds to a liquid species.

This partial differential equation (PDE) is solved using a finite difference scheme, and with the one-dimensional simplification, the resulting system in matrix form is given by:

$$\begin{aligned} \frac{\partial Y}{\partial t} = & (A \cdot (Y \times M_{\text{derivative1}}) + Y \cdot (A \times M_{\text{derivative1}})) + \\ & B \cdot (Y \times M_{\text{derivative2}}) + (Y \times M_{\text{derivative1}}) \cdot (B \times M_{\text{derivative1}}) + (C \cdot Y) + \sum_{i=1}^k V_i \end{aligned} \quad (2.2)$$

In this matrix form, $(Y, A, B, C, V_i) \in \mathcal{M}(p, q)(\mathbb{R})^{(k+4)}$ represents the unknown vector and coefficients, where k is the number of species. Additionally, $(M_{\text{derivative2}}, M_{\text{derivative1}}) \in \mathcal{M}_{q,q}(\mathbb{R})^2$ are matrices generated by the solver itself. It's important to note that this formulation is quite general, allowing for non-constant coefficients a_j and b_j . The solver can adapt to a simplified version when coefficients are constant. The matrices A and B are typically calculated by hydrodynamics (advection and diffusion coefficients), while V_i represents reaction rates or mass transfer rates. The user usually provides coefficients A , B , C , and V . Most of the time, only coefficient V will be supplied by user. This implementation allows for easy modification of coefficients without requiring detailed knowledge of the specific equations being solved at the cost of some optimisation.

Important: This resolution is managed by the `fd1dsolver` module (Finite Difference 1D Solver), and no modifications should be made by the user. The public API consists solely of the `solve` function, along with the `ODEmethod` format.

2.1 Finite difference scheme

Differential operator are discretized using finite difference method, we have $\forall j \in \llbracket 1, p \rrbracket$ the general discretization.

The first order derivative can be written using an upwind scheme as:

$$\frac{\partial(a_j^2 y_j^2)}{\partial z} = a_j^2 \frac{y_j^3 - y_j^1}{2\Delta z} + y_j^2 \frac{a_j^3 - a_j^1}{2\Delta z} \quad (2.3)$$

$$\forall i \in \llbracket 3, q \rrbracket \quad \frac{\partial(a_j^i y_j^i)}{\partial z} = a_j^i \frac{y_j^{i-2} - 4y_j^{i-1} + 3y_j^i}{2\Delta z} + y_j^i \frac{a_j^{i-2} - 4a_j^{i-1} + 3a_j^i}{2\Delta z} \quad (2.4)$$

That can be reduced to the following equation if the coefficient $a_j = \text{cst} \quad \forall i \in \llbracket 1, q \rrbracket$:

$$\frac{\partial(a_j^2 y_j^2)}{\partial z} = a \frac{y_j^3 - y_j^1}{2\Delta z} \quad (2.5)$$

$$\forall i \in \llbracket 3, q \rrbracket \quad \frac{\partial(a_j^i y_j^i)}{\partial z} = a \frac{y_j^{i-2} - 4y_j^{i-1} + 3y_j^i}{2\Delta z} \quad (2.6)$$

$$(2.7)$$

The second order derivative can be written using an upwind scheme as:

$$\forall i \in \llbracket 2, q-1 \rrbracket \quad \frac{\partial a_j^i y_j^i}{\partial z} = \frac{y_j^{i-2} a_j^{i-2} - 4y_j^{i-1} a_j^{i-1} + 3y_j^i a_j^i}{2\Delta z} \quad (2.8)$$

2.1.1 Boundary condition calculation

When working with partial differential equations (PDEs) and employing finite difference schemes, boundary conditions can be specified in two ways. The Dirichlet condition prescribes a constant value at the boundaries, while the Neumann condition dictates a fixed rate of change or gradient at the boundaries, $\frac{\partial y_{j,1}}{\partial z} = f(t)$ and $\frac{\partial y_{j,q}}{\partial z} = g(t)$.

For Dirichlet, Let $(\alpha, \beta) \in \mathbb{R}^2$

$$\frac{\partial y_j^1}{\partial t} = \alpha \quad (2.9)$$

$$\frac{\partial y_j^q}{\partial t} = \beta \quad (2.10)$$

$$(2.11)$$

For Neumann, Let $(y_0, y_{q+1}) \in \mathbb{R}^2$

$$y_0 = y_j^2 - 2\Delta z f(t) \quad (2.12)$$

$$y_{q+1} = y_j^{q-1} + 2\Delta z g(t) \quad (2.13)$$

$$\frac{\partial y}{\partial t}|_{n=1} = -\frac{a_j^2 y_j^2 - a_j^1 y_0}{2\Delta z} + \frac{b_j^1 (y_j^2 - 2y_j^1 + y_0)}{\Delta z^2} \quad (2.14)$$

$$\frac{\partial y}{\partial t}|_{n=q} = -\frac{a_j^q y_{q+1} - a_j^{q-1} y_j^{q-1}}{2\Delta z} + \frac{b_j^q (y_{q+1} - 2y_j^q + y_j^{q-1})}{\Delta z^2} \quad (2.15)$$

3 Currently implemented model details

As of version 1.1.X, the available models are:

3.1 General Physics Implementation

Certain physics models are shared among different models. For instance, there is a common implementation for mass transfer.

3.1.1 Mass transfer

The mass transfer from liquid to gas is described by the following equations:

$$C^* = \frac{H \cdot C_G}{M} \cdot R \cdot T \quad (3.1)$$

$$kl = 2 \cdot \sqrt{\frac{D \cdot u_{rel}}{\pi \cdot db}} \quad (3.2)$$

$$kla = kl \cdot a \quad (3.3)$$

$$MTR = kla \cdot (C^* - C_L) \quad (3.4)$$

where

The interfacial area has to be calculated using a model specific equation.

C^*	:	Maximum solubility of gas in liquid, calculated using Henry's Law.
kl	:	Liquid-side mass transfer coefficient.
kla	:	Overall mass transfer coefficient.
MTR	:	Mass transfer rate.
R	:	Ideal gas constant.
T	:	Temperature.
M	:	Molar mass.
C_G	:	Concentration of gas.
C_L	:	Concentration of liquid.
a	:	Interfacial area coefficient.
u_{rel}	:	Relative velocity.
db	:	Diameter of the bubble.

3.2 Available Models

1. Bubbly Flow Model
2. IATE Bubbly Flow Model

3.2.1 Bubbly Flow

The bubbly flow model simulates columns, both cocurrent and countercurrent, with or without reactions. Specific closure equations are implemented for simulation, covering hydrodynamics, bubble size, and bubble velocity. Additionally, specific arguments for calculating mass transfer are embedded. All equations are taken from original paper (Ngu et al., 2022). However, some modification have been made.

The overall system solved here can be expressed as follows:

Let $p \in \mathbb{N}$ represent the number of species in each phase, and let $q \in \mathbb{N}$ denote the number of nodes. For each pair $(i, j) \in [1, p] \times [1, q]$. Let $y_{G,i} = \epsilon_G C_{G,i}$ represent the gas phase mass fraction of species i , and $y_{L,i} = (\epsilon_L) C_{L,i}$ represent the liquid phase mass fraction of species i . In this context, ϵ_G and ϵ_L are the void fractions or volume fractions of gas and liquid phases respectively. $C_{G,i}$ and $C_{L,i}$ are the concentrations¹ of species i in the gas and liquid phases respectively. This representation allows us to describe the behavior of each species in both gas and liquid phases within the system.

$$\begin{cases} \frac{\partial y_{G,i}}{\partial t} = -\frac{\partial(u_G y_{G,i})}{\partial z} + \frac{\partial}{\partial z}(D_G \frac{\partial y_{G,i}}{\partial z}) - k_{L,i}a(C_i^* - C_{L,i}) \\ \frac{\partial y_{L,i}}{\partial t} = -\frac{\partial(u_L y_{L,i})}{\partial z} + \frac{\partial}{\partial z}(D_L \frac{\partial y_{L,i}}{\partial z}) + k_{L,i}a(C_i^* - C_{L,i}) + \epsilon_L R_i \end{cases}$$

In addition, we also consider the biomass X which is a specific phase as the biomass behaves as a liquid component but cannot transfer nor react in the same way. So we have:

$$\begin{cases} \frac{\partial y_{G,i}}{\partial t} = -\frac{\partial(u_G y_{G,i})}{\partial z} + \frac{\partial}{\partial z}(D_G \frac{\partial y_{G,i}}{\partial z}) - k_{L,i}a(C_i^* - C_{L,i}) \\ \frac{\partial y_{L,i}}{\partial t} = -\frac{\partial(u_L y_{L,i})}{\partial z} + \frac{\partial}{\partial z}(D_L \frac{\partial y_{L,i}}{\partial z}) + k_{L,i}a(C_i^* - C_{L,i}) + \epsilon_L R_i \\ \frac{\partial y_X}{\partial t} = -\frac{\partial(u_L y_X)}{\partial z} + \frac{\partial}{\partial z}(D_L \frac{\partial y_X}{\partial z}) + \epsilon_L R_X \end{cases} \quad (3.5)$$

This system of equations describes the evolution of mass fractions in the gas phase ($y_{G,i}$), the liquid phase ($y_{L,i}$), and the biomass phase (y_X) over time and space.

¹Users should ensure the use of consistent units, particularly when inputting initial concentrations, molar masses, and Henry's law constants, especially in cases involving transfers. Presently, the recommended unit is mg/L, as it aligns well with the (biological) reaction models currently integrated.

Pressure and mixture density Let's calculate the gas fraction using the relationship between pressure and the amount of substance. We start with the expression for the pressure gradient with respect to the vertical coordinate z :

$$\frac{\partial P}{\partial z} = -\rho(z, t) \cdot g \quad (3.6)$$

Here, P represents the pressure, z is the vertical coordinate, $\rho(z, t)$ is the density at height z and time t , and g is the acceleration due to gravity.

The pressure at a certain height z and time t can be expressed as:

$$P(z, t) = P_0 + g \int_z^H \rho(x, t) dx \quad (3.7)$$

In this equation, P_0 is the reference pressure, H is the total height, and $\rho(x, t)$ represents the density at any height x and time t .

We can express the density $\rho(z, t)$ as a combination of the gas phase density $\rho_g(z, t)$ weighted by the gas volume fraction ϵ_g , and the liquid phase density ρ_l weighted by $(1 - \epsilon_g)$. This gives us:

$$\rho(z, t) = \epsilon_g \rho_g(z, t) + (1 - \epsilon_g) \rho_l \quad (3.8)$$

Here, ϵ_g represents the gas fraction or gas volume fraction.

Therefore, the local gas retention can be expressed as follows: Let $q \in \mathbb{N}$ nodes, $\forall j \in [1, q]$:

A *fixed-point iteration algorithm* is used to solve the nonlinear system:

$$\begin{cases} y_g^j &= \epsilon_g C_g^j \\ P_i &= P_0 + g \int_{z^i}^H \rho(x, t) dx \\ \frac{P_j}{R \cdot T} &= \sum_k \frac{C_{g,k}^j}{M_k} \end{cases} \quad (3.9)$$

Here, y_{gas}^i represents the gas phase at node i , P_i is the pressure at node i , P_0 is the reference pressure, g is the acceleration due to gravity, z^i is the height at node i , H is the total height, $\rho(x, t)$ is the density at height x and time t , R is the ideal gas constant, T is the temperature, C_{gas}^i is the gas concentration at node i , M_j is the molar mass of gas species j , and ϵ_{gas} is the void gas fraction.

This system of equations represents the equilibrium conditions. Some bounded optimisation method can be used to solve however currently, system is solved iteratively for y_{gas}^i , P_i , and $C_{\text{gas},j}^i$. Since we expect small variation between time steps, we assume that reusing the current gas retention as algorithm initialization results in rapid convergence, even with a naive method.

Boundary Condition The boundary conditions have been updated to accommodate non-constant coefficients and allow for time varying boundary conditions, particularly in scenarios involving recycling of outling phase.

In the absence of recycling, the boundary conditions remain unchanged. Specifically:

$$\frac{\partial y_{gas}^1}{\partial t} = 0 \quad (3.10)$$

$$\frac{\partial y_{gas}}{\partial t} \Big|_{n=1} = -\frac{u_g^2 y_g^2 - u_g^1 y_g^1}{2\Delta z} + 2 \frac{D_g^1 (y_g^2 - y_g^1)}{\Delta z^2} \quad (3.11)$$

$$\frac{\partial y_{liq}}{\partial t} \Big|_{n=1} = -u_l \frac{y_l^2 - y_l^1}{2\Delta z} + 2 \frac{D_l^1 (y_l^2 - y_l^1)}{\Delta z^2} \quad (3.12)$$

$$(3.13)$$

However, in the presence of recycling, a non-constant boundary condition must be applied to the relevant equation. Users are required to provide a specific function f or g that fit with their specific conditions, as detailed in Equation 2.12 .

Currently, a gas recycling boundary condition is implemented and can be activated. Further details are provided below.

Let use write the gas flow rate F at the bottom of the column with a recycling rate $0 \leq \lambda < 1$

Let $n \in \mathbb{N}$ the number of nodes and $T_f \in \mathbb{R}_+^*$ the final time.

$$\begin{aligned} F: [1, n] \times [0, T_f] &\longrightarrow \mathbb{R} \\ x, t &\mapsto y(x, t) \cdot Q(x, t), \end{aligned}$$

Where Q is the volumetric gas flow rate

$$\forall x, t \in [1, n] \times [0, T_f]; y(0, t) = y_{init} \quad (3.14)$$

$$\forall t \in [0, T_f] \quad \frac{\partial y(n, t)}{\partial z} = 0 \quad (3.15)$$

$$\forall x, t \in [1, n] \times [0, T_f]; y(xi, t) = y_i(t) \quad (3.16)$$

$$\forall t \in [0, T_f] \quad F(0, t) = y(N, t - dt) \lambda \frac{P_N}{P_0} + F_{init} \quad (3.17)$$

Now, consider the flux boundary condition on the first node ($z=0$) using finite difference. The ghost node approach is used here to express the derviative at $z=0$ (below the first node, where the feed flow and recycled flow mix.)

As we cannot calculate it, we have to perform some approximation. To this end, we assume that all our functions are continuous² and use continuity extension. We approximate the derivative as follows:

To simplify, let $y_i(t) = y_i$ (as the following equation is true $\forall t \in [0, T_f]$)

$$\frac{\partial y}{\partial z}|_{z=0} = f(t) \Rightarrow \quad (3.18)$$

$$\frac{\partial y}{\partial z}|_{z=0} \approx \frac{y_2 - y_0}{2\Delta z} = f(t) \Rightarrow \quad (3.19)$$

$$y_0 = y_2 - 2f(t)\Delta Z \quad (3.20)$$

Replace this in the PDE for $z=0$ for the gas species:

$$\frac{\partial y}{\partial z} = -\frac{\partial uy}{\partial z} + \frac{\partial^2 Dy}{\partial z^2} \Rightarrow \quad (3.21)$$

$$\frac{\partial y}{\partial z} = -\frac{u_2 y_2 - u_0 y_0}{\Delta z} + \frac{D}{\Delta z^2} (y_2 - 2y_1 + y_0) \quad (3.22)$$

Now, find the value of $f(t)$: To this end, we assume that the gas velocity $u_0 = u_1 = u_m$. This hypothesis

²This assumption holds true, considering we are dealing with mass flux, which inherently exhibits continuity.

is correct as long as we don't consider any change in the column surface.

$$\frac{\partial y}{\partial z} = f(t), \quad \text{we know the flow rate } F \quad (3.23)$$

$$\frac{\partial F}{\partial z} = \frac{\partial y u S}{\partial z} = S u_m \frac{\partial y}{\partial z} = g(t) \quad (3.24)$$

$$\frac{\partial F}{\partial z} \approx \frac{\Delta F}{\Delta z} \approx \frac{F_1 - F_{recycling}}{\Delta z} = g(t) \quad (3.25)$$

$$F_{recycling} = \lambda F_N \frac{P_N}{P_1} + F_{init} \quad (3.26)$$

$$\Rightarrow f(t) = \frac{g(t)}{S U_m} \quad (3.27)$$

Where S is the column section.

3.2.2 IATE Bubbly Flow

This model is a direct extension of the previous Bubbly Flow Model, enabling simulations within the same framework. A notable addition is the implementation of the Interfacial Area Transport Equation, which, in specific scenarios, can improve the accuracy of bubble diameter changes and gas fraction evolution over time and space.

The formulation of the Interfacial Area Transport Equation is adapted from FLUENT documentation. However, for simplicity, the model currently neglects coalescence and breakage phenomena inherent in bubbly flow. The computation of $\frac{\partial \rho}{\partial z}$ is done with an upwind scheme, while $\frac{\partial \rho}{\partial t}$ is presently neglected. This simplification assumes that rapid density changes at the same nodes are uncommon.

$$\frac{\partial(\rho_g a)}{\partial t} = -\frac{\partial(u_g \rho_g a)}{\partial z} + \frac{\partial^2(D \rho_g a)}{\partial z^2} + \frac{a}{3} \left(\frac{\partial \rho_g}{\partial t} + u_g \frac{\partial \rho_g}{\partial z} \right) + \frac{2}{3} \cdot \frac{\dot{m} tr_g}{\epsilon_g} \cdot a \quad (3.28)$$

This equation represents the transport of the gas-phase density multiplied by the interfacial area with respect to height (z). It considers advection, diffusion, and sources related to the change in gas density and mass transfer rate.

As a result, we can derive the following closure equation:

$$\rho_g = \frac{m_g}{V} = \sum_i C_g^i = \sum_i \frac{y_g^i}{\epsilon_g} \quad (3.29)$$

$$\text{So } \rho_g = \frac{1}{\epsilon_g} \sum_i y_g^i \quad (3.30)$$

$$\Rightarrow \rho_g a = \frac{a}{\epsilon_g} \sum_i y_g^i \quad (3.31)$$

$$\text{Furthermore, we know that } a = \frac{6\epsilon_g}{d_b} \Rightarrow \frac{a}{\epsilon_g} = \frac{6}{d_b} \quad (3.32)$$

$$\Rightarrow d_b = \frac{6 \cdot \sum_i y_g^i}{\rho_g \cdot a} \quad \text{This is directly accessible during simulation} \quad (3.33)$$

These equations establish the closure relation for gas density and interfacial area (a) in terms of gas fraction and bubble diameter. The closure relation indicates that the bubble diameter is inversely proportional to the product of gas density and interfacial area.

Equation Removal : The equation $(\frac{db(z,t)}{db(0,t)})^3 = \frac{\epsilon_g(z,t)}{\epsilon_g(0,t)}$ is no longer necessary due to the closure relation 3.33. The advantage of eliminating this equation lies in the decoupling of bubble diameter and gas fraction, potentially leading to different system behaviors. Therefore the following equation is no longer needed :

In summary, these equations describe the transport of gas-phase density and interfacial area, provide closure relations involving gas fraction and bubble diameter, and justify the removal of a redundant equation for improved computational efficiency and system behavior exploration.

3.3 Available Reactions

The biological models that have been implemented use a biological base class, which serves as an abstraction layer for calculating reaction rates. The calculated reaction rate is determined by the product of two coefficients, denoted as S and R . These coefficients are computed within the class through a method provided by the user. In mathematical terms, the reaction rate (R_{bio}) is expressed as the product of S and R :

$$R_{bio} = S \times R$$

It's the the quantity $R_{bio} \times C_X$ that is used in as PDE coefficient where C_X is the biomass concentration in liquid.

1. Methanation Model
2. COBRA Methanation Model

3.3.1 Methanation Model

As well as the Bubbly Flow model, the Methanation Model is taken from (Ngu et al., 2022).

3.3.2 COBRA Methanation Model

The COBRA Methanation Model is an extension of the Methanation Model, incorporating a .sbml file to describe the reaction stoichiometry. This .sbml file serves as a structured representation of the reaction network, providing information about the reactants, products, and their respective quantities in grams. By utilizing this standardized format, the COBRA Methanation Model enhances the clarity and accessibility of the reaction stoichiometry, facilitating a more systematic and interoperable approach to studying the methanation process.

This model proves valuable for researchers and practitioners involved in the field of methanation, offering a detailed and standardized representation of the underlying chemical reactions involved in methane synthesis.

4 Current implementation

4.1 Key Package Dependencies

Basic linear algebra operations (e.g., `gemm`, `gemv`) NumPy (Harris et al., 2020) is used with default BLAS implementation. However, for more specialized algorithms such as nonlinear system solvers and ordinary differential equations (ODE), the Scipy package is used (Virtanen et al., 2020). Additionally, the model may have dependencies on the COBRAP (Ebrahim et al., 2013) package for running biological simulations.

The broader application relies on dependencies like Matplotlib (Hunter, 2007). Furthermore, OpenPyxl³, lxml⁴, and defusedxml are utilized to handle input requirements.

4.2 Overview and data layout

This package mainly relies on the NumPy interface for linear algebra and array processing. In this context, all containers within the package are treated as NumPy arrays by default. Additionally, since NumPy inherits from the C API, the default array layout is Row-Major (Right-layout). Hence, we have adopted this layout for our containers.

In the context of Partial Differential Equations (PDEs), where the number of spatial nodes typically exceeds the number of equations, the chosen layout proves advantageous for users. This is because the first index corresponds to the system equation (which generally requires no iteration throughout the equation), and the second index (contiguous) represents the spatial nodes. While certain LAPACK routines, utilized by SciPy, may employ a column-major layout, there is no need for user transposition. Consequently, when dealing with a system of n_c equations and n_x spatial nodes, the package represents it as a NumPy array 'Y' of size n_c, n_x . Thus, 'y[0,0]' denotes the first equation's first node, and 'Y[1,-1]' signifies the second equation's last node.

Moreover, to enhance the efficiency of finite quotient Jacobian evaluation, parallel evaluation⁵ is used. This involves using a n-dimensional NumPy array. Functions handling vectorized evaluations may use NumPy arrays of size (n_c, n_x, n_v) , where ' n_v ' represents the quantity of parallel evaluations. It's important to note that ODE resolution and PDE discretization automatically manage this behavior, with the option 'Vectorized' available in the 'fd1dsolver.solve' function.

4.3 Package Overview

We can see in Fig 1 an overview of the package.

Case information is loaded within the main function. Each model is accessible and can be selected by the user through the *run_case* argument and subsequently solved. Post-processing can be performed in another script using the *dump* module to export and import results.

A more detailed structure can be seen in Fig 2

The structure of the solver module is as follows, with functions in the public API: *solve*, *neumann* in and out, and *upwind_scheme*.

³<https://openpyxl.readthedocs.io/en/stable/index.html>

⁴<https://lxml.de/>

⁵Note that this approach differs from parallel execution. Instead of calling the same function n_v times, we evaluate the function once. Information from multiple time steps is gathered within the same variable, allowing the NumPy backend (BLAS) to determine the feasibility of parallel processing. In cases where multithreading is not viable, the computation seamlessly transitions to serial processing along the array dimension. This approach reduces the number of function calls, optimizes cache utilization, and enhances linear optimization, all without relying on traditional multithreading. See Scipy documentation for more information

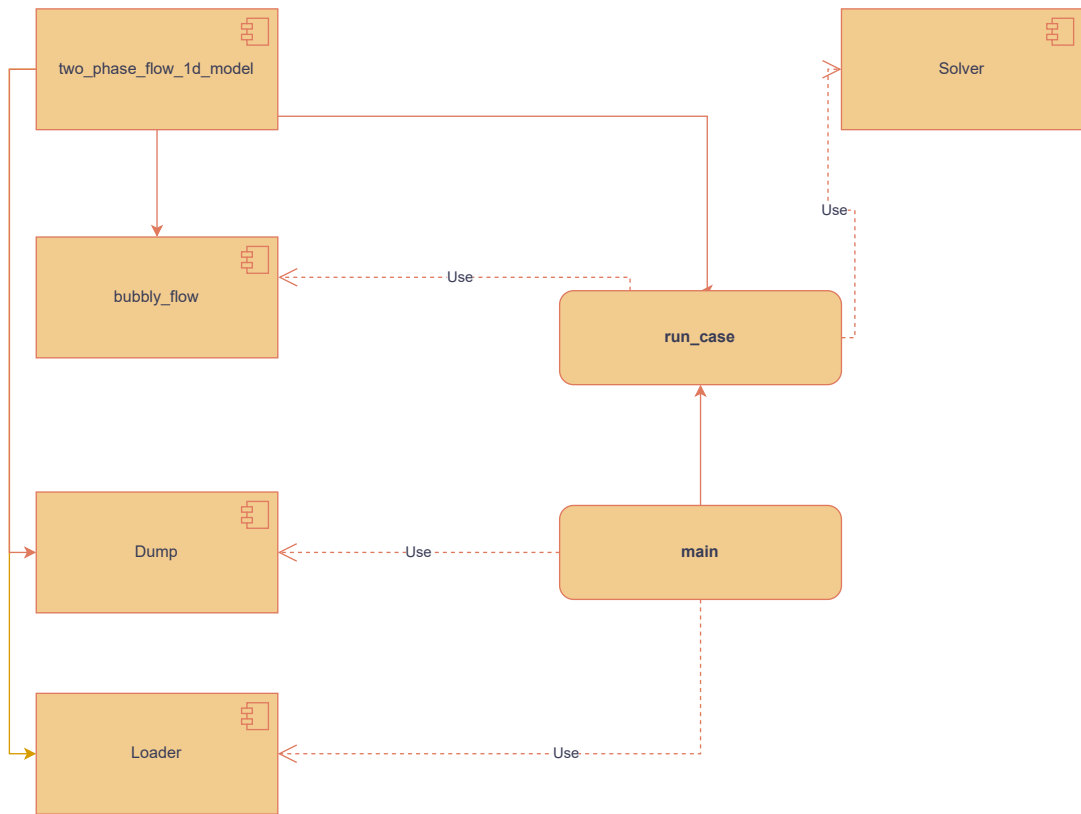


Figure 1: Library overview

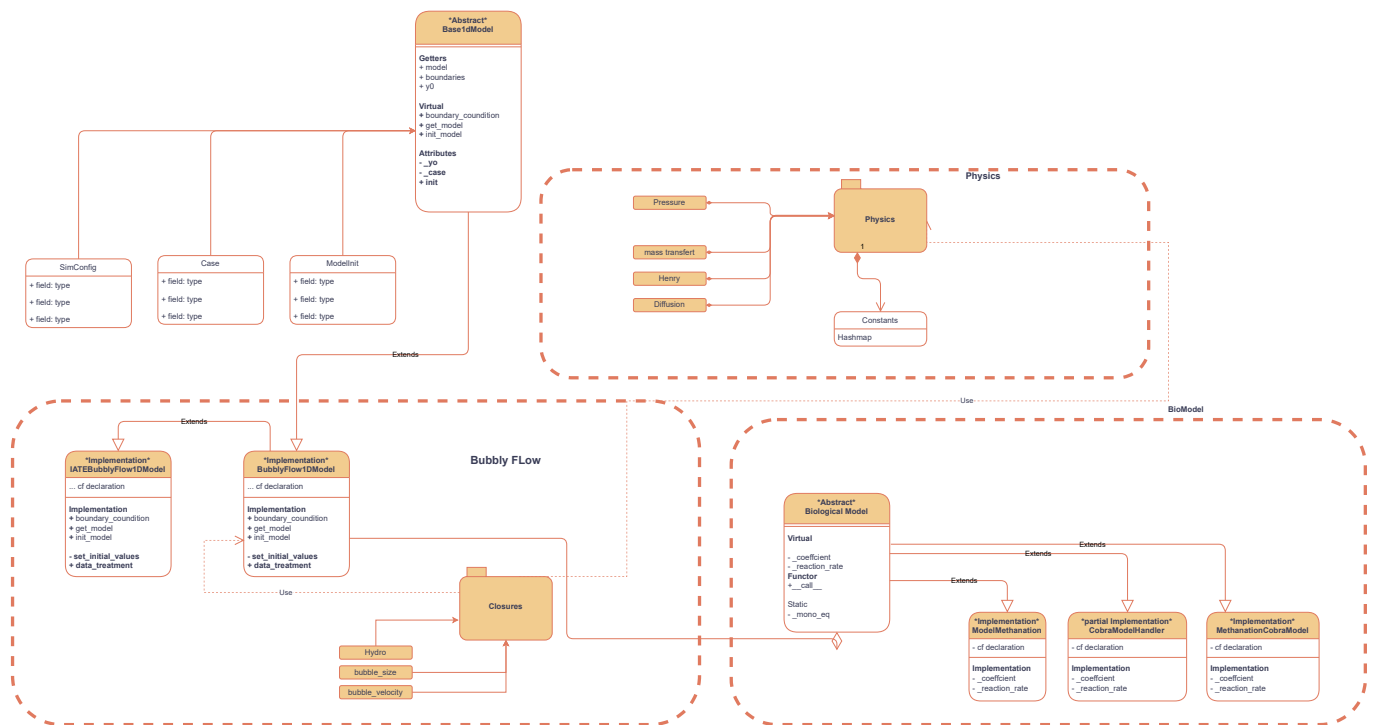


Figure 2: Main class diagram

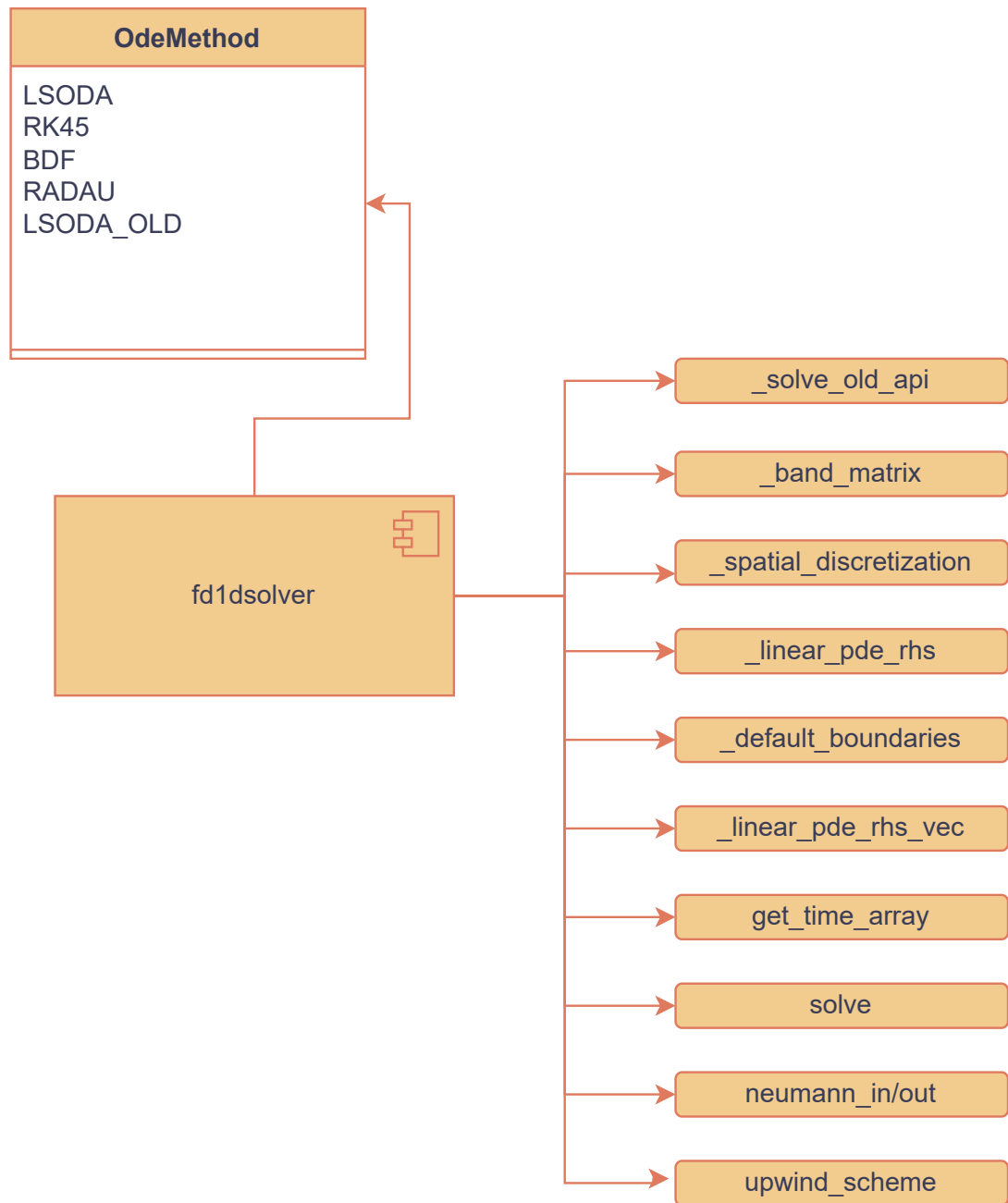


Figure 3: Solver package member

List of Figures

1	Library overviewy	14
2	Main class diagram	14
3	Solver package member	15

List of Tables

References

- Ebrahim, Ali et al. (2013). “COBRApy: CONstraints-Based Reconstruction and Analysis for Python”. In: *BMC Systems Biology* 7.1, p. 74. ISSN: 1752-0509. DOI: 10.1186/1752-0509-7-74. URL: <https://doi.org/10.1186/1752-0509-7-74>.
- Harris, Charles R. et al. (Sept. 2020). “Array programming with NumPy”. In: *Nature* 585.7825, pp. 357–362. DOI: 10.1038/s41586-020-2649-2. URL: <https://doi.org/10.1038/s41586-020-2649-2>.
- Hunter, J. D. (2007). “Matplotlib: A 2D graphics environment”. In: *Computing in Science & Engineering* 9.3, pp. 90–95. DOI: 10.1109/MCSE.2007.55.
- Ngu, Vincent et al. (2022). “Spatio-temporal 1D gas-liquid model for biological methanation in lab scale and industrial bubble column”. In: *Chemical Engineering Science* 251, p. 117478. ISSN: 0009-2509. DOI: <https://doi.org/10.1016/j.ces.2022.117478>. URL: <https://www.sciencedirect.com/science/article/pii/S0009250922000628>.
- Virtanen, Pauli et al. (2020). “SciPy 1.0: Fundamental Algorithms for Scientific Computing in Python”. In: *Nature Methods* 17, pp. 261–272. DOI: 10.1038/s41592-019-0686-2.