HYDROGEN STORAGE SIMULATION – PART 2

Marco Talice

Revision 01

Date: 09/08/2025

Author: Marco Talice

(1/5): Domenico: To do: check whether rhoEqn.H in rhoPimpleFoam has an fvOptions that allows a user defined source term without modifying the source code. yes!

(2/5): Domenico: check how totalPressure inlet condition has been implemented.

(3/5) Domenico: check for models allowing to compute the effective heat capacity of a mixture of two components.

in the thermodynamics settings, specify pure mixture or reactingMixture!
Does rhoPimpleFoam allow to specify the thermodynamics of a pure mixture? If not, should rhoPimpleFoam be replaced by reactingFoam with reactions and chemistry switches off?

(4/5) Domenico: check how a user-defined defined function is able to read a user defined dictionary.

(5/5) Domenico fails to understand the requirements of placing the function computeMdot in global scope as well as constructions made to accomplish this. Domenico wonders about the following construction:

A/ ensure that rhoPimpleFoam is able handle thermodynamics of a pure mixture (or reacting mixture if needed) such that the effective heat capacity of the mixture of H2-gas and MOF solid can be computed as the weighted average of the heat capacity of the components. Switch to reactingFoam (or alternative) if needed;

B/ define a coded source function that computes Mdot (defined inside of fvOptions, thus without modifying the base solver code);

C/ include the source function computeMdot as source function in the equation for rho (gas density, thus rhoEqn.H). This should be possible given (1/5) above. One should be able to test this approach assuming no time-evolution for rho_s (i.e., assuming a constant value for rho_s);

D/ define a coded transport equation for rho_s (solid density) (inside fvOptions, again without modifying source code). Foresee that this transport equation has a source term Mdot.

E/ Define a code source term for the energy equation in terms in H/M = (\rho_s - \rho_empty) / (\rho_sat - \rho_empty);

F/ include porosity of a porous media (as currently done);

1/ Not sure whether to include element of Part-1 to obtain a more complete overview;
2/ Not sure whether to describe rhoPimpleFoam or to refer to documentation on rhoPimpleFoam prior to describing its modifications. Might be useful in supporting further plans;
3/ Suggest to cite work on kiln to highlight previous work on conjugate heat transfer in OpenFoam;

Contents

Seperate Subsection to high light same mesh used for absorption and desorption case?

Do not understand fixed value for U at inlet and fixed value for p at inlet.

Modeling Hydrogen Absorption/Desorption in Metal Hydrides using OpenFOAM

1 Introduction

Metal hydride (MH)–based hydrogen storage is attractive for energy systems because it can store hydrogen at very high densities under safe, moderate conditions. Unlike compressed or liquid hydrogen, MHs absorb hydrogen into a solid matrix, yielding extremely high volumetric hydrogen densities and good gravimetric capacity (Kukkapalli 2023). For example, typical metal–hydrogen compounds can achieve H_2 densities on the order of hundreds of kg-H_2/m^3, which are far above those typical of compressed gas, and even exceed liquid hydrogen's density (≈71 kg-H_2/m^3) (Kukkapalli 2023). At the same time, MH systems operate at relatively low pressures (often <10-20 bar) and near-ambient temperatures (Larpruenrudee 2022). This lowers the risk of high-pressure leaks or cryogenic loss. In practice, an MH tank can contain far more hydrogen by volume than an equivalent-volume pressurized cylinder, yet remain lighter and simpler than liquid storage. These features (high storage capacity and inherent safety), make metal hydrides excellent for stationary and mobile energy storage. If compared with alternative technologies for hydrogen storage (i.e., compressed or liquid hydrogen), MH-based systems offer three main advantages:

High Volumetric and Gravimetric Density: MHs pack hydrogen atoms into metal lattices, so the hydrogen volume shrinks dramatically. Theoretical MHs can reach ≈245 kg-H_2/m^3, versus only tens of kg/m^3 for compressed gas. Even "simple" hydrides like La Ni_5H_6 or MgH_2 yield very high energy per unit volume (Kukkapalli 2023).

Safety (Low Pressure/Temperature): MH tanks are typically charged at modest pressures (1-20 bar) and release hydrogen only when heated. Because hydrogen is chemically bound in the solid, the risk of sudden release or explosion is much lower than with high-pressure vessels (Larpruenrudee 2022). Many metal hydrides can absorb/release at near-ambient conditions (e.g. $LaNi_5$, TiFe at T<100 ^oC) (Kukkapalli 2023), (Larpruenrudee 2022). This "cold" operation leverages waste heat (e.g. from a fuel cell) to drive hydrogen release, further enhancing system safety and efficiency (Larpruenrudee 2022), (Spiegel 2019).

Reversibility: The MH hydrogenation reaction is fully reversible and can cycle many times. On charging (hydriding) the metal absorbs H_2 exothermically, and on discharging (dehydriding) it releases H_2 with heat input (Spiegel 2019). This chemistry can be repeated with minimal degradation (especially with improved alloys), making MH storage robust for long-term use (Larpruenrudee 2022).

The performance of a hydride-based hydrogen storage system is primarily determined by its storage capacity and the rate at which hydrogen can be absorbed and released. While the storage capacity largely depends on the specific material used, the charging and discharging rates are affected by various factors, the most important of which is the thermal management. This is because the reaction by which a metal (M) reacts with hydrogen (H_2) to form a hydride (MH_x):

$$M(s) + \frac{x}{2} H_2\,(g) \rightleftharpoons MH_x(s) \pm \Delta H$$

Is strongly influenced by ΔH. The absorption process is an exothermic chemical reaction that releases a significant amount of heat (ΔH). If not properly dissipated, this heat eventually reverses the chemical equilibrium toward the desorption reaction (hydrogen release), severely impacting the performance of the metal hydride tank. Equivalent chemical principles govern the desorption reaction, but now heat should be provided to the tank to favor the release of H_2 as a gas.

2 Heat Management Strategies in Literature

As previously mentioned, efficient thermal management is essential to sustain the heat exchange required for hydrogen absorption and desorption processes in metal hydride tanks (MHTs). To optimize these processes, various strategies have been developed to manage the heat generated or absorbed during hydrogen sorption. These thermal regulation systems can be categorized into three main types: **active systems**, which employ embedded spiral or finned heat exchangers; **passive systems**, which use phase change materials (PCMs) to absorb or release latent heat, and **hybrid systems**, where PCMs are combined with metal foams or particles to improve thermal conductivity (Liu 2025). Figure 1, adapted from (Liu 2025), provides an overview of hydride-based hydrogen storage systems along with their typical performance metrics. Additionally, Table 1 summarizes key literature references for each system type.

Table 1: Some literature references for hydride-based hydrogen storage systems

System description	Reference
Cooling tubes - external water jacket	(Karmakar 2021)
Double-layered reactor - cooling tubes - N2 as heat transfer fluid	(Lin 2021)
Honeycomb hexagonal fins	(Afzal, Design and computational analysis of a metal hydride hydrogen storage system with hexagonal honeycomb based heat transfer enhancements-part A 2021)
Internal conical fins - Embedded cooling tubes	(Chandra 2020)
With phase change material	(Jemni 1995), (Nasrallah 1997), (Freni 2009), (Ye 2020), (Darzi 2016)
Multi-tubular storage	(Afzal 2018) 3
Dual coil cooling tube	(Tong 2019)

Compartmentalized reactor	(Chippar 2018)
Concentric finned tube	(Askri 2009)
Water outer jacket, Embedded spiral cooling tube	(Mellouli 2009)
Cooling tubes interconnected by aluminum fins	(Raju 2011)

Storage materials	Tank design	System description	Charging rate [kg H_2 per min]
10 kg $LaNi_5$		Cooling tubes External water jacket	0.004
110 g $LaNi_{4.25}Al_{0.75}$		Double-layered reactor Cooling tubes N_2 as heat transfer fluid	<0.001
50 kg $La_{0.9}Ce_{0.1}Ni_5$		Honeycomb hexagonal fins	0.005
5 kg $LaNi_5$		Internal conical fins Embedded cooling tubes	0.008
10 kg MgH_2		With phase change material	0.029
210 kg Ti—Mn		Multi-tubular storage	0.078
$LaNi_5$		Dual coil cooling tube	0.001
$LaNi_5$		Compartmentalised reactor	0.001
$LaNi_5$		Concentric finned tube	0.003
1 kg $LaNi_5$		Water outer jacket, Embedded spiral cooling tube	0.001
100 kg $NaAlH_4$		Cooling tubes interconnected by aluminum fins	0.272

Figure 1: Summary of reported metal hydride storage systems performance, as reported in the literature (Liu 2025).

3 Theoretical Model

The mathematical model presented hereinafter is based on the following assumptions:

• Hydrogen is the only fluid, and it behaves like an ideal gas;

• The hydride is considered as an isotropic bed with homogeneous porosity and no change in volume;

Is it sufficiently clear what properties are intended here?

• Th[illegible] main physical properties [illegible]remain constant during the reaction;

• Local thermal equilibrium [illegible]between the hydride bed and the hydrogen gas is assumed.

Same question: is it clear what is meant here? Or refer to next page for more elaborate explanation?

Under these assumptions, conservation equations of mass, momentum and energy for gas and solid can be written as:

Equation 1: Solid hydride mass balance

$$(1-\varepsilon)\frac{\partial \rho_s}{\partial t} = \dot{m}$$

Equation 2: Gas phase mass balance

$$\varepsilon\frac{\partial \rho_g}{\partial t} + \nabla\cdot\left(\rho_g\vec{u}\right) = \dot{m}$$

Where ε is the material's porosity, and ρ_g and ρ_s are the densities of hydrogen and solid.

Equation 3: Reaction kinetics for absorption

$$\dot{m}_{abs} = C_a \exp\left(-\frac{E_a}{RT}\right)\ln\left(\frac{p}{p_{\mathrm{eq,abs}}}\right)(\rho_{\mathrm{sat}} - \rho_s)$$

Equation 4: Reaction kinetics for desorption

$$\dot{m}_{des} = C_d \exp\left(-\frac{E_d}{RT}\right)\left(\frac{p - p_{\mathrm{eq,des}}}{p_{\mathrm{eq,des}}}\right)(\rho_s - \rho_{\mathrm{emp}})$$

Where C_a and C_d are dimensional constants (1/s), and ρ_{emp} and ρ_{sat} are the bed's densities at the initial (empty) and final (saturated) conditions.

Equation 5: Van't Hoff equilibrium pressure (as in (Darzi 2016))

$$\ln\left(\frac{p_{\mathrm{eq,abs/des}}}{p_{\mathrm{ref}}}\right) = A_{abs/des} - \frac{B_{abs/des}}{T}$$

Equation 6: polynomial equilibrium pressure (as in (Jemni 1995))

$$\frac{p_{eq,abs/des}}{p_{ref}} = \sum_{j=1}^{n=5} a_j (H/M)^j \exp\left(\frac{\Delta H_{abs/des}}{R_g}\left(\frac{1}{T} - \frac{1}{T_0}\right)\right)$$

Where coefficients A, B, and a_j in Equation 5 and Equation 6 are either taken from a material properties data base or experimentally determined. The quantity H/M that appears in Equation 6 is given by:

Equation 7: H/M definition

$$H/M = \frac{\rho_s - \rho_{emp}}{\rho_{sat} - \rho_{emp}}$$

Equation 8: Momentum equation (hydrogen)

$$\frac{\partial \rho_g \vec{v}}{\partial t} + \nabla \rho_g \overrightarrow{v\vec{v}} = -\nabla p + \nabla \bar{\tau} + \phi_D$$

Equation 9: Darcy's law

$$\phi_D = -\frac{K}{\mu}\nabla p$$

Darzi Eqn. (8) formulates this differently We imagine this to be the correct expression.

Where K is the material's permeability that can be experimentally determined from the particles' average size (d_p) and the porosity ϵ, via the Kozeny-Carman equation:

Equation 10: Kozeny-Carman equation

$$K = \frac{d_p^2}{150}\frac{\varepsilon^3}{(1-\varepsilon)^2}$$

Equation 11: Gas energy equation

$$\varepsilon \rho_g C_{p,g} \frac{\partial T_g}{\partial t} = \varepsilon k_g \nabla^2 T_g - \rho_g C_{p,g}\left(\vec{v}\nabla T_g\right) + \phi_{Q_g}$$

Equation 12: Solid energy equation

$$(1-\varepsilon)\rho_s C_{p,s}\frac{\partial T_s}{\partial t} = (1-\varepsilon)k_s \nabla^2 T_s + \phi_{Q_s}$$

Where $C_{p,g}$, $C_{p,s}$, k_g, and k_s are the specific heat coefficients of hydrogen and solid, and their thermal conductivity coefficients. The source term in Equation 11 and Equation 12 is computed as:

Equation 13: Energy equations' source term

Definition?

$$\phi_{Q,abs/des} = \phi_{abs/des}\left(\frac{\Delta H_{abs/des}}{M_{H_2}} + T\left(C_{p,g} - C_{p,s}\right)\right)$$

Where M_{H_2} is the molecular mass of hydrogen. Because of the assumption of local thermal equilibrium between hydrogen and hydride (i.e., $T_g = T_s$), their individual specific heat capacities and thermal conductivities can be condensed into a single effective specific heat capacity $\left(\rho C_p\right)_{eff}$ and a single effective thermal conductivity $(k)_{eff}$:

Equation 14: Effective specific heat

$$(\rho C_p)_{eff} = \varepsilon \rho_g C_{p,g} + (1-\varepsilon)\rho_s C_{p,s}$$

Equation 15: Effective thermal conductivity

$$k_{eff} = \varepsilon k_g + (1-\varepsilon)k_s$$

Equation 16: Merged energy equation

$$(\rho C_p)_{\text{eff}} \frac{\partial T}{\partial t} + \rho_g C_{p,g} \vec{v} \cdot \nabla T = \nabla \cdot (k_{\text{eff}} \nabla T) - \dot{m} \left[(1-\varepsilon) \frac{\Delta H_{abs/des}}{M_{H_2}} - T(C_{p,g} - C_{p,s}) \right]$$

Is rhoPimpleFoam best suited for development? Should a two-phase solver be choosen instead?

4 Implementation in OpenFOAM Solver

The original energy equation in rhoPimpleFoam is designed for general compressible flows and does not account for the specific thermochemical phenomena involved in hydrogen absorption and desorption in metal hydrides. As a result, the standard formulation is incapable of solving the full set of governing equations relevant to this problem, namely, Equation 1 - Equation 9 together with Equation 11 and Equation 12, or alternatively Equation 14 to Equation 16.

To enable rhoPimpleFoam to solve the hydrogen absorption/desorption model, the following modifications are necessary:

1. **Reaction kinetics and equilibrium pressures**: Evaluate the absorption and desorption rates ($\dot{m}_{abs}$, $\dot{m}_{des}$), and the corresponding equilibrium pressures ($P_{eq,abs}$, and $P_{eq,des}$) using Equation 3, Equation 4 and Equation 5;
2. **Solid-phase mass conservation**: Introduce a new scalar transport equation for the solid hydride density ρ_s based on Equation 1;
3. **Inclusion of the source terms** $\frac{\dot{m}}{(1-\epsilon)}$ **and** $\frac{\dot{m}}{\epsilon}$ in Equation 1 and Equation 2;
4. **Computation of the source term** $\boldsymbol{\phi_D}$ and its inclusion in the momentum equation via the Darcy law (Equation 9), and eventually using Equation 10 to evaluate experimentally the hydride permeability K from the hydride's particles' diameter d_p if not directly known;
5. **Resolution of the energy gas and solid energy equations**, either in the form of Equation 11 and Equation 12, or as the merged energy equation (Equation 16).

The first four tasks can be implemented using OpenFOAM's built-in explicitPorositySource[1] and scalarCodedSource[2] function objects within constant/fvOptions. This approach avoids modifying the original solver code, except for the addition of the ρ_s, equation, since all source terms are handled externally. However, a notable limitation is that the reaction rate $\dot{m}$ (and thus P_{eq}) must

Equilibrium pressure, either absoprtion or desorption.

[1] https://www.openfoam.com/documentation/guides/latest/doc/guide-fvoptions-sources-explicit-porosity.html

[2] https://www.openfoam.com/documentation/guides/latest/doc/guide-fvoptions-sources-coded.html

Can scalarCodedSource be instructed to read properties from a user-defined dictionary?

Do not understand, Can \dot{m} be stored in the database of runtime object instead?

be recalculated wherever it is used, due to the local scope of function objects (meaning values computed in one source term cannot be reused in another).

The treatment of the energy equations (Equation 11 and Equation 12, or of Equation 16 with Equation 14 and Equation 15) requires further consideration. One option is to retain the original energy equation in rhoPimpleFoam, formulated either in terms of internal energy **e** or enthalpy **h**, to represent the gas-phase energy balance (Equation 11). Thermophysical properties such as $C_{p,g}$ and κ_g[3] would be directly assigned in constant/thermophysicalProperties, and the only further modification needed would be the introduction of a scalarCodedSource object in fvOptions to compute the source term $\frac{\Phi_{Q,abs/des}}{}$ using Equation 13. Solving the solid-phase energy equation (Equation 12) , however, necessitates the addition of a separate scalar transport equation. Unlike the gas phase, the solid thermophysical properties ($C_{p,s}$ and κ_s) cannot be defined in constant/thermophysicalProperties, and as of OpenFOAM v2306, no existing function objects allow them to be specified externally in constant/fvOption. These parameters must therefore be hard-coded into the new scalar equation source file—an approach that reduces modularity and code reusability. The corresponding heat source term can again be included using an ad hoc function object, similar to the gas-phase energy source.

Do not understand this part.

What method is meant here?

The advantage of this method is that the original energy equation can be reused for the gas phase (Equation 11) , minimizing solver modification. However, the downside is the increased complexity due to solving two additional scalar equations (for ρ_s and the solid-phase energy), with the added constraint of embedding solid thermophysical properties directly in the source code.

Read solid properties from file as done for properties of the lining for the cement kiln?

Do not understand. The number of equation is dictated by the model, not by the solver.

To avoid increasing the number of equations in the solver, an alternative approach is to adapt OpenFOAM's built-in energy equation to represent Equation 16 , in which the effective specific heat capacity (Equation 14) and thermal conductivity (Equation 15) account for the combined thermal behavior of the gas and solid phases. The heat source term can once again be implemented using a function object in constant/fvOptions. This method appears optimal, as it avoids introducing additional scalar transport equations and handles all necessary modifications externally via function objects.

However, this approach presents a fundamental limitation: while the effective thermal conductivity κ_{eff} remains constant once ϵ, κ_g, and κ_s are specified, the effective heat capacity, $\left(\rho\, C_p\right)_{eff}$ varies over time, as it depends on the evolving fields ρ_g and ρ_s (see Equation 14). In OpenFOAM v2306, the specific heat capacity can only be defined as a constant or as a polynomial function of temperature, but not as a general function of other field variables. As a result, despite the conceptual elegance and implementation efficiency of this strategy, it is ultimately not feasible due to the limitations imposed by the current OpenFOAM framework.

[3] Compressible solvers such as rhoPimpleFoam do not directly use κ, but compute its value from the values of C_p, μ, and Prandtl number assigned in the constant/thermophysicalProperties file ($\kappa = \frac{C_p\,\mu}{Pr}$).

To overcome these limitations, we propose the modifications to the original rhoPimpleFoam shown in Figure 2. The following sections provide a detailed explanation of each functional block depicted in the figure, highlighting the specific parts of the code where the absorption/desorption model, described by Equation 1 to Equation 9, and Equation 13 to Equation 16, is implemented.

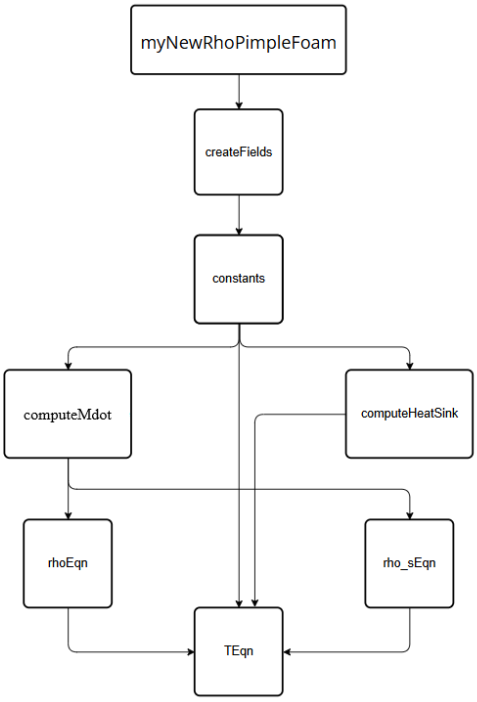

Figure 2: Flow chart of the modification made to the original rhoPimpleFoam solver

4.1 myNewRhoPimpleFoam.H (main program)

Modifications to the original rhoPimpleFoam.H are highlighted in Figure 3.

1. The object used to read-in the model's constants in included and executed.
2. Computation of $\dot{m}$ (Equation 3 and Equation 4) is included.
3. The newly defined equations for rho_s (Equation 1) and T (Equation 16) are included in the solver, while the original energy equation is removed.

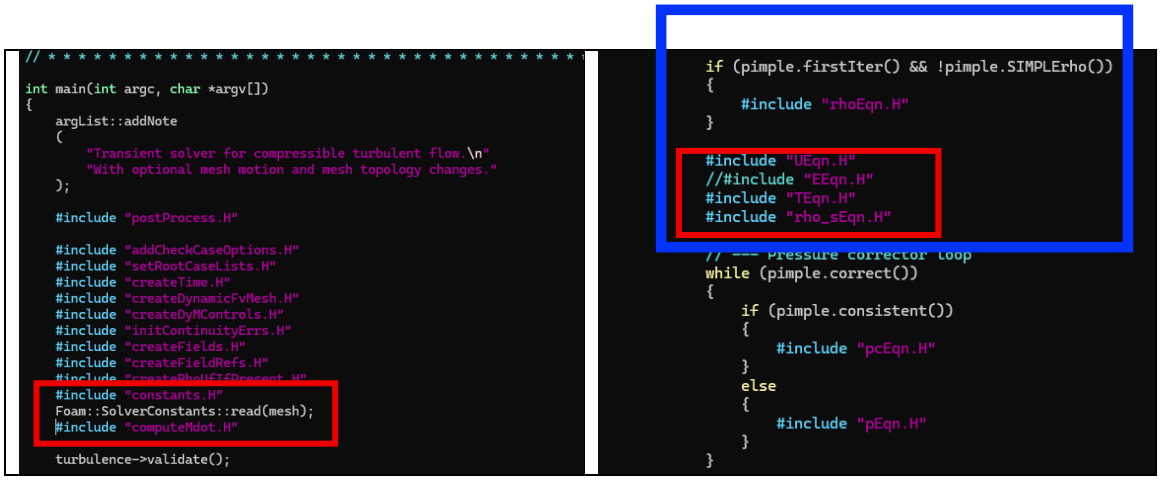

Figure 3: Modifications made to rhoPimpleFoam.H

4.2 createFields.H

Modifications to the original createFields.H are highlighted in Figure 4.

1. Validations of h, e, and T is commented out to avoid conflicts with the original code.
2. The instruction T.read() forces the solver to read the temperature field from disk to ensure that the temperature boundary conditions are read and applied.
3. The volScalarField rho_s is defined here for later use in the newly defined equation solved in rho_sEqn.h

Why not treat temperature T in same way as p, U, rho and rho_s (for consistency)?

```
//thermo.validate(args.executable(), "h", "e");
//thermo.validate(args.executable(), "T");

volScalarField& T = thermo.T();  // ✅ mutable reference for solving
T.read();  // 🔥 Force-read the field from disk

volScalarField rho
(
    IOobject
    (
        "rho",
        runTime.timeName(),
        mesh,
        IOobject::READ_IF_PRESENT,
        IOobject::AUTO_WRITE
    ),
    thermo.rho()
);

Info<< "Reading field U\n" << endl;
volVectorField U
(
    IOobject
    (
        "U",
        runTime.timeName(),
        mesh,
        IOobject::MUST_READ,
        IOobject::AUTO_WRITE
    ),
    mesh
);

Info << "Reading field rho_s (metal hydride loading)" << endl;
volScalarField rho_s
(
    IOobject
    (
        "rho_s",
        runTime.timeName(),
        mesh,
        IOobject::READ_IF_PRESENT,
        IOobject::AUTO_WRITE
    ),
    mesh,
    dimensionedScalar("rho_s", dimDensity, 7164.0)  // default to rhoEmp if field missing
);
```

Figure 4: Changes to createFields.H

4.3 constants.H

The physical and thermodynamic parameters required by the mathematical model (such as activation energies, reaction enthalpies, specific heat capacities, thermal conductivities, and porosity), are initialized in constants.H and constants.C, and are loaded dynamically at runtime. As shown in Figure 5, the header file constants.H is included in the main solver file myNewRhoPimpleFoam.C, where its contents are instantiated and accessed during the simulation. The definition of the SolverConstants class provided in constants.H is illustrated in Figure 6, where each model parameter is declared as a static dimensionedScalar. This approach ensures that all derived quantities maintain consistent physical units throughout the solver, in accordance with OpenFOAM's unit-checking system. At the beginning of the constants.C file, the physical units of the previously declared constants are explicitly assigned, as illustrated in Figure 7. The two

different approaches supported by OpenFOAM for defining the units of a **dimensionedScalar** are demonstrated. In the first method, the units are specified by directly providing the exponents of each base dimension using the **dimensionSet** constructor, for example:

dimensionedScalar SolverConstants::M_H2("M_H2", dimensionSet(1, 0, 0, 0, -1, 0, 0), 0);

In the second method, OpenFOAM's predefined dimensional constants are used to simplify unit specification. An example of this more concise form is:

dimensionedScalar SolverConstants::rhoEmp("rhoEmp", dimMass/dimVolume, 0);

Both approaches are functionally equivalent and serve to enforce dimensional consistency within OpenFOAM's type-checking system, thereby reducing the likelihood of unit-related errors in numerical operations.

Figure 8 highlights the section of constants.C where the IOdictionary object is constructed to retrieve parameter values from the customConstants dictionary, which is located in the **constant/** directory of the simulation case. This mechanism allows model parameters to be externally configured without recompiling the solver, thereby enhancing flexibility and reproducibility.

Finally, Figure 9 shows how the constants' values are actually read from the dictionary with statements like:

constant-name.value() = constDict.get<scalar>("*constant-name*");

```
#include "fluidThermo.H"
#include "turbulentFluidThermoModel.H"
#include "bound.H"
#include "pimpleControl.H"
#include "pressureControl.H"
#include "CorrectPhi.H"
#include "fvOptions.H"
#include "localEulerDdtScheme.H"
#include "fvcSmooth.H"

// * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * //

int main(int argc, char *argv[])
{
    argList::addNote
    (
        "Transient solver for compressible turbulent flow.\n"
        "With optional mesh motion and mesh topology changes."
    );

    #include "postProcess.H"

    #include "addCheckCaseOptions.H"
    #include "setRootCaseLists.H"
    #include "createTime.H"
    #include "createDynamicFvMesh.H"
    #include "createDyMControls.H"
    #include "initContinuityErrs.H"
    #include "createFields.H"
    #include "createFieldRefs.H"
    #include "createRhoUfIfPresent.H"
    #include "constants.H"
    Foam::SolverConstants::read(mesh);
    #include "calculateReactionRate.H"
```

Figure 5: The new routine used to read-in the mathematical model's constant is included and read-in at the beginning of myNewRhoPimpleFoam.H

```
#ifndef CONSTANTS_H
#define CONSTANTS_H

#include "dictionary.H"
#include "dimensionedScalar.H"
#include "IOdictionary.H"
#include "IOobject.H"
#include "dimensionedTypes.H"
#include "fvMesh.H"

namespace Foam
{
    class SolverConstants
    {
    public:
        static dimensionedScalar Ca;         | static dimensionedScalar Cd;
        static dimensionedScalar Ea;           static dimensionedScalar Ed;
        static dimensionedScalar Aabs;         static dimensionedScalar Babs;
        static dimensionedScalar Ades;         static dimensionedScalar Bdes;
        static dimensionedScalar Pref;         static dimensionedScalar M_H2;
        static dimensionedScalar rhoEmp;       static dimensionedScalar rhoSat;
        static dimensionedScalar porosity;     static dimensionedScalar Cp_g;
        static dimensionedScalar Cp_s;         static dimensionedScalar k_g;
        static dimensionedScalar k_s;          static dimensionedScalar R;
        static dimensionedScalar dH_mol;       static dimensionedScalar T_amb;
        static dimensionedScalar C;            static dimensionedScalar rho_p;
        static dimensionedScalar Cp_p;         static dimensionedScalar volRatio;
        static dimensionedScalar useFunction; static dimensionedScalar Tmin;
        static dimensionedScalar Tmax;

        static void read(const fvMesh& mesh);
    };
}

#endif
```

Figure 6: The "constant.H" file

```
#include "constants.H"
#include "dimensionedTypes.H"
#include "fvMesh.H"
#include "basicThermo.H"

namespace Foam
{
    dimensionedScalar SolverConstants::Ca("Ca", dimless/dimTime, 0);
    dimensionedScalar SolverConstants::Cd("Cd", dimless/dimTime, 0);
    dimensionedScalar SolverConstants::Ea("Ea", dimEnergy/dimMoles, 0);
    dimensionedScalar SolverConstants::Ed("Ed", dimEnergy/dimMoles, 0);
    dimensionedScalar SolverConstants::Aabs("Aabs", dimless, 0);
    dimensionedScalar SolverConstants::Babs("Babs", dimEnergy/dimMoles, 0);
    dimensionedScalar SolverConstants::Ades("Ades", dimless, 0);
    dimensionedScalar SolverConstants::Bdes("Bdes", dimEnergy/dimMoles, 0);
    dimensionedScalar SolverConstants::Pref("Pref", dimPressure, 0);
    dimensionedScalar SolverConstants::M_H2("M_H2", dimensionSet(1, 0, 0, 0, -1, 0, 0), 0);
    dimensionedScalar SolverConstants::rhoEmp("rhoEmp", dimMass/dimVolume, 0);
    dimensionedScalar SolverConstants::rhoSat("rhoSat", dimMass/dimVolume, 0);
    dimensionedScalar SolverConstants::porosity("porosity", dimless, 0);
    dimensionedScalar SolverConstants::Cp_g("Cp_g", dimEnergy/dimMass/dimTemperature, 0);
    dimensionedScalar SolverConstants::Cp_s("Cp_s", dimEnergy/dimMass/dimTemperature, 0);
    dimensionedScalar SolverConstants::k_g("k_g", dimEnergy/dimTime/dimLength/dimTemperature, 0);
    dimensionedScalar SolverConstants::k_s("k_s", dimEnergy/dimTime/dimLength/dimTemperature, 0);
    dimensionedScalar SolverConstants::R("R", dimEnergy/dimMoles/dimTemperature, 0);
    dimensionedScalar SolverConstants::dH_mol("dH_mol", dimEnergy/dimMoles, 0);
    dimensionedScalar SolverConstants::T_amb("T_amb", dimTemperature, 0);
    dimensionedScalar SolverConstants::C("C", dimless, 0);
    dimensionedScalar SolverConstants::rho_p("rho_p", dimMass/dimVolume, 0);
    dimensionedScalar SolverConstants::Cp_p("Cp_p", dimEnergy/dimMass/dimTemperature, 0);
    dimensionedScalar SolverConstants::volRatio("volRatio", dimless, 0);
    dimensionedScalar SolverConstants::useFunction("useFunction", dimless, 0);
    dimensionedScalar SolverConstants::Tmin("Tmin", dimTemperature, 273.0);
    dimensionedScalar SolverConstants::Tmax("Tmax", dimTemperature, 10000.0);
```

Figure 7: Units assignment to the defined dimensionedScalar constants at the beginning of constants.C

```
void SolverConstants::read(const fvMesh& mesh)
{
    // Add to mesh object registry first
    IOdictionary customDict
    (
        IOobject
        (
            "customConstants",
            mesh.time().constant(),
            mesh,
            IOobject::MUST_READ,
            IOobject::NO_WRITE
        )
    );

    const dictionary& constDict = customDict.subDict("reactionParameters");
```

Figure 8: definition of the IOobject custom dictionary used to read the "customConstants" dictionary in constants.C

```
Ca.value() = constDict.get<scalar>("Ca");
Cd.value() = constDict.get<scalar>("Cd");
Ea.value() = constDict.get<scalar>("Ea");
Ed.value() = constDict.get<scalar>("Ed");

// van't Hoff equation constants
Aabs.value() = constDict.get<scalar>("Aabs");
Babs.value() = constDict.get<scalar>("Babs"); // J/mol
Ades.value() = constDict.get<scalar>("Ades");
Bdes.value() = constDict.get<scalar>("Bdes"); // J/mol
Pref.value() = constDict.get<scalar>("Pref");         // Pa

// Material constants
M_H2.value()   = constDict.get<scalar>("M_H2");   // kg/mol
rhoEmp.value() = constDict.get<scalar>("rhoEmp"); // kg/m³
rhoSat.value() = constDict.get<scalar>("rhoSat"); // kg/m³
porosity.value() = constDict.get<scalar>("porosity");

// Thermophysical properties
Cp_g.value() = constDict.get<scalar>("Cp_g"); // J/kg/K
Cp_s.value() = constDict.get<scalar>("Cp_s"); // J/kg/K
k_g.value()  = constDict.get<scalar>("k_g"); // W/m/K
k_s.value()  = constDict.get<scalar>("k_s"); // W/m/K

// Misc
R.value()      = constDict.get<scalar>("R"); // J/mol/K
dH_mol.value() = constDict.get<scalar>("dH_mol"); // J/mol
T_amb.value()  = constDict.get<scalar>("T_amb"); // K
C.value() = constDict.get<scalar>("C");
rho_p.value() = constDict.get<scalar>("rho_p"); // kg/m³
Cp_p.value() = constDict.get<scalar>("Cp_p"); // J/kg/K
volRatio.value() = constDict.get<scalar>("volRatio");
useFunction.value() = constDict.get<scalar>("useFunction");

// Temperature limits
Tmin.value() = constDict.get<scalar>("Tmin");
Tmax.value() = constDict.get<scalar>("Tmax");
```

Figure 9: The section of constants.C where constants' values are read from the user defined dictionary

4.4 computeMdot.H (Reaction Kinetics)

Equation 3, Equation 4, together with the van't Hoff Equation 5 are implemented in computeMdot.H and computeMdot.C, shown in Figure 10 and Figure 11. Values of p, T and ρ_s used in the calculation of p_{eq} and $\dot{m}$ are those coming from the previous iteration (or from the initial conditions at the beginning of the simulation). The computed value of $\dot{m}$ is multiplied by volRatio, which is the ratio of the wedge volume to the full domain volume:

$$volRatio = \frac{\alpha\,\pi\,R^2\,L}{360\,\pi\,R^2\,L} = \frac{\alpha}{360}$$

```
#ifndef COMPUTE_MDOT_H
#define COMPUTE_MDOT_H

#include "volFields.H"
#include "fvCFD.H"          // <-- this brings in volScalarField and mesh access
#include "constants.H"      // <-- SolverConstants used inside .C file

void computeMdot
(
    const volScalarField& T,
    const volScalarField& p,
    const volScalarField& rho_s,
    volScalarField& mdot
);

#endif
```

Figure 10: computeMdot.H

```
#include "computeMdot.H"
#include "constants.H"
#include "fvCFD.H"

void computeMdot
(
    const volScalarField& T,
    const volScalarField& p,
    const volScalarField& rho_s,
    volScalarField& mdot
)
{
    forAll(rho_s, c)
    {
        scalar Tc = Foam::max(273.0, Foam::min(1000.0, T[c]));
        scalar pc = p[c];
        scalar rhoSc = Foam::max(SolverConstants::rhoEmp.value(), Foam::min(rho_s[c], SolverConstants::rhoSat.value()));

        scalar PeqAbs = SolverConstants::Pref.value() * Foam::exp(SolverConstants::Aabs.value() - SolverConstants::Babs.
value() / Tc);
        scalar PeqDes = SolverConstants::Pref.value() * Foam::exp(SolverConstants::Ades.value() - SolverConstants::Bdes.
value() / Tc);

        if (pc > PeqAbs)
        {
            scalar lnTerm = Foam::log(Foam::max(pc / PeqAbs, SMALL));
            mdot[c] = SolverConstants::Ca.value()
                    * Foam::exp(-SolverConstants::Ea.value() / (SolverConstants::R.value() * Tc))
                    * lnTerm * (SolverConstants::rhoSat.value() - rhoSc);
        }
        else if (pc < PeqDes)
        {
            scalar frac = (pc - PeqDes) / Foam::max(PeqDes, SMALL);
            mdot[c] = SolverConstants::Cd.value()
                    * Foam::exp(-SolverConstants::Ed.value() / (SolverConstants::R.value() * Tc))
                    * frac * (rhoSc - SolverConstants::rhoEmp.value());
        }
        mdot[c] = SolverConstants::volRatio.value() * mdot[c];

    }
}
```

Figure 11: computeMdot.C

4.5 computeHeatSink.H (modelled volumetric heat sink)

In Darzi et al. (Darzi 2016) and Jemni et al. (Jemni 1995), heat is removed from the tank during the absorption reaction by means of a PCM (molten salt) material or a cooling fluid, as it is shown in Figure 12.

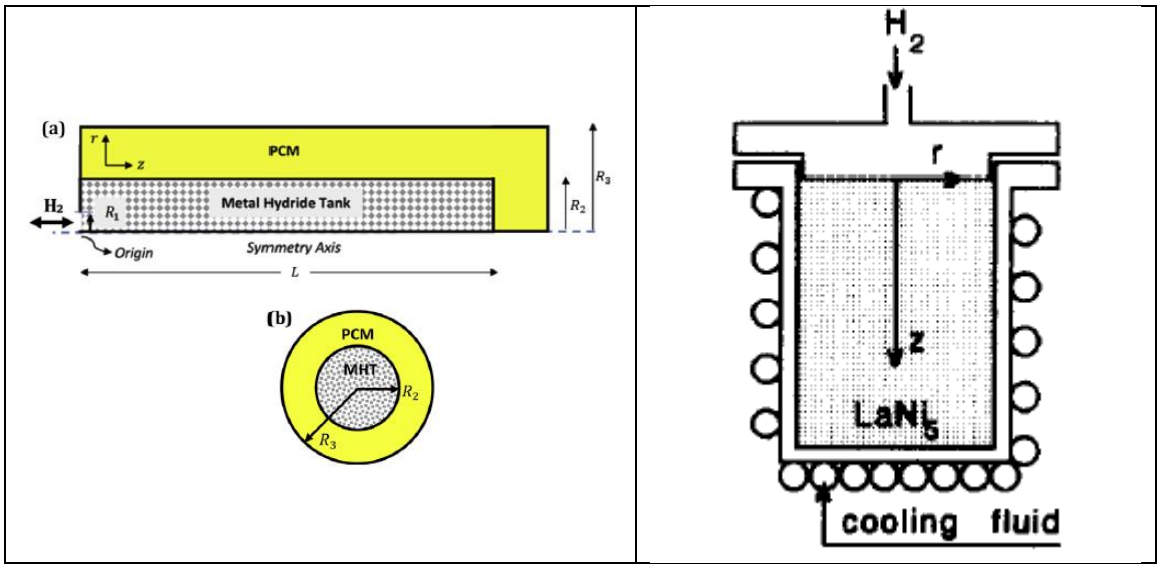

Figure 12: the cooling systems used in (Darzi 2016) (left) and (Jemni 1995) (right)

Simulation of these systems require the resolution of a conjugate heat transfer problem, which cannot be done using the chosen OpenFoam solver (rhoPimpleFoam). In addition to that, the PCM material has its own dynamic, and its associated energy equation is:

Equation 17: PCM energy equation

$$\frac{\partial(\rho_p H_p)}{\partial t} = \kappa_p \nabla^2 T + \kappa_p \nabla T$$

This is a typo. Not used further.

Here, H_p is the PCM's total enthalpy, sum of the sensible enthalpy (h) and the PCM's latent heat content (ΔH_p):

$$H_p = h + \Delta H_p$$

And ρ_p is the PCM density. andf \kappa_p the thermal diffusion coefficient.

Because the sensible enthalpy (h) is generally much smaller than the latent heat (ΔH_p) (see (Darzi 2016)), the term $\frac{\partial(\rho_p H_p)}{\partial t}$ in Equation 17 reduces to $\frac{\partial(\rho_p \Delta H_p)}{\partial t}$. The latent heat ($\Delta H_p$) varies between zero (solid PCM) and the value of the latent heat of fusion (L_p) (liquid PCM), and the PCM melt fraction (β) can be computed as (Darzi 2016):

Equation 18: PCM melted fraction

$$\beta = \begin{cases} \frac{\Delta H_p}{L_p} = 0 \;\; T < T_{sp} \\ \frac{\Delta H_p}{L_p} = 1 \;\; T > T_{lp} \\ \frac{\Delta H_p}{L_p} = \frac{T - T_{sp}}{T_{lp} - T_{sp}} \;\; T_{sp} < T < T_{lp} \end{cases}$$

Shouid T be denoted as T_{PCM} for increased transparancy?

The PCM latent heat (ΔH_p) can be written in terms of β and L_p as:

$$\Delta H_p = \beta\, L_p$$

And Equation 17 becomes:

$$\frac{\partial(\rho_p \beta L_p)}{\partial t} = \kappa_p\, \nabla^2 T + \kappa_p\, \nabla T$$

same typo as before.

Or, considering that L_p is a constant:

Equation 19: PCM's melted fraction dynamic

$$L_p \frac{\partial(\rho_p \beta)}{\partial t} = \kappa_p\, \nabla^2 T + \kappa_p\, \nabla T$$

same typo as before

Equation 19 describes how the PCM melted fraction varies in time as a function of its thermodynamic properties (L_p) and (κ_p) and its temperature (T).

The dynamic of β is shown in Figure 13, taken from (Darzi 2016).

\beta depends on spatial coordinates as well. Does graph a volume fraction?

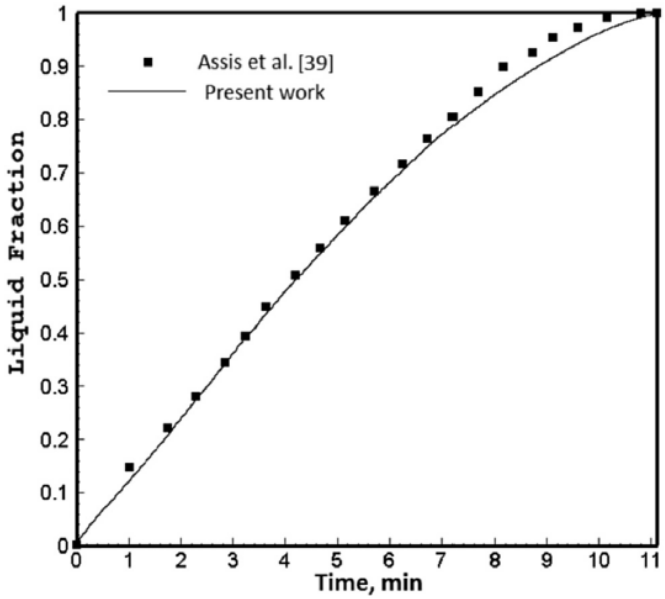

Figure 13: PCM liquid fraction (β) vs time (Darzi 2016)

It is worth noting that, by recalling the definition of β from Equation 18, Equation 19 can also be interpreted as governing the spatial and temporal evolution of the PCM temperature, provided its thermodynamic properties, as well as initial and boundary conditions, are specified. In the context of a full conjugate heat transfer problem (e.g., using chtMultiRegionFoam), these equations would be solved within the solid region.

Because the present work does not include the solid PCM region in the computational domain, nor does it solve the full conjugate heat transfer problem between the tank and the PCM region, we must find a different approach to include the effects of the PCM region (and of its dynamics) on the system.

A first possible way to reach this objective would be to govern the system's heat management via the temperature boundary conditions, but this would require a fine tuning of the heat flux value, and, more importantly, would not be possible to include the dynamics shown in Figure 13, unless a "coded" boundary condition type is used. Is a boundary condition that tabulates Figure 13 intended here?

We propose a different approach based on the assumption that the effect of the PCM zone can be modelled by a volumetric lumped term (Q_{loss}) of the form:

Equation 20: The heat sink model equation[4]

$$\frac{Q_{loss}}{t} = \frac{C\beta\,\rho_p\,C_{p,p}\,(T - T_{amb})}{t}$$

Division by zero causes difficulties later.
Is a coded boundary condition a better option?

Where β is not defined from Equation 18 (as we are not solving Equation 17 or Equation 19), but is reconstructed from Figure 13. The constant (C) in Equation 20 is totally arbitrary and used to tune the model against the reference results of Jemni et al., as reported by Darzi et al. in (Darzi 2016) (see figure 2 of the paper).

The function $\beta = \beta(t)$ of Figure 13 has been digitized and interpolated with the polynomial:

Equation 21: polynomial reconstruction of β

$$\beta = -2x10^{-09}\,t^3 + 4x10^{-07}\,t^2 + 0.002122351\;\; t$$

Constrained by:

Equation 22: β function constraints

$$\beta = max(0.0, min(beta, 1.0))$$

Should 1.0 be replaced by \Delta H_p / L_p?

The proposed heat sink model is implemented in computeHeatSink.H and computeHeatSink.C like in Figure 14 and Figure 15.

To enhance numerical stability, (Q_{loss}) is implemented in linearized form:

Equation 23: linearized form of the modelled energy sink term

Energy source in case of H2-gas release.

$$\frac{Q_{loss}}{t} = \frac{C\,\beta\,\rho_p\,C_{p,p}\,(T - T_{amb}) = C\,\beta\,\rho_p\,C_{p,p}\,T - C\,\beta\,\rho_p\,C_{p,p}\,T_{amb}}{t} = a_l\,T + b_l$$

Where a_l and b_l write:

$$a_l = \frac{C\,\beta\,\rho_p\,C_{p,p}}{t}$$

$$b_l = \frac{C\,\beta\,\rho_p\,C_{p,p}\,T_{amb}}{t} = a_l\,T_{amb}$$

[4] Note that the required heat sink units are $\frac{kg}{m\,s^3}$, or, in OpenFoam notation, [1 0 -3 0 0 0 0], so the numerator of Equation 20, whose units are $\frac{kg}{m\,s^2}$ needs to be divided by time.

Why global scope here?

```
#ifndef COMPUTEHEATSINK_H
#define COMPUTEHEATSINK_H

#include "fvCFD.H"
#include "constants.H"

// Global scope declaration (no namespace here!)
void computeHeatSink(const Foam::Time& runTime, Foam::scalar& al, Foam::scalar& bl);

#endif
```

Figure 14: computeHeatSink.H

```
#include "computeHeatSink.H"

void computeHeatSink(const Foam::Time& runTime, Foam::scalar& al, Foam::scalar& bl)
{

    Foam::scalar t = runTime.value();
    Foam::scalar beta = 0.0;
    Foam::scalar a = -0.000000002; Foam::scalar b = 0.0000004; Foam::scalar c = 0.002122351;

    if (t < Foam::SMALL)
    {
        FatalErrorInFunction
            << "Simulation time too small or zero, division by t would be unsafe." << Foam::nl << Foam::exit(Foam::FatalError);
    }

    if (Foam::SolverConstants::useFunction.value() == 1)
      {
    // Time-based function
        if (t <= 660.0)
          {
              beta = a * Foam::pow(t, 3) + b * Foam::pow(t, 2) +  c * t;
              beta = Foam::max(0.0, Foam::min(beta, 1.0));
          }
        else
          {
              beta = 1.0;
          }
      }
    else if (Foam::SolverConstants::useFunction.value() == 0)
      {
        beta = 1.0;
      }

    const Foam::scalar C     = Foam::SolverConstants::C.value();
    const Foam::scalar rho_p = Foam::SolverConstants::rho_p.value();
    const Foam::scalar Cp_p  = Foam::SolverConstants::Cp_p.value();
    const Foam::scalar T_amb = Foam::SolverConstants::T_amb.value();

    al = C * beta * rho_p * Cp_p / t;
    bl = al * T_amb;

}
```

Figure 15: computeHeatSink.C

4.6 rhoEqn.H (hydrogen mass conservation)

The gas phase (ρ_g) continuity is solved using Equation 2 modifying the original rhoEqn.H to include the source term $\frac{\dot{m}}{\epsilon}$, where $\dot{m}$ is computed in computeMdot.H. The resulting rhoEqn.H is provided in Figure 16.

```
\*---------------------------------------------------------------------------*/

#include "fvCFD.H"  // This includes all the necessary fvm/fvc headers
#include "constants.H"

// === Reaction rate model based on Darzi et al. (2016) ===
// Refer to Eq. (1), (3), and (4) from the paper.

volScalarField T = thermo.T();
volScalarField p = thermo.p();

volScalarField dRhoSdt
(
    IOobject
    (
        "dRhoSdt",
        runTime.timeName(),
        mesh,
        IOobject::NO_READ,
        IOobject::NO_WRITE
    ),
    mesh,
    dimensionedScalar("zero", dimMass/dimVolume/dimTime, 0.0)
);

volScalarField mdot
(
    IOobject
    (
        "mdot",
        runTime.timeName(),
        mesh,
        IOobject::NO_READ,
        IOobject::NO_WRITE
    ),
    mesh,
    dimensionedScalar("zero", dimMass/dimVolume/dimTime, 0.0)
);

// 🔁 Compute mdot and source term
computeMdot(T, p, rho_s, mdot);

dRhoSdt = mdot/SolverConstants::porosity.value();
//-------------------------------------------------

    fvScalarMatrix rhoEqn
    (
        fvm::ddt(rho)
      + fvc::div(phi)
      ==
        dRhoSdt
      + fvOptions(rho)
    );

    fvOptions.constrain(rhoEqn);

    rhoEqn.solve();

    fvOptions.correct(rho);

// ************************************************************************* //
```

Figure 16: rhoEqn.H

4.7 rho_sEqn.H (hydride mass conservation)

The hydride phase (ρ_s) continuity is solved using Equation 1, and the resulting rho_sEqn.H is provided in Figure 17.

```
#include "fvCFD.H"  // This includes all the necessary fvm/fvc headers
#include "constants.H"

// === Reaction rate model based on Darzi et al. (2016) ===
// Refer to Eq. (1), (3), and (4) from the paper.

volScalarField dRhoSdt
(
    IOobject
    (
        "dRhoSdt",
        runTime.timeName(),
        mesh,
        IOobject::NO_READ,
        IOobject::NO_WRITE
    ),
    mesh,
    dimensionedScalar("zero", dimMass/dimVolume/dimTime, 0.0)
);

// 🔄 Compute mdot and source term
computeMdot(T, p, rho_s, mdot);

dRhoSdt = mdot/(1 - SolverConstants::porosity.value());

fvScalarMatrix rho_sEqn
(
    fvm::ddt(rho_s)
    ==
    dRhoSdt
);

rho_sEqn.relax();

fvOptions.constrain(rho_sEqn);

rho_sEqn.solve();

fvOptions.correct(rho_s);
```

Figure 17; rho_sEqn.H

4.8 TEqn.H (Energy Equation)

TEqn.H implements the merged energy equation (Equation 16) and includes calls to computeMdot.H and computeHeatSink.H to retrieve the terms needed by the energy balance equation, as shown in Figure 18.

```
// --- Energy equation
fvScalarMatrix TEqn
(
    fvm::ddt(rhoCpEff, T)
  - fvm::laplacian(kEff, T)
  + convectiveTerm
  + a*T
  ==
  -b
);

TEqn.relax();
fvOptions.constrain(TEqn);
TEqn.solve();
T.correctBoundaryConditions();
fvOptions.correct(T);
```

Figure 18: the energy equation solved by TEqn.H

Terms rhoCpEff and kEff are computed using Equation 14 and Equation 15 in TEqn.H as shown in Figure 19.

```
// --- Effective rho*Cp (volumetric heat capacity)
tmp<volScalarField> tRhoCpEff =
    SolverConstants::porosity.value() * rho * SolverConstants::Cp_g
  + (scalar(1.0) - SolverConstants::porosity.value()) * rho_s * SolverConstants::Cp_s;

volScalarField rhoCpEff
(
    IOobject("rhoCpEff", runTime.timeName(), mesh, IOobject::NO_READ, IOobject::NO_WRITE),
    tRhoCpEff()
);

volScalarField kEff
(
    IOobject("kEff", runTime.timeName(), mesh, IOobject::NO_READ, IOobject::NO_WRITE),
    mesh,
    dimensionedScalar
    (
        "kEff",
        SolverConstants::k_g.dimensions(),
        SolverConstants::porosity.value() * SolverConstants::k_g.value() + (scalar(1.0) - SolverConstants::porosity.valu
e()) * SolverConstants::k_s.value()
    )
);
```

Figure 19: calculation of $(\rho\, C_p)_{eff}$ and κ_{eff} in TEqn.H with Equation 14 and Equation 15

Use thermodynamics of a mixture?

It is worth noting that in Figure 18 the convective term $\rho_g C_{p,g} \vec{v} \cdot \nabla T$ of Equation 16 cannot be included via a standard OpenFoam call like fvm::div(phi, T), as the thermodynamic which is implicitly included in (phi), does not match the model's requirement. For this reason, Figure 18 shows the usage of the term **convectiveTerm** at the left-hand-side of the equation, where convectiveTerm is computed in TEqn.h, as in Figure 20.

```
volVectorField EpsRhoCpU
(
    IOobject("EpsRhoCpU", runTime.timeName(), mesh, IOobject::NO_READ, IOobject::NO_WRITE),
    SolverConstants::porosity * rho * SolverConstants::Cp_g * U
);
volScalarField magEpsRhoCpU("magEpsRhoCpU", mag(EpsRhoCpU));

volVectorField gradT = fvc::grad(T);

volScalarField magGradT("magGradT", mag(gradT));

volScalarField convectiveTerm
(
    IOobject("convectiveTerm", runTime.timeName(), mesh, IOobject::NO_READ, IOobject::NO_WRITE),
    EpsRhoCpU & gradT
);
```

Figure 20: calculation of the convectiveTerm ($\rho_g C_{p,g} \vec{v} \cdot \nabla T$) in TEqn.H

It is also worth noting that the reaction energy source term of Equation 16:

$$Q_{reaction} = -\dot{m}\left[(1-\varepsilon)\frac{\Delta H_{abs/des}}{M_{H_2}} - T\left(C_{p,g} - C_{p,s}\right)\right]$$

is linearized for enhancing the solver stability as:

$$Q_{reaction} = \dot{m}T\left(C_{p,g} - C_{p,s}\right) - \dot{m}\,(1-\varepsilon)\frac{\Delta H_{abs/des}}{M_{H_2}} = a_{reaction}\,T + b_{reaction}$$

and finally, the two coefficients (a) and (b) are defined:

$$a = a_{reaction} + a_{loss}$$

$b = b_{reaction} + b_{loss}$

Where a_{loss} and b_{loss} come from the previously discussed computeHeatSink.H.

5 Validation

5.1 Absorption case

5.1.1 Mesh

To reduce computational cost, the axial symmetry of the flow is exploited by modeling only a 5-degree wedge of the actual device, as shown in Figure 21. Only a single cell is used in the tangential direction. As in (Darzi 2016), the tank's inner radius and length measure 0.04 m, and 0.8 m, respectively. A short pipe (0.1 m long) with radius 0.01 m is used for a more convenient imposition of the inlet conditions.

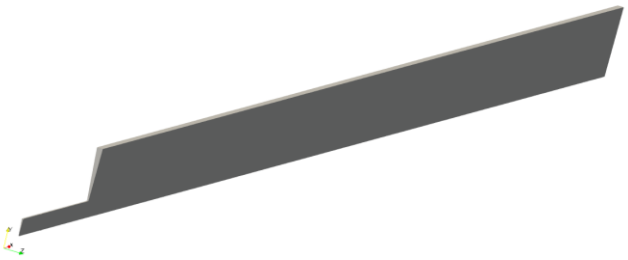

Figure 21: The five-degree wedge of the device used in the axis symmetric simulations

The blockMeshDict file used to generate the mesh is provided in APPENDIX I.

A mesh sensitivity analysis has been performed on three grids, obtained by recursively doubling the number of cells in the x and y directions of the previous refinement level. The computational domain is divided in three blocks, covering the feeding pipe, the tank's region that extends from the axis to the feeding pipe's radius, and the remaining tank's volume, respectively. Table 2 summarizes each block's discretization used in the axial and radial directions, and the respective total number of computational cells. Only one cell is used in the circumferential direction. Figure 22 shows the three grids on the x–y plane.

Table 2: Computational mesh size

Block	Coarse	Medium	Fine
1	5 x 3	10 x 6	20 x 12
2	20 x 3	40 x 6	80 x 12
3	20 x 7	40 x 14	80 x 28
Total number of cells	215	860	3440

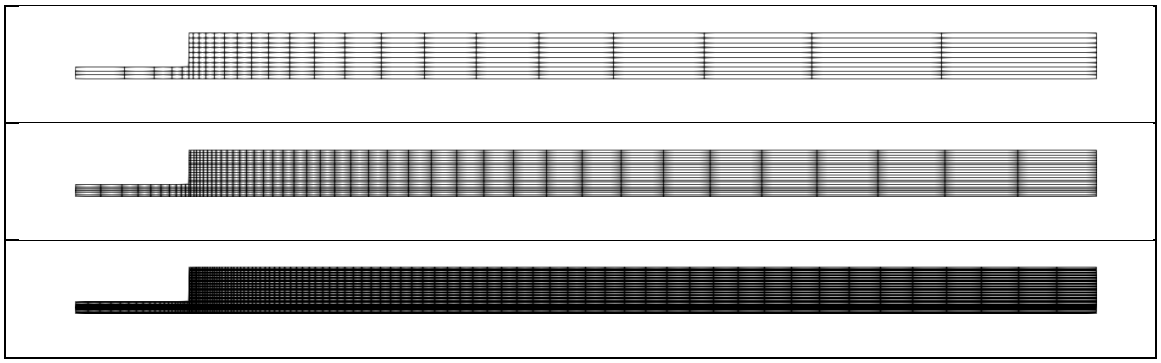

Figure 22: Wedge (5 degrees) computational mesh; from top to bottom: coarse, medium, and fine grid

5.1.2 Boundary and Initial Conditions

The boundary condition files used throughout the simulations presented here are reported in APPENDIX II. Initial conditions are reported in Table 3.

Table 3: Initial conditions

U [m/s]	P [Pa]	T [K]	ρ_s [kg/m^3]
(0 0 0)	10^6	290.0	7164.0

5.1.3 Numerical settings (fvSchemes and fvSolution)

Numerical settings are given in APPENDIX III.

5.1.4 Simulation ((Jemni 1995), as in (Darzi 2016))

The implemented model was tested using the same absorption case employed by Darzi et al. (Darzi 2016) to validate their own model. Although the original reference data come from the work of Jemni et al. (Jemni 1995), their presentation in that study makes the data difficult to extract and use directly. Therefore, for comparison purposes, Darzi's processed version of the data was used instead of the original dataset.

In a first round of simulations, the same coefficients presented in (Darzi 2016) were used, and the arbitrary constant (C) that appears in the proposed heat sink model (Equation 23) was set to one. In a second set of simulations, the Van't Hoff equation coefficients[5] and the constant (C) has been tuned to match the reference data.

[5] We are fully aware that this choice is in general questionable, as those coefficients reflect the physical behavior of the specific material of which the metal bed is made of. A more formally consistent solution would be pre-multiplying the original Van't Hoff equation coefficients by some arbitrary quantities, but the final effects would remain the same. Because we consider the present work as a first step towards the development of a solver with full conjugate heat transfer capabilities for this class of applications, we have decided to disregard in the present context the actual physical meaning of the Van't Hoff equation coefficients.

Because the model used to simulate the absorption phase acts as a volumetric lumped term, and because of the nature of the boundary conditions used at the tank's walls ($\frac{\partial T}{\partial n} = 0$), the solution is expected to be independent of the computational grid.

5.1.4.1 Results with original coefficients from (Darzi 2016)

Table 4 reports the model's coefficients as used in the work of Darzi's et al. (Darzi 2016). Table 5 summarizes the computational time on the three grids used for the mesh independence study. Figure 23 shows the comparison of the simulated temperature and H/M values against the reference data of Jemni et al. (Jemni 1995), as reported in (Darzi 2016). The figure shows that:

1. Results are grid independent.
2. Predicted temperature and H/M value do not fit the reference data, so the model needs to be calibrated.

The first finding is expected. Given the applied boundary conditions, the model effectively behaves as a one-dimensional system in which energy rapidly redistributes throughout the domain. As a result, spatial effects become negligible and the outcome is largely independent of the mesh resolution. Additionally, heat is removed from the system through a volumetric sink term that functions as a lumped parameter, entirely decoupled from the computational grid.

The second finding is also unsurprising. The proposed heat sink model, in its current form, cannot fully replicate the detailed conjugate heat transfer approach used by Darzi et al. in their study and therefore requires careful, case-specific tuning of its coefficients.

Table 4: original coefficients as in (Darzi 2016)

Parameter	Description	Value	Units
Ca	Absorption rate coefficient	59.187	s^{-1}
Cd	Desorption rate coefficient	9.57	s^{-1}
Cp_g	Heat capacity of hydrogen	14890	J/kg.K
Cp_s	Heat capacity of metal	419	J/kg.K
Ea	Activation energy in absorption	21179.6	J/mol
Ed	Activation energy in desorption	16473	J/mol
K	Permeability	109	m^2
K_g	Hydrogen thermal conductivity	0.815	W/m.K
k_{MH}	Metal bed thermal conductivity	2	W/m.K
M_g	Hydrogen molecular weight	2.016	kg/kmol
ϵ	Porosity	0.4, 0.5, 0.6	
ρ_{emp}	Empty bed density	7164	kg/m^3
ρ_{sat}	Saturated bed density	7259	kg/m^3
μ_g	Hydrogen viscosity	8.4E-06	kg/m.s
A_{abs}	Van't Hoff equation coefficient	10.70	-
B_{abs}	Van't Hoff equation coefficient	3704.6	-
A_{des}	Van't Hoff equation coefficient	10.57	-
B_{des}	Van't Hoff equation coefficient	3704.6	-
C	Heat Sink Coefficient (Equation 23)	1.0	-

useFunction	Switch between β functions	1	-

Table 5: Grid independence study, original coefficients, simulation time[6]

Refinement level	Coarse	Medium	Fine
Execution time [s]	17.7	241.2	4329.2

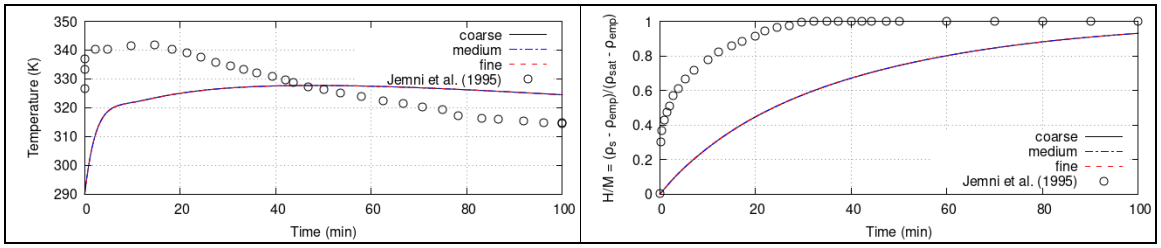

Figure 23: grid independence study, absorption case, original coefficients, temperature (left) and M/H ratio (right) at probe point vs time

5.1.4.2 Results with calibrated coefficients

With the caveat of footnote 5, it was decided to adjust A_{des} and B_{des}, and the heat sink model coefficient (C), in order to calibrate the developed model against the reference dataset. Table 6 shows the calibrated coefficients' values. Having already demonstrated the method's independence of the computational grid, the simulation has been carried out on the coarse mesh only. Simulation time was 16.18 seconds. Figure 24 compares temperature and H/M results obtained using the calibrated set of coefficients when the coefficient β is computed using Equation 21 or set to a constant value (β = 1) . Results obtained with the reconstructed value of β show excellent agreement with the reference data.

Table 6: calibrated coefficients

A_{abs}	Van't Hoff equation coefficient	123.00	-
B_{abs}	Van't Hoff equation coefficient	42500	-
C	Heat Sink Coefficient (Equation 23)	0.500	-

Are these quantities dimensionless?

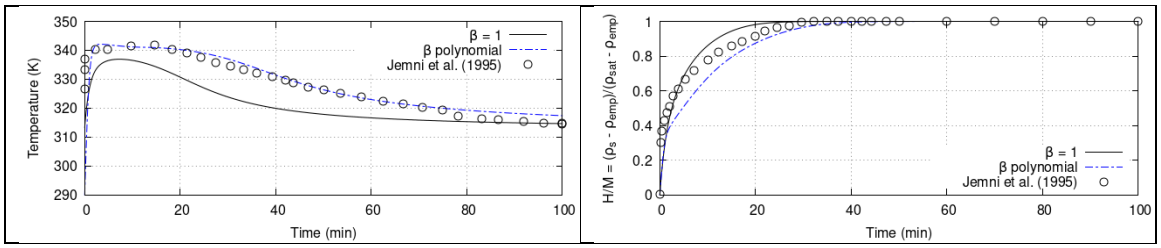

Figure 24: Calibrated coefficients, absorption case, β = 1 vs β computed using Equation 21, temperature[7] *(left) and M/H ratio (right) at probe point vs time*

[6] On a single Intel® Core ™ i7-9750H CPU @ 2.60 GHz

[7] Note that Fig. 2 from (Darzi 2016) reports Jemni's data points only until time = 100 min, whereas the original data from Jemni et al. (Jemni 1995) extends to 120 min.

5.2 Desorption simulation ((C.A. Chung 2009))

In his paper, Chung (C.A. Chung 2009) simulate the desorption process inside the device shown in Figure 25.

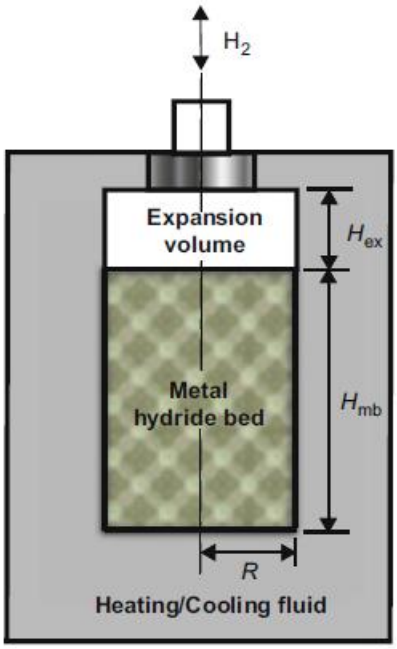

Figure 25: the device used by Chung to simulate the absorption and desorption phases (figure taken from (C.A. Chung 2009)).

Chung considers the presence of an expansion volume (H_{ex} = 20 mm) on the top of the hydride bed tank (R = 25 mm, H_{mb} = 60 mm). The system exchanges energy (being cooled or heated) during the absorption and desorption phases, with a co-axial vessel filled with heating/cooling fluid. In our study, we neglect the presence of the expansion volume. The mesh parameters are the same used for the absorption case.

In his work Chung account for the energy exchange between the metal bed and the cooling/heating fluid through a boundary condition of the type:

Equation 24

$$-\kappa\frac{\partial T}{\partial n} = h\left(T - T_{cf}\right)$$

Where the constant effective heat transfer coefficient h is 1652 W/m^2.K.

Because in this case the value of h is known, we have decided to account for the thermal exchange via boundary condition, rather than using the previously described volumetric model. To this aim, we have implemented the mixed boundary condition described in the following section 5.2.1.

Is this BC only relevant in the desorption case?

5.2.1 myWallHeatFluxTemperatureFvPatchScalarField

The file myWallHeatFluxTemperatureFvPatchScalarField.C defines a custom mixed-type boundary condition for temperature in OpenFOAM to represent a convective wall heat transfer model with additional safety features like flux limiting and adaptive switching between fixed-value and fixed-gradient modes depending on the heat generation rate inside the domain.

The core functionality of the newly developed boundary condition enforces the Newton's law of heat transfer at a wall, but with some added complexity:

Equation 25

$$q'' = -h_{\text{eff}}(T_p - T_a)$$

Where q'' is the heat flux at the wall (W/m²), h_{eff} the effective heat transfer coefficient (W/m²·K), T_p the patch (wall) temperature, and T_a the ambient (external) temperature.

This is applied through the mixed boundary condition formulation in OpenFOAM:

Equation 26

$$\phi = \alpha\, T_{\text{ref}} + (1 - \alpha)\left(-\kappa \frac{\partial T}{\partial n}\right)$$

Where ϕ is the total boundary condition value, α the `valueFraction` (between 0 and 1), T_{ref} is the refValue (used when $\alpha = 1$), κ is the effective thermal conductivity, and $\frac{\partial T}{\partial n}$ the normal temperature gradient (used when $\alpha = 0$).

The effective heat transfer coefficient in Equation 25 combines the convective coefficient h and conduction through a wall of thickness δ and conductivity κ_{wall}:

Equation 27

$$h_{\text{eff}} = \left(\frac{1}{h} + \frac{\delta}{\kappa_{\text{wall}}}\right)^{-1}$$

This allows modeling of a wall material between the fluid and the external temperature.

To prevent unphysical heat fluxes, a flux limiter is introduced. The allowed maximum flux is:

Equation 28

$$q''_{\text{max,adaptive}} = \min\left(q''_{\text{max}}, \frac{Q_{\text{reaction}}}{A_{\text{patch}} + \varepsilon}\right)$$

Where Q_{reaction} is the volume-averaged reaction heat generation $[W]$, A_{patch} the total wall area, and q''_{max} a user-defined hard flux cap (default: 20000 W/m²). The quantity ε in Equation 28 indicates a small number, not the material's porosity. Then the heat flux is clamped:

Equation 29

$$q'' = \text{clamp}\left(-h_{\text{eff}}\left(T_p - T_a\right), -q''_{\text{max,adaptive}}, +q''_{\text{max,adaptive}}\right)$$

5.2.1.1 An Adaptive Mixed Fraction Formulation of α

The boundary condition switches between fixed temperature ($\alpha = 1$) and fixed gradient ($\alpha = 0$) behavior based on the total heat generation Q_{reaction}. A smoothing/relaxation algorithm is used, so that if $Q_{\text{reaction}} > Q_{\text{limit}}$ → full Neumann ($\alpha = 0$) else → smooth switching:

Equation 30

$$\alpha_{\text{new}} = \min\left(1.0, \frac{\tau_{\text{relax}}}{\tau_{\text{relax}} + |Q_{\text{reaction}}|}\right)$$

And then, to suppresses abrupt changes and ensures stability in the heat feedback loop:

Equation 31

$$\alpha = 0.9 \cdot \alpha_{\text{old}} + 0.1 \cdot \alpha_{\text{new}}$$

5.2.1.2 Usage in Dictionary

```
T
{
    type          myWallHeatFluxTemperature;
    Ta            290;
    h             1000;
    thickness     0.005;
    kappaWall     205;
    maxFlux       15000;
    tauRelax      0.01;
    volQ_reaction_limit  100;
    value         uniform 300;
}
```

5.2.2 Setup and Results

The boundary condition files for p, T, U, and ρ_s are provided in APPENDIX IV. Numerical parameters (fvSolution and fvSchemes files) are the same used for the absorption case and

reported in APPENDIX III. For numerical stabilization, the inlet pressure was linearly ramped from 0.143 MPa to the final value of 1.0 MPa over the first 100 seconds of the simulated time, following a similar approach to that used by Chung in (C.A. Chung 2009).

Table 7: Grid independence study, desorption case, simulation time[8]

Refinement level	Coarse	Medium	Fine
Execution time [s]	21.8	126.9	488.9

What happens here? Hydrogen does not leave domain. Mass in domain is conserved.

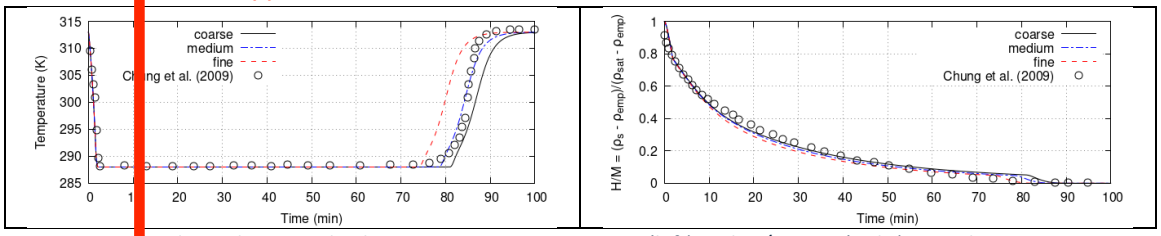

Figure 26: grid independence study, desorption case, temperature (left) and M/H ratio (right) at probe point vs time

Comparison of Table 5 and Table 7 shows that, as the grid is refined, the volumetric lumped model used for simulating the absorption process is more time-demanding than using the temperature boundary condition to account for heat exchange. This behavior can be explained with the following observations:

- The volumetric sink term is inversely proportional to the simulation time (t): in the absorption case, the PCM-effect is lumped into the volumetric heat-sink of Equation 23, that we rewrite here for convenience:

$$\frac{Q_{loss}}{t} = \frac{C\,\beta\,\rho_p\,C_{p,p}\,(T - T_{amb}) = C\,\beta\,\rho_p\,C_{p,p}\,T - C\,\beta\,\rho_p\,C_{p,p}\,T_{amb}}{t}$$
$$= a_l\,T + b_l$$

  Where a_l and b_l write:

$$a_l = \frac{C\,\beta\,\rho_p\,C_{p,p}}{t}$$

$$b_l = \frac{C\,\beta\,\rho_p\,C_{p,p}\,T_{amb}}{t} = a_l\,T_{amb}$$

  At very early times $t \to 0$, a_l becomes extremely large, making Q_{loss} a very strong, rapidly changing source in the temperature equation. This effect is particularly severe when β is given the constant value of one. Moreover, as the grid is refined, the CFL condition results in a smaller allowed time-step, so that the value of Q_{loss} stays high on a greater number of iterations.

- The explicit (or semi-explicit) treatment of source terms demands small Δt:

[8] On a single Intel® Core ™ i7-9750H CPU @ 2.60 GHz

OpenFOAM's PIMPLE loop solves

Equation 32

hugh?

$$\rho C_p \frac{\partial T}{\partial t} + \cdots = \nabla! \cdot (k \nabla T) + \underbrace{a_1 T + b_1}_{\text{volumetric source}} + \text{reaction source}$$

and because that volumetric source is implemented outside the implicit matrix via computeHeatSink.H, stability requires:

$$\Delta t \lesssim \frac{1}{a_1} \quad (\text{so that } \rho C_p \, \Delta t \, a_1 < 1)$$

When a_1 is huge at small t, the solver drastically reduces Δt to keep $\rho C_p \, \Delta t \, a_1$ below unity, otherwise the temperature update would "overshoot" and diverge.

The boundary-condition approach avoids the stiff source issue because the developed custom mixed convective boundary condition

$$-k \frac{\partial T}{\partial n} = h \, (T - T_{\text{cf}})$$

is handled implicitly in the finite-volume wall-flux matrix. It does not inject a large, time-singular source into the cell interior, so the energy equation remains far less stiff. The only time-step constraint there is the usual Courant (and possibly diffusion) limit, allowing Δ to stay much larger.

On the other hand, it should be kept in mind that results obtained using the volumetric lumped model are grid independent, and on the coarser grid level, the lumped model shows a smaller execution time (16.18 s vs 21.7).

Exam of Figure 26 shows a strong grid dependence of the solution. This should not be surprising, as the wall boundary condition is of Robin's type, so that its gradient-dependent part is sensitive to the grid's resolution in the wall-normal direction.

6 Conclusions

This work presented a modified compressible OpenFOAM solver aimed at simulating hydrogen absorption and desorption in metal hydride beds. The principal milestones that the work has reached are (i) the extension of rhoPimpleFoam to include hydride-phase mass transport and the reaction kinetics (computeMdot), (ii) a merged energy formulation that accounts for an effective heat capacity and thermal conductivity of the two-phase medium, (iii) a simple volumetric model to mimic PCM heat removal during absorption (computeHeatSink), and (iv) a bespoke mixed (Robin-type) thermal boundary condition for wall heat exchange during desorption. The solver changes, case setup and implementation details are documented and example validation cases against literature data were provided.

The validation work produced two main findings. For the absorption case, the lumped volumetric heat-sink approach yields a grid-independent global response and can be tuned to reproduce published time-series with modest computational cost. For the desorption case, the wall heat exchange implemented via a Robin-type boundary condition reproduces the expected behavior but shows a marked sensitivity to near-wall grid spacing, a predictable outcome since the discrete wall flux depends on the wall-normal gradient approximation. The study also highlighted numerical stiffness introduced by the chosen form of the volumetric heat sink at very early times.

While the current implementation is functional and useful for rapid parametric studies, several limitations must be addressed before the solver can be relied upon for design-level predictions. The lumped PCM model includes an inverse-time factor that produces an artificial early-time stiffness; the wall Robin condition requires either careful grid design or a higher-order/implicit discretization to remove mesh dependence. In addition, the calibration needed to match reference data (notably adjustments to van't Hoff coefficients) indicates that further, physics-based validation against raw experimental measurements is necessary to avoid parameter overfitting.

Looking forward, the most valuable next steps are: (1) replacing the lumped PCM sink with a physically consistent enthalpy-based conjugate heat-transfer model (e.g., via chtMultiRegionFoam or a coupled region approach), (2) implementing an implicit/ghost-node treatment of the Robin boundary condition to remove grid sensitivity, (3) possibly moving stiff source terms into the implicit matrix or solving the PCM energy ODE implicitly to remove early-time numerical stiffness, and (4) performing systematic verification (manufactured solution) and validation (raw experimental datasets), together with uncertainty quantification and GCI-based convergence assessment. These improvements will substantially increase the solver's predictive capability and make it suitable for engineering design and optimization studies.

(1/2): how is the Robin boundary condition currently discretized in space? What is meant by an implicit treatment?

(2/2): unclear how an implicit solution algorithm would be implemented.

References

Afzal, M., & Sharma, P. 2021. "Design and computational analysis of a metal hydride hydrogen storage system with hexagonal honeycomb based heat transfer enhancements-part A." *International Journal of Hydrogen Energy* 46 (24): 13116-13130.

Afzal, M., & Sharma, P. 2018. "Design of a large-scale metal hydride based hydrogen storage reactor: simulation and heat transfer optimization." *International Journal of Hydrogen Energy* 43 (29): 13356-13372.

Askri, F., Salah, M. B., Jemni, A., & Nasrallah, S. B. 2009. "Optimization of hydrogen storage in metal-hydride tanks." *International journal of hydrogen Energy* 34 (2): 897-905.

C.A. Chung, Ci-Jyun Ho. 2009. "Thermal–fluid behavior of the hydriding and dehydriding processes in a metal hydride hydrogen storage canister." *International Journal of Hydrogen Energy* 34: 4351-4364.

Chandra, S., Sharma, P., Muthukumar, P., & Tatiparti, S. S. V. 2020. "Modeling and numerical simulation of a 5 kg LaNi5-based hydrogen storage reactor with internal conical fins." *International journal of hydrogen energy* 45 (15): 8794-8809.

Chippar, P., Lewis, S. D., Rai, S., & Sircar, A. 2018. "Numerical investigation of hydrogen absorption in a stackable metal hydride reactor utilizing compartmentalization." *International journal of hydrogen energ* 43 (16): 8007-8017.

Darzi, A. A. R., Afrouzi, H. H., Moshfegh, A., & Farhadi, M. 2016. "Absorption and desorption of hydrogen in long metal hydride tank equipped with phase change material jacket." *International journal of hydrogen energy* 41 (22): 9595-9610. doi:10.1016/j.ijhydene.2016.04.051.

Freni, A., Cipiti, F., & Cacciola, G. 2009. "Finite element-based simulation of a metal hydride-based hydrogen storage tank." *International Journal of Hydrogen Energy* 34 (20): 8574-8582.

Jemni, A., & Nasrallah, S. B. 1995. "Study of two-dimensional heat and mass transfer during absorption in a metal-hydrogen reactor." *International Journal of Hydrogen Energy* 20 (1): 43-52.

K. Holz, K. Boie, H. Christoph. 1953. *Laws of Turbulent Pipe Flow (Beginning at Small Reynolds Numbers).* NACA Technical Memorandum No. 1346, NACA. NACA TM-1346 (PDF).

Karmakar, A., Mallik, A., Gupta, N., & Sharma, P. 2021. "Studies on 10kg alloy mass metal hydride based reactor for hydrogen storage." *International Journal of Hydrogen Energy* 46 (7): 5495-5506.

Kukkapalli, V. K., Kim, S., & Thomas, S. A. 2023. "Thermal Management Techniques in Metal Hydrides for Hydrogen Storage Applications: A Review." *Energies* 16 (8): 3444. doi:https://doi.org/10.3390/en16083444.

Larpruenrudee, P., Bennett, N. S., Gu, Y., Fitch, R., & Islam, M. S. 2022. "Design optimization of a magnesium-based metal hydride hydrogen energy storage system." *Scientific Reports* 12 (1): 1-16. doi:https://doi.org/10.1038/s41598-022-17120-3.

Lin, X., Xie, W., Zhu, Q., Yang, H., & Li, Q. 2021. "Rational optimization of metal hydride tank with LaNi4. 25Al0. 75 as hydrogen storage medium." *Chemical Engineering Journal* 421.

Liu, W., Tupe, J. A., & Aguey-Zinsou, F. 2025. "Metal Hydride Storage Systems: Approaches to Improve Their Performances." *Particle & Particle Systems Characterization* 42 (3). doi:https://doi.org/10.1002/ppsc.202400163.

Mellouli, S., Askri, F., Dhaou, H., Jemni, A., & Nasrallah, S. B. 2009. "Numerical study of heat exchanger effects on charge/discharge times of metal–hydrogen storage vessel." *International Journal of Hydrogen Energy* 34 (7): 3005-3017.

Nasrallah, S. B., & Jemni, A. 1997. "Heat and mass transfer models in metal-hydrogen reactor." *International Journal of Hydrogen Energy* 22 (1): 67-76.

Nikuradze, J. 1932. *Gesetzmäßigkeiten der turbulenten Strömung in glatten Rohren (Laws of Turbulent Flow in Smooth Pipes).* Heft 356 (Issue 356), Forschungsarbeiten auf dem Gebiete des Ingenieurwesens (Research Works in the Field of Engineering), Berlin: VDI-Verlag.

Nikuradze, J. 1933. *Strömungsgesetze in rauhen Rohren (Laws of Flow in Rough Pipes).* Heft 361 (Issue 361), Forschungsarbeiten auf dem Gebiete des Ingenieurwesens (Research Works in the Field of Engineering), Berlin: VDI-Verlag.

Rabienataj Darzi AA, et al. 2016. "Absorption and desorption of hydrogen in long metal hydride tank equipped with phase change material jacket." *International Journal of Hydrogen Energy* 1-16. doi:http://dx.doi.org/10.1016/j.ijhydene.2016.04.051.

Raju, M., & Kumar, S. 2011. "System simulation modeling and heat transfer in sodium alanate based hydrogen storage systems." *International Journal of Hydrogen Energy* 36 (2): 1578-1591.

Reynolds, O. 1883. "An Experimental Investigation of the Circumstances Which Determine Whether the Motion of Water Shall Be Direct or Sinuous, and of the Law of Resistance in Parallel Channels." *Philosophical Transactions of the Royal Society of London* 174: 935-982. doi:10.1098/rstl.1883.0029 .

Spiegel, Colleen. 2019. *Metal Hydrides.* February 26. https://www.fuelcellstore.com/blog-section/component-information/metal-hydrides-blog#:~:text=The%20use%20of%20metal%20hydrides,05%20W%2FmK.

Tong, L., Xiao, J., Bénard, P., & Chahine, R. 2019. "Thermal management of metal hydride hydrogen storage reservoir using phase change materials." *International Journal of Hydrogen Energy* 44 (38): 21055-21066.

Ye, Y., Lu, J., Ding, J., Wang, W., & Yan, J. 2020. "Numerical simulation on the storage performance of a phase change materials based metal hydride hydrogen storage tank." *Applied Energy* 278: 115682.

APPENDIX I

blockMeshDict

Here is a brief explanation of how the grading works in blockMesh. We want to cluster the grid cells over the bottom and upper walls of a channel, with the same, but oppose, grading ratio. In this example[9] the block is divided uniformly in the x and z -directions and split into three grading sections in the y-direction described by three triples: ((0.2 0.3 4) (0.6 0.4 1) (0.2 0.3 0.25)). Each of the grading sections is described by a triple consisting of the fraction of the block, the fraction of the divisions and the grading ratio (size of first division/size of last division). Both the fraction of the block and the fraction of the divisions are normalized automatically so they can be specified scaled in anyway, e.g. as percentages, and they need not sum to 1 or 100.

```
blocks
  (
    hex (0 1 2 3 4 5 6 7) (20 60 20)
    simpleGrading
    (
      1
      ((2 3 4) (6 4 1) (2 3 0.25))
      1
    )
);
```

In the example above the total size “H” along y-direction is split in three parts. The first part starts from the bottom wall and covers 20% of H, contains 30% of the total number of cells along y, Ny, and has a grading ratio of 4. The second part covers 60% of H, contains 40% of the total number of cells along y, Ny, uniformly spaced (grading ratio of 1). Finally, the third and last part covers 20% of H, contains 30% of the total number of cells along y, Ny, and has a grading ratio of 0.25, (which is equal to ¼, the reciprocal of the grading ratio used on the bottom wall).

In our case, we do not intend to have an equally spaced distribution in the mid-height of the channel, so we can use the following two triples to impose the desired grading in the y-direction: ((0.5 0.5 32) (0.5 0.5 0.03125)). This statement says that the two divisions cover 50% of H each, and 50% of the total number of cells in the y-direction, with grading ratio of 32 on the bottom wall and 1/32 = 0.03125 on the top wall. The blockMesh dictionary shown below here includes the

[9] See https://openfoamwiki.net/index.php/BlockMesh

grading section, even though the grids generated for the tests presented in this work use a uniform grading.

```
/*--------------------------------*- C++ -*----------------------------------*\
| =========                 |                                                 |
| \\      /  F ield         | OpenFOAM: The Open Source CFD Toolbox           |
|  \\    /   O peration     | Version:  v2306                                 |
|   \\  /    A nd           | Website:  www.openfoam.com                      |
|    \\/     M anipulation  |                                                 |
\*---------------------------------------------------------------------------*/
FoamFile
{
    version     2.0;
    format      ascii;
    class       dictionary;
    object      blockMeshDict;
}

scale 0.01;  // Convert cm to m

// Geometry parameters
xmin  -10;  // Feeding pipe start (cm)
R1     1;   // Feeding pipe radius (cm)
R2     4.0; // Cylinder radius (cm)
L     80;   // Cylinder length (cm)

// Precomputed values for 2.5° wedge angle
cs    0.9990482216;  // cos(2.5°)
sn    0.04361938736; // sin(2.5°)
vertices
(
    // Front vertices (wedge angle = -2.5°)
    (-10 0 0)                          // 0
    (-10 0.9990482216 -0.04361938736)        // 1
    (0    0.9990482216 -0.04361938736)       // 2
    (0    3.9961928863 -0.17447754946)       // 3
    (80   3.9961928863 -0.17447754946)       // 4
    (80   0 0)                         // 5
    (0    0 0)                         // 6
    (80   0.9990482216 -0.04361938736)       // 7

    // Back vertices (wedge angle = +2.5°)
    (-10 0 0)                          // 8
    (-10 0.9990482216 0.04361938736)         // 9
    (0    0.9990482216 0.04361938736)        // 10
    (0    3.9961928863 0.17447754946)        // 11
    (80   3.9961928863 0.17447754946)        // 12
    (80   0 0)                         // 13
```

```
   (0    0 0)                          // 14
   (80   0.9990482216 0.04361938736)        // 15
);
blocks
(
   // Feeding pipe (5, 10, 20, 40 axial, 3, 6, 12, 24 radial for the coarse, medium, fine , super fine
mesh levels)
   hex (0 6 2 1 8 14 10 9)
   (5   3 1)
   simpleGrading
   (
      0.125   // Axial clustering toward inlet
      1 //0.03125 // Radial clustering toward wall
      1
   )

   // Cylinder inner (20, 40, 80, 160 axial, 3, 6, 12, 24 radial, for the coarse, medium, fine, and super
fine mesh levels)
   hex (6 5 7 2 14 13 15 10)
   (20   3 1 )
   simpleGrading
   (
      32      // Axial uniform
      1 //0.03125 // Radial clustering toward inner wall
      1
   )

   // Cylinder outer (20, 40, 80, 160, axial, 7, 14, 28, 56 radial, for the coarse, medium, fine, and
super fine mesh levels)
   hex (2 7 4 3 10 15 12 11)
   (20   7  1)
   simpleGrading
   (
      32                          // Axial uniform
      1 //((0.25 0.5 4) (0.25 0.5 0.25) ) //((0.25 0.5 32) (0.25 0.5 0.03125) ) // Radial clustering
      1
   )
);
edges ();

boundary
(
   wedgeFront
   {
      type wedge;
      faces
      (
         (0 1 2 6)
```

```
        (6 2 7 5)
        (2 3 4 7)
    );
  }

  wedgeBack
  {
    type wedge;
    faces
    (
      (8 9 10 14)
      (14 10 15 13)
      (10 11 12 15)
    );
  }

  inlet
  {
    type patch;
    faces
    (
      (0 8 9 1)   // Feeding pipe inlet
    );
  }
  walls
  {
    type wall;
    faces
    (
      (1 9 10 2)   // Feeding pipe wall
      (3 11 12 4)  // Cylinder outer wall
      (5 13 15 7)  // Outlet inner (x = L)
      (7 15 12 4)  // Outlet outer (x = L)
      (2 3 11 10)  // Left face (x = 0)
    );
  }

  axis
  {
    type empty;
    faces
    (
      (0 6 14 8)  // Axis face at feeding pipe start
      (6 5 13 14)
    );
  }
);
mergePatchPairs ();
```

APPENDIX II

```
/*--------------------------------*- C++ -*----------------------------------*\
| =========                 |                                                 |
| \\      /  F ield         | OpenFOAM: The Open Source CFD Toolbox           |
|  \\    /   O peration     | Version:  v2306                                 |
|   \\  /    A nd           | Website:  www.openfoam.com                      |
|    \\/     M anipulation  |                                                 |
\*---------------------------------------------------------------------------*/
FoamFile
{
    version     2.0;
    format      ascii;
    class       volVectorField;
    object      U;
}
// * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * //
dimensions      [0 1 -1 0 0 0 0];
internalField   uniform (0 0 0);
boundaryField
{
    inlet
    {
        type            fixedValue;
        value           $internalField;
    }
    wedgeFront
    {
        type            wedge;
    }
    wedgeBack
    {
        type            wedge;
    }
    walls
    {
        type            noSlip;
    }
    axis
    {
        type            empty;
    }
}
// ************************************************************************* //
```

```
/*--------------------------------*- C++ -*----------------------------------*\
| =========                 |                                                 |
| \\      /  F ield         | OpenFOAM: The Open Source CFD Toolbox           |
|  \\    /   O peration     | Version:  v2306                                 |
|   \\  /    A nd           | Website:  www.openfoam.com                      |
|    \\/     M anipulation  |                                                 |
\*---------------------------------------------------------------------------*/
FoamFile
{
   version    2.0;
   format     ascii;
   class      volScalarField;
   object     p;
}
// * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * //
dimensions     [1 -1 -2 0 0 0 0];
internalField   uniform 1.0e06;
boundaryField
{
   inlet
   {
      type          fixedValue;
      value         uniform 1.0e06;
   }
   wedgeFront
   {
      type          wedge;
   }
   wedgeBack
   {
      type          wedge;
   }
   walls
   {
      type          zeroGradient;
   }
   axis
   {
      type          empty;
   }
}
// ************************************************************************* //
```

```
/*--------------------------------*- C++ -*----------------------------------*\
| =========                 |                                                 |
| \\      /  F ield         | OpenFOAM: The Open Source CFD Toolbox           |
|  \\    /   O peration     | Version:  v2306                                 |
|   \\  /    A nd           | Website:  www.openfoam.com                      |
|    \\/     M anipulation  |                                                 |
\*---------------------------------------------------------------------------*/
FoamFile
{
   version    2.0;
   format     ascii;
   class      volScalarField;
   object     T;
}
// * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * //
dimensions     [0 0 0 1 0 0 0];
internalField   uniform 290.0;
boundaryField
{
   inlet
   {
      type          fixedValue;
      value         uniform 290;
   }
   wedgeFront
   {
      type          wedge;
   }
   wedgeBack
   {
      type          wedge;
   }
   walls
   {
   type zeroGradient;
   }
   axis
   {
      type          empty;
   }
}
// ************************************************************************* //
```

```
/*--------------------------------*- C++ -*----------------------------------*\
| =========                 |                                                 |
| \\      /  F ield         | OpenFOAM: The Open Source CFD Toolbox           |
|  \\    /   O peration     | Version:  v2306                                 |
|   \\  /    A nd           | Website:  www.openfoam.com                      |
|    \\/     M anipulation  |                                                 |
\*---------------------------------------------------------------------------*/
FoamFile
{
   version     2.0;
   format      ascii;
   class       volScalarField;
   object      rho_s;
}
// * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * //
dimensions      [1 -3 0 0 0 0 0];  // [kg/m³]
internalField   uniform 7164.0;  // start at rhoEmp
boundaryField
{
   inlet
   {
      type zeroGradient;
   }
   wedgeFront
   {
      type       wedge;
   }
   wedgeBack
   {
      type       wedge;
   }
   walls
   {
      type zeroGradient;
   }
   axis
   {
      type          empty;
   }
}
// ************************************************************************* //
```

APPENDIX III

```
/*--------------------------------*- C++ -*----------------------------------*\
| =========                 |                                                 |
| \\      /  F ield         | OpenFOAM: The Open Source CFD Toolbox           |
|  \\    /   O peration     | Version:  v2306                                 |
|   \\  /    A nd           | Website:  www.openfoam.com                      |
|    \\/     M anipulation  |                                                 |
\*---------------------------------------------------------------------------*/
FoamFile
{
    version     2.0;
    format      ascii;
    class       dictionary;
    object      fvSchemes;
}
// * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * //
ddtSchemes
{
    default         Euler;
}
gradSchemes
{
    default         Gauss linear;

    limited         cellLimited Gauss linear 1;
    grad(U)         $limited;
    grad(k)         $limited;
    grad(omega)     $limited;
}
divSchemes
{
    default         none;
    div(phi,U)      Gauss linearUpwind limited;
    energy          Gauss linearUpwind limited;
    div(phi,he)     $energy;
    div(phi,e)      $energy;
    div(phi,K)      $energy;
    div(phi,Ekp)    $energy;
    turbulence      Gauss linearUpwind limited;
    div(phi,k)      $turbulence;
    div(phi,omega)  $turbulence;
    div(phi,epsilon) $turbulence;
    div(phi,rho_s)   Gauss limitedLinear 1;
    div(phi_energy,T) $energy;
    div(phi,T) $energy;
    div(phiv,p)     Gauss linearUpwind limited; \\linear;
```

```
    div((phi|interpolate(rho)),p) Gauss linearUpwind limited; \\linear;
    div(((rho*nuEff)*dev2(T(grad(U))))    Gauss linear;
}
laplacianSchemes
{
    default        Gauss linear corrected;
    laplacian(kEff,T) Gauss linear corrected;
}
interpolationSchemes
{
    default        linear;
}
snGradSchemes
{
    default        corrected;
}
wallDist
{
    method         meshWave;
}
// ************************************************************************* //
```

```
/*--------------------------------*- C++ -*----------------------------------*\
| =========                 |                                                 |
| \\      /  F ield         | OpenFOAM: The Open Source CFD Toolbox           |
|  \\    /   O peration     | Version:  v2306                                 |
|   \\  /    A nd           | Website:  www.openfoam.com                      |
|    \\/     M anipulation  |                                                 |
\*---------------------------------------------------------------------------*/
FoamFile
{
   version     2.0;
   format      ascii;
   class       dictionary;
   object      fvSolution;
}
// * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * //
solvers
{
   p
   {
      solver         FPCG;//GAMG;(FPCG GAMG PBiCGStab PCG PPCG PPCR smoothSolver)
      preconditioner
      {
         preconditioner  GAMG;
         tolerance       1e-05;
         relTol          0;
         smoother                        DICGaussSeidel;//(DIC    DICGaussSeidel    FDIC    GaussSeidel
nonBlockingGaussSeidel symGaussSeidel)       }
      tolerance       1e-6;
      relTol          0.01;
   }
   pFinal
   {
      $p;
      relTol          0;
   }
   "(U|k|omega|epsilon|e|he)"
   {
      solver         PBiCGStab;
      preconditioner  DILU;
      tolerance       1e-6;
      relTol          0.1;
      minIter         1;
   }
   "(U|k|omega|epsilon|e|he)Final"
   {
      $U;
      relTol          0;
      minIter         1;
```

```
    }
    "(rho|rho_s|T)"
    {
        solver          PBiCGStab;
        preconditioner  DIC;
        tolerance       1e-6;
        relTol          0.1;
        minIter         1;
    }
    "(rho|rho_s|T)Final"
    {
        $rho_s;
        relTol          0;
        minIter         1;
    }
}
PIMPLE
{
    nCorrectors             1;
    nNonOrthogonalCcorrectors 1;
    nOuterCcorrectors       1;
}
relaxationFactors
{
    fields
    {
        p     0.3;
        rho   1;//0.3;
    }
    equations
    {
        U       1;
        e       1;//0.7;
        T       1;
        rho_s   1;//0.3;
    }
}
// ************************************************************************* //
```

APPENDIX IV

```
/*--------------------------------*- C++ -*----------------------------------*\
| =========                 |                                                 |
| \\      /  F ield         | OpenFOAM: The Open Source CFD Toolbox           |
|  \\    /   O peration     | Version:  v2306                                 |
|   \\  /    A nd           | Website:  www.openfoam.com                      |
|    \\/     M anipulation  |                                                 |
\*---------------------------------------------------------------------------*/
FoamFile
{
    version     2.0;
    format      ascii;
    class       volScalarField;
    object      T;
}
// * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * //
dimensions      [0 0 0 1 0 0 0];
internalField   uniform 313.0;
boundaryField
{
    inlet
    {
        type          fixedValue;
        value         uniform 313;
    }
    wedgeFront
    {
        type          wedge;
    }
    wedgeBack
    {
        type          wedge;
    }
    walls
    {
      type          myWallHeatFluxTemperature;
      Ta            313.0;
      h             1652.0;
      kappa         kEff;
      maxFlux       1500;
      tauRelax      1;                  // control flux–value transition:
                                        // tauRelax to control how fast the BC "gives up"
                                        // flux control when Q_reaction → 0.
                                        // Lower values = more aggressive switch to fixedValue
      volQ_reaction_limit 0.0925;  // Lower limit below which alpha is computed and not set to zero
      thickness     0.003;              // 3 mm aluminum wall
```

```
    kappaWall      237;              // W/m·K (typical for aluminum)
    value          $internalField;
  }
  axis
  {
    type           empty;
  }
}
// ************************************************************************* //
```

```
/*--------------------------------*- C++ -*----------------------------------*\
| =========                 |                                                 |
| \\      /  F ield         | OpenFOAM: The Open Source CFD Toolbox           |
|  \\    /   O peration     | Version:  v2306                                 |
|   \\  /    A nd           | Website:  www.openfoam.com                      |
|    \\/     M anipulation  |                                                 |
\*---------------------------------------------------------------------------*/
FoamFile
{
   version     2.0;
   format      ascii;
   class       volScalarField;
   object      p;
}
// * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * //
dimensions      [1 -1 -2 0 0 0 0];
internalField   uniform 0.28e06;
boundaryField
{
   inlet
   {
   type codedFixedValue;
   value uniform 0.28e6;  // initial value
   name timeInterpolatedPressure;
   code
   #{
      const scalar t = this->db().time().value(); // current time
      scalar p;
      if (t <= 100.0)
      {
         // Linear interpolation from 280000 to 100000
         p = 280000 - (180000.0 * t / 100.0);
      }
      else
      {
         p = 100000;
      }
      operator==(p);
   #};
   }
   wedgeFront
   {
      type          wedge;
   }
   wedgeBack
   {
      type          wedge;
   }
```

```
    walls
    {
        type            zeroGradient;
    }
    axis
    {
        type            empty;
    }
}
// ************************************************************************* //
```

```
/*--------------------------------*- C++ -*----------------------------------*\
| =========                 |                                                 |
| \\      /  F ield         | OpenFOAM: The Open Source CFD Toolbox           |
|  \\    /   O peration     | Version:  v2306                                 |
|   \\  /    A nd           | Website:  www.openfoam.com                      |
|    \\/     M anipulation  |                                                 |
\*---------------------------------------------------------------------------*/
FoamFile
{
    version     2.0;
    format      ascii;
    class       volVectorField;
    object      U;
}
// * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * //
dimensions      [0 1 -1 0 0 0 0];
internalField   uniform (0 0 0);
boundaryField
{
    inlet
    {
        type            fixedValue;
        value           $internalField;
    }
    wedgeFront
    {
        type            wedge;
    }
    wedgeBack
    {
        type            wedge;
    }
    walls
    {
        type            noSlip;
    }
    axis
    {
        type            empty;
    }
}
// ************************************************************************* //
```

```
/*--------------------------------*- C++ -*----------------------------------*\
| =========                 |                                                 |
| \\      /  F ield         | OpenFOAM: The Open Source CFD Toolbox           |
|  \\    /   O peration     | Version:  v2306                                 |
|   \\  /    A nd           | Website:  www.openfoam.com                      |
|    \\/     M anipulation  |                                                 |
\*---------------------------------------------------------------------------*/
FoamFile
{
    version     2.0;
    format      ascii;
    class       volScalarField;
    object      rho_s;
}
// * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * //
dimensions      [1 -3 0 0 0 0 0];  // [kg/m³]
internalField   uniform 7259.0;  // rhoEmp (7164) for absorption; rhoSat (7259) for desorption

boundaryField
{
    inlet
    {
        type zeroGradient;
    }
    walls
    {
        type zeroGradient;
    }
    wedgeFront
    {
        type        wedge;
    }
    wedgeBack
    {
        type        wedge;
    }
    axis
    {
        type           empty;
    }
}
```