Cite as: Kandel P.: Combining Density-Based Compressible Solver `rhoCentralFoam` with Reacting Flow Solver `reactingFoam`. In Proceedings of CFD with OpenSource Software, 2024, Edited by Nilsson. H.,
http://dx.doi.org/10.17196/OS_CFD#YEAR_2024

CFD with OpenSource software

A course at Chalmers University of Technology
Taught by Håkan Nilsson

Combining Density-Based Compressible Solver `rhoCentralFoam` with Reacting Flow Solver `reactingFoam`

Developed for OpenFOAM-v2406

Author:
Pablo Kandel
Technische Universität Berlin
pablo.kandel@tu-berlin.de

Peer reviewed by:
Mahdi Lavari
Abdulla Ghani
Saeed Salehi

Licensed under CC-BY-NC-SA, `https://creativecommons.org/licenses/`

Disclaimer: This is a student project work, done as part of a course where OpenFOAM and some other OpenSource software are introduced to the students. Any reader should be aware that it might not be free of errors. Still, it might be useful for someone who would like to learn some details similar to the ones presented in the report and in the accompanying files. The material has gone through a review process. The role of the reviewer is to go through the tutorial and make sure that it works, that it is possible to follow, and to some extent correct the writing. The reviewer has no responsibility for the contents.

January 19, 2025

Learning outcomes

The main requirements of a tutorial in the course is that it should teach the four points: How to use it, The theory of it, How it is implemented, and How to modify it. Therefore the list of learning outcomes is organized with those headers.

The reader will learn:

How to use it:

- The main differences between pressure-based solvers and density-based solvers will be discussed, their respective range of application, the different space and time discretization techniques, and the important variables of interest.
- The `forwardStep` test case will be used for comparison.

The theory of it:

- The reacting flow governing equations for combustion.
- The theory of a pressure-based reacting flow solver `reactingFoam`.
- The theory of a density-based compressible solver `rhoCentralFoam`.

How it is implemented:

- There is one density-based solvers available in the OpenFOAM-ESI branch, `rhoCentralFoam`. It uses the central scheme discretization technique of Kurganov et al. [1] to discretize the advective fluxes.

How to modify it:

- The transport equations for reacting flow will be added in `rhoCentralFoam`. The procedure to add the reacting transport equations within a density-based framework will be described step by step.
- The fluxes discretization in density-based solvers is different than in the standard PISO/SIMPLE solvers and special care will be taken for this aspect.
- The implementation will be validated on the one-dimensional reacting shock tube case.

Prerequisites

- How to run standard document tutorials like `forwardStep` and `counterFlowFlame2D` tutorials.
- Fundamentals of Computational Methods for Fluid Dynamics, Book by J.H.Ferziger and M.Peric [2] and The Finite Volume Method in Computational Fluid Dynamics, Book by F.Moukalled, L.Mangani and M.Darwish [3].
- Basic knowledge of numerical combustion theory: Poinsot, T. (2005). Theoretical and Numerical Combustion. RT Edwards. [4].
- Have a basic awareness of the structure of OpenFOAM and how it utilises applications and libraries.
- How to customize a solver and do top-level application programming.

Contents

Nomenclature

Acronyms

CFD	Computational Fluid Dynamics
CFL	Courant–Friedrichs–Lewy
KNP	Kurganov, Noelle and Petrova
KT	Kurganov and Tadmor
TVD	Total Variation Diminishing

English symbols

$\tilde{R}$	Specific gas constant	$\mathrm{J}/(\mathrm{kg}\cdot\mathrm{K})$
C_p	Specific heat capacity at constant pressure	$\mathrm{J}/(\mathrm{kg}\cdot\mathrm{K})$
C_v	Specific heat capacity at constant volume	$\mathrm{J}/(\mathrm{kg}\cdot\mathrm{K})$
D_k	Mass diffusivity of species k	m^2/s
e_s	Sensible internal energy	J
h_s	Sensible enthalpy	J
J	Mass diffusive flux	kg/s
m	Mass of gas mixture	kg
m_k	Mass of species k	kg
Ma	Mach number	
p	Pressure	Pa
q_i	i-component of heat flux	$\mathrm{J}/(\mathrm{m}^2\mathrm{s})$
R	Ideal gas constant	$\mathrm{J}/(\mathrm{mol}\cdot\mathrm{K})$
T	Temperature of gas mixture	K
u_i	i-component of velocity vector	m/s
W	Mean molecular weight of gas mixture	kg/mol
x_i	i-component of space vector	m
Y_k	Mass fraction of species k	

Greek symbols

$\dot{\omega}_k$	Reaction rate of species k	$\mathrm{kg}/(\mathrm{m}^3\cdot\mathrm{s})$
$\dot{\omega}_T$	Heat release due to combustion	$\mathrm{J}/(\mathrm{m}^3\cdot\mathrm{s})$
γ	Heat capacity ratio	
λ	Heat conductivity of gas mixture	$\mathrm{W}/(\mathrm{K}\cdot\mathrm{m})$
μ	Fluid dynamic viscosity	$\mathrm{Pa}\cdot\mathrm{s}$
ν	Fluid kinematic viscosity	m^2/s
ϕ	Volumetric flux due to fluid flow	m^3/s
Ψ	Tensor field of any rank	
ψ	Compressibility	$\mathrm{s}^2/\mathrm{m}^2$
ρ	Fluid density	kg/m^3
τ	Viscous stress tensor	Pa
a	Volumetric fluxes relative to the local speed of sound	m^3/s

Subscripts

f	Value interpolated on the cell face
$f\pm$	Direction of interpolation
i	i-direction in the Cartesian frame
k	Species k

Chapter 1

Introduction

Compressible flows occur when the fluid density changes significantly due to pressure or temperature variations. The Mach number Ma, is the ratio of the flow velocity U to the speed of sound c in the medium given by

$$\mathrm{Ma} = \frac{U}{c}. \tag{1.1}$$

It is used to determine whether compressible effects should be considered. A value of $\mathrm{Ma} > 0.3$ is often taken as a threshold at which compressibility becomes important.

These flows are often encountered in high-speed aerodynamics, propulsion systems, and gas dynamics. In such flows, the behavior of the fluid can be governed by the compressibility of the fluid, which affects the speed of sound, shock waves, and other phenomena like expansion fans and rarefaction waves. For analyzing compressible flows, different numerical methods can be employed, which can broadly be categorized into pressure-based and density-based solvers.

Pressure-based solvers such as SIMPLE [5] (Semi-Implicit Method for Pressure Linked Equations) and PISO [6] (Pressure-Implicit with Splitting of Operators) algorithms, focus on solving the pressure-velocity coupling by iteratively updating the pressure and velocity fields. They use a pressure correction Poisson equation derived from the continuity and momentum equations. These methods are known for their stability and robustness across a wide range of Courant–Friedrichs–Lewy (CFL) numbers, making them highly effective for solving steady-state problems and transient flows with large time steps. The implicit time-stepping schemes used in these algorithms, enable stable simulations and flexible CFL condition. This stability makes pressure-based solvers highly attractive for simulations where accurate resolution of small-scale time-dependent features is not critical. However, the implicit time-stepping scheme, while enabling large time steps, can introduce numerical diffusion, which tends to smear sharp gradients in transient or convection-dominated flows. Furthermore, the use of upwind numerical schemes for the advective terms in the governing equations to ensure stability contributes to this diffusive behavior, reducing the accuracy of the solution in problems with strong convection.

Density-based solvers, directly solve the governing equations for mass, momentum, and energy, often employing explicit time-stepping. This approach offers several advantages, particularly for compressible flows. In this context, density-based solvers are more precise due to the absence of numerical diffusion inherent in implicit time-stepping schemes. Therefore, explicit time-stepping makes these solvers well-suited for problems with sharp gradients, such as shocks and contact discontinuities. Additionally, implementing higher-order spatial and temporal schemes is generally more straightforward in the explicit framework, enhancing solution accuracy in complex flow fields. However, density-based solvers come with notable challenges. Explicit time-stepping imposes a stringent restriction on the time step size for stability, especially in highly transient flows where flow variables change significantly over very short timescales. Moreover, density-based solvers can be less robust in low-speed or highly transient flows, where convergence is harder to achieve. Special treatment for acoustic boundary conditions is also typically required, adding complexity to simulations with open

or reflective boundaries [7].

Historically, OpenFOAM has been primarily based on pressure-based algorithms. In OpenFOAM version 1.5, Greenshields et al. [8] introduced `rhoCentralFoam`, a density-based compressible flow solver. This solver implements central-upwind schemes of Kurganov et al. [1] to efficiently handle high-speed compressible flows, offering an alternative to the traditionally pressure-based solvers in OpenFOAM.

The most widely used reacting flow solver in OpenFOAM, namely `reactingFoam`, employs a pressure-based approach. However, there is currently no density-based reacting flow solver available in the official OpenFOAM release. This is particularly relevant in the context of turbulent reacting flows in combustion chambers, where there is a strong coupling between acoustic waves and other flow mechanisms, such as heat release and turbulent mixing [4]. This coupling can significantly influence the overall dynamics of the system, making density-based solvers, which naturally account for such wave phenomena, an interesting choice for accurately capturing these interactions.

In this work, we aim to bridge this gap by extending the density-based framework of `rhoCentralFoam` to incorporate reacting flow capabilities, enabling it to solve combustion problems.

Chapter 2

Theoretical Background

2.1 Reacting Flow Governing Equations

The governing equations for a reacting mixture with N species are the continuity, momentum, species and energy equations given by [4]

$$\frac{\partial \rho}{\partial t} + \frac{\partial \rho u_j}{\partial x_j} = 0, \tag{2.1}$$

$$\frac{\partial \rho u_i}{\partial t} + \frac{\partial \rho u_i u_j}{\partial x_j} = -\frac{\partial p}{\partial x_j} + \frac{\partial \tau_{ij}}{\partial x_j}, \tag{2.2}$$

$$\frac{\partial \rho Y_k}{\partial t} + \frac{\partial \rho u_i Y_k}{\partial x_i} = -\frac{\partial J_{k,i}}{\partial x_i} + \dot{\omega}_k. \tag{2.3}$$

In OpenFOAM, conservation of energy can be solved in terms of the sensible internal energy e_s or the sensible enthalpy $h_s = e_s + p/\rho$ defined by

$$\frac{\partial \rho \left(e_s + \frac{1}{2} u_j u_j\right)}{\partial t} + \frac{\partial}{\partial x_i}\left(\rho u_i \left(e_s + \frac{1}{2} u_j u_j\right)\right) = -\frac{\partial u_i p}{\partial x_i} - \frac{\partial q_i}{\partial x_i} + \dot{\omega}_T, \tag{2.4}$$

$$\frac{\partial \rho \left(h_s + \frac{1}{2} u_j u_j\right)}{\partial t} + \frac{\partial}{\partial x_i}\left(\rho u_i \left(h_s + \frac{1}{2} u_j u_j\right)\right) = \frac{\partial p}{\partial t} - \frac{\partial q_i}{\partial x_i} + \dot{\omega}_T. \tag{2.5}$$

In the above equations, ρ is the gas density and u is the fluid velocity, p is the pressure and τ is the viscous stress tensor

$$\tau_{ij} = -\frac{2}{3}\mu \frac{\partial u_k}{\partial x_k}\delta_{ij} + \mu\left(\frac{\partial u_i}{\partial x_j} + \frac{\partial u_j}{\partial x_i}\right), \tag{2.6}$$

μ is the dynamic viscosity and δ_{ij} is the Kronecker symbol: $\delta_{ij} = 1$ if $i = j$, 0 otherwise. Y_k is the mass fraction of species defined by

$$Y_k = \frac{m_k}{m}. \tag{2.7}$$

Here m_k is the mass of species k in a given volume V and m the total mass of gas in this volume. $J_{k,i}$ is the i-component of the species k diffusive flux assuming Fick's law

$$J_{k,i} = \rho D_k \frac{\partial Y_k}{\partial x_i}. \tag{2.8}$$

There are different models to compute the mass diffusivities D_k. This discussion is out of the scope of this project, and we assume Lewis and Prandtl number are equal to one as it is done in the OpenFOAM-v2406 version

$$J_{k,i} = \mu \frac{\partial Y_k}{\partial x_i}. \tag{2.9}$$

$\dot{\omega}_k$ is the reaction rate of species k, which is dependent on the chemical reactions present in the chemistry mechanism used. We don't intend to give an in-depth description of this term here. $\dot{\omega}_T$ is the heat release rate due to combustion and q_i is the i-component of the energy flux

$$q_i = -\lambda \frac{\partial T}{\partial x_i}. \tag{2.10}$$

Eq. (2.10) is the heat diffusion from Fourier's Law with λ and T the heat conductivity and temperature of the mixture.

2.2 Central Scheme Theory

In this section, we present the theory behind the central schemes introduced by Kurganov et al. [1] also referred to as Kurganov-Tadmor's scheme (KT) and Kurganov-Noelle-Petrova scheme (KNP). This was implemented in OpenFOAM by Greenshields et al. in the solver `rhoCentralFoam` [8] and also described in the work of Kraposhin et al. [9].

In Fig. 2.1, we use the finite volume method and represent in one dimension two control volumes, or cells where P indicates the center of the cell with coordinate x_i in which balances of mass, momentum and energy are considered. Point N is the center of the neighboring cell with coordinate x_{i+1}. The face area vector $\mathbf{S_f}$ is a vector normal to the face surface pointing out of the P cell, whose magnitude is that of the area of the face. In this collocated system, all dependent variables and material properties are stored at each cell centroid, P and N. We describe the discretization of a general dependent tensor field Ψ by interpolation of values Ψ_P at cell centers to values Ψ_f at cell faces.

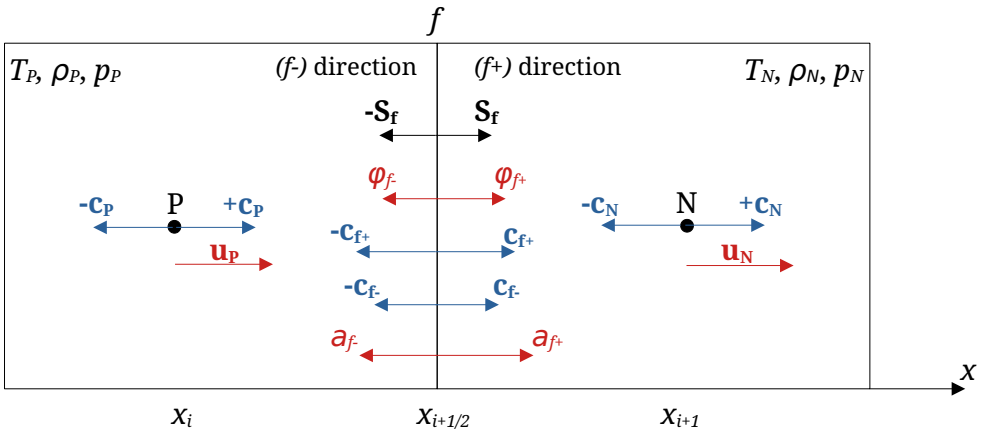

Figure 2.1: Finite volume discretization highlighting volumetric fluxes ϕ and a interpolated at the face f with respect to $f-$ and $f+$ directions and local speed of sound $\mathbf{c}$.

In particular, the advective fluxes include not only the macroscopic velocity of the medium $\mathbf{u}$, but also the acoustic disturbance that propagates at the speed of sound $\mathbf{c}$ in every direction. As part of the central discretization process and to take into account the propagation of waves in any direction, the discretization of the convective fluxes $\nabla \cdot (\mathbf{u}\rho)$, $\nabla \cdot (\mathbf{u}(\rho\mathbf{u}))$, $\nabla \cdot (\mathbf{u}(\rho E))$ and $\nabla \cdot (\mathbf{u}p)$ in Eqs. (2.1)–(2.5) is critical. They are integrated over a control volume and linearized as follows

$$\int_V \nabla \cdot (\mathbf{u}\Psi)\, \mathrm{d}V = \int_S \mathbf{u}\Psi \cdot \mathrm{d}\mathbf{S} \approx \sum_f \mathbf{S}_f \cdot \mathbf{u}_f \Psi_f = \sum_f \phi_f \Psi_f. \tag{2.11}$$

Here, $\sum_f$ denotes a summation over cell faces and $\phi_f = \mathbf{S}_f \cdot \mathbf{u}_f$ is the volumetric flux, i.e., the volume of fluid flowing through the face per second. Often in incompressible flow, $\mathbf{u_f}$ is linearly interpolated using a central differencing method and Ψ_f is interpolated according to a selected scheme that usually incorporates upwinding to some degree to stabilize. In incompressible flow, the velocity

and therefore the sign of ϕ_f determines the upwinding direction. However, in compressible flow, propagation of waves at the local speed of sound also impacts the transport of fluid properties. This means stabilization must take into account that transport can occur in any direction. Interpolation is therefore carried out in both directions for each face, in the $f+$ and $f-$ directions

$$\sum_f \phi_f \Psi_f = \sum_f \alpha \phi_{f+} \Psi_{f+} + (1-\alpha)\, \phi_{f-} \Psi_{f-} + \omega_f \left(\Psi_{f-} - \Psi_{f+}\right). \tag{2.12}$$

Here, the subscript $f\pm$ indicates the cell, which value is used for interpolation on the face f, if $f+$ then the cell P is used, with respect to which the normal is external ($\mathbf{S_f}$), if $f-$ then the cell N is used, with respect to which the normal is internal ($-\mathbf{S_f}$). To calculate α, we introduce the volumetric fluxes relative to the local speed of propagation, noting that they are both defined here as positive in their respective directions $f+$ and $f-$:

$$\begin{aligned} a_{f+} &= \max(c_{f+}|S_f| + \phi_{f+},\ \ c_{f-}|S_f| + \phi_{f-},\ \ 0), \\ a_{f-} &= -\min(-c_{f+}|S_f| + \phi_{f+},\ \ -c_{f-}|S_f| + \phi_{f-},\ \ 0). \end{aligned} \tag{2.13}$$

$c_{f\pm}$ are the speeds of sound at the face, outward and inward of the owner cell

$$c_{f\pm} = \sqrt{\gamma \tilde{R} T_{f\pm}}. \tag{2.14}$$

Here, $\tilde{R}$ is the specific gas constant and $\gamma = C_p/C_v$ is the ratio of specific heats at constant pressure and volume, C_p and C_v, respectively. The purpose of a_{f-} and a_{f+} is to combine the convective and acoustic contributions to represent the outward and inward propagation fluxes, ensuring stability in the compressible flow calculations. The weighting factor α is

$$\alpha = \begin{cases} \frac{1}{2}, & \text{for the KT method,} \\ \frac{a_{f+}}{a_{f+}+a_{f-}}, & \text{for the KNP method.} \end{cases} \tag{2.15}$$

And finally the diffusive volumetric flux

$$\omega_f = \begin{cases} \alpha \max(a_{f+}, a_{f-}), & \text{for the KT method,} \\ \alpha\,(1-\alpha)\,(a_{f+} + a_{f-}), & \text{for the KNP method.} \end{cases} \tag{2.16}$$

The method involves $f+$ and $f-$ face interpolations of a number of variables (T, ρ, p, $\mathbf{u}$) from values at neighboring cell centers. We don't intend to describe in detail the schemes used here. As explained in [8], we use symmetric Total Variation Diminishing (TVD) schemes such as Minmod [10] and van Leer [11] for linear interpolations.

Chapter 3

Compressible Solvers in OpenFOAM

3.1 Thermophysical Models

As explained by Choquet [12], thermophysical models are used to describe cases where the thermal energy, compressibility, or mass transfer is important. This is the case for compressible flow.

OpenFOAM allows thermophysical properties to be constant, or functions of temperature, pressure and composition. Thermal energy can be described in form of enthalpy or internal energy. The $p - v - T$ relation can be described with various equations of state or as an isobaric system. The `thermophysicalProperties` dictionary is the file where the user can specify the entry values for any solver that uses the `thermophysicalModels` library. In a simulation case, this file can be found in the `constant` folder. The structure of this file begins with the chosen thermophysical model, formed by a combination of each of the thermophysical properties submodels.

The following example from `combustion/reactingFoam/laminar/counterFlowFlame2D` tutorial of a `thermophysicalProperties` dictionary gives a brief explanation of how this dictionary is constituted. There are many combinations possible for the thermophysical model and we don't intend to give a full description of the thermophysical library here. The reader should refer to Choquet [12] for a more complete description.

Example of `thermophysicalProperties` file

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

thermoType
{
    type            hePsiThermo;
    mixture         reactingMixture;
    transport       sutherland;
    thermo          janaf;
    energy          sensibleEnthalpy;
    equationOfState perfectGas;
    specie          specie;
}

inertSpecie         N2;

chemistryReader     foamChemistryReader;

foamChemistryFile   "<constant>/reactions";

foamChemistryThermoFile "<constant>/thermo.compressibleGas";


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

3.2 reactingFoam

3.2.1 Introduction

reactingFoam is a pressure-based solver designed for transient simulations of compressible, reacting flows. It handles laminar and turbulent, multispecies flows with temperature and density variations. This solver is well-suited for the simulation of combustion processes and chemical reactions within fluids, accommodating a variety of reaction kinetics and species transport mechanisms. In Fig. 3.1, we show an example of a methane flame obtained with reactingFoam.

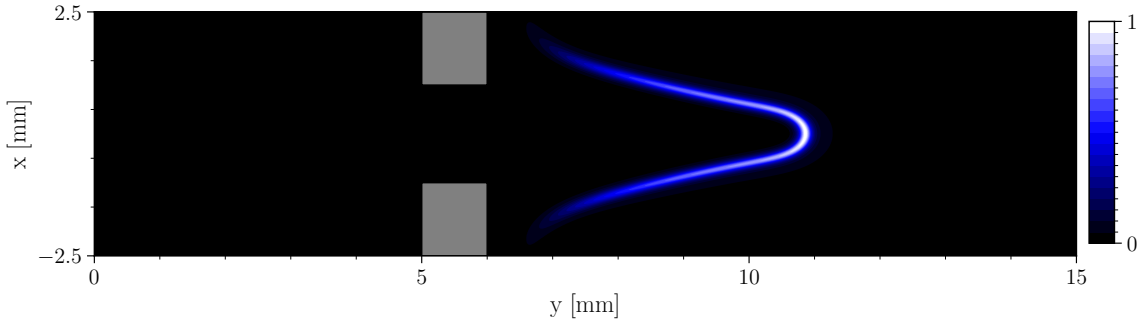

Figure 3.1: Normalized heat release of the laminar slit flame of [13].

Descriptions of reactingFoam were given in previous reports [14, 15, 16, 17, 18]. For this reason we don't intend to give an in-depth description of the reactingFoam solver but rather describe the points and parts of the code that we consider the most relevant for this study.

3.2.2 createFields.H

The createFields.H file is a header file included with all solvers, it initializes the variables and fields, as well as the turbulence and thermophysical models if needed. reactingFoam, as most of compressible flow solver, has two such files, createFields.H and createFieldRefs.H. We describe both files in this section.

$FOAM_SOLVERS/combustion/reactingFoam/createFields.H

```
Info<< "Reading thermophysical properties\n" << endl;
autoPtr<psiReactionThermo> pThermo(psiReactionThermo::New(mesh));
psiReactionThermo& thermo = pThermo();
thermo.validate(args.executable(), "h", "e");

basicSpecieMixture& composition = thermo.composition();
PtrList<volScalarField>& Y = composition.Y();
```

The thermophysical pointer pThermo is constructed using the selector New from the psiReactionThermo class. It reads the thermoType entry in the thermophysicalProperties dictionary which specifies the complete thermophysical model. A reference called thermo of the thermophysical model is created, this reference can be used to access and update all the thermophysical properties according to the chosen model. The line thermo.validate(args.executable(), "h", "e"); ensures that the thermodynamics package is consistent with energy forms supported by the application. As mentioned in Section 2.1, the user can choose to solve the energy conservation in term of sensible internal energy or sensible enthalpy, here noted as e and h.

Next, the mass fraction fields are constructed. The line basicSpecieMixture& composition = thermo.composition(); creates an object composition from the basicSpecieMixture class by calling the composition() function on the thermo reference. From the newly constructed composition object we can create the list of pointers to store the species mass fractions Y with

the following line: `PtrList<volScalarField>& Y = composition.Y();`. The total number and names of the species will depend on the files `constant/reactions` and `constant/thermo`.

`$FOAM_SOLVERS/combustion/reactingFoam/createFields.H`

```
const word inertSpecie(thermo.get<word>("inertSpecie"));
if (!composition.species().found(inertSpecie))
{
    FatalIOErrorIn(args.executable().c_str(), thermo)
        << "Inert specie " << inertSpecie << " not found in available species "
        << composition.species() << exit(FatalIOError);
}
```

The block of code above ensures that the inert species indicated in the `thermophysicalProperties` file, for example `inertSpecie N2;`, is found in the list of species of the `composition` object.

`$FOAM_SOLVERS/combustion/reactingFoam/createFieldRefs.H`

```
const volScalarField& psi = thermo.psi();
const volScalarField& T = thermo.T();
const label inertIndex(composition.species().find(inertSpecie));
```

The temperature T, and compressibility ψ, are set up as references in the `createFieldRefs.H` file as seen above. The compressibility is defined as

$$\psi = \frac{W}{RT}, \tag{3.1}$$

and can be found in

`$FOAM_SRC/thermophysicalModels/equationOfState/perfectGas/perfectGasI.H`

```
template<class Specie>
inline Foam::scalar Foam::perfectGas<Specie>::psi(scalar p, scalar T) const
{
    return 1.0/(this->R()*T);
}
```

With `this->R()` returning the specific gas constant of the mixture defined in

`$FOAM_SRC/thermophysicalModels/specie/specie/specieI.H`

```
inline Foam::scalar Foam::specie::R() const
{
    return RR/molWeight_;
}
```

`RR` being the ideal gas constant and `molWeight_` the molecular weight of the mixture noted as R and W in Eq. (3.1). A `label` variable, `inertIndex`, is created and determined based on the `termophysicalProperties` file of Section 3.1 (`N2` in this case) and on the `composition` object.

$FOAM_SOLVERS/combustion/reactingFoam/createFields.H

```
volScalarField rho
(
    IOobject
    (
        "rho",
        runTime.timeName(),
        mesh
    ),
    thermo.rho()
);

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

volScalarField& p = thermo.p();

#include "compressibleCreatePhi.H"
```

The pressure, density and velocity fields are then created (note the differences in how those fields are declared). Pressure field is created as a reference from the `p()` member function of `thermo` which returns the `volScalarField` member data `p_` of the `thermo` object. This `volScalarField p_` was created when the selector `New` from the `psiReactionThermo` class was called, this is done in the line `p_(lookupOrConstruct(mesh, "p", pOwner_))` of `basicThermo.C` file.

Density field ρ is then created using the constructor of the `volScalarField` class, which we can find in the `DimenssionedField.C` file of the templated class `dimenssionedField` which takes an `IOobject` and a `volScalarField` as argument. In this case the `volScalarField rho` is initialized with the function `thermo.rho()` field. This function has different implementations depending on the thermophysical model chosen. We consider here that the `hePsiThermo` type was chosen with the `perfectGas` option as in Section 3.1 so the density is calculated according to the ideal gas law

$$\rho = \frac{Wp}{RT} = \psi p. \tag{3.2}$$

The velocity field `U` is then created similarly as in most OpenFOAM solvers, it will check in the time folder if a `U` file is present and create a `volVectorField` from it. The `#include "compressibleCreatePhi.H"` will create the flux `phi` interpolated on the cell faces. It differs from the usual `#include "createPhi.H"` line for incompressible flow since it includes the density in its calculation in the line `linearInterpolate(rho*U) & mesh.Sf()`

$$\phi_f = \rho_f \left(\mathbf{u_f} \cdot \mathbf{S_f} \right). \tag{3.3}$$

With the subscript f indicating the value of a variable interpolated on the cell face.

$FOAM_SOLVERS/combustion/reactingFoam/createFields.H

```
pressureControl pressureControl(p, rho, pimple.dict(), false);

mesh.setFluxRequired(p.name());

Info << "Creating turbulence model.\n" << nl;
autoPtr<compressible::turbulenceModel> turbulence
(
    compressible::turbulenceModel::New
    (
        rho,
        U,
        phi,
        thermo
    )
);

Info<< "Creating reaction model\n" << endl;
autoPtr<CombustionModel<psiReactionThermo>> reaction
(
    CombustionModel<psiReactionThermo>::New(thermo, turbulence())
);

multivariateSurfaceInterpolationScheme<scalar>::fieldTable fields;

forAll(Y, i)
{
    fields.add(Y[i]);
}
fields.add(thermo.he());

volScalarField Qdot
(
    IOobject
    (
        "Qdot",
        runTime.timeName(),
        mesh,
        IOobject::READ_IF_PRESENT,
        IOobject::AUTO_WRITE
    ),
    mesh,
    dimensionedScalar(dimEnergy/dimVolume/dimTime, Zero)
);

#include "createDpdt.H"

#include "createK.H"
```

Following, control parameters regarding pressure and density are read from the `fvSolution` file through the `pressureControl(p, rho, pimple.dict(), false);` line. The next block creates a pointer called `reaction`, which points to the `CombustionModel<psiReactionThermo>` class (an instantiation of `CombustionModel` using `psiReactionThermo`). The turbulence-chemistry interaction models are described in more detail by A.Äkerblom [16]. The pointer reaction is used during the solver loop when the species equations Eq. (2.9) are solved. Next, a `fields` variable is created to store scalar fields for multivariate surface interpolation. Using a loop (`forAll(Y, i)`), all species mass fraction fields (`Y[i]`) are added to `fields`. The specific enthalpy or energy field (`thermo.he()`) is also added to `fields`, ensuring all relevant properties are prepared for interpolation during the simulation. The heat release rate field `Qdot` is then created followed by `#include "createDpdt.H"` and `#include "createK.H"` for the time derivative of the pressure field p and the kinetic energy field K with the line `volScalarField K("K", 0.5*magSqr(U));` as

$$K = \frac{1}{2}|\mathbf{u}^2|. \tag{3.4}$$

3.2.3 reactingFoam.C

In this section, we describe how the density, velocity, energy and pressure are solved in the pressure-based compressible solver `reactingFoam`. Below, we show the the main lines of the `reactingFoam.C` file including the important header files to solve the governing equation presented in Section 2.1.

`$FOAM_APP/solvers/combustion/reactingFoam/reactingFoam.C` Main lines

```
#include "rhoEqn.H"

while (pimple.loop())
{
    #include "UEqn.H"
    #include "YEqn.H"
    #include "EEqn.H"

    // --- Pressure corrector loop
    while (pimple.correct())
    {

        [...]

        else
        {
            #include "pEqn.H"
        }
    }

    [...]

}

rho = thermo.rho();

[...]
```

We will provide an explanation of the header files: `rhoEqn.H` for continuity, `UEqn.H` for momemtum, `YEqn.H` for species transport, `EEqn.H` for energy conservation and `pEqn.H` for pressure-velocity-density coupling.

3.2.3.1 rhoEqn.H

First, the density field is calculated by solving the continuity equation Eq. (2.1) by including the `rhoEqn.H` header file.

`$FOAM_SRC/finiteVolume/cfdTools/compressible/rhoEqn.H`

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

    fvOptions.constrain(rhoEqn);

    rhoEqn.solve();

    fvOptions.correct(rho);
}
```

The linear matrix system is created as `fvScalarMatrix rhoEqn`, with all the terms in the continuity equation. The system is solved and the density field computed through the line `rhoEqn.solve();`.

3.2.3.2 UEqn.H

The momentum equation Eq. (2.2) is solved in order to compute the velocity field **u** in the UEqn.H header file.

$FOAM_SOLVERS/combustion/reactingFoam/UEqn.H

```
// Solve the Momentum equation

MRF.correctBoundaryVelocity(U);

tmp<fvVectorMatrix> tUEqn
(
    fvm::ddt(rho, U) + fvm::div(phi, U)
  + MRF.DDt(rho, U)
  + turbulence->divDevRhoReff(U)
 ==
    fvOptions(rho, U)
);
fvVectorMatrix& UEqn = tUEqn.ref();

UEqn.relax();

fvOptions.constrain(UEqn);

if (pimple.momentumPredictor())
{
    solve(UEqn == -fvc::grad(p));

    fvOptions.correct(U);
    K = 0.5*magSqr(U);
}
```

A reference UEqn to the tmp<fvVectorMatrix> object is constructed, which builds the matrix system to solve the momentum equation. The momentum equation is then solved by adding the pressure source term explicitly with the line solve(UEqn == -fvc::grad(p));.

3.2.3.3 YEqn.H

The reacting species transport equations Eq. 2.9 are solved in the YEqn.H header file.

$FOAM_SOLVERS/combustion/reactingFoam/YEqn.H

```
tmp<fv::convectionScheme<scalar>> mvConvection
(
    fv::convectionScheme<scalar>::New
    (
        mesh,
        fields,
        phi,
        mesh.divScheme("div(phi,Yi_h)")
    )
);
```

First, the mvConvection object is created. This is done in order to have the same convection scheme for every species as a convective flux scheme called here div(phi,Yi_h) and selected in the fvScheme file.

$FOAM_SOLVERS/combustion/reactingFoam/YEqn.H

```
    reaction->correct();
    Qdot = reaction->Qdot();
    volScalarField Yt(0.0*Y[0]);
```

Before the species equation is defined and solved for each species, we find the lines reaction->correct(); and Qdot = reaction->Qdot();. The first line tells the turbulence-chemistry interaction

model to correct all species reaction rates, and the second retrieves the updated heat release rate per unit volume, `Qdot`, from the turbulence-chemistry interaction model. In the next block is declared and solved one transport equation for each reacting species (note the `forAll(Y, i)` loop).

`$FOAM_SOLVERS/combustion/reactingFoam/YEqn.H`

```
forAll(Y, i)
{
    if (i != inertIndex && composition.active(i))
    {
        volScalarField& Yi = Y[i];

        fvScalarMatrix YiEqn
        (
            fvm::ddt(rho, Yi)
          + mvConvection->fvmDiv(phi, Yi)
          - fvm::laplacian(turbulence->muEff(), Yi)
         ==
            reaction->R(Yi)
          + fvOptions(rho, Yi)
        );

        YiEqn.relax();

        fvOptions.constrain(YiEqn);

        YiEqn.solve("Yi");

        fvOptions.correct(Yi);

        Yi.clamp_min(0);
        Yt += Yi;
    }
}

Y[inertIndex] = scalar(1) - Yt;
Y[inertIndex].clamp_min(0);
```

We find the temporal derivative `fvm::ddt(rho, Yi)`, the advection term `mvConvection->fvmDiv(phi, Yi)` and the diffusion term `fvm::laplacian(turbulence->muEff(), Yi)`. We note that according to the unity Lewis and Prandtl number assumption, already discussed in Section 2.1, the diffusion coefficient is the effective kinematic viscosity `muEff`, which is the sum of the turbulent and laminar kinematic viscosity and depends on the chosen turbulence and thermophysical models.

The term `reaction->R(Yi)` is the species production rate and corresponds to the term $\dot{\omega}_k$ in Eq. (2.9). To avoid negative nonphysical mass fractions values their minimum is set to 0 with `Yi.clamp_min(0);` and mass fractions are added to the sum `Yt += Yi;`. This allow to compute the mass fraction of the inert species without solving an extra transport equation with the line `Y[inertIndex] = scalar(1) - Yt;`.

3.2.3.4 EEqn.H

The energy equation Eq. (2.4) or Eq. (2.5) is solved in the `EEqn.H` header file.

`$FOAM_SOLVERS/combustion/reactingFoam/EEqn.H`

```
volScalarField& he = thermo.he();
```

First, a reference `volScalarField& he` is constructed by calling the `he()` function on the `thermo` object. Depending on the model chosen in the `thermophysicalProperties` file, this will be the sensible enthalpy `h` or the sensible internal energy `e`.

$FOAM_SOLVERS/combustion/reactingFoam/EEqn.H

```
fvScalarMatrix EEqn
(
    fvm::ddt(rho, he) + mvConvection->fvmDiv(phi, he)
  + fvc::ddt(rho, K) + fvc::div(phi, K)
  + (
        he.name() == "e"
      ? fvc::div
        (
            fvc::absolute(phi/fvc::interpolate(rho), U),
            p,
            "div(phiv,p)"
        )
      : -dpdt
    )
  - fvm::laplacian(turbulence->alphaEff(), he)
 ==
    Qdot
  + fvOptions(rho, he)
);
```

Similar to `rhoEqn.H`, the `fvScalarMatrix EEqn` object is created to solve the matrix system for the chosen energy variable. The only difference between the equations Eq. (2.4) and Eq. (2.5) is in the pressure term in the right hand side of the equations. This is taken into account in `C++` syntax in the line 5 to 14 of the above code. If the variable `he` is the sensible internal energy `e` then the term $\frac{\partial u_i p}{\partial x_i}$ is added to the left hand side of the equation. If it's the sensible enthalpy, then the time derivative of the pressure field is subtracted to the left hand side.

The `fvm::laplacian(turbulence->alphaEff(), he)` term corresponds to the energy flux in Eq. (2.4) with `turbulence->alphaEff()` the effective thermal conductivity being the sum of the turbulent and laminar thermal conductivity which also depends on the turbulence and thermophysical models chosen.

The `Qdot` term calculated at the beginning of `YEqn.H` and corresponding to the heat release rate due to combustion $\dot{\omega_T}$ is added to the right hand side of the equation.

$FOAM_SOLVERS/combustion/reactingFoam/EEqn.H

```
EEqn.relax();

fvOptions.constrain(EEqn);

EEqn.solve();

fvOptions.correct(he);

thermo.correct();

Info<< "min/max(T) = "
    << min(T).value() << ", " << max(T).value() << endl;
```

Finally, the energy equation is solved in the block above, and the temperature is calculated thanks to the `thermo.correct()` line, the minimum and maximum temperature are printed out with `Info<< "min/max(T) = " << min(T).value() << ", " << max(T).value() << endl;`

3.2.3.5 pEqn.H

In this file, the pressure field is computed by solving the pressure equation. This is the main difference with density-based solvers. The pressure-velocity-density coupling is solved through the SIMPLE and PISO algorithms instead of calculating the pressure from an equation of state. At this point, we entered the `while (pimple.correct())` loop or the inner corrector loop, where the PISO algorithm is executed.

$FOAM_SOLVERS/combustion/reactingFoam/pEqn.H

```
rho = thermo.rho();

volScalarField rAU(1.0/UEqn.A());
surfaceScalarField rhorAUf("rhorAUf", fvc::interpolate(rho*rAU));
volVectorField HbyA(constrainHbyA(rAU*UEqn.H(), U, p));
```

First the density `rho` is updated according to the new temperature field since `T` was updated at the end of `EEqn.H`. `thermo.rho()` calls the ideal gas law Eq. (3.2) to compute the new density field.

The next three lines declare new variables needed to solve the pressure equation. The inverse `rAU` of the `A` matrix of the momentum linear system is computed as `volScalarField rAU(1.0/UEqn.A());`. The compressible flux of `rAU` on the cell faces is then computed on the next line by interpolating `rho*rAU` on the cell faces with `surfaceScalarField rhorAUf("rhorAUf", fvc::interpolate(rho*rAU));`. The last line calculates the velocity field without pressure gradient, `HbyA` with corrected boundary conditions (`contrainHbyA`).

$FOAM_SOLVERS/combustion/reactingFoam/pEqn.H

```
{
    surfaceScalarField phiHbyA
    (
        "phiHbyA",
        (
            fvc::flux(rho*HbyA)
          + MRF.zeroFilter(rhorAUf*fvc::ddtCorr(rho, U, phi))
        )
    );

    MRF.makeRelative(fvc::interpolate(rho), phiHbyA);

    // Update the pressure BCs to ensure flux consistency
    constrainPressure(p, rho, U, phiHbyA, rhorAUf, MRF);

    while (pimple.correctNonOrthogonal())
    {
        fvScalarMatrix pEqn
        (
            fvm::ddt(psi, p)
          + fvc::div(phiHbyA)
          - fvm::laplacian(rhorAUf, p)
         ==
            fvOptions(psi, p, rho.name())
        );

        pEqn.solve(p.select(pimple.finalInnerIter()));

        if (pimple.finalNonOrthogonalIter())
        {
            phi = phiHbyA + pEqn.flux();
        }
    }
}
```

In the code above the pressure equation is built and solved. First the face flux of `HbyA`, `surfaceScalarField phiHbyA` is created using `fvc::flux(rho*HbyA)` and an additional transient correction term for moving meshes (`MRF` for Multiple Reference Frame) that we neglect here. The line `constrainPressure(p, rho, U, phiHbyA, rhorAUf, MRF);`, as indicated in the comment enforces global conservation of `phiHbyA` and coherent pressure boundary conditions. Next, we enter the `while (pimple.correctNonOrthogonal())` loop which can be specified in the `fvSolution` file as `nNonOrthogonalCorrectors 0;`. The `fvScalarMatrix pEqn` is created to solve the pressure equation for compressible flow. Despite similarity to the pressure-correction equation for incompressible flows, this compressible formulation is different and is described in details in [2, 3].

The linear solver is read from `fvSolution` in `p.select(pimple.finalInnerIter())` that returns

a word type which is either p or pFinal. In the final non-orthogonal correction loop, the conservative face fluxes are corrected with phi = phiHbyA + pEqn.flux();.

$FOAM_SOLVERS/combustion/reactingFoam/pEqn.H

```
#include "rhoEqn.H"
#include "compressibleContinuityErrs.H"

// Explicitly relax pressure for momentum corrector
p.relax();

U = HbyA - rAU*fvc::grad(p);
U.correctBoundaryConditions();
fvOptions.correct(U);
K = 0.5*magSqr(U);

if (pressureControl.limit(p))
{
    p.correctBoundaryConditions();
}

rho = thermo.rho();

if (thermo.dpdt())
{
    dpdt = fvc::ddt(p);
}
```

In the last lines of the file, the density is recalculated by solving the continuity equation with the freshly updated fluxes phi and the continuity errors are calculated and printed out with #include "compressibleContinuityErrs.H". Following, the pressure field is explicitly relaxed, the velocity field is corrected as well as its boundary conditions and the kinetic energy field K is recalculated with the new velocity field. If pressure bounds are specified in the fvSolution file the pressure is limited and its boundary conditions corrected as well. Finally, the density is computed again from the equation of state to be consistent with the pressure and the time derivative of the pressure field is updated for use in the energy equation.

3.3 rhoCentralFoam

We present a description of the implementation of the solver rhoCentralFoam, summarizing that given in full by Greenshields et al. [8]. In-depth details regarding the solver rhoCentralFoam have been provided in a previous report by Harvey, E. [19]. For a more comprehensive understanding of the solver's formulation and implementation, the reader is referred to this work. Here, we will only go through the major differences between the implementation of rhoCentralFoam with regard to reactingFoam and make the link between the equations introduced in Section 2.2 and the code.

3.3.1 createFields.H

Similarly as in reactingFoam we have createFields.H and createFieldRefs.H files.

$FOAM_SOLVERS/compressible/rhoCentralFoam/createFieldRefs.H

```
volScalarField& p = thermo.p();
const volScalarField& T = thermo.T();
const volScalarField& psi = thermo.psi();

bool inviscid(true);
if (max(thermo.mu().cref().primitiveField()) > 0.0)
{
    inviscid = false;
}
```

The temperature T, pressure p, compressibility psi, and viscosity mu, are set up as references as seen above. A further boolean variable, inviscid, is created as true and then determined based on the viscosity. If the viscosity is read as zero from the thermophysicalProperties file, then this boolean remains true and a simplified set of the Navier-Stokes equations is solved. If mu is greater than 0, then the simulation is run with the inviscid variable set to false.

$FOAM_SOLVERS/compressible/rhoCentralFoam/createFields.H

```
#include "createRDeltaT.H"

Info<< "Reading thermophysical properties\n" << endl;

autoPtr<psiThermo> pThermo
(
    psiThermo::New(mesh)
);
psiThermo& thermo = pThermo();

volScalarField& e = thermo.he();

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

Like in reactingFoam, the thermo reference is created to access and update the thermophysical properties. It is also based on compressibility psi but since we are not dealing with reacting flow here, the thermodynamic class is psiThermo and not psiReactionThermo. We note that the energy variable here is called e instead of he, for rhoCentralFoam we should only choose the internal energy as energy variable.

$FOAM_SOLVERS/compressible/rhoCentralFoam/createFields.H

```
volVectorField rhoU
(
    IOobject
    (
        "rhoU",
        runTime.timeName(),
        mesh,
        IOobject::NO_READ,
        IOobject::NO_WRITE
    ),
    rho*U
);

volScalarField rhoE
(
    IOobject
    (
        "rhoE",
        runTime.timeName(),
        mesh,
        IOobject::NO_READ,
        IOobject::NO_WRITE
    ),
    rho*(e + 0.5*magSqr(U))
);
```

The solver solves the conservative variables or density weighted fields, ρ (`rho`), $\hat{\mathbf{u}} = \rho\mathbf{u}$ (`rhoU`), $\hat{E} = \rho E$ (`rhoE`). These fields are setup in the createFields.H file using the `rho`, `U`, `e` fields and formula for the total energy $E = e + \frac{\|\mathbf{u}^2\|}{2}$ as seen above.

`$FOAM_SOLVERS/compressible/rhoCentralFoam/createFields.H`

```
surfaceScalarField pos
(
    IOobject
    (
        "pos",
        runTime.timeName(),
        mesh
    ),
    mesh,
    dimensionedScalar("pos", dimless, 1.0)
);

surfaceScalarField neg
(
    IOobject
    (
        "neg",
        runTime.timeName(),
        mesh
    ),
    mesh,
    dimensionedScalar("neg", dimless, -1.0)
);
```

Also included in the header file are the creation of fields `pos` and `neg`, as `surfaceScalarField` variables which are used in the solver as part of the central discretization process and to take into account the propagation of waves in any direction (previously introduced as $f+$ and $f-$ in Section 2.2). The discretization of the convective fluxes $\nabla \cdot (\mathbf{u}\rho)$, $\nabla \cdot (\mathbf{u}(\rho\mathbf{u}))$, $\nabla \cdot (\mathbf{u}(\rho E))$, $\nabla \cdot (\mathbf{u}p)$ and the implementation of the central scheme is discussed further in Section 3.3.2.

`$FOAM_SOLVERS/compressible/rhoCentralFoam/createFields.H`

```
surfaceScalarField phi("phi", fvc::flux(rhoU));

Info<< "Creating turbulence model\n" << endl;
autoPtr<compressible::turbulenceModel> turbulence
(
    compressible::turbulenceModel::New
    (
        rho,
        U,
        phi,
        thermo
    )
);
```

The face flux `phi` is created, this implementation is equivalent to the one in `reactingFoam` but uses the `fvc::flux()` function on the `rhoU` field.

3.3.2 rhoCentralFoam.C

3.3.2.1 Central Flux Scheme Implementation

$FOAM_SOLVERS/compressible/rhoCentralFoam/createFields.H

```
        // --- Directed interpolation of primitive fields onto faces

        surfaceScalarField rho_pos(interpolate(rho, pos));
        surfaceScalarField rho_neg(interpolate(rho, neg));

        surfaceVectorField rhoU_pos(interpolate(rhoU, pos, U.name()));
        surfaceVectorField rhoU_neg(interpolate(rhoU, neg, U.name()));

        volScalarField rPsi("rPsi", 1.0/psi);
        surfaceScalarField rPsi_pos(interpolate(rPsi, pos, T.name()));
        surfaceScalarField rPsi_neg(interpolate(rPsi, neg, T.name()));

        surfaceScalarField e_pos(interpolate(e, pos, T.name()));
        surfaceScalarField e_neg(interpolate(e, neg, T.name()));

        surfaceVectorField U_pos("U_pos", rhoU_pos/rho_pos);
        surfaceVectorField U_neg("U_neg", rhoU_neg/rho_neg);

        surfaceScalarField p_pos("p_pos", rho_pos*rPsi_pos);
        surfaceScalarField p_neg("p_neg", rho_neg*rPsi_neg);

        surfaceScalarField phiv_pos("phiv_pos", U_pos & mesh.Sf());
        // Note: extracted out the orientation so becomes unoriented
        phiv_pos.setOriented(false);
        surfaceScalarField phiv_neg("phiv_neg", U_neg & mesh.Sf());
        phiv_neg.setOriented(false);
```

At the beginning of the time loop, the above code interpolates primitive variables (e.g., density, velocity, and energy) from cell centers to face centers in the positive (`pos`) and negative (`neg`) directions, which correspond to outward and inward fluxes at the face. The interpolation is performed using the discretization scheme specified in the `fvSchemes` dictionary, for example:

$FOAM_TUTORIALS/compressible/rhoCentralFoam/forwardStep/controlDict/fvSchemes

```
interpolationSchemes
{
    default         linear;

    reconstruct(rho) vanLeer;
    reconstruct(U)  vanLeerV;
    reconstruct(T)  vanLeer;
}
```

The fluxes ϕ associated with the fluid velocity through the face are calculated as:

- $\phi_{f+} = \mathbf{u}_{f+} \cdot \mathbf{S}_f$ (`U_pos & mesh.Sf()`, flux outward of the owner cell)
- $\phi_{f-} = \mathbf{u}_{f-} \cdot \mathbf{S}_f$ (`U_neg & mesh.Sf()`, flux inward of the neighbor cell)

The fluxes are then adjusted:

- `phiv_pos.setOriented(false)` ensures that `phiv_pos` is an unoriented scalar.
- `phiv_neg.setOriented(false)` ensures that `phiv_neg` is an unoriented scalar.

These interpolated fluxes are essential for the central scheme, which computes fluxes using information from both sides of the face.

$FOAM_SOLVERS/compressible/rhoCentralFoam/createFields.H

```
    volScalarField c("c", sqrt(thermo.Cp()/thermo.Cv()*rPsi));
    surfaceScalarField cSf_pos
    (
        "cSf_pos",
        interpolate(c, pos, T.name())*mesh.magSf()
    );

    surfaceScalarField cSf_neg
    (
        "cSf_neg",
        interpolate(c, neg, T.name())*mesh.magSf()
    );

    surfaceScalarField ap
    (
        "ap",
        max(max(phiv_pos + cSf_pos, phiv_neg + cSf_neg), v_zero)
    );

    surfaceScalarField am
    (
        "am",
        min(min(phiv_pos - cSf_pos, phiv_neg - cSf_neg), v_zero)
    );

    surfaceScalarField a_pos("a_pos", ap/(ap - am));

    surfaceScalarField amaxSf("amaxSf", max(mag(am), mag(ap)));

    surfaceScalarField aSf("aSf", am*a_pos);

    if (fluxScheme == "Tadmor")
    {
        aSf = -0.5*amaxSf;
        a_pos = 0.5;
    }

    surfaceScalarField a_neg("a_neg", 1.0 - a_pos);

    phiv_pos *= a_pos;
    phiv_neg *= a_neg;

    surfaceScalarField aphiv_pos("aphiv_pos", phiv_pos - aSf);
    surfaceScalarField aphiv_neg("aphiv_neg", phiv_neg + aSf);

    // Reuse amaxSf for the maximum positive and negative fluxes
    // estimated by the central scheme
    amaxSf = max(mag(aphiv_pos), mag(aphiv_neg));

    #include "centralCourantNo.H"
```

Similarly as in Eq. (2.14), the local speed of sound is calculated as

$$c = \sqrt{\frac{\gamma}{\psi}}. \tag{3.5}$$

This represents the local speed at which pressure disturbances propagate in the medium. Similarly as with the previous primitive variables, the local speed of sound is interpolated in the positive and negative sides of the face and multiplied by the surface area of the cell face to obtain the associated volumetric fluxes `cSf_pos` and `cSf_neg`. The volumetric fluxes a_+ and a_- defined in Eq. (2.13) are defined as `a_p` and `a_m` in the code. The diffusive volumetric flux ω_f is then calculated under the variable name `aSf` and updated together with α (`a_pos`), in case the `Tadmor` scheme is chosen. To recover the formulation of Eq. (2.12), the fluxes `phiv_pos` and `phiv_pos` (ϕ_{f+} and ϕ_{f-}) are multiplied by `a_pos` and `a_neg` (α and $1-\alpha$) and `aSf` is added and subtracted respectively to obtain `aphiv_pos` and `aphiv_neg`.

We can then calculate the different convective terms $\nabla \cdot (\mathbf{u}\Psi)$ of Eq. (2.11):

- $\nabla \cdot (\mathbf{u}\rho)$:

$FOAM_SOLVERS/compressible/rhoCentralFoam/rhoCentralFoam.C

```
phi = aphiv_pos*rho_pos + aphiv_neg*rho_neg;
```

- $\nabla \cdot (\mathbf{u}(\rho\mathbf{u}))$:

$FOAM_SOLVERS/compressible/rhoCentralFoam/rhoCentralFoam.C

```
surfaceVectorField phiU(aphiv_pos*rhoU_pos + aphiv_neg*rhoU_neg);
```

The pressure gradient is then integrated:

$FOAM_SOLVERS/compressible/rhoCentralFoam/rhoCentralFoam.C

```
surfaceVectorField phiUp(phiU + (a_pos*p_pos + a_neg*p_neg)*mesh.Sf());
```

- $\nabla \cdot (\mathbf{u}(\rho E))$:

$FOAM_SOLVERS/compressible/rhoCentralFoam/rhoCentralFoam.C

```
surfaceScalarField phiEp
(
    "phiEp",
    aphiv_pos*(rho_pos*(e_pos + 0.5*magSqr(U_pos)) + p_pos)
  + aphiv_neg*(rho_neg*(e_neg + 0.5*magSqr(U_neg)) + p_neg)
  + aSf*p_pos - aSf*p_neg
);
```

This is also corrected with `+ aSf*p_pos` and `- aSf*p_neg` to counteract the `aSf` (ω_f) terms applied through the `aphiv` fields for the pressure term. This modification is used since the additional diffusive flux terms with `aSf` (ω_f) are only needed when the convective term is applied as part of a substantive derivative, $\frac{D}{Dt}$. This has been true for all other fields but must be corrected for p.

Last step before solving the governing equations is to declare the fields related to viscosity:

$FOAM_SOLVERS/compressible/rhoCentralFoam/rhoCentralFoam.C

```
volScalarField muEff("muEff", turbulence->muEff());
volTensorField tauMC("tauMC", muEff*dev2(Foam::T(fvc::grad(U))));
```

3.3.2.2 Solve Governing Equations

$FOAM_SOLVERS/compressible/rhoCentralFoam/rhoCentralFoam.C

```
// --- Solve density
solve(fvm::ddt(rho) + fvc::div(phi));
```

First, the continuity equation is solved explicitly to update the density.

$FOAM_SOLVERS/compressible/rhoCentralFoam/rhoCentralFoam.C

```
// --- Solve momentum
solve(fvm::ddt(rhoU) + fvc::div(phiUp));

U.ref() =
    rhoU()
   /rho();
U.correctBoundaryConditions();
rhoU.boundaryFieldRef() == rho.boundaryField()*U.boundaryField();

if (!inviscid)
{
    solve
    (
        fvm::ddt(rho, U) - fvc::ddt(rho, U)
      - fvm::laplacian(muEff, U)
      - fvc::div(tauMC)
    );
    rhoU = rho*U;
}
```

The momentum equation is solved in two steps. The diffusive terms, which are functions of $\mathbf{u}$ and T, cannot be evaluated implicitly with the other terms due to the variables, `rhoU`, `rhoE` being density weighted. To include them explicitly would mean a fully explicit solution which was found by C.Greenshields et al. [8] to suffer a severe time-step limit. Instead, the diffusive terms are applied as implicit corrections to the inviscid equation if the viscosity is non-zero.

$FOAM_SOLVERS/compressible/rhoCentralFoam/rhoCentralFoam.C

```
// --- Solve energy
surfaceScalarField sigmaDotU
(
    "sigmaDotU",
    (
        fvc::interpolate(muEff)*mesh.magSf()*fvc::snGrad(U)
      + fvc::dotInterpolate(mesh.Sf(), tauMC)
    )
  & (a_pos*U_pos + a_neg*U_neg)
);

solve
(
    fvm::ddt(rhoE)
  + fvc::div(phiEp)
  - fvc::div(sigmaDotU)
);

e = rhoE/rho - 0.5*magSqr(U);
e.correctBoundaryConditions();
thermo.correct();
rhoE.boundaryFieldRef() ==
    rho.boundaryField()*
    (
        e.boundaryField() + 0.5*magSqr(U.boundaryField())
    );

if (!inviscid)
{
    solve
    (
        fvm::ddt(rho, e) - fvc::ddt(rho, e)
      - fvm::laplacian(turbulence->alphaEff(), e)
    );
    thermo.correct();
    rhoE = rho*(e + 0.5*magSqr(U));
}
```

The solution of the energy equation follows a largely similar method with the variable `sigmaDotU` corresponding to the viscous heating term. The inviscid equation is first solved and the energy `e` updated. The boundary conditions are corrected as well as the thermophysical properties and temperature. Finally, the non-inviscid element is added implicitly and the new equation is solved.

`$FOAM_SOLVERS/compressible/rhoCentralFoam/rhoCentralFoam.C`

```
        p.ref() =
            rho()
           /psi();
        p.correctBoundaryConditions();
        rho.boundaryFieldRef() == psi.boundaryField()*p.boundaryField();

        turbulence->correct();

        runTime.write();

        runTime.printExecutionTime(Info);
    }

    Info<< "End\n" << endl;

    return 0;
}
```

After solving all the governing equations, the pressure field is updated using the ideal gas law. The pressure boundary conditions are then corrected in order to update the boundary conditions for `rho` again. The time loop ends with the correction of the turbulence properties and the writing of relevant variables.

Chapter 4

Implementation of `reactingRhoCentralFoam`

In this chapter, we describe the implementation of the reacting species transport equations in the solver `rhoCentralFoam`.

4.1 `createFields.H`

The first step is to modify the `createFields.H` header file since this is where the fields are constructed and the pointers are created.

4.1.1 `thermo` Object and Species Mass Fractions

The first modification is to change the thermophysical model from `psiThermo` to `psiReactionThermo` since it takes into account chemical reactions and to declare the species mass fractions fields. We replace the lines 5 to 11 with the following block:

`reactingRhoCentralFoam/createFields.H`

```
Info<< "Reading thermophysical properties\n" << endl;
autoPtr<psiReactionThermo> pThermo(psiReactionThermo::New(mesh));
psiReactionThermo& thermo = pThermo();
volScalarField& e = thermo.he();

basicSpecieMixture& composition = thermo.composition();
PtrList<volScalarField>& Y = composition.Y();
PtrList<volScalarField> rhoY(Y.size());

const word inertSpecie(thermo.get<word>("inertSpecie"));
if (!composition.species().found(inertSpecie))
{
    FatalIOErrorIn(args.executable().c_str(), thermo)
        << "Inert specie " << inertSpecie << " not found in available species "
        << composition.species() << exit(FatalIOError);
}
```

This block is identical to that in `reactingFoam`, with the exception of the new line `PtrList<volScalarField> rhoY(Y.size());` which declares the list of pointers to the density weighted species mass fractions

$$\hat{Y_k} = \rho Y_k, \tag{4.1}$$

which are needed for the implementation of central convective fluxes in the species mass fractions equations.

reactingRhoCentralFoam/createFields.H

```
multivariateSurfaceInterpolationScheme<scalar>::fieldTable fields;

forAll(Y, i)
{
    volScalarField& Yi = Y[i];
    fields.add(Yi);

    const word Yiname = Yi.name();
    rhoY.set
    (
        i,
        new volScalarField
        (
            IOobject
            (
                "rho"+Yiname,
                runTime.timeName(),
                mesh,
                IOobject::NO_READ,
                IOobject::NO_WRITE
            ),
            rho*Yi
        )
    );
}
fields.add(thermo.he());
```

The last difference with the original file can be seen in the code above. The block `rhoY.set(i, new volScalarField(...))` creates a new field for the density-weighted species mass fraction `rhoY[i]` and assigns it to the `rhoY` field list. The `volScalarField` is initialized with an `IOobject` that specifies its name `"rho"+Yiname`, the current simulation time `runTime.timeName()`, and the computational mesh. The `IOobject::NO_READ` and `IOobject::NO_WRITE` flags ensure that the field is not read from or written to disk automatically. The value of `rhoY[i]` is computed as the product of the density field `rho` and the corresponding species mass fraction field `Yi`. This step ensures that the density-weighted species fields are properly initialized for use in the simulation.

4.2 reactingRhoCentralFoam.C

Here we go through the modification of the main file `reactingRhoCentralFoam.C`.

4.2.1 Header Files

reactingRhoCentralFoam/reactingRhoCentralFoam.C

```
#include "fvCFD.H"
#include "turbulentFluidThermoModel.H"
#include "psiReactionThermo.H" //added
#include "CombustionModel.H" //added
#include "dynamicFvMesh.H"
#include "fixedRhoFvPatchScalarField.H"
#include "directionInterpolate.H"
#include "localEulerDdtScheme.H"
#include "fvcSmooth.H"
```

We replace the `#include "psiThermo.H"` with `#include "psiReactionThermo.H"` and add `#include "CombustionModel.H"` for combustion modeling.

4.2.2 rhoYEqn.H

reactingRhoCentralFoam/reactingRhoCentralFoam.C

```
        // --- Solve species
        #include "rhoYEqn.H"
```

We solve the species equations after solving the momentum equation in reactingRhoCentralFoam.C by including a new rhoYEqn.H header file.

reactingRhoCentralFoam/rhoYEqn.H

```
    reaction->correct();
    Qdot = reaction->Qdot();
    volScalarField Yt(0.0*Y[0]);

    forAll(Y, i)
    {
        if (i != inertIndex && composition.active(i))
        {
            volScalarField& rhoYi = rhoY[i];
            volScalarField& Yi = Y[i];

            surfaceScalarField rhoYi_pos(interpolate(rhoYi, pos, T.name()));
            surfaceScalarField rhoYi_neg(interpolate(rhoYi, neg, T.name()));

            surfaceScalarField phiYi(aphiv_pos*rhoYi_pos + aphiv_neg*rhoYi_neg);

            // --- Solve Yi
            solve(fvm::ddt(rhoYi) + fvc::div(phiYi));
            rhoYi.clamp_min(0);
            Yi.ref() = rhoYi()/rho();
            Yi.correctBoundaryConditions();
            rhoYi.boundaryFieldRef() == rho.boundaryField()*Yi.boundaryField();

            if (!inviscid)
            {
                fvScalarMatrix YiEqn
                (
                    fvm::ddt(rho, Yi) - fvc::ddt(rho, Yi)
                  - fvm::laplacian(muEff, Yi)
                    ==
                    reaction->R(Yi)
                );
                YiEqn.solve("Yi");
            }
            else
            {
                fvScalarMatrix YiEqn
                (
                    fvm::ddt(rho, Yi) - fvc::ddt(rho, Yi)
                    ==
                    reaction->R(Yi)
                );
                YiEqn.solve("Yi");
            }

            Yi.clamp_min(0);

            rhoYi = rho*Yi;

            Yt += Yi;
        }
    }

    Y[inertIndex] = scalar(1) - Yt;
    Y[inertIndex].clamp_min(0);
    rhoY[inertIndex] = rho*Y[inertIndex];
```

This is a modified version of the YEqn.H from the reactingFoam solver to include the central scheme discretization of the advective fluxes $\nabla \cdot (\mathbf{u}(\rho Y_k))$. We already described the implementation of the advective fluxes $\nabla \cdot (\mathbf{u}\rho)$, $\nabla \cdot (\mathbf{u}(\rho\mathbf{u}))$ and $\nabla \cdot (\mathbf{u}(\rho E))$ in rhoCentralFoam in Section 3.3.2.1 and we will follow a similar procedure here:

- Interpolation of rhoYi in both pos and neg directions:
  - surfaceScalarField rhoYi_pos(interpolate(rhoYi, pos, T.name()));
  - surfaceScalarField rhoYi_neg(interpolate(rhoYi, neg, T.name()));
- $\nabla \cdot (\mathbf{u}(\rho Y_k))$ is calculated with the volumetric fluxes a_+, a_- and the diffusive term ω_f integrated in aphiv_pos and aphiv_neg computed previously in reactingRhoCentralFoam.C:
  - surfaceScalarField phiYi(aphiv_pos*rhoYi_pos + aphiv_neg*rhoYi_neg);
- The inviscid transport equation is first solved without the chemical source term. The species mass fraction Yi is updated and its boundary conditions corrected.
- The diffusive term is then applied as implicit corrections to the inviscid equation if the viscosity is non-zero. Like in reactingFoam, we add the species production rate reaction->R(Yi) to the transport equation in both inviscid and viscous formulations.
- The rest of the code is similar to reactingFoam, with the mass fractions minimum limited to zero and the inert species solved as Y[inertIndex] = scalar(1) - Yt;.

4.3 Make/files and Make/options

We modify the Make/files file accordingly to the new solver name reactingRhoCentralFoam.

reactingRhoCentralFoam/Make/files

```
reactingRhoCentralFoam.C

EXE = $(FOAM_USER_APPBIN)/reactingRhoCentralFoam
```

We modify the Make/options file to link the necessary libraries for combustion simulations.

reactingRhoCentralFoam/Make/options Part 1

```
EXE_INC = \
    -I$(FOAM_SOLVERS)/compressible/rhoCentralFoam/BCs/lnInclude \
    -I$(LIB_SRC)/finiteVolume/lnInclude \
    -I$(LIB_SRC)/meshTools/lnInclude \
    -I$(LIB_SRC)/TurbulenceModels/turbulenceModels/lnInclude \
    -I$(LIB_SRC)/TurbulenceModels/compressible/lnInclude \
    -I$(LIB_SRC)/thermophysicalModels/specie/lnInclude \
    -I$(LIB_SRC)/thermophysicalModels/reactionThermo/lnInclude \
    -I$(LIB_SRC)/transportModels/compressible/lnInclude \
    -I$(LIB_SRC)/thermophysicalModels/basic/lnInclude \
    -I$(LIB_SRC)/thermophysicalModels/chemistryModel/lnInclude \
    -I$(LIB_SRC)/dynamicFvMesh/lnInclude \
    -I$(LIB_SRC)/ODE/lnInclude \
    -I$(LIB_SRC)/combustionModels/lnInclude
```

We first show the included libraries for compilation. The first line was modified to link the special boundary conditions used in rhoCentralFoam. Since we don't want to recompile this library, we link our new solver with the already compiled library located in the BCs folder in the original rhoCentralFoam directory. This allows us to delete the BCs folder in the reactingRhoCentralFoam directory. Additionally, the following libraries are linked to the solver: reactionThermo, chemistryModel, ODE and combustionModels.

reactingRhoCentralFoam/Make/options Part 1

```
EXE_LIBS = \
    -lfiniteVolume \
    -lfvOptions \
    -lmeshTools \
    -lcompressibleTransportModels \
    -lfluidThermophysicalModels \
    -lspecie \
    -lrhoCentralFoam \
    -lturbulenceModels \
    -lcompressibleTurbulenceModels \
    -lthermoTools \
    -lreactionThermophysicalModels \
    -lchemistryModel \
    -lODE \
    -lcombustionModels \
    -ldynamicFvMesh \
    -ltopoChangerFvMesh
```

Here, we specify the compiled libraries that the executable needs to link with during the building process. The added libraries are `lreactionThermophysicalModels`, `lchemistryModels`, `lODE` and `lcombustionModels`. We can finally compile the new solver by executing `wmake` in the terminal from the `reactingRhoCentralFoam` directory.

Chapter 5

Test Cases

5.1 Cold Flow: `forwardStep` case

5.1.1 Setup

The flow is moving at Mach 3 in a wind tunnel containing a forward-facing step was originally introduced by Emery [20] as a test for numerical schemes. It uses a gas initialized with an inlet velocity $U_{\text{in}} = 3\,\text{m/s}$, pressure $p = 1\,\text{Pa}$ and temperature $T = 1\,\text{K}$. The properties are set such that this is an inviscid gas for which the speed of sound is $1\,\text{m/s}$ at a temperature of $1\,\text{K}$ and $\gamma = 7/5$. This is described in more detail in the following section. We have a structured uniform mesh with the cell length in the x and y direction equal to $1.25\,\text{cm}$, and we ran the simulation at a CFL number of 0.2 for a duration of $4\,\text{s}$.

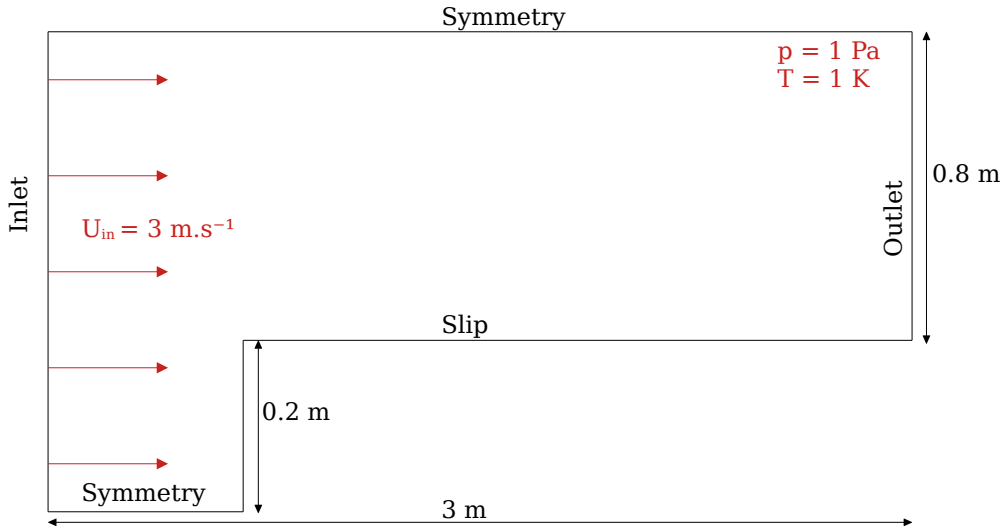

Figure 5.1: Numerical domain of the `forwardStep` tutorial, with initial conditions in red and boundary conditions in black.

This case is originally located in `$FOAM_TUTORIALS/compressible/rhoCentralFoam/forwardStep`. We adapt and run the case with `reactingRhoCentralFoam` and compare the results with the original `rhoCentralFoam` solver. This is done in order to have a first check of the implementation without reacting species. In the next section, we describe the modified input files to run the case with `reactingRhoCentralFoam`.

5.1.2 Thermophysical Files

reactingRhoCentralFoam/run/forwardStep/constant/thermophysicalProperties

```
thermoType
{
    type            hePsiThermo;
    mixture         reactingMixture;
    transport       const;
    thermo          eConst;
    equationOfState perfectGas;
    specie          specie;
    energy          sensibleInternalEnergy;
}

inertSpecie         N2;

chemistryReader     foamChemistryReader;

foamChemistryFile   "<constant>/reactions";

foamChemistryThermoFile "<constant>/thermo.compressibleGas";
```

The `thermophysicalProperties` file defines the thermodynamic, transport, and chemistry models used in the simulation. The `thermoType` block first specifies `type hePsiThermo`, a thermodynamic model for compressible flows where the pressure is treated as a primary thermodynamic variable. The `mixture reactingMixture` defines a chemically reactive mixture. `transport const` sets a constant transport model for properties like viscosity and thermal conductivity. `thermo eConst` uses a constant heat capacity model where specific heat capacities (`Cp` and `Cv`) are independent of temperature. `equationOfState perfectGas` assumes the gas obeys the ideal gas law Eq. (3.2). `specie specie` indicates the use of a `specie` object for molecular weights and gas constants. Finally, `energy sensibleInternalEnergy` specifies that the solved energy variable is the sensible internal energy `e`. The `inertSpecie N2` defines nitrogen `N2` as the inert species, which does not participate in chemical reactions. For chemistry and thermodynamic data, `chemistryReader foamChemistryReader` specifies the OpenFOAM-native chemistry file reader. Reaction definitions are located in `<constant>/reactions`, while species thermophysical properties, such as heat capacities and molecular weights, are provided in `<constant>/thermo.compressibleGas`.

reactingRhoCentralFoam/run/forwardStep/constant/thermo.compressibleGas

```
// Note: these are the properties for a "normalised" inviscid gas
//       for which the speed of sound is 1 m/s at a temperature of 1K
//       and gamma = 7/5
N2
{
    specie
    {
        molWeight       11640.3;
    }
    elements
    {
        N       2;
    }
    thermodynamics
    {
        Cv              1.7857;
        Hf              0;
    }
    transport
    {
        mu              0;
        Pr              1;
    }
}
```

As we see in the note at the beginning of the file above, the thermodynamic and transport properties are set for nitrogen `N2` under a "normalized" inviscid gas assumption, such that the speed of sound is equal to 1 m/s at a temperature of 1 K. Thus, those are not realistic properties of nitrogen. The molecular weight is set equal to 11.640 kg/mol. The specific heat at constant volume `Cv` is set to $1.7857\,\mathrm{J}\cdot\mathrm{K/kg}$, while the formation enthalpy `Hf` is set to zero as a reference. Transport properties reflect the inviscid assumption, with viscosity `mu` set to $0\,\mathrm{Pa}\cdot\mathrm{s}$. The Prandtl number `Pr` is defined as 1, so the thermal conductivity is equal to the viscosity. We justify the choice of those values in the next derivation.

Assuming ideal gas assumption, the speed of sound c is computed as

$$c = \sqrt{\gamma \frac{R}{W} T}. \tag{5.1}$$

Here, γ is the ratio of specific heats

$$\gamma = \frac{C_p}{C_v}. \tag{5.2}$$

To calculate C_p, we can use Mayer's relation

$$C_p = C_v + \frac{R}{W} = 1.7857 + \frac{8.314}{11.64} = 2.5, \tag{5.3}$$

and then calculate γ

$$\gamma = \frac{2.5}{1.786} = 1.4, \tag{5.4}$$

and finally the speed of sound in such mixture at 1 K

$$c = \sqrt{1.4 \times \frac{8.314}{11.64} \times 1} = 1\,\mathrm{m/s}. \tag{5.5}$$

`reactingRhoCentralFoam/run/forwardStep/constant/reactions`

```
elements
(
    N
);

species
(
    N2
);

reactions
{
}
```

The `reactions` file defines the chemical elements, species, and reactions. In this case, the `elements` section specifies the chemical elements involved in the reactions, with nitrogen `N` being the only element listed. The `species` section defines the chemical species, which is the normalized N_2 in this case. The `reactions` section would typically list the chemical reactions, but it is empty here, indicating that no specific reactions are defined for this system.

5.1.3 Initial T, U Files

reactingRhoCentralFoam/run/forwardStep/0/T

```
dimensions      [0 0 0 1 0 0 0];

internalField   uniform 1;

boundaryField
{
    inlet
    {
        type            fixedValue;
        value           uniform 1;
    }

    outlet
    {
        type            inletOutlet;
        inletValue      uniform 1;
        value           uniform 1;
    }
```

reactingRhoCentralFoam/run/forwardStep/0/U

```
dimensions      [0 1 -1 0 0 0 0];

internalField   uniform (3 0 0);

boundaryField
{
    inlet
    {
        type            fixedValue;
        value           uniform (3 0 0);
    }

    outlet
    {
        type            inletOutlet;
        inletValue      uniform (3 0 0);
        value           uniform (3 0 0);
    }
```

In the above files, we show the important parts of the `0/T` and `0/U` files. As mentioned previously, the temperature is set to 1 K so the speed of sound in the gas is equal to 1 m/s. The initial velocity is set to 3 m/s in the whole domain in order to have a supersonic configuration with Ma = 3.

The `inletOutlet` boundary condition is chosen for the outlet for both `T` and `U`. This is an usual boundary condition when the flow can be either entering or exiting the domain. It behaves as a `fixedValue` boundary condition if the flow is entering the domain and as a `zeroGradient` boundary condition if the flow is leaving the domain. The `inletValue` keyword specifies the value of the variable on the boundary in case the flow is entering the domain. Since both `inletValue` and `value` are set to `(3,0,0)`, the flow is initialized to have a constant velocity of 3 m/s in the positive x-direction at the outlet and since there is no domain beyond the outlet, the flow is simply forced to exit at the specified rate. The same treatment is applied for `T`. We don't present the other initial files `0/p` and `0/N2` here since their setup is trivial.

5.1.4 Results

Figure 5.2 shows the state after 4 s of the `forwardStep` with `rhoCentralFoam` and, for comparison, mirrored along the x-axis, with the new solver `reactingRhoCentralFoam`. Initially, the shock curves toward the step's upper surface, then flattens, strikes the upper boundary, reflects downward, and hits the step. As it continues to flatten, a Mach reflection forms, and the intersection of the waves

moves upstream, creating a horizontal slip surface. A weak shock also forms where the overexpanded flow hits the step's upper surface. The results match perfectly, with identical pressure fields observed throughout the domain for both solvers.

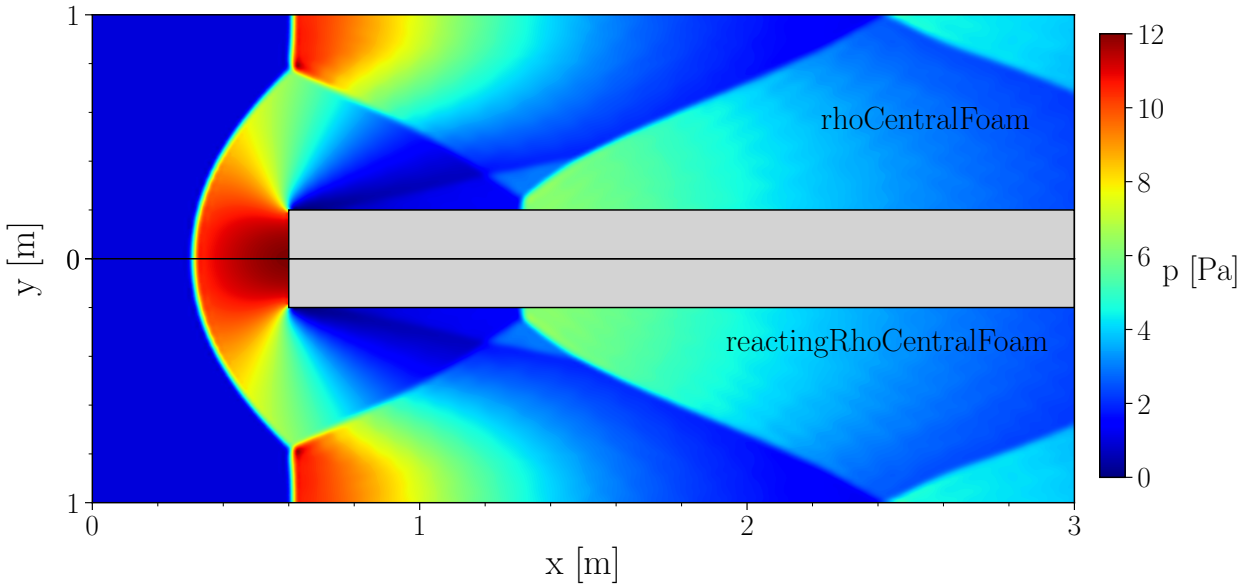

Figure 5.2: `forwardStep` case results comparison between `rhoCentralFoam` (Top) and mirrored `reactingRhoCentralFoam` (Bottom).

This provides an initial validation of the implementation of the `rhoCentralFoam` framework within the reacting thermodynamic package, demonstrating its applicability to non-reacting cases.

5.2 Reacting Flow: One-Dimensional Reacting Shock Tube

5.2.1 Setup

Here we validate the implementation of the transport equation of the reacting species. We use the one-dimensional reacting shock tube case from Ferrer et al. [21]. It involves a reactive mixture in a closed tube, where a shock reflects off a solid boundary, triggering a reaction wave that grows and merges with the shock structure.

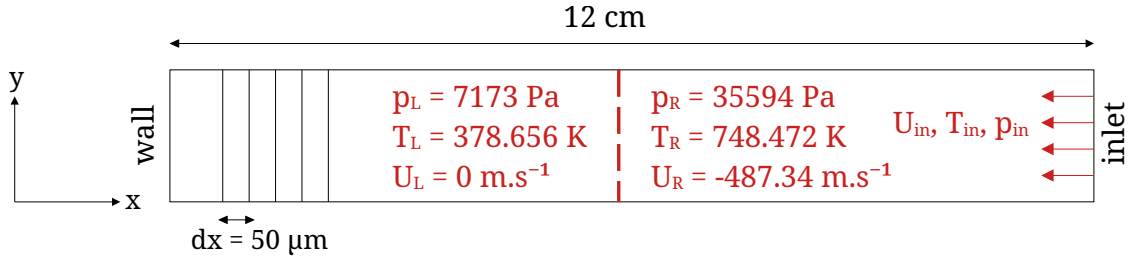

Figure 5.3: Numerical domain of the one-dimensional reacting shock tube case, with initial conditions in red, boundaries in black and schematic grid discretization.

The mixture consists of a 2:1:7 molar ratio of $H_2 : O_2 :$ Ar. Initial conditions are $(\rho_L,\ u_L,\ p_L) = (0.072\,\mathrm{kg/m^3},\ 0\,\mathrm{m/s},\ 7173\,\mathrm{Pa})$ and $(\rho_R, u_R, p_R) = (0.18075\,\mathrm{kg/m^3},\ -487.34\,\mathrm{m/s},\ 35594\,\mathrm{Pa})$, with the left and right states initially separated at the midpoint of a 12 cm domain. The domain is discretized with 2400 uniform grid points, with wall boundary conditions at $x = 0\,\mathrm{cm}$ and `fixedValue`

boundary conditions at the inlet at $x = 12\,\text{cm}$. The convective CFL number is set to 0.1 in order to minimize the numerical diffusion.

5.2.2 Thermophysical Files

reactingRhoCentralFoam/run/reactingShockTube/thermophysicalProperties

```
thermoType
{
    type            hePsiThermo;
    mixture         reactingMixture;
    transport       sutherland;
    thermo          janaf;
    energy          sensibleInternalEnergy;
    equationOfState perfectGas;
    specie          specie;
}

inertSpecie         AR;

chemistryReader     foamChemistryReader;

foamChemistryFile   "<constant>/reactions";

foamChemistryThermoFile "<constant>/thermo";
```

The `thermoType` dictionary is similar to the file from the previous case, with two differences: the `transport` keyword is set to `sutherland`, which calculates viscosity and thermal conductivity based on Sutherland's law [22], suitable for temperature-dependent gas properties, and the `thermo` keyword is set to `janaf`, which uses JANAF polynomials [23] for temperature-dependent thermodynamic property calculations, ideal for detailed chemical reactions.

Unlike the previous non-reacting case, this simulation involves chemical reactions of combustion, requiring a proper chemistry mechanism. The mechanism of Li et al. [24] for hydrogen combustion is provided in CHEMKIN format and will be used by the solver and associated libraries to compute chemical reaction rates. To use this CHEMKIN mechanism with the OpenFOAM-native chemistry file reader, the `chemkinToFoam` utility is employed to convert the CHEMKIN files into the OpenFOAM format. This involves the following steps:

- Prepare the CHEMKIN input files, including the reaction mechanism (`chem.inp`) and thermodynamic properties (`therm.dat`).
- Run `chemkinToFoam` in the desired OpenFOAM case directory, providing the CHEMKIN files as input.
- The utility generates two output files: `reaction` (defining chemical reactions) and `thermo` (containing species thermophysical properties), which are then stored in the `constant` folder of the case.

The commands to generate the `reactions` and `thermo` files as well as the `thermo` and `reactions` are not shown here, but the reader can refer to them in the appendix A.

5.2.3 Initial Files

The initial fields for pressure `p`, temperature `T`, velocity `U`, and species mass fractions are not described here, as their setup is relatively straightforward. The `setFields` utility is used to assign the appropriate values to the left and right halves of the domain.

5.2.4 Results

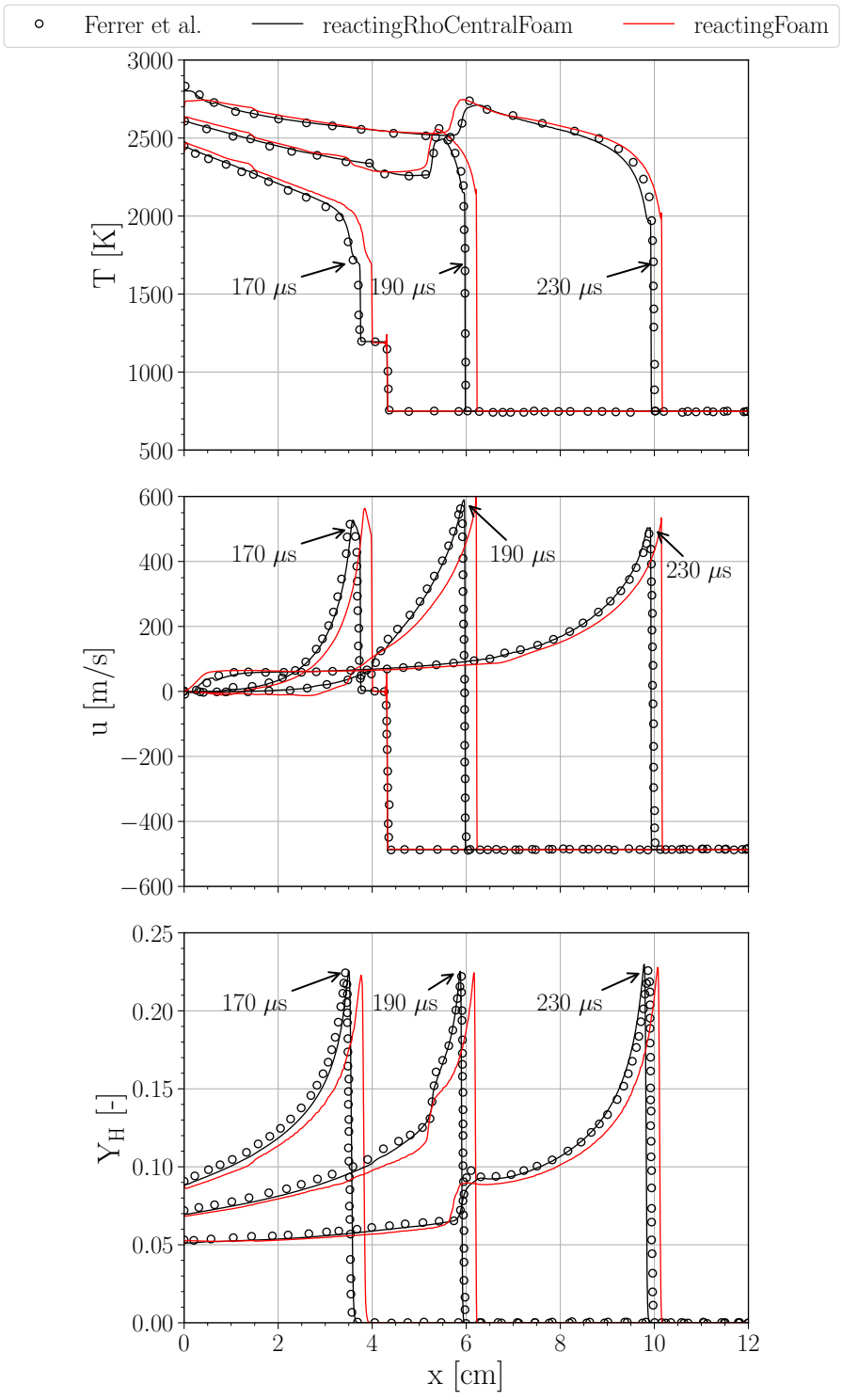

Figure 5.4: Multi-species reacting shock tube of H2/O2/Ar mixture with a discontinuity initially placed at 6 cm. Profiles of Temperature (Top), velocity (Middle) and mass fraction of hydrogen (Bottom).

In Fig 5.4, we show results for temperature, velocity and hydrogen mass fraction obtained at three different times, 170 µs, 190 µs and 230 µs. We compare our simulation with a reference solution from Ferrer et al. [21] and with `reactingFoam`. The `reactingRhoCentralFoam` solver shows excellent agreement with the reference solution across all analyzed fields. In contrast, the second order linear spatial schemes and implicit time-stepping employed by `reactingFoam` introduce significant numerical diffusion, leading to smeared gradients and reduced accuracy in resolving shock waves. While increasing the number of outer and inner iterations in `reactingFoam` enhances accuracy by mitigating some of the numerical diffusion, it comes at the cost of a substantial increase in computational time.

Chapter 6

Conclusion

In this work, we successfully extended the density-based solver `rhoCentralFoam` to include reacting flow capabilities, resulting in the development of the `reactingRhoCentralFoam` solver. Through a detailed theoretical formulation and implementation, the solver was validated against well-established test cases, including non-reacting and reacting flows. The results demonstrated excellent agreement with reference solutions, confirming the accuracy and reliability of the implemented framework.

The developed solver provides a robust tool for simulating high-speed compressible flows with chemical reactions, offering improved capabilities for addressing challenges such as shock interactions, wave dynamics, and combustion phenomena. Compared to `reactingFoam`, the new solver showed superior performance in the supersonic reacting case, accurately capturing sharp gradients and shocks that `reactingFoam` could not resolve effectively due to its numerical schemes, which are not specifically designed for handling such features. While the explicit nature of the solver imposes certain stability constraints, it also enables precise capturing of sharp gradients and facilitates the incorporation of advanced spatial and temporal schemes.

Future work could explore the optimization of the solver for low-speed flows, as well as the integration of more complex turbulence-chemistry interaction models to further enhance its versatility. The implementation and supporting materials for this work will also be available at the following Git address: https://git.tu-berlin.de/pkandel/reactingRhoCentralFoam/-/tree/main.

Bibliography

[1] A. Kurganov, S. Noelle, and G. Petrova, "Semidiscrete central-upwind schemes for hyperbolic conservation laws and hamilton–jacobi equations," *SIAM Journal on Scientific Computing*, vol. 23, no. 3, pp. 707–740, 2001.

[2] J. H. Ferziger and M. Perić, *Computational methods for fluid dynamics*. Springer, 2002.

[3] F. Moukalled, L. Mangani, M. Darwish, F. Moukalled, L. Mangani, and M. Darwish, *The finite volume method*. Springer, 2016.

[4] T. Poinsot, "Theoretical and numerical combustion," *RT Edwards*, 2005.

[5] L. Caretto, A. Gosman, S. Patankar, and D. Spalding, "Two calculation procedures for steady, three-dimensional flows with recirculation," in *Proceedings of the Third International Conference on Numerical Methods in Fluid Mechanics: Vol. II Problems of Fluid Mechanics*. Springer, 1973, pp. 60–68.

[6] R. I. Issa, A. Gosman, and A. Watkins, "The computation of compressible and incompressible recirculating flows by a non-iterative implicit scheme," *Journal of Computational Physics*, vol. 62, no. 1, pp. 66–82, 1986.

[7] T. J. Poinsot and S. Lelef, "Boundary conditions for direct simulations of compressible viscous flows," *Journal of computational physics*, vol. 101, no. 1, pp. 104–129, 1992.

[8] C. J. Greenshields, H. G. Weller, L. Gasparini, and J. M. Reese, "Implementation of semi-discrete, non-staggered central schemes in a colocated, polyhedral, finite volume framework, for high-speed viscous flows," *International journal for numerical methods in fluids*, vol. 63, no. 1, pp. 1–21, 2010.

[9] M. Kraposhin, A. Bovtrikova, and S. Strijhak, "Adaptation of kurganov-tadmor numerical scheme for applying in combination with the piso method in numerical simulation of flows in a wide range of mach numbers," *Procedia Computer Science*, vol. 66, pp. 43–52, 2015.

[10] P. L. Roe, "Characteristic-based schemes for the euler equations," *Annual review of fluid mechanics*, vol. 18, no. 1, pp. 337–365, 1986.

[11] B. Van Leer, "Towards the ultimate conservative difference scheme. ii. monotonicity and conservation combined in a second-order scheme," *Journal of computational physics*, vol. 14, no. 4, pp. 361–370, 1974.

[12] Choquet, "Thermophysicalmodels library in openfoam-2.3.x (or 2.4.x)," http://www.tfd.chalmers.se/~hani/kurser/OS_CFD/.

[13] V. Kornilov, R. Rook, J. ten Thije Boonkkamp, and L. De Goey, "Experimental and numerical investigation of the acoustic response of multi-slit bunsen burners," *Combustion and Flame*, vol. 156, no. 10, pp. 1957–1970, 2009.

[14] C. Andersen, "Numerical investigation of a bfr using openfoam," https://projekter.aau.dk/projekter/files/14411784/Report.pdf.

[15] S. M. Mousavi, "Combination of reactingfoam and chtmultiregionfoam as a first step toward creating a multiregionreactingfoam, suitable for solid/gas phase," https://www.tfd.chalmers.se/~hani/kurser/OS_CFD_2019/Seyed_Morteza_Mousavi/FinalReport_S_M_Mousavi_CFDWithOSS.pdf, 2019.

[16] A. Äkerblom, "Turbulence-chemistry interaction in openfoam and how to implement a dynamic pasr model for les of turbulent combustion," https://www.tfd.chalmers.se/~hani/kurser/OS_CFD_2022/ArvidAkerblom/OF_Report_Arvid_Final.pdf, 2022.

[17] M. Bertsch, "Description of the reacting flow solver fgmfoam," https://www.tfd.chalmers.se/~hani/kurser/OS_CFD_2019/Michael_Bertsch/report_FGMFoam.pdf, 2019.

[18] A. Lundström, "Simple gas phase reaction," https://www.tfd.chalmers.se/~hani/kurser/OS_CFD_2007/AndreasLundstrom/reactingFoam.pdf, 2007.

[19] H. E., "Combining a density-based compressible solver with a multiphase model," https://www.tfd.chalmers.se/~hani/kurser/OS_CFD_2020/EleanorHarvey/Project_Report.pdf, 2020.

[20] A. F. Emery, "An evaluation of several differencing methods for inviscid fluid flow problems," *Journal of Computational Physics*, vol. 2, no. 3, pp. 306–331, 1968.

[21] P. J. M. Ferrer, R. Buttay, G. Lehnasch, and A. Mura, "A detailed verification procedure for compressible reactive multicomponent navier–stokes solvers," *Computers & Fluids*, vol. 89, pp. 88–110, 2014.

[22] W. Sutherland, "Lii. the viscosity of gases and molecular force," *The London, Edinburgh, and Dublin Philosophical Magazine and Journal of Science*, vol. 36, no. 223, pp. 507–531, 1893.

[23] D. R. Stull, *JANAF Thermochemical Tables*. Clearinghouse, 1965, vol. 1.

[24] J. Li, Z. Zhao, A. Kazakov, and F. L. Dryer, "An updated comprehensive kinetic model of hydrogen combustion," *International journal of chemical kinetics*, vol. 36, no. 10, pp. 566–575, 2004.

Study questions

1. What are the key differences between pressure-based and density-based solvers?
2. What are the key differences between `reactingFoam`, and `rhoCentralFoam`?
3. How are the advective fluxes calculated in `rhoCentralFoam`?
4. How to add reacting species transport equations to `rhoCentralFoam` in OpenFOAM?

Appendix A

reactingShockTube Additional Files

A.1 Allpre

reactingRhoCentralFoam/run/Allpre

```
#!/bin/sh
cd "${0%/*}" || exit                                # Run from this directory
. ${WM_PROJECT_DIR:?}/bin/tools/RunFunctions        # Tutorial run functions
#------------------------------------------------------------------------------

runApplication chemkinToFoam \
               chemkin/chem.inp chemkin/therm.dat chemkin/transportProperties \
               constant/reactions constant/thermo

cp -r 0.orig/ 0/
runApplication blockMesh
runApplication setFields
runApplication decomposePar

#------------------------------------------------------------------------------
```

A.2 thermo

reactingRhoCentralFoam/run/constant/thermo

```
OH
{
    specie
    {
        molWeight       17.00737;
    }
    thermodynamics
    {
        Tlow            200;
        Thigh           6000;
        Tcommon         1000;
        highCpCoeffs    ( 2.86472886 0.00105650448 -2.59082758e-07 3.05218674e-11 -1.33195876e-15
     3718.85774 5.70164073 );
        lowCpCoeffs     ( 4.12530561 -0.00322544939 6.52764691e-06 -5.79853643e-09 2.06237379e-12
     3381.53812 -0.69043296 );
    }
    transport
    {
        As              0;
        Ts              0;
    }
```

```
    elements
    {
        H               1;
        O               1;
    }
}

H2O2
{
    specie
    {
        molWeight       34.01474;
    }
    thermodynamics
    {
        Tlow            200;
        Thigh           3500;
        Tcommon         1000;
        highCpCoeffs    ( 4.16500285 0.00490831694 -1.90139225e-06 3.71185986e-10 -2.87908305e-14
     -17861.7877 2.91615662 );
        lowCpCoeffs     ( 4.27611269 -0.000542822417 1.67335701e-05 -2.15770813e-08 8.62454363e-12
     -17702.5821 3.43505074 );
    }
    transport
    {
        As              0;
        Ts              0;
    }
    elements
    {
        H               2;
        O               2;
    }
}

O2
{
    specie
    {
        molWeight       31.9988;
    }
    thermodynamics
    {
        Tlow            200;
        Thigh           3500;
        Tcommon         1000;
        highCpCoeffs    ( 3.28253784 0.00148308754 -7.57966669e-07 2.09470555e-10 -2.16717794e-14
     -1088.45772 5.45323129 );
        lowCpCoeffs     ( 3.78245636 -0.00299673416 9.84730201e-06 -9.68129509e-09 3.24372837e-12
     -1063.94356 3.65767573 );
    }
    transport
    {
        As              0;
        Ts              0;
    }
    elements
    {
        O               2;
    }
}

H2
{
    specie
    {
        molWeight       2.01594;
    }
```

```
    thermodynamics
    {
        Tlow            200;
        Thigh           3500;
        Tcommon         1000;
        highCpCoeffs    ( 3.3372792 -4.94024731e-05 4.99456778e-07 -1.79566394e-10 2.00255376e-14
     -950.158922 -3.20502331 );
        lowCpCoeffs     ( 2.34433112 0.00798052075 -1.9478151e-05 2.01572094e-08 -7.37611761e-12
     -917.935173 0.683010238 );
    }
    transport
    {
        As              0;
        Ts              0;
    }
    elements
    {
        H               2;
    }
}

HO2
{
    specie
    {
        molWeight       33.00677;
    }
    thermodynamics
    {
        Tlow            200;
        Thigh           3500;
        Tcommon         1000;
        highCpCoeffs    ( 4.0172109 0.00223982013 -6.3365815e-07 1.1424637e-10 -1.07908535e-14
     111.856713 3.78510215 );
        lowCpCoeffs     ( 4.30179801 -0.00474912051 2.11582891e-05 -2.42763894e-08 9.29225124e-12
     294.80804 3.71666245 );
    }
    transport
    {
        As              0;
        Ts              0;
    }
    elements
    {
        H               1;
        O               2;
    }
}

O
{
    specie
    {
        molWeight       15.9994;
    }
    thermodynamics
    {
        Tlow            200;
        Thigh           3500;
        Tcommon         1000;
        highCpCoeffs    ( 2.56942078 -8.59741137e-05 4.19484589e-08 -1.00177799e-11 1.22833691e-15
     29217.5791 4.78433864 );
        lowCpCoeffs     ( 3.1682671 -0.00327931884 6.64306396e-06 -6.12806624e-09 2.11265971e-12
     29122.2592 2.05193346 );
    }
    transport
    {
        As              0;
```

```
        Ts              0;
    }
    elements
    {
        O               1;
    }
}

H2O
{
    specie
    {
        molWeight       18.01534;
    }
    thermodynamics
    {
        Tlow            200;
        Thigh           3500;
        Tcommon         1000;
        highCpCoeffs    ( 3.03399249 0.00217691804 -1.64072518e-07 -9.7041987e-11 1.68200992e-14
     -30004.2971 4.9667701 );
        lowCpCoeffs     ( 4.19864056 -0.0020364341 6.52040211e-06 -5.48797062e-09 1.77197817e-12
     -30293.7267 -0.849032208 );
    }
    transport
    {
        As              0;
        Ts              0;
    }
    elements
    {
        H               2;
        O               1;
    }
}

H
{
    specie
    {
        molWeight       1.00797;
    }
    thermodynamics
    {
        Tlow            200;
        Thigh           3500;
        Tcommon         1000;
        highCpCoeffs    ( 2.50000001 -2.30842973e-11 1.61561948e-14 -4.73515235e-18 4.98197357e-22
     25473.6599 -0.446682914 );
        lowCpCoeffs     ( 2.5 7.05332819e-13 -1.99591964e-15 2.30081632e-18 -9.27732332e-22 25473.6599
      -0.446682853 );
    }
    transport
    {
        As              0;
        Ts              0;
    }
    elements
    {
        H               1;
    }
}

AR
{
    specie
    {
        molWeight       39.948;
```

```
    }
    thermodynamics
    {
        Tlow            200;
        Thigh           5000;
        Tcommon         1000;
        highCpCoeffs    ( 2.5 0 0 0 0 -745.375 4.366 );
        lowCpCoeffs     ( 2.5 0 0 0 0 -745.375 4.366 );
    }
    transport
    {
        As              0;
        Ts              0;
    }
    elements
    {
        Ar              1;
    }
}
```

A.3 reactions

reactingRhoCentralFoam/run/constant/thermo

```
elements        3(H O Ar);

species         9(H2 O2 H O OH HO2 H2O2 H2O AR);

reactions
{
    un-named-reaction-0
    {
        type            reversibleArrheniusReaction;
        reaction        "H2 + O2 = 2OH";
        A               1.7e+10;
        beta            0;
        Ta              24042.47739;
    }
    un-named-reaction-1
    {
        type            reversibleArrheniusReaction;
        reaction        "H2 + OH = H + H2O";
        A               1170000;
        beta            1.3;
        Ta              1824.571432;
    }
    un-named-reaction-2
    {
        type            reversibleArrheniusReaction;
        reaction        "H + O2 = O + OH";
        A               5.13e+13;
        beta            -0.816;
        Ta              8306.177779;
    }
    un-named-reaction-3
    {
        type            reversibleArrheniusReaction;
        reaction        "H2 + O = H + OH";
        A               18000000;
        beta            1;
        Ta              4441.165874;
    }
    un-named-reaction-4
    {
        type            reversiblethirdBodyArrheniusReaction;
```

```
        reaction        "H + O2 = HO2";
        A               2.1e+12;
        beta            -1;
        Ta              0;
        coeffs
9
(
(H2 3.3)
(O2 0)
(H 1)
(O 1)
(OH 1)
(HO2 1)
(H2O2 1)
(H2O 21)
(AR 1)
)
;
    }
    un-named-reaction-5
    {
        type            reversibleArrheniusReaction;
        reaction        "H + O2 + O2 = HO2 + O2";
        A               6.7e+13;
        beta            -1.42;
        Ta              0;
    }
    un-named-reaction-6
    {
        type            reversibleArrheniusReaction;
        reaction        "HO2 + OH = H2O + O2";
        A               5e+10;
        beta            0;
        Ta              503.1912388;
    }
    un-named-reaction-7
    {
        type            reversibleArrheniusReaction;
        reaction        "H + HO2 = 2OH";
        A               2.5e+11;
        beta            0;
        Ta              956.0633538;
    }
    un-named-reaction-8
    {
        type            reversibleArrheniusReaction;
        reaction        "HO2 + O = O2 + OH";
        A               4.8e+10;
        beta            0;
        Ta              503.1912388;
    }
    un-named-reaction-9
    {
        type            reversibleArrheniusReaction;
        reaction        "2OH = H2O + O";
        A               600000;
        beta            1.3;
        Ta              0;
    }
    un-named-reaction-10
    {
        type            reversiblethirdBodyArrheniusReaction;
        reaction        "H2 = 2H";
        A               2230000000;
        beta            0.5;
        Ta              46595.50872;
        coeffs
9
```

```
(
(H2 3)
(O2 1)
(H 2)
(O 1)
(OH 1)
(HO2 1)
(H2O2 1)
(H2O 6)
(AR 1)
)
;
    }
    un-named-reaction-11
    {
        type            reversiblethirdBodyArrheniusReaction;
        reaction        "O2 = 2O";
        A               185000000;
        beta            0.5;
        Ta              48084.95478;
        coeffs
9
(
(H2 1)
(O2 1)
(H 1)
(O 1)
(OH 1)
(HO2 1)
(H2O2 1)
(H2O 1)
(AR 1)
)
;
    }
    un-named-reaction-12
    {
        type            reversiblethirdBodyArrheniusReaction;
        reaction        "H + OH = H2O";
        A               7.5e+17;
        beta            -2.6;
        Ta              0;
        coeffs
9
(
(H2 1)
(O2 1)
(H 1)
(O 1)
(OH 1)
(HO2 1)
(H2O2 1)
(H2O 20)
(AR 1)
)
;
    }
    un-named-reaction-13
    {
        type            reversibleArrheniusReaction;
        reaction        "H + HO2 = H2 + O2";
        A               2.5e+10;
        beta            0;
        Ta              352.2338672;
    }
    un-named-reaction-14
    {
        type            reversibleArrheniusReaction;
```

```
        reaction        "2HO2 = H2O2 + O2";
        A               2000000000;
        beta            0;
        Ta              0;
    }
    un-named-reaction-15
    {
        type            reversiblethirdBodyArrheniusReaction;
        reaction        "H2O2 = 2OH";
        A               1.3e+14;
        beta            0;
        Ta              22895.20137;
        coeffs
9
(
(H2 1)
(O2 1)
(H 1)
(O 1)
(OH 1)
(HO2 1)
(H2O2 1)
(H2O 1)
(AR 1)
)
;
    }
    un-named-reaction-16
    {
        type            reversibleArrheniusReaction;
        reaction        "H + H2O2 = H2 + HO2";
        A               1600000000;
        beta            0;
        Ta              1912.126708;
    }
    un-named-reaction-17
    {
        type            reversibleArrheniusReaction;
        reaction        "H2O2 + OH = H2O + HO2";
        A               1e+10;
        beta            0;
        Ta              905.7442299;
    }
}
```