Cite as: Sehole, HAH.: Implementation of the Dynamically Thickened Flame LES Model for Premixed and Non-Premixed Turbulent Combustion in OpenFOAM . 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

Implementation of the Dynamically Thickened Flame LES Model for Premixed and Non-Premixed Turbulent Combustion in OpenFOAM

Developed for OpenFOAM-v10

Author:
Hafiz Ali Haider Sehole
Aalto University
ali.haider@aalto.fi

Peer reviewed by:
Ville Vuorinen
Saeed salehi
Paria Khosravifar

Library on GitHub: https://github.com/hahspk/DTF

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.

February 4, 2025

Learning outcomes

The main requirements of a tutorial in the course are 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:

- How to use the turbulence-chemistry interaction (TCI) model in OpenFOAM.
- How to use the new TCI model, Dynamic Thickened Flame (DTF), when setting up a case.

The theory of it:

- The role of turbulence-chemistry interaction (TCI) models and how they impact the chemical of combustion simulations.
- Overview of TCI models available in OpenFOAM, along with their advantages and limitations.
- The theory behind the modification of reaction rates and diffusion coefficients in the species transport equation and their connection with TCI models.
- Description of calculating the reaction rates and diffusion coefficient modification for DTF. Details in the modification section.

How it is implemented:

- In OpenFOAM, reactive flow solvers obtain chemical reaction information through integrated chemistry models, which compute species concentrations, reaction rates, and heat release.
- A TCI model in OpenFOAM is required by the solver to handle the coupling between turbulent eddy structures and chemical reaction rates.

How to modify it:

- Implementing a new TCI (Turbulence-Chemistry Interaction) model in OpenFOAM requires two modifications to the species transport equation:

  1. Adjustment of the reaction rate source term by multiplying it with the efficiency function (E) and dividing it by the thickening factor (F).
  2. Modification of the diffusion coefficient by multiplying it with the efficiency function (E) and the thickening factor (F).

  The PaSR model serves as a suitable template for this implementation, as it already includes the necessary framework for modifying the reaction rate source term.

- Guidance on setting up a simple, coarse LES (Large Eddy Simulation) case to test the newly implemented TCI model.

Prerequisites

To gain the most benefit from this report, the reader is expected to have:

- A foundational understanding of fluid mechanics, combustion, and turbulence
- Familiarity with CFD (Computational Fluid Dynamics) and turbulence modeling, particularly Large Eddy Simulation (LES)
- Understanding of turbulence-chemistry interaction models in OpenFOAM, especially PaSR.
- Thickened Flame Model code [1] implemented by Rintanen [2] during his thesis work.
- Experience in setting up and running cases in OpenFOAM
- Basic knowledge of object-oriented programming and C++ syntax
- The ability to locate files, classes, and functions within the OpenFOAM source code

Contents

Nomenclature

Acronyms

DTF	Dynamically Thickened Flame
LES	Large Eddy Simulation
PaSR	Partially Stirred Reactor
TCI	Turbulence-Chemistry Interaction

English symbols

$\vec{j}_i$	Diffusive flux of species i	$\mathrm{kg/m^2/s}$
$\vec{v}$	Velocity vector	$\mathrm{m/s}$
c_p	Specific heat capacity at constant pressure	$\mathrm{J/(kg \cdot K)}$
c_{ms}	Wrinkling factor model constant	
D	Thermal diffusivity	$\mathrm{m^2/s}$
D_i	Diffusion coefficient for species i	$\mathrm{m^2/s}$
D_{iF}	Modified diffusion coefficient (DTF model)	$\mathrm{m^2/s}$
E	Efficiency function	
F	Flame thickening factor	
F_s	Dynamically calculated thickening factor	
$F_{\max}$	Maximum thickening factor	
h	Heat release rate	$\mathrm{J/s}$
$h_{\max}$	Maximum heat release rate	$\mathrm{J/s}$
Le_i	Lewis number for species i	
R_{iF}	Modified reaction rate (DTF model)	$\mathrm{kg/m^3/s}$
Re_t	Turbulent Reynolds number	
S_l^0	Premixed laminar flame speed	$\mathrm{m/s}$
u'	Subgrid-scale velocity fluctuations	$\mathrm{m/s}$
Y_i	Mass fraction of species i	

Greek symbols

α	Adjustment factor for efficiency function	
β	Scaling parameter for flame sensor	
Δ	Filter width	m
Δ_e	Test filter width	m
δ_g	Geometric cell size	m
δ_l	Laminar flame thickness	m
δ_l^0	Unthickened laminar flame thickness	m
δ_l^1	Thickened flame thickness	m
ϵ	Turbulent dissipation rate	$\mathrm{m^2/s^3}$
Γ	Stretch function	
κ	Reacting volume fraction	
λ	Thermal conductivity	$\mathrm{W/(m \cdot K)}$
ν_{eff}	Effective viscosity	$\mathrm{kg/(m \cdot s)}$

ρ	Density	$\mathrm{kg/m^3}$
τ_c	Chemical time scale	s
τ_k	Mixing time scale	s
Ξ	Wrinkling factor	

Chapter 1

Introduction

1.1 Motivation and Background

Turbulent combustion is one of the most challenging problems in physics due to the turbulence and highly non-linear nature of chemical kinetics. To solve practical problems, engineers use turbulence models that combine mathematical equations with experimental data, making them part science and part approximation. These models rely on the concept that large vortices break down into smaller ones until they dissipate, as in Large Eddy Simulations (LES) .

LES achieves a balance between computational cost and accuracy by resolving the larger scales of turbulence while modeling the smaller, sub-grid scales. Pope [3] recommended resolving at least 80% of the turbulent kinetic energy to ensure realistic large-scale results. To enhance accuracy, the LES filter width can be reduced, approaching the Kolmogorov scales [4], which decreases the influence of unresolved scales and reduces reliance on sub-grid models. However, as long as turbulence remains unresolved, sub-grid turbulence models are required. LES offers a range of well-validated sub-grid turbulence models [5], such as the WALE model [6] , which performs well for many engineering applications.

Combining turbulence with combustion introduces additional complexities. Combustion requires the fuel and oxidizer to mix at the molecular level, which occurs through turbulent mixing. As eddies of different sizes break down into smaller ones, strain and shear at their interfaces enhance mixing and molecular diffusion [7]. In the presence of a flame, heat and radicals diffuse from the reaction zone, steepening gradients further. However, chemical reactions at these interfaces may alter the mixing process in ways that are not fully understood, as combustion involves multiple reactions occurring on different time scales. It is the reason LES requires sub-grid combustion models to account for unresolved small-scale turbulence and its interaction with chemical reactions. In OpenFOAM, the known combustion models [8] for large eddy simulations are as follows,

- **Laminar**: The effect of turbulent fluctuations on kinetic rates is neglected, and reaction rates are determined by general finite-rate chemistry directly.
- **Eddy-Dissipation Concept (EDC)**: It assumes that reactions occur in regions of the flow where turbulence kinetic energy dissipation takes place. [9, 10, 11, 12].
- **Partially Stirred Reactor (PaSR)**: Computes reaction rates considering both turbulent mixing and chemical timescales.

The PaSR model serves as a approach for simulating turbulent premixed and partially premixed flames in OpenFOAM. The PaSR is a model used in LES to approximate the combustion process. It helps simulate how the turbulent fluid mixes and reacts chemically without having to resolve every small-scale turbulent structure in the simulation. The PaSR model requires a skeletal mechanism for fuel/air combustion and the transport properties of all species. It also includes a closure model for the filtered source terms that arise from LES. The consumption rates for each reaction are

determined by both the mixing time scale and the chemical time scale, which are computed using global flame parameters and the turbulent time scale, respectively.

In comparison to PaSR, the newly implemented LES subgrid-scale turbulent combustion model in OpenFOAM, Dynamically Thickened Flame (DTF) [13], is capable of handling combustion regimes that fall between perfectly premixed and non-premixed conditions. Unlike other models, DTF does not rely on prior assumptions about the flame structure, enabling it to accurately simulate scenarios where premixed and non-premixed combustion coexist. The theoretical framework of the DTF model is detailed in Chapter 2, Section 2.3.3. DTF requires two modifications to the species transport equation, Adjustment of the reaction source term by multiplying it with efficiency function (E) and divided by thickening factor (F). Modification of the diffusion coefficient by multiplying it with an efficiency function (E) and a thickening factor (F). In comparison to other TCI models available in OpenFOAM that are widely used for combustion simulations, their capability difference is represented in Table 1.1.

Table 1.1: Comparison of `DTF` with other TCI Models

TCI Model	Non-Premixed	Partially-Premixed	Premixed
Laminar	✓		
EDC	✓	✓	
PaSR		✓	✓
DTF	✓	✓	✓

From Table 1.1, the Laminar model is primarily used for non-premixed combustion in OpenFOAM, though it can also be applied to premixed flames in Direct Numerical Simulation (DNS). Similarly, the functionality of the Eddy-Dissipation Concept (EDC) and Partially Stirred Reactor (PaSR) models for different flow regimes is summarized in Table 1.1, as per OpenFOAM's documentation [14]. The Dynamic Thickened Flame (DTF) model [13], proposed in this report, represents an implementation in OpenFOAM capable of handling multi-regime flows.

The PaSR implementation serves as the optimal template for implementing the DTF model. Analogous to PaSR (see Eq. 2.5), where the reaction rate source term is scaled by κ, the DTF model scales the reaction rate source term by $\frac{E}{F}$. Additionally, the DTF model necessitates reimplementation of the `ThermophysicalTransportModels` library, where the product (EF) scales the diffusion coefficient.

1.2 Report Outline

In Chapter 2, the required theory related to the species transport equation is given, and further it explains the diffusion coefficient and reaction rate source term calculation using different TCI models. The details of TCI models Laminar, PaSR and DTF are explained in a subsequent section. Chapter 3, explains the PaSR implementation in OpenFOAM. Chapter 4 describes the implementation of DTF over the existing code of TFMFoam. The case setup and results are described in Chapter 5.

Chapter 2

Theory

2.1 Governing Equations

Combustion is the process of converting chemical energy stored in fossil fuels into heat through chemical reactions. The governing equations for any combustion model are based on the balance laws for energy and chemical species. For a detailed derivation of these equations, readers are referred to Williams [15]. The conservation equation for species i is expressed as

$$\frac{\partial}{\partial t}(\rho Y_i) + \nabla \cdot (\rho \overrightarrow{v} Y_i) = -\nabla \cdot \overrightarrow{j_i} + R_i, \tag{2.1}$$

where ρ is the mixture density, $\overrightarrow{v}$ is the velocity vector, R_i is the net production rate of species i due to chemical reactions, and $\overrightarrow{j_i}$ represents the diffusive flux of species i.

2.2 Diffusion

The diffusive flux $\overrightarrow{j_i}$ in Eq.2.1 is driven by gradients in concentration and temperature. The molecular transport process that causes these fluxes is complex, and a detailed explanation can be found in Williams [15]. In turbulent combustion, however, turbulent transport dominates over molecular transport, making it practical to use simplified representations of diffusive fluxes. One common simplification is the binary flux or mass diffusion approximation

$$\overrightarrow{j_i} = -\rho D_i \nabla Y_i, \tag{2.2}$$

where Y_i is the mass fraction of species i and D_i is the mass diffusion coefficient of species i in the mixture. However, in multicomponent systems, this approximation can violate mass conservation if the diffusion coefficients (D_i) are not equal. This occurs because the sum of all fluxes must vanish, and the total of all mass fractions must equal unity. Although Eq.2.2 is convenient for notation, it is not suitable for laminar flame calculations. To simplify further, it is often assumed that all diffusion coefficients D_i are proportional to the thermal diffusivity D, given by

$$D = \frac{\lambda}{\rho c_p}, \tag{2.3}$$

where λ is the thermal conductivity and c_p is the specific heat capacity at constant pressure. Using this assumption, the Lewis number L_{e_i} for species i is defined as

$$L_{e_i} = \frac{\lambda}{\rho c_p D_i} = \frac{D}{D_i}. \tag{2.4}$$

The Lewis number is assumed to remain unity. The current implementation of the Dynamic Thickened Flame (DTF) model uses a unity Lewis number approximation, where the diffusion coefficient

is modeled using `unityLewisEddyDiffusivity` model. Further, this report is not further getting into the details of energy balance; the reader can refer to Peters [7].

2.3 Reaction Rate

The reaction rate source term R_i from Eq. 2.1 can be computed using TCI models. In OpenFOAM, reaction rates are calculated using different TCI models, such as Laminar, EDC, and PaSR. This report will focus on the Laminar and PaSR models. A solid understanding of these models in the context of OpenFOAM will assist readers in implementing a new TCI model called DTF. This section describes the equations used to calculate the reaction rate source term in these TCI models.

2.3.1 Laminar Model

The laminar model represents the simplest approach to modeling reaction rates. The model assumes each computational cell is treated as a homogeneous reactor, and the reaction rate R_i is taken to be identical to the unfiltered reaction rate $\overline{R_i} = R_i$. This assumption is valid for laminar flows or turbulent flows where the computational grid is fine enough to resolve all turbulence-chemistry interactions. In OpenFOAM, this approach is implemented in the laminar class under the namespace of combustionModel.

2.3.2 Partially Stirred Reactor (PaSR)

The Partially Stirred Reactor (PaSR) from OpenFOAM [16] model modifies the reaction source term by kappa (κ), a reacting volume fraction based on the ratio of chemical and turbulent mixing timescales. The reacting source term is defined as

$$R_i = \kappa \times R_i(Y_i), \tag{2.5}$$

where κ is

$$\kappa = \frac{\tau_c}{\tau_c + \tau_k}, \tag{2.6}$$

where R_i is the reaction rate of specie i, Y_i is the specie that passes to reaction rate, τ_c is the chemical time scale, and τ_k represents the turbulent mixing time scale. The τ_k is calculated as the Kolmogorov time scale, it is defined in PaSR [16] at line 93

$$\tau_k = C_{mix}\sqrt{\frac{\nu_{\text{eff}}}{\epsilon}}, \tag{2.7}$$

where C_{mix} is the constant value provided by the user, default is 1, ν_{eff} is the effective viscosity accounting for turbulence, and ϵ is the turbulent dissipation rate. The chemical time scale τ_c which is implemented in OpenFOAM defined in `chemistryModel` [17, 18] at line 446 is derived based on reaction kinetics

$$\tau_c = \frac{c_{\text{tot}}}{\sum_{j=1}^{n_R}\sum_{i=1}^{N_{s,\text{RHS}}}\nu_{i,j}k_{f,j}}, \tag{2.8}$$

where c_{tot} is the total species concentration, n_R is the number of chemical reactions, $N_{s,\text{RHS}}$ is the number of product species, $\nu_{i,j}$ is the stoichiometric coefficient, and $k_{f,j}$ is the forward reaction rate constant. The reacting volume fraction κ is updated as

$$\kappa = \begin{cases} 1.0 & \text{if } \tau_k < \text{small}, \\ \frac{\tau_c}{\tau_c+\tau_k} & \text{otherwise.} \end{cases} \tag{2.9}$$

This approach is particularly used for modeling of premixed and partially premixed flames.

2.3.3 Dynamic Thickened Flame (DTF)

The DTF model is based on the thickened flame (TF) approach as described by Legier et al. [13], which is traditionally used to make flames resolvable on coarse grids in large eddy simulations (LES). In the TF approach, the flame thickness is artificially increased by a factor F while maintaining the correct flame propagation speed. This is achieved by multiplying the species and thermal diffusion coefficients by F and reducing the reaction rate source term by the same factor. The species transport equation from Eq.2.1 for the DTF model will be written as

$$\frac{\partial(\rho Y_i)}{\partial t} + \nabla \cdot (\rho \overrightarrow{v} Y_i) = \nabla \cdot (\rho D_{iF} \nabla Y_i) - R_{iF}, \tag{2.10}$$

where, ρ is density, $\overrightarrow{v}$ is velcoity, D_{iF} is modified diffusion coefficient scaled by the thickening factor (F), and R_{iF} is reaction rate source term. D_{iF} and R_{iF} are represented as

$$D_{iF} \rightarrow DF, \quad R_{iF} \rightarrow \frac{1}{F} R_i. \tag{2.11}$$

2.3.3.1 Thickening Factor

The DTF model introduces a dynamic thickening factor (F) that varies both spatially and temporally, depending on the local combustion conditions. A sensor based on heat release, as described in Eq.2.13, detects the presence of the flame front and activates the thickening only within reaction zones. The thickening factor (F) is defined as

$$F = 1 + (min(F_{\max}, F_s) - 1)\Omega, \tag{2.12}$$

where Ω is the heat release sensor, F is the thickening factor, $F_{\max}$ is the maximum thickening factor set by the user and F_s is the thickening factor calculated dynamically over time and space.

2.3.3.2 Flame Sensor

The sensor Ω is based on the heat release rate and is defined as

$$\Omega = tanh(\beta \frac{h}{h_{max}}), \tag{2.13}$$

where h is the heat release rate and h_{max} is the maximum heat release and β controls the transition layer thickness between thickened and non-thickened regions.

2.3.3.3 Efficiency Function

This dynamic adjustment ensures that the flame remains resolvable in the grid while preserving accurate mixing and diffusion characteristics in non-reactive regions. To account for the effects of unresolved turbulence, the DTF model uses an efficiency function (E), which modifies the diffusion coefficients and reaction rate in Eq. 2.14 as

$$D_{iEF} \rightarrow EFD, \quad R_{iEF} \rightarrow \frac{E}{F} R_i, \tag{2.14}$$

where D_{iEF} is the modified diffusion coefficient, R_{iEF} is the modified reaction rate source term, F is the thickening factor, and E is the efficiency function. The efficiency function E is expressed as the ratio between the wrinkling factor Ξ of the real flame $\delta_l = \delta_l^0$ and that of the thickened flame with thickness, $\delta_l^1 = F\delta_l^0$ as

$$E = \frac{\Xi|_{\delta_l = \delta_l^0}}{\Xi|_{\delta_l = \delta_l^1}} \geq 1, \tag{2.15}$$

where δ_l is flame thickness, δ_l^0 is the laminar flame speed, and δ_l^1 is the turbulent flame thickening. To estimate a dimensionless wrinkling factor Ξ, which is proposed by Colin et. al. [19, 20] defined as

the flame surface to its projection in the propagating direction. It depends on velocity fluctuations ($u'_{\Delta e}$) and flame parameters, which include laminar flame speed (δ_l^0), laminar flame thickness (δ_l) turbulent flame thickening (δ_l^1) and thickening Factor (F) as

$$\Xi = 1 + \alpha \frac{u'_{\Delta e}}{S_l^0} \Gamma \left(\frac{\Delta_e}{\delta_l}, \frac{u'_{\Delta e}}{S_l^0} \right), \tag{2.16}$$

where

$$\alpha = \frac{2 \ln 2}{3 c_{ms} \left(\sqrt{Re_t} - 1 \right)}, \quad c_{ms} = 0.28, \tag{2.17}$$

and Γ is a dimensionless stretch function

$$\Gamma \left(\frac{\Delta_e}{\delta_l}, \frac{u'_{\Delta e}}{S_l^0} \right) \approx 0.75 \exp \left[-1.2 \left(\frac{u'_{\Delta e}}{S_l^0} \right)^{-0.3} \right] \left(\frac{\Delta_e}{\delta_l} \right)^{2/3}. \tag{2.18}$$

The test filter scale Δ_e is chosen such that $\Delta_e = \delta_l^1 = F \delta_l^0 > \Delta x$, which ensures the proper flame resolution. The α as desired in Eq. 2.17 is not implemented in the present implementation. α is set by the user as constant value.

2.3.3.4 Laminar Flame Speed

In premixed combustion, the most important quantity is the velocity at which the flame front propagates normal to itself and relative to the flow into the unburnt mixture. This velocity is called the laminar flame speed [21]. It is proportional to the thermal diffusivity (D) and the reaction rate (R). Mathematically, the laminar flame speed is expressed as

$$S_l^0 = \sqrt{DR}, \tag{2.19}$$

where D represents the thermal diffusivity of the species and R represents the total reaction rate of all the species. The thermal diffusivity is given by

$$D = \frac{\lambda}{\rho c_p}, \tag{2.20}$$

where λ is the thermal conductivity, ρ is the density, and c_p is the specific heat capacity. The laminar flame thickness (δ_L) is proportional to $\frac{D}{S_l^0}$ and is expressed as

$$\delta_l = \frac{D}{S_l^0}. \tag{2.21}$$

To simulate the laminar flame on a coarse mesh without altering its speed, the flame can be artificially thickened by proportionally increasing the diffusivity (D) and decreasing the reaction rate ($1/R$). The dynamic thickening factor (F_s) is defined as

$$F_s = \frac{\Delta N}{\delta_l}, \tag{2.22}$$

where Δ is the grid size, δ_L is the laminar flame thickness, and N is the user-specified number of points in the flame. The grid size (Δ) is calculated as

$$\Delta = V^{\frac{1}{N_d}}, \tag{2.23}$$

where V is the cell volume and N_d is the spatial dimension (2D or 3D).

.

Chapter 3

PaSR: OpenFOAM Implementation

The combustion models in OpenFOAM are designed to simulate various types of reactive flows depending on the physical and chemical characteristics of the problem. The `PaSR` model in OpenFOAM is implemented to solve premixed or partially premixed reactive flows. These combustion models reside in the `combustionModels` directory:

`src/combustionModels`

This chapter provides a detailed explanation of the `PaSR` model's implementation, its dependencies, and its role within OpenFOAM. Figure 3.1 illustrates the basic workflow of `PaSR` and its interaction with other components of the OpenFOAM source code.

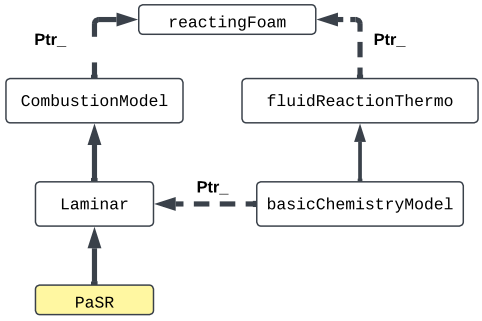

Figure 3.1: The `PaSR` workflow and its implementation structure in OpenFOAM.

As shown in Fig. 3.1, the `PaSR` class inherits from the `laminar` class. The `laminar` class utilize pointer to dynamically allocate memory on the heap during runtime for calculating reaction rates and heat release rates. The `combustionModels` class manages available combustion models based on user specifications. The `reactingFoam` solver creates a pointer to `combustionModels` to compute reaction rates. During execution, `combustionModels` retrieves the user-specified combustion model from the `combustionProperties` configuration file from `constant` directory of case setup to calculate the reaction rate.

The `PaSR.C` file implements the reaction rate function, inheriting this functionality from the base `laminar` class. The `PaSR` implementation file is located in:

`src/combustionModels/PaSR/PaSR.C`

The `PaSR` source code comprises the following components:

- Registration
- Constructor for initializing the `PaSR` model with required parameters
- Destructor
- Member functions implementing calculation methods for the combustion model

3.1 Registration

In the first block, the `PaSR` combustion model is registered as `PaSR` and added to the runtime selection table in OpenFOAM as listed in Listing 3.1.

Listing 3.1: PaSR.C Registration

```
namespace Foam
{
namespace combustionModels
{
    defineTypeNameAndDebug(PaSR, 0);
    addToRunTimeSelectionTable(combustionModel, PaSR, dictionary);
}
}
```

OpenFOAM uses `namespaces` to organize its code. Think of it like creating folders to keep related files together. The `Foam` namespace is the top-level container that holds all of OpenFOAM's functionality. Inside `Foam`, there's a specific namespace called `combustionModels`. This namespace is where all the combustion-related code is placed, making it easier to find and manage models like `PaSR`. Now, to make the `PaSR` model available in OpenFOAM, it needs to be registered in the code. That's where `defineTypeNameAndDebug(PaSR, 0)` comes in. This line does two things:

1. It registers the `PaSR` model so OpenFOAM recognizes it as a combustion model.
2. The `0` means that debugging for this model is turned off by default.

The final piece is adding the model to the *runtime selection table*.

```
addToRunTimeSelectionTable(combustionModel, PaSR, dictionary);
```

The runtime selection table allows users to choose the combustion model they want to use directly from their input files (called *dictionaries* in OpenFOAM) without having to modify or recompile the code. For example, if you want to use the `PaSR` model in a simulation, you simply include this line in your dictionary file, which is `combustionProperties` located in `constant` directory of the case setup:

```
combustionModel PaSR;
```

With these steps, OpenFOAM makes the `PaSR` model available and easy to configure for users. It's a smart way to keep the framework flexible and user-friendly.

3.2 Constructor

The constructor initializes the `PaSR` model by reading configuration files and setting up model parameters like `Cmix` and `kappa` (κ) and initializes the `laminar` combustion model as listed in Listing 3.2.

Listing 3.2: PaSR.C Constructor

```
Foam::combustionModels::PaSR::PaSR
(
    const word& modelType,
```

```
    const fluidReactionThermo& thermo,
    const compressibleMomentumTransportModel& turb,
    const word& combustionProperties
)
:
    laminar(modelType, thermo, turb, combustionProperties),
    Cmix_(this->coeffs().template lookup<scalar>("Cmix")),
    kappa_
    (
        IOobject
        (
            thermo.phasePropertyName(typeName + ":kappa"),
            this->mesh().time().timeName(),
            this->mesh(),
            IOobject::NO_READ,
            IOobject::AUTO_WRITE
        ),
        this->mesh(),
        dimensionedScalar(dimless, 0)
    )
{}
```

The `Foam::combustionModels::PaSR::PaSR(...)` function is the constructor for the `PaSR` class. It initializes the model and sets up its essential parameters. This constructor is based on four key arguments:

- `const word& modelType`: Specifies which type of combustion model is being initialized.
- `const fluidReactionThermo& thermo`: Provides critical thermodynamic properties like temperature and chemical composition.
- `const compressibleMomentumTransportModel& turb`: Represents the turbulence model that the combustion model will use.
- `const word& combustionProperties`: Refers to the name of the file where the combustion properties are stored.

When the constructor is called, it ensures that all these components are correctly initialized to prepare the model for simulation. The constructor also invokes the base class constructor, `laminar(modelType, thermo, turb, combustionProperties)`. This step is crucial because it initializes all the shared components of the combustion model. One of the parameters in the `PaSR` model is `Cmix_`. This value is fetched to `PaSR` using:

```
Cmix_(this->coeffs().template lookup<scalar>("Cmix"))
```

Here, `Cmix_` is read from the model's coefficients file. It represents the mixing rate co-efficient, which plays an essential role in how the model simulates the interaction between turbulence and chemical reactions. Another important property in the `PaSR` model is `kappa_`, which is a field representing a model-specific property. Its setup involves a few steps:

1. **Creating the `IOobject` for `kappa_`:**
    - `thermo.phasePropertyName(typeName + ":kappa")`: Defines the name of the file that will store the `kappa` property.
    - `this->mesh().time().timeName()`: Specifies the current time directory for the simulation.
    - `this->mesh()`: Associates the `kappa` property with the computational mesh.
    - `IOobject::NO_READ`: Indicates that the `kappa` file will not be read initially.
    - `IOobject::AUTO_WRITE`: Ensures that the file is automatically updated and saved during the simulation.

2. **Initializing `kappa_`:** The line `dimensionedScalar(dimless, 0)` sets `kappa_` to a dimensionless scalar value of 0 by default.

By combining these elements, OpenFOAM ensures that the `PaSR` model is not only well-organized but also highly configurable and efficient during simulations.

3.3 Destructor

The destructor is called when an object of the `PaSR` class is destroyed. In this case, the destructor does not perform any actions because it is empty:

```
Foam::combustionModels::PaSR::~PaSR()
```

An empty destructor means no special cleanup is needed for this class.

3.4 Member Functions

The member functions consist of various methods to perform the required calculations for the combustion model. These include functions to calculate `kappa`, reaction rate (`R`) inherited from the `laminar` class, heat release rate (`Qdot`), and a function for reading model parameters.

3.4.1 `correct()`

The `correct()` function updates the `kappa` field based on turbulence and chemistry data. The `laminar::correct()` function calls the `correct()` function of the base `laminar` class to perform necessary updates. Then, it retrieves the desired variables `epsilon` and `nuEff` from the turbulence model. The pointer to the chemical time scale (`tc`) definition is implemented in the `baseChemistryModel` class. The code retrieves the output of the chemical time scale and stores it in `tc`. These variables are required to calculate the `kappa`, which is described in Eq. 2.6. The `kappa` is calculated as shown in Listing 3.3.

Listing 3.3: PaSR.C correct()

```
void Foam::combustionModels::PaSR::correct()
{
    laminar::correct();

    tmp<volScalarField> tepsilon(this->turbulence().epsilon());
    const scalarField& epsilon = tepsilon();

    tmp<volScalarField> tnuEff(this->turbulence().nuEff());
    const scalarField& nuEff = tnuEff();

    tmp<volScalarField> ttc(this->chemistryPtr_->tc());
    const scalarField& tc = ttc();

    forAll(epsilon, i)
    {
        const scalar tk =
            Cmix_*sqrt(max(nuEff[i]/(epsilon[i] + small), 0));

        if (tk > small)
        {
            kappa_[i] = tc[i]/(tc[i] + tk);
        }
        else
        {
            kappa_[i] = 1.0;
        }
    }
}
```

3.4.2 R(volScalarField& Y)

The function `Foam::combustionModels::PaSR::R` returns a modified reaction rate for the `PaSR` combustion model after multiplying it with `kappa_`. The reaction rate term is inherited from the `laminar` combustion model. The implementation is shown in Listing 3.4.

Listing 3.4: PaSR.C R(volScalarField& Y)

```
Foam::tmp<Foam::fvScalarMatrix>
Foam::combustionModels::PaSR::R(volScalarField& Y) const
{
    return kappa_*laminar::R(Y);
}
```

3.4.3 Qdot()

Similarly, the `Foam::combustionModels::PaSR::Qdot()` function modifies and returns the heat release rate for the PaSR model after multiplying by `kappa`. It is implemented as shown in Listing 3.5.

Listing 3.5: PaSR.C Qdot()

```
Foam::tmp<Foam::volScalarField>
Foam::combustionModels::PaSR::Qdot() const
{
    return volScalarField::New
    (
        this->thermo().phasePropertyName(typeName + ":Qdot"),
        kappa_*laminar::Qdot()
    );
}
```

3.4.4 read()

After `Qdot`, the `read()` function is implemented to look for the `Cmix` value in the `combustionProperties` file. It will return `true` if it finds the value and `false` if not mentioned. The `read()` method ensures the model's parameters are correctly updated during runtime.

To summarize, the `PaSR` model in OpenFOAM is a well-organized and efficient framework for simulating premixed or partially premixed reactive flows. By following the structure and implementation details provided, users can effectively utilize and modify the `PaSR` model to suit their specific simulation needs. Based on this learning, in the next chapter, we will implement the `DTF` model.

Chapter 4

DTF: Implementation

In this chapter, based on the learning from Chapter 2 and Chapter 3 the new TCI model will be implemented that will work for non-premixed, partially premixed and premixed combustion.The baseline code used in DTF is implemented by Rintanen [1] in his master thesis work [2], as Thickened Flame Model (`TFM`). In comparison to the `PaSR`, the `reactingFoam` will calculate the reaction rates in a similar way as computed for `PaSR`, beside this, `DTF` requires modifications to the diffusion coefficient, which requires the implementation of `thermoPhysicalTransport` library. The `DTF` code interaction with other source code components in OpenFOAM is shown in Fig. 4.1.

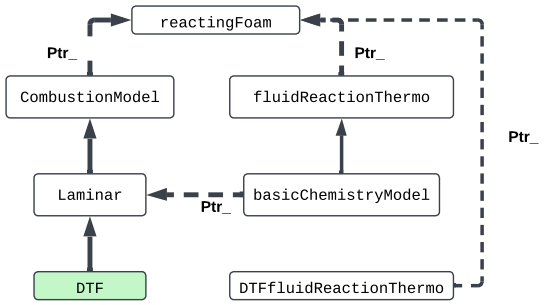

Figure 4.1: `DTF` interaction with other source code components in OpenFOAM

4.1 Introduction to `PaSR` & `TFM`

Before getting involved in any implementation, it is important to understand what `PaSR` and `TFM` provide for implementing the `DTF` model. The explanation of `PaSR` in Chapter 3 is meant to give the user an understanding of how the `combustionModel` is implemented in OpenFOAM. `PaSR` was specifically chosen for this report due to its similarity with the implementation of `TFM` or `DTF`.

The next question is: If `TFM` is used as the base code for `DTF`, how does it compare with the `PaSR` implementation? In `PaSR`, the `correct()` function computes the `kappa` field. This `kappa` field is then used as a parameter to modify the reaction rate source term function (`Foam::combustionModels::PaSR::R`) as `return kappa_*laminar::R(Y);`, which modifies the reaction rates, as well as the heat release (`Qdot`).

In comparison, the `TFM` `correct()` function calculates the `FlameSensor` (S), `ThickeningFactor` (F), and `EfficiencyFunction` (E). `TFM` will initialize the member variables and functions necessary for `TFM` calculations. Similarly, it will pass the `ThickeningFactor` parameter (F) and the `EfficiencyFunction` parameter (E) to the reaction rate source term and heat release instead of `kappa`.

4.2 Introduction to TFM & DTF

The key differences between TFM and DTF are summarized in Table 4.1.

Table 4.1: Comparison of TFM and DTF Parameters

Parameters	TFM	DTF
Mesh	Uniform	Non-uniform (Tetra, Poly, Hexa)
Physics	Laminar/DNS	Laminar/DNS/LES
Laminar Flame Speed (S_l^0)	User-defined (Fixed Value)	Computed (update/iteration)
Laminar Flame Thickness (δ_l)	User-defined (Fixed Value)	Computed (update/iteration)
Thickening Factor (F)	User-defined (Fixed Value)	Computed (update/iteration)
Efficiency Factor (E)	Power-law	Colin
Flame Sensor (S)	Based on heat release	Optimized TFM sensor*
ThermophysicalTransportModels	Modified Library	Same implementation as TFM
Flame Speed (S_l^0 & δ_l)	Not implemented	Implemented

* *Optimized TFM sensor: The same sensor as in TFM is used, but filters are applied to smooth the values. The magnitude of the sensor is also taken to ensure positive values.*

Other than the dynamic flame sensor described in Eq. 2.13 and referenced in Table 4.1, a fixed sensor is also part of the code. It is implemented for TFM and remains unchanged in the code for DTF. The calculation of laminar flame speed (S_l^0) and laminar flame thickness (δ_l) requires the reaction rates of all species involved in the simulation and thermal diffusivity (D), as described in Eq. 2.20. To perform these calculations, a new function, `flameSpeed()`, has been implemented. This function calculates the laminar flame speed (S_l^0) and laminar flame thickness (δ_l).

The DTF used the same tree structure for coding as TFM represented in Lisitng 4.1. The implementation of the DTF will take a start from the existing TFMFoam code. Begin by downloading the TFMFoam code from GitHub:

```
git clone https://github.com/arintanen/TFMFoam.git
```

After cloning the repository, verify the directory structure by running the following command inside the TFMFoam folder:

```
tree -L 2
```

Executing this command will generate a directory listing, as shown in Listing 4.1:

Listing 4.1: TFMFoam Directory

```
|-- Allwclean
|-- Allwmake
|-- README.md
|-- src
|   |-- combustionModels
|   |-- lnInclude
|   |-- ThermophysicalTransportModels
|-- tutorials
    |-- combustionProperties
    |-- thermophysicalTransport
```

In the first step, compile TFMFoam using the script `./Allwmake`. Next, navigate to the `combustionModels` directory and move the TFM code to a folder named DTF.

```
mv -rp TFM DTF
```

Change the directory and move to the newly created DTF directory and rename the header and constructor files from TFM to DTF:

```
cd DTF
mv TFM.H DTF.H
mv TFM.C DTF.C
```

Make the initial modifications to the `TFMFoam` directories and files to set up the `DTF` framework. Update the file names and references using the following commands:

```
sed -i 's/\bTFM\b/DTF/g' DTF.H
sed -i 's/\bTFM\b/DTF/g' DTF.C
sed -i '1i DTF/DTF.C' ../Make/files
```

Finally, compile the updated code with:

```
./../../../Allwmake
```

Once the framework for `DTF` is set up, the next step is to modify the `DTF` code. In the implementation process of `DTF`, the report will try to connect with `PaSR` implementation in Chapter 3. Follow the steps below to update the code accordingly.

4.3 Registration

During the process of renaming the files and updating file content, the `TFM` registration is updated to the `DTF`, as represented in Lisitng 4.2.

Listing 4.2: "src/combustionModels/DTF/DTF.C", DTF Registration

```
namespace Foam
{
namespace combustionModels
{
    defineTypeNameAndDebug(DTF, 0);
    addToRunTimeSelectionTable(combustionModel, DTF, dictionary);
}
}
```

4.4 Constructor

4.4.1 Update the `DTF.H`

Add the following `volScalarField` variables for the reaction rate (`tRR_`), diffusivity (`D_`) and thickening factor (`Fs_`) after the first line from Listing 4.3.

Listing 4.3: "src/combustionModels/DTF/DTF.H", Private volScalarField

```
volScalarField F_, EF_; //first line
volScalarField tRR_;
volScalarField D_;
volScalarField Fs_;
```

Next, add the following scalar variables after the first line of Lisitng 4.4.

Listing 4.4: "src/combustionModels/DTF/DTF.H",Private Scalars

```
autoPtr<efficiencyFunction> efficiencyFunction_; //first line
scalar upperLimit_;
scalar lowerLimit_;
scalar filters_;
```

Now, add the following protected member for the chemistry model.

Listing 4.5: "src/combustionModels/DTF/DTF.H", Protected Chemistry Pointer

```
autoPtr<basicChemistryModel> chemistryPtr_;
```

Finally, define the member function for flame speed calculations in the member function section.

Listing 4.6: "src/combustionModels/DTF/DTF.H", Member Function flameSpeed()

```
void flameSpeed();
```

4.4.2 Update the `DTF.C`

Initialize the newly added members in the header file. Add the `volScalarField` after `EF_` initialization

Listing 4.7: "src/combustionModels/DTF/DTF.C", EF Initialization

```
    EF_
     (
         IOobject
         (
             "EF",
             this->mesh().time().timeName(),
             this->mesh(),
             IOobject::NO_READ,
             IOobject::NO_WRITE
         ),
         this->mesh(),
         dimensionedScalar(dimless, 1)
     ),
```

and add the scalar after the following `efficiencyFunction_` initialization.

Listing 4.8: "src/combustionModels/DTF/DTF.C", Efficiency Function Initialization

```
efficiencyFunction_(efficiencyFunction::New(turb.mesh(), coeffs(), turb, Fmax_)),
```

Here is the initialization of new variables, Listing 4.9 is the representation of `volScalarField` and Listing 4.10 represents `scalar`.

Listing 4.9: `"src/combustionModels/DTF/DTF.C", Constructor Members`

```
tRR_
(
    IOobject
    (
        "tRR",
        mesh_.time().timeName(),
        mesh_,
        IOobject::NO_READ,
        IOobject::NO_WRITE
    ),
    mesh_,
    dimensionedScalar("tRR", dimMass/dimVolume/dimTime, Zero)
),
D_
(
    IOobject
    (
        "D",
        mesh_.time().timeName(),
        mesh_,
        IOobject::NO_READ,
        IOobject::NO_WRITE
    ),
    mesh_,
    dimensionedScalar("D", dimLength*dimLength/dimTime, Zero)
```

```
),
Fs_
(
    IOobject
    (
        "Fs",
        this->mesh().time().timeName(),
        this->mesh(),
        IOobject::NO_READ,
        IOobject::NO_WRITE
    ),
    this->mesh(),
    dimensionedScalar(dimless, 1)
),
```

Listing 4.10: "src/combustionModels/DTF/DTF.C", Constructor Members

```
deltaL_(0.001),
SL_(3),
upperLimit_(this->coeffs().template lookup<scalar>("upperLimit")),
lowerLimit_(this->coeffs().template lookup<scalar>("lowerLimit")),
filters_(this->coeffs().template lookup<scalar>("filters")),
chemistryPtr_(basicChemistryModel::New(thermo))
```

4.5 Member Functions

4.5.1 Implementation of the `flameSpeed()` Function

The `flameSpeed()` function is implemented to calculate the parameters required for the thickening factor (F) and efficiency function (E). Specifically, it computes the laminar flame speed (Eq. 2.19) and the laminar flame thickness (Eq. 2.21). This section describes the step-by-step implementation of the `flameSpeed()` function.

First, the `flameSpeed()` function is initiated inside the `correct()` function. This initiation ensures that the required parameters are calculated during each time step. The initiation of `flameSpeed()` within `correct()` is shown in Listing 4.11.

Listing 4.11: "src/combustionModels/DTF/DTF.C", correct();

```
/// FlameSpeed switch, default is true. Set to false for cold (non-reactive) flows.
const bool FlameSpeed = this->coeffs().lookupOrDefault("FlameSpeed", true);
if (FlameSpeed)
{
    flameSpeed();
}
```

Next, the `flameSpeed()` function is implemented after the `Qdot` function. The `flameSpeed()` function is defined as a `void` function, meaning it does not return any value and does not take any arguments. The implementation of the `flameSpeed()` function is shown in Listing 4.12.

Listing 4.12: src/combustionModels/DTF/DTF.C, flameSpeed(); Implementation

```
void Foam::combustionModels::DTF::flameSpeed()
{
    ... //flameSpeed() implementation
}
```

The laminar flame speed (S_l^0) is proportional to the reaction rate and thermal diffusivity. The reaction rate function (`calculateRR`) is implemented in the `basicChemistryModel` class of OpenFOAM. To calculate the reaction rate in the `flameSpeed()` function, a `chemistryPtr_` pointer is created to dynamically allocate memory for the reaction rate function (`calculateRR`) on the heap. The `calculateRR` function requires two input parameters: the species index number and the reaction number. These parameters are extracted from the `basicChemistryModel` class through the `chemistryPtr_` pointer, as shown in Listing 4.13.

Listing 4.13: src/combustionModels/DTF/DTF.C, flameSpeed(); Species Index and Reaction Number

```
    // Get the number of species
    const label nSpecie = chemistryPtr_->nSpecie();
    Info << "Number of species: " << nSpecie << endl;

    // Get the number of reactions
    const label nReaction = chemistryPtr_->nReaction();
    Info << "Number of reactions: " << nReaction << endl;
```

The reaction rate function (`calculateRR`) is implemented in Listing 4.14. The absolute value of the reaction rate is taken using `Foam::mag` to ensure positive values for summation.

Listing 4.14: src/combustionModels/DTF/DTF.C, flameSpeed(); Reaction Rate (calculateRR)

```
    ////////////////////////// Reaction Rate Calculations ////////////////////

    // Reset reaction rate for all cells to zero to ensure accurate summation for the current time
     step
    forAll(tRR_.internalField(), celli)
    {
        tRR_[celli] = 0.0;
    }

    // Calculate the reaction rate for all reactions and species
    for (label ri = 0; ri < nReaction; ++ri)
    {
        for (label si = 0; si < nSpecie; ++si)
        {
            volScalarField::Internal RR = Foam::mag(chemistryPtr_->calculateRR(ri, si));
            forAll(RR, celli)
            {
                if (!std::isnan(F_[celli]) && F_[celli] > SMALL)
                {
                    tRR_[celli] += RR[celli];
                }
            }
        }
    }
```

The total reaction rate `tRR_` is reset to zero after every time loop to ensure that only the values of the present instant are considered. After summing all absolute reaction rates, the results are passed through a smoothing filter. To use the smoothing filter, include the header file `"fvcSmooth.H"` along with the other header files at the top of the `DTF.C` file. The implementation of the smoothing filter is shown in Listing 4.15.

Listing 4.15: src/combustionModels/DTF/DTF.C, flameSpeed(); Reaction Rate (smoothFilter)

```
    volScalarField smoothed_tRR = tRR_; // filter variable tRR_

    // Smoothing filter
    fvc::smooth(smoothed_tRR, filters_);
    tRR_ = smoothed_tRR;
```

The `flameSensor_->S()` function is used to extract the values of the total reaction rate from the flame front. Listing 4.16 demonstrates how the reaction rate is extracted and calculated across all processors.

Listing 4.16: src/combustionModels/DTF/DTF.C, flameSpeed(); Reaction Rate (calculateRR)

```
    ////////////////////////// Flame Sensor //////////////////////////
    volScalarField q_sensor = flameSensor_->S();
    scalar sumReactionRate = 0.0;
    scalar count = 0.0;
    Info << "Internal field size: " << q_sensor.internalField().size() << endl;

```

```
    // Extract reaction rates between the set values of the flame sensor
    forAll(q_sensor.internalField(), celli)
    {
        if (q_sensor[celli] >= lowerLimit_ && q_sensor[celli] <= upperLimit_)
        {
            sumReactionRate += tRR_[celli];
            count++;
        }
    }

    // Parallel reduction for sum and count
    scalar globalSumReactionRate = sumReactionRate;
    scalar globalCount = count;
    reduce(globalSumReactionRate, sumOp<scalar>());
    reduce(globalCount, sumOp<scalar>());

    // Compute average reaction rate across all processors
    scalar avgReactionRate = (globalCount > 0) ? (globalSumReactionRate / globalCount) : 0.0;

    Info << "Average reaction rate (based on sensor): " << avgReactionRate << endl;
```

Similarly, the thermal diffusivity (D) is calculated, as described in Eq.2.3. While the reaction rate calculation uses the `calculateRR` function implemented in the `basicChemistryModel` class, the thermal diffusivity calculation accesses the values of thermal conductivity (λ), density (ρ), and heat capacity (C_p) through the `thermo_` pointer from the `fluidReactionThermo` class. Afterward, the smoothing filter and extraction of diffusion values from the reaction front follow the same procedure as the reaction rate `tRR_` calculation. The thermal diffusivity calculations are shown in Listing 4.17.

Listing 4.17: src/combustionModels/DTF/DTF.C, flameSpeed(); Thermal Diffusivity Calculation

```
    ////////////////////////// Diffusion Calculations ////////////////////////
    const volScalarField& rhoField = Foam::combustionModel::thermo_.rho();
    const volScalarField& kappaField = Foam::combustionModel::thermo_.kappa();
    const volScalarField& CpField = Foam::combustionModel::thermo_.Cp();

    forAll(this->mesh().V(), celli)
    {
        scalar rho = rhoField[celli];
        scalar lambda = kappaField[celli];
        scalar cp = CpField[celli];

        if (rho > SMALL && cp > SMALL)
        {
            D_[celli] =  lambda / (rho * cp);
        }
        else
        {
            D_[celli] = 0;
        }
    }
    Info << "Maximum Diffusion, D_: " << Foam::gMax(D_) << endl;
    Info << "Minimum Diffusion, D_: " << Foam::gMin(D_) << endl;

    // Smoothing filter
    volScalarField smoothed_D = D_;
    fvc::smooth(smoothed_D, filters_);
    D_ = smoothed_D;
    Info << "Smoothed Diffusion: Maximum: " << Foam::gMax(D_) << ", Minimum: " << Foam::gMin(D_) <<
     endl;

    scalar sumD = 0.0;
    scalar countD = 0.0;
    // Extracting values of diffusion between the set values of the flame sensor
    forAll(q_sensor.internalField(), celli)
    {
        if (q_sensor[celli] >= lowerLimit_ && q_sensor[celli] <= upperLimit_)
        {
```

```
            sumD += D_[celli];
            countD++;
        }
    }

    // Parallel reduction for sumD and countD
    scalar globalSumD = sumD;
    scalar globalCountD = countD;
    reduce(globalSumD, sumOp<scalar>());
    reduce(globalCountD, sumOp<scalar>());

    // Compute average D across all processors
    scalar avgD = (globalCountD > 0) ? (globalSumD / globalCountD) : 0.0;

    Info << "Average Diffusion (based on sensor): " << avgD << endl;
```

After calculating the reaction rate and thermal diffusivity, the laminar flame speed (S_l^0) and laminar flame thickness (δ_l) are computed inside the `flameSpeed()` function, as shown in Listing 4.18.

Listing 4.18: src/combustionModels/DTF/DTF.C, flameSpeed(); Laminar Flame Speed and Thickness Calculation

```
void Foam::combustionModels::DTF::flameSpeed()
{
    ////////////////////////// Diffusion Calculations //////////////////////////
    const volScalarField& rhoField = Foam::combustionModel::thermo_.rho();
    const volScalarField& kappaField = Foam::combustionModel::thermo_.kappa();
    const volScalarField& CpField = Foam::combustionModel::thermo_.Cp();

    forAll(this->mesh().V(), celli)
    {
        scalar rho = rhoField[celli];
        scalar lambda = kappaField[celli];
        scalar cp = CpField[celli];

        if (rho > SMALL && cp > SMALL)
        {
            D_[celli] = lambda / (rho * cp);
        }
        else
        {
            D_[celli] = 0;
        }
    }
    Info << "Maximum Diffusion, D_: " << Foam::gMax(D_) << endl;
    Info << "Minimum Diffusion, D_: " << Foam::gMin(D_) << endl;

    // Smoothing filter
    volScalarField smoothed_D = D_;
    fvc::smooth(smoothed_D, filters_);
    D_ = smoothed_D;
    Info << "Smoothed Diffusion: Maximum: " << Foam::gMax(D_) << ", Minimum: " << Foam::gMin(D_) <<
     endl;

    ////////////////////////// Flame Speed and Thickness //////////////////////
    Info << "Starting laminar flame speed (SL_) and thickness (deltaL_) calculation..." << endl;

    if (avgReactionRate > SMALL && avgD > SMALL)
    {
        SL_ = sqrt(avgReactionRate * avgD);
        deltaL_ = avgD / SL_;

        // Clamp values to avoid unrealistic results
        if (SL_ > 50)
        {
            Info << "SL_ exceeds 50, clamping to 50." << endl;
            SL_ = 50;
```

```
        }

        if (deltaL_ > 10)
        {
            Info << "deltaL_ exceeds 10, clamping to 10." << endl;
            deltaL_ = 10;
        }
    }
    else
    {
        // Use default values if calculations fail
        SL_ = this->coeffs().template lookup<scalar>("SL");
        deltaL_ = this->coeffs().template lookup<scalar>("deltaL");
    }

    Info << "Completed laminar flame speed (SL_) and thickness (deltaL_) calculation." << endl;
}
```

4.5.2 Update Thickening Factor

The next step is to update the thickening factor (F) (Eq. 2.12) and efficiency (E) (Eq. 2.18). Replace the following code inside the `correct();` function:

```
F_ = (Fmax_-1)*flameSensor_->S() +1;
```

with the following implementation in Listing 4.19

Listing 4.19: "src/combustionModels/DTF/DTF.C", Thickening Factor Update

```
forAll(F_, celli)
{
    scalar delta_g = pow(mesh_.V()[celli], 1.0 / static_cast<scalar>(mesh_.nGeometricD()));
    if (deltaL_ > SMALL)
    {
        scalar Fvalue = (delta_g * this->coeffs().template lookup<scalar>("N")) / deltaL_;
        Fs_[celli] = min(Fvalue, Fmax_);
        scalar sensorValue = flameSensor_->S()[celli];
        F_[celli] = 1 + (Fs_[celli] - 1) * sensorValue;

        if (F_[celli] < 1) F_[celli] = 1;
    }
    else
    {
        F_[celli] = 1.0;
    }
}
```

Next, replace the efficiency-thickening factor multiplication code:

```
EF_ = efficiencyFunction_->E()*F_;
```

with the following implementation as listed in Listing 4.20.

Listing 4.20: "src/combustionModels/DTF/DTF.C", EF Factor Update

```
forAll(EF_, celli)
{
    EF_[celli] = efficiencyFunction_->E()[celli] * F_[celli];
}
```

with these changes, the implementation inside the `correct();` function is now complete. The next step is to implement the efficiency function (E). The efficiency function (E) requires the laminar flame speed (S_l^0), the laminar flame thickness (δ_l), and the thickening factor (F).

4.5.3 Efficiency Function Implementation in DTF

Previously, the efficiency function (E) was implemented outside the TFM class. To make the efficiency function dynamic, there are several ways to introduce the variables calculated inside the DTF class to the efficiency function class. One approach is through getter functions. In this implementation, the required parameters are introduced to the efficiency function (E) via getter functions. This requires modifications to the header and constructor files of DTF.

4.5.3.1 Adding Getter Functions to DTF.H

Add the following getter functions to the member section of the DTF.H header file as shown in Listing 4.21.

Listing 4.21: "src/combustionModels/DTF/DTF.H", DTF.H Getter Functions

```
scalar deltaL() const { return deltaL_; }
scalar SL() const { return SL_; }
scalar filters() const { return filters_; }
const volScalarField& F() const { return F_; }
```

4.5.3.2 Update the Efficiency Function Constructor DTF.C

Next, update the constructor to include a pointer (*this) for initializing the efficiency function inside the DTF.C file. This pointer makes the getter functions accessible to the efficiency function class. The updated constructor will look as represented in Listing 4.22.

Listing 4.22: "src/combustionModels/DTF/DTF.C", Efficiency Function Constructor

```
efficiencyFunction_(efficiencyFunction::New(turb.mesh(), coeffs(), turb, Fmax_, *this)),
```

4.5.3.3 Update efficiencyFunction.H

Add the following line as a private member as listed in Listing 4.23.

Listing 4.23: "src/combustionModels/EfficiencyFunctions/efficiencyFunction/efficiencyFunction.H", DTF Member

```
const Foam::combustionModels::DTF& dtfModel_;
```

Next, update the runtime declaration of the efficiency function to include the DTF parameter as shown in Listing 4.24.

Listing 4.24: "src/combustionModels/EfficiencyFunctions/efficiencyFunction/efficiencyFunction.H", Efficiency Function Runtime Update

```
declareRunTimeSelectionTable
(
    autoPtr,
    efficiencyFunction,
    dictionary,
    (
        const dictionary& dict,
        const fvMesh& mesh,
        const compressibleMomentumTransportModel& turb,
        scalar F,
        const Foam::combustionModels::DTF& dtfModel // Addition as the fifth member
    ),
    (dict, mesh, turb, F, dtfModel) // Pass DTF parameter
);
```

4.5.3.4 Update `efficiencyFunction.C`

Add `#include "DTF.H"` in the header, then update the constructor with the `DTF` parameter as listed in Listing 4.25.

Listing 4.25: "src/combustionModels/DTF/DTF.C", Efficiency Function Constructor Update

```
Foam::efficiencyFunction::efficiencyFunction
(
    const dictionary& dict,
    const fvMesh& mesh,
    const compressibleMomentumTransportModel& turb,
    scalar Fmax,
    const Foam::combustionModels::DTF& dtfModel // DTF as fifth member
)
:
    mesh_(mesh),
    E_(
         IOobject
         (
             "E",
             mesh().time().timeName(),
             mesh(),
             IOobject::NO_READ,
             IOobject::NO_WRITE
         ),
         mesh,
         dimensionedScalar(dimless, 1)
     ),
     F_(Fmax),
     turb_(turb),
     dtfModel_(dtfModel)  // DTF initialization
{
}
```

4.5.3.5 Update `efficiencyFunctionNew.C`

Add `#include "DTF.H"` in the header, then update the constructor with the `DTF` parameter as described in Listing 4.26.

Listing 4.26: "src/combustionModels/EfficiencyFunctions/efficiencyFunction/efficiencyFunction.H", Efficiency Function Constructor Update

```
Foam::autoPtr<Foam::efficiencyFunction> Foam::efficiencyFunction::New
(
    const fvMesh& mesh,
    const dictionary& dict,
    const compressibleMomentumTransportModel& turb,
    scalar F,
    const Foam::combustionModels::DTF& dtfModel // DTF
)
{
    const word modelType(dict.lookup("efficiencyFunction"));

    Info<< "Selecting efficiency function model " << modelType << endl;

    dictionaryConstructorTable::iterator cstrIter =
        dictionaryConstructorTablePtr_->find(modelType);

    if (cstrIter == dictionaryConstructorTablePtr_->end())
    {
        FatalIOErrorInFunction
        (
            dict
        )   << "Unknown efficiency function type "
            << modelType << nl << nl
            << "Valid efficiency function types are :" << endl
```

```
            << dictionaryConstructorTablePtr_->sortedToc()
            << exit(FatalIOError);
    }

    return autoPtr<efficiencyFunction>(cstrIter()(dict, mesh, turb, F, dtfModel)); // DTF parameter
}
```

4.5.3.6 Colin Efficiency Function Implementation

Now, implement the Colin efficiency function inside the `EfficiencyFunction` directory. Create the folder `ColinEfficiencyFunction`, and then create two files, `colin.H` and `colin.C`, in the following directory:

```
src/combustionModels/EfficiencyFunctions

mkdir ColinEfficiencyFunction
cd ColinEfficiencyFunction
touch colin.H
touch colin.C
```

The code to calculate the `Colin` efficiency is implemented in the `colin.C` file. The complete content of `colin.H` and `colin.C` can be found in Appendix A. The Colin efficiency function is described in Sec. 2.3.3.3, and this section explains its implementation as described in Listing 4.27.

Listing 4.27: "src/combustionModels/EfficiencyFunctions/colin.C", correct() Function

```
void Foam::efficiencyFunctionModels::colin::correct() {
    scalar delta_L = dtfModel_.deltaL(); //Laminar Flame Thickness from DTF class
    scalar SL_ = dtfModel_.SL(); //Laminar Flame Speed from DTF class
    scalar Filters_ = dtfModel_.filters(); //Filters to smooth the velocity fluctuations from DTF
     class
    const volScalarField& delta_ = mesh_.lookupObject<volScalarField>("delta"); // delta from LES co-
     efficient dict
    const volScalarField& Fc_ = dtfModel_.F(); //Dynamic thickening Factor from DTF class

    // Check validity of inputs
    if (delta_L <= 0) {
        FatalErrorInFunction << "Invalid flame thickness (delta_L): " << delta_L << exit(FatalError);
    }
    if (SL_ <= 0) {
        FatalErrorInFunction << "Invalid flame speed (SL_): " << SL_ << exit(FatalError);
    }
    Info << "Flame Thickness (delta_L): " << delta_L << endl;
    Info << "Flame Speed (SL_): " << SL_ << endl;

    // Calculate uPrime_ using filtered U field
    volVectorField U_hat(turb_.U());
    Info << "Uhat_ computed successfully." << endl;

    for (int i = 0; i < Filters_; i++) {
        U_hat = sFilter_(U_hat);
    }

    // Ensure delta_ is valid
    if (delta_.internalField().size() == 0) {
        FatalErrorInFunction << "delta_ field is empty or uninitialized." << exit(FatalError);
    }
    uPrime_ = 2.0 * mag(pow(delta_, 3) * fvc::laplacian(fvc::curl(U_hat)));

    // Calculate the efficiency
    forAll(E_, celli) {
        scalar delta_e = Fc_[celli] * delta_L;

        if (delta_e <= 0) {
```

```
            WarningInFunction << "Invalid delta_e at cell " << celli << endl;
            E_[celli] = 1.0; // Assign minimum efficiency to avoid invalid calculations
            continue;
        }
        scalar delta_r = delta_e / delta_L;
        scalar delta_R = delta_e / (delta_L * Fc_[celli]);

        if (delta_r <= 0 || delta_R <= 0) {
            FatalErrorInFunction << "Invalid delta_r or delta_R values." << exit(FatalError);
        }

        scalar up_SL_r = (uPrime_[celli] * delta_e) / SL_;

        if (up_SL_r <= 0) {
            WarningInFunction << "Invalid up_SL_r at cell " << celli << endl;
            E_[celli] = 1.0; // Assign minimum efficiency to avoid invalid calculations
            continue;
        }

        scalar numerator = 1.0 + alpha_ * 0.75 * exp(-1.2 / pow(up_SL_r, 0.3)) * pow(delta_r, 2.0 /
    3.0) * up_SL_r;
        scalar denominator = 1.0 + alpha_ * 0.75 * exp(-1.2 / pow(up_SL_r, 0.3)) * pow(delta_R, 2.0 /
    3.0) * up_SL_r;

        if (denominator <= 0) {
            WarningInFunction << "Invalid denominator at cell " << celli << endl;
            E_[celli] = 1.0; // Assign minimum efficiency to avoid division by zero
            continue;
        }

        scalar efficiency = numerator / denominator;

        // Enforce the condition E >= 1
        E_[celli] = max(efficiency, 1.0);
    }
}
```

4.5.4 R(volScalarField& Y)

The function `Foam::combustionModels::DTF::R` returns a modified reaction rate for the `DTF` combustion model after multiplying it with `efficiencyFunction_->E()/F_`. The reaction rate term is inherited from the `laminar` combustion model. The implementation is shown in Listing 4.28.

Listing 4.28: DTF.C R(volScalarField& Y)

```
Foam::tmp<Foam::fvScalarMatrix>
Foam::combustionModels::DTF::R(volScalarField& Y) const
{
    return efficiencyFunction_->E()/F_*laminar::R(Y);
}
```

4.5.5 Qdot()

Similarly, the `Foam::combustionModels::DTF::Qdot()` function modifies and returns the heat release rate for the DTF model after multiplying by `efficiencyFunction_->E()/F_`. It is implemented as shown in Listing 4.29.

Listing 4.29: DTF.C Qdot()

```
Foam::tmp<Foam::volScalarField>
Foam::combustionModels::DTF::Qdot() const
{
    return volScalarField::New
    (
```

```
6         this->thermo().phasePropertyName(typeName + ":Qdot"),
7         fficiencyFunction_->E()/F_*laminar::Qdot()
8     );
9 }
```

4.6 DTFThermophysicalTransportModels

The modification of the diffusion coefficient requires scaling the mass diffusion coefficient and thermal diffusion coefficient. These modifications of the diffusion coefficient are implemented in `TFM` and `DTF` will use the same implementation to modify the diffusion coefficient. The `ThermophysicalTransportModels` code is available in `src` directory of the provided code. The modification done to the diffusivity term is represented in Listing 4.30.

Listing 4.30: "src/ThermophysicalTransportModels/turbulence/DTFunityLewisEddyDiffusivity/DTFunityLewisEddyDiffusivity.H",Modification to Diffusivity

```
1         virtual tmp<volScalarField> DEff(const volScalarField& Yi) const
2         {
3         const volScalarField& EF = mesh_.lookupObject<volScalarField>("EF"); // look for EF value
4             return volScalarField::New
5             (
6                 "DEff",
7                 this->thermo().kappa()/this->thermo().Cp()*EF + this->alphat()  //EF multiplied to
    scale the diffusivity term
8             );
9         }
```

Now, change the folder to `DTF` main folder, where it contains the `Allwmake` and `Allwclean` files and compile the code by running `./Allwmake` command.

Chapter 5

Test Cases and Results

In order to evaluate the implemented combustion model, two test cases have been prepared. These test cases are designed to address different levels of complexity and computational resource requirements. The details of the test cases are as follows:

- **Case 01**: This is a simplified test case to run simulation on a personal computer. It involves the simulation of a 3D freely propagating hydrogen flame using `DTF`.
- **Case 02**: This case focuses on a turbulent hydrogen flame based on the AHEAD burner. The simulation was conducted using the `DTF` model, and the results were compared with experimental data and reference numerical results from the literature.

Readers can use the accompanying files to run the cases. The step-by-step process is to first source the OpenFOAM-v10 and then run the Allrun script to simulate the individual case.

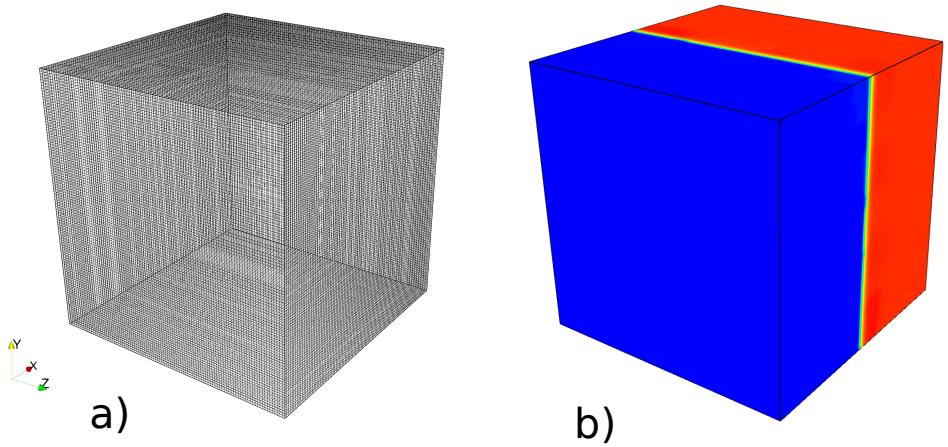

Figure 5.1: Case 01: Computational Domain

5.1 Case 01: 3D Freely Propagating Hydrogen Flame

The case setup consists of a 3D box, as illustrated in Fig. 5.1. Fig. 5.1(a) shows the mesh used for the simulation, while Fig. 5.1(b) displays the patch values for temperature. The red region represents the patch with burnt gas temperature and burnt species (water), while the blue patch corresponds to unburnt gas temperature and species (hydrogen, nitrogen, and oxygen). The box has dimensions of 0.1 m in width, height, and length along the x, y, and z directions. The grid is uniformly

distributed, with each direction divided into 100 cells. The flow moves along the x-direction. The boundary conditions include `inlet` and `outlet` along the flow direction. The top and bottom of the domain are defined as `symmetryPlane` and are labeled `symm1` and `symm2`, respectively. The left and right sides, named `frontAndBack`, are modeled as `zeroGradient` walls. The flow is initialized with a fixed velocity of 0.026 m/s at a temperature of 300 K. The composition of the flow consists of 1.45% hydrogen, 22.96% oxygen, and 75.58% nitrogen from the inlet by mass. The `setFields` utility is used to patch the values of unburnt and burnt mixtures to the domain. The patch scripts are placed in `system` directory of case setup. The simulation is run for 1 second with a time step size of 0.0001 s. The combustion model settings are provided in the `combustionProperties` file, where the combustion model name and other required coefficients are specified, as shown in Listing 5.1.

Listing 5.1: `constant/combustionProperties`

```
combustionModel  DTF;

DTFCoeffs
{
    //efficiencyFunction none;
    efficiencyFunction colin;

    colinCoeffs
    {
        alpha 0.02;
        n_filters 10; // Number of filtering operations
    }
    Fmax          15; // Maximum thickening factor
    N             3;  // Control parameter for flame thickening
    FlameSpeed    true;
    lowerLimit    0.99; // Flame sensor upper value
    upperLimit    1;    // Flame sensor lower value
    filters       2;
    //flameSensor none;
    flameSensor tanh;
    tanhCoeffs {
        beta 10;
        n_filters 10;
    }
}
```

The coefficients required for the `DTF` combustion model include the following:

- `F_max`: The maximum threshold set by the user.
- `N`: Represents the number of cells.
- `FlameSpeed`: A switch to enable or disable the `FlameSpeed` function.
- `lowerLimit`: The value provided to the `flameSensor` to extract reaction rate and diffusivity values.
- `upperLimit`: The upper threshold for the `flameSensor`.
- `filters`: Specifies the number of filters used to smooth fluctuations in reaction rate, diffusivity, and velocity.

The `flameSensor` offers two options: `none` and `tanh`. If `tanh` is selected, additional parameters `beta` and `n_filters` must be provided. Similarly, the efficiency function can be set to either `none` or `colin`. If `colin` is chosen, the `alpha` parameter must be specified.The configuration of the `thermophysicalTransportModel` is provided in Listing 5.2.

Listing 5.2: `constant/thermophysicalTransport`

```
    LES
    {
```

```
        model           DTFunityLewisEddyDiffusivity;
        Prt             0.85;
    }
```

The implemented library names are specified at the end of the `controlDict` file. When users provide these names, the implemented `combustionModel` and `thermoPhysicalTransport` libraries become accessible to OpenFOAM, as shown in Listing 5.3.

Listing 5.3: `system/controlDict`

```
libs
(
    "libDTFcombustion.so"
    "libDTFtransport.so"
);
```

The simulations are carried out using the hydrogen mechanism provided by Capurso et al. [22]. The remaining case setup files, including `chemistryProperties` and `thermoPhyscialProperties`, along with other case setup files, are listed in Appendix B.

5.1.1 Results

Fig. 5.2 represents the flame propagation, showing how the temperature propagates from the middle of the plane towards the fuel inlet.

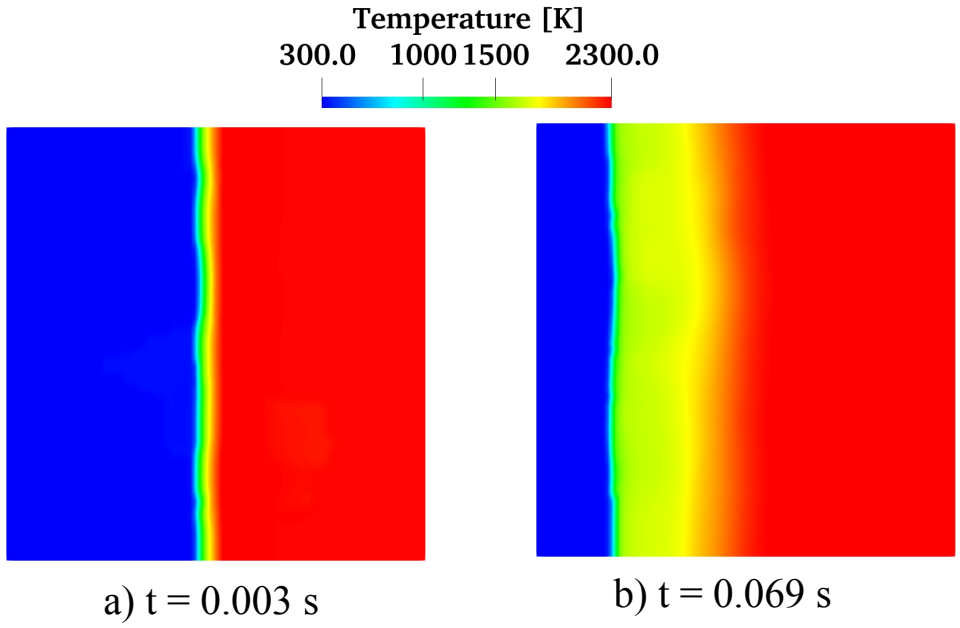

Figure 5.2: z-plane at $z = 0$, at different instants, showing the temperature field at times $t = 0.003$ s and $t = 0.069$ s.

5.2 Case 02: Turbulent Hydrogen Flame

The computational domain for the AHEAD burner, considered for the simulation, is represented in Fig. 5.3. The domain consists of fuel inlets, an air inlet, fuel injectors, swirl generators, dilution holes, a mixing chamber, and a combustion chamber. Experimental studies on this setup were conducted

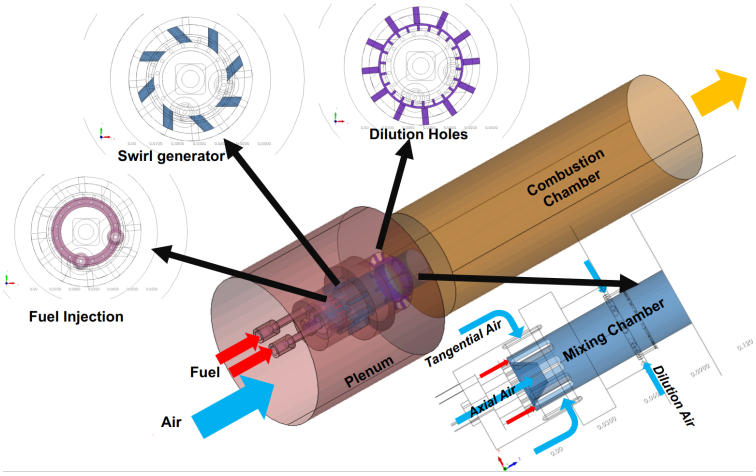

Figure 5.3: AHEAD burner, computational domain

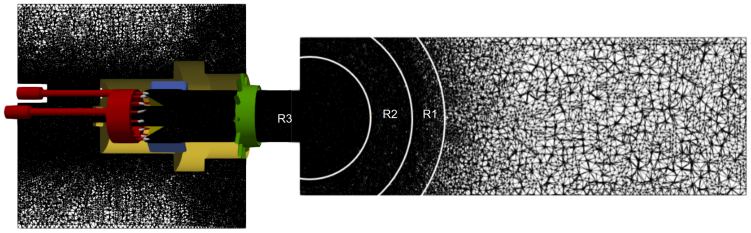

Figure 5.4: Computational grid of the AHEAD burner.

by Reichel et al. [23], and detailed information about the numerical setup can be found in the study by Sehole et al. [24]. This setup was chosen because it produces highly turbulent premixed flames, making it suitable for the intended analysis. The computational grid is based on a tetrahedral mesh, which is created using the Ennova CFD tool [25] as shown in Fig. 5.4. The mesh is refined in the R3 region and gradually coarsens towards R1. In the R3 region, the cell size is 0.75 mm, which then increases to 1 mm. The maximum cell size in the domain is 5 mm. The `DTF` combustion model setup for this case is similar to that of Case 01, so it is not necessary to explain the setup again. The simulation is run for 0.45 s with a time step size of 0.000001 s. The case is initialized from an existing simulation, with the `DTF` simulation starting at 0.35 s and running for an additional 0.1 s. The case setup is provided, with tutorials available to the user. The `polyMesh` folder, which contains the mesh files and simulated data files at time step 0.35, is available as a separate download link from the server due to its large size. After downloading those folders, copy the `polyMesh` folder to `constant` directory and keep the `orig_0.35` folder in the Case 02 directory.

5.2.1 Validation

The `DTF` simulation model utilizes the laminar flame speed (S_l^0) and laminar flame thickness (δ_0) to compute the thickening factor (F) and the Colin efficiency function (E). The results obtained from the `DTF` model are compared with experimental data and reference numerical datasets, including the AVBP thickened flame model [22] and the PaSR [24] implementation in OpenFOAM with $C_{\text{mix}} = 1$. Figure 5.5 presents contour plots of the thickening factor (F), flame sensor (S), and the absolute total reaction rate of all species. The values of F and S are dynamically calculated for each cell and iteration.

The PaSR study conducted by Sehole et al. [24], investigates premixed hydrogen flames in an AHEAD burner. These simulations were performed on three different mesh resolutions: coarse, medium, and fine. The results from the fine mesh are shown in Fig. 5.6 for comparison with the `DTF`

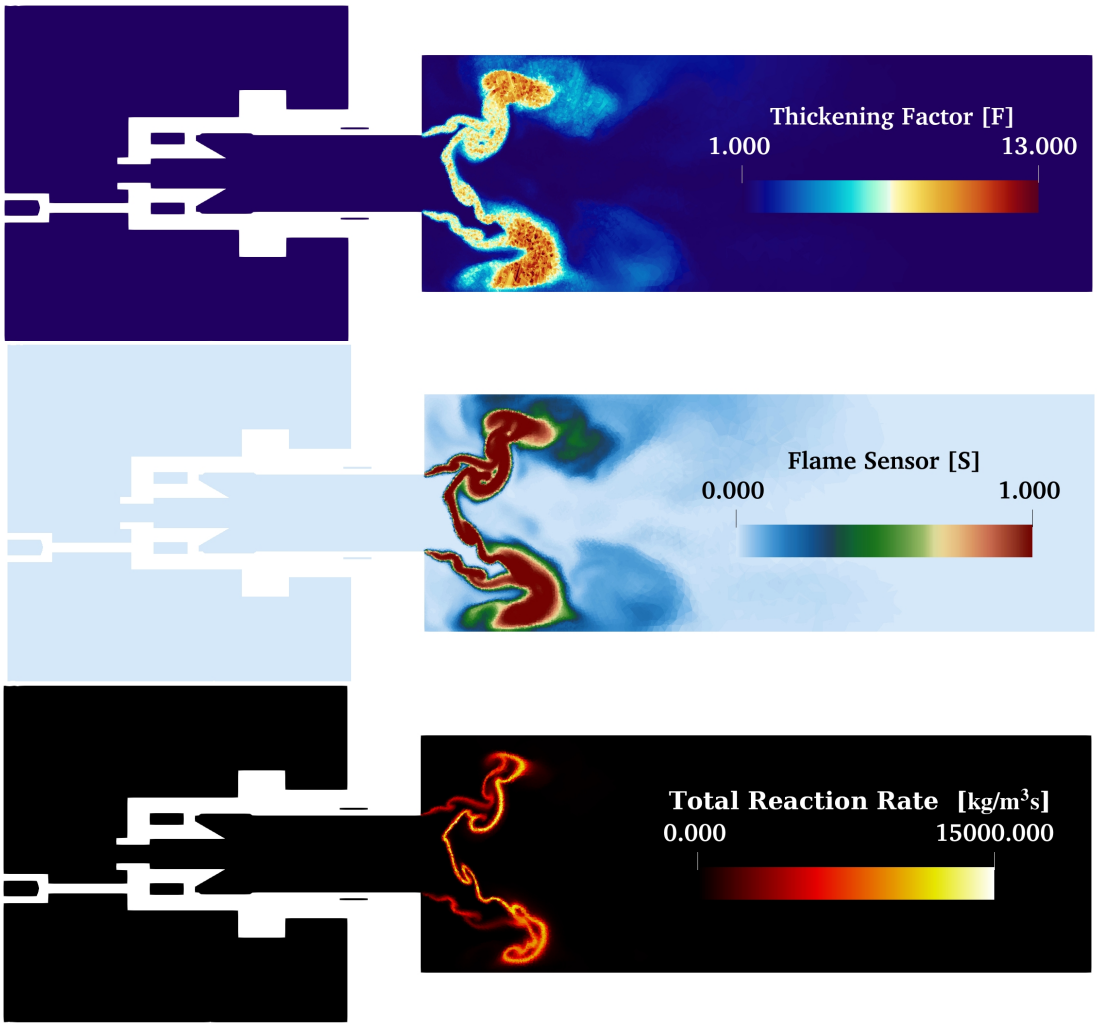

Figure 5.5: Contour plots showing (from top to bottom) the flame thickening factor (F), flame sensor (S), and absolute total reaction rate.

model results.

In this report, the DTF results, based on a coarse mesh, demonstrate better performance than the PaSR results obtained on a fine mesh, as illustrated in Fig. 5.6. A detailed analysis reveals that the velocity profiles predicted by the DTF model closely match those produced by the AVBP model. However, significant differences are observed in the temperature and OH species profiles between the PaSR and DTF models. Although additional references are required to fully assess these results, a comparison with the experimental OH-LIF contour plots by Reichel et al. [23] indicates improvements in the DTF results relative to the PaSR model. The DTF model predicts lower OH values in the outer recirculation zone, aligning more closely with experimental observations.

It is worth mentioning that the DTF method discussed herein still needs to be further studied. Currently, the author is working on a reactive hydrogen jet in cross-flow case in order to understand better.

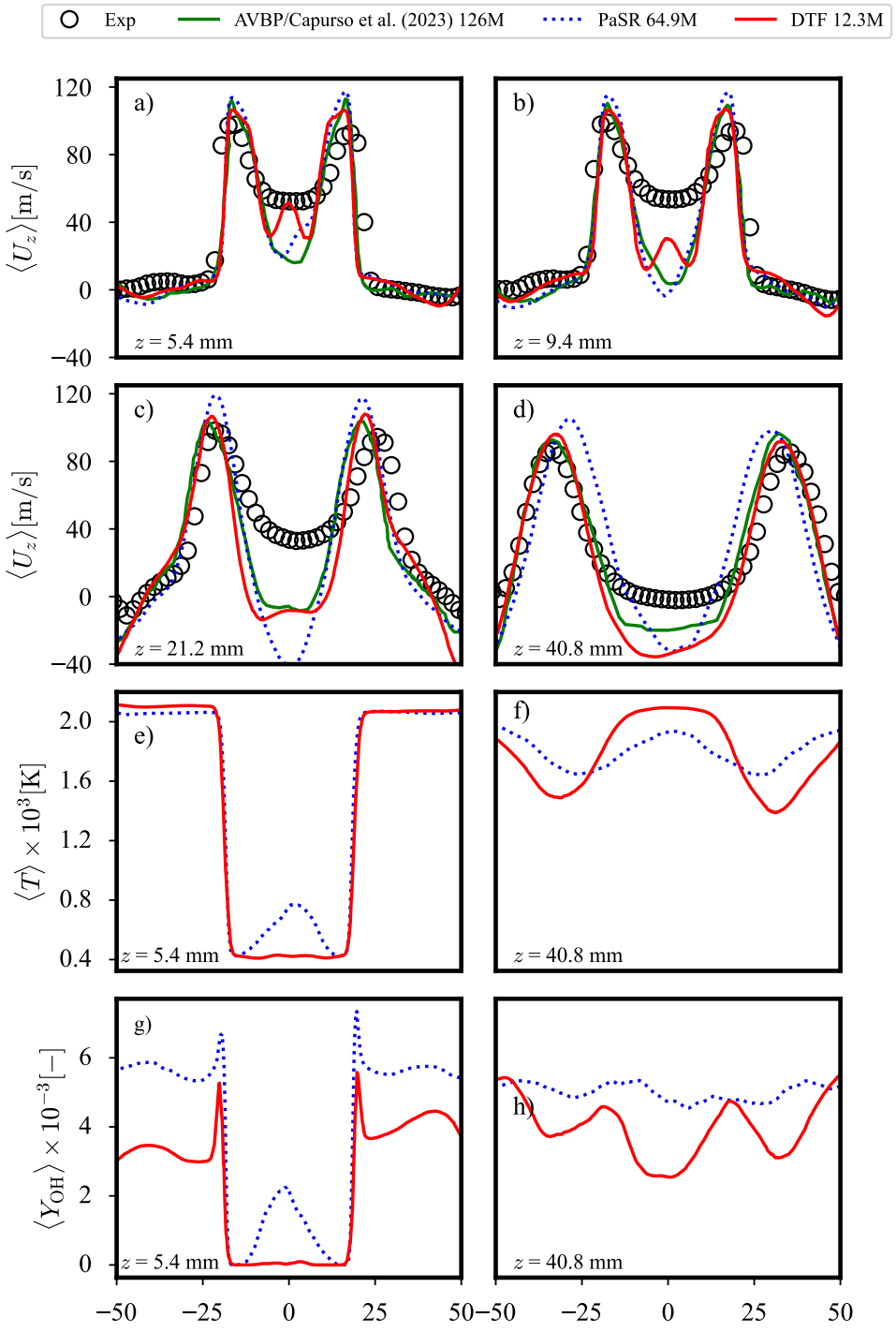

Figure 5.6: Plots (a)–(d) present numerical results for the mean axial velocity at various heights of the burner, compared with experimental data, the AVBP study by Capurso et al. [22], the PaSR study [24], and the current `DTF` implementation. Plots (e)–(h) show the mean temperature and OH mass fraction, comparing the `DTF` and PaSR results.

Index

Bibliography

[1] Aleksi Rintanen, "TFMFoam," 2023. [Online]. Available: https://github.com/arintanen/TFMFoam

[2] A. Rintanen, "Numerical study on hydrogen flame instabilities in 2D using thickened flame approach," 2023.

[3] S. B. Pope, "Turbulent Flows," 8 2000. [Online]. Available: https://www.cambridge.org/core/product/identifier/9780511840531/type/book

[4] "The local structure of turbulence in incompressible viscous fluid for very large Reynolds numbers," *Proceedings of the Royal Society of London. Series A: Mathematical and Physical Sciences*, vol. 434, no. 1890, pp. 9–13, 7 1991.

[5] V. John, "Large Eddy Simulation of Turbulent Incompressible Flows," vol. 34, 2004. [Online]. Available: http://link.springer.com/10.1007/978-3-642-18682-0

[6] F. Nicoud and F. Ducros, "Subgrid-scale stress modelling based on the square of the velocity gradient tensor," *Flow, Turbulence and Combustion*, vol. 62, no. 3, pp. 183–200, 1999. [Online]. Available: https://link.springer.com/article/10.1023/A:1009995426001

[7] N. Peters, "Turbulent Combustion," *Turbulent Combustion*, 8 2000. [Online]. Available: https://www.cambridge.org/core/books/turbulent-combustion/4A93A00CCA922A28D8E5316744D8CF8F

[8] "combustionModel Class Reference — OpenFOAM Source Code Guide." [Online]. Available: https://cpp.openfoam.org/v10/classFoam_1_1combustionModel.html

[9] B. MAGNUSSEN, "On the structure of turbulence and a generalized eddy dissipation concept for chemical reaction in turbulent flow," 1 1981. [Online]. Available: https://arc.aiaa.org/doi/10.2514/6.1981-42

[10] I. R. Gran and B. F. Magnussen, "A Numerical Study of a Bluff-Body Stabilized Diffusion Flame. Part 2. Influence of Combustion Modeling And Finite-Rate Chemistry," *COMBUSTION SCIENCE AND TECHNOLOGY*, vol. 119, no. 1-6, pp. 191–217, 1996. [Online]. Available: https://www.tandfonline.com/doi/abs/10.1080/00102209608951999

[11] B. M. E. t. c. o. computational and u. 2005, "The eddy dissipation concept: A bridge between science and technology," *folk.ntnu.noBF MagnussenECCOMAS thematic conference on computational combustion, 2005•folk.ntnu.no*. [Online]. Available: https://folk.ntnu.no/ivarse/edc/BFM_ECOMAS2005_Lisboa.pdf

[12] A. Parente, M. R. Malik, F. Contino, A. Cuoci, and B. B. Dally, "Extension of the Eddy Dissipation Concept for turbulence/chemistry interactions to MILD combustion," *Fuel*, vol. 163, pp. 98–111, 1 2016.

[13] Legier. J.P., "Dynamically thickened flame LES model for premixed and non-premixed turbulent combustion." [Online]. Available: https://web.stanford.edu/group/ctr/ctrsp00/poinsot.pdf

[14] "The OpenFOAM Source Code Guide — OpenFOAM v10 — The OpenFOAM Foundation." [Online]. Available: https://cpp.openfoam.org/v10/

[15] F. A. Williams, "Combustion theory: The fundamental theory of chemically reacting flow systems, second edition," *Combustion Theory: The Fundamental Theory of Chemically Reacting Flow Systems, Second Edition*, pp. 1–680, 1 2018.

[16] "PaSR Class Reference — OpenFOAM Source Code Guide." [Online]. Available: https://cpp.openfoam.org/v10/classFoam_1_1combustionModels_1_1PaSR.html

[17] "OpenFOAM: src/thermophysicalModels/chemistryModel/chemistryModel/chemistryModel.C Source File." [Online]. Available: https://cpp.openfoam.org/v10/chemistryModel_8C_source.html#l00446

[18] E.-M. Wartha, M. Bösenhofer, and M. Harasek, "Characteristic Chemical Time Scales for Reactive Flow Modeling," *Combustion Science and Technology*, vol. 193, no. 16, pp. 2807–2832, 2020. [Online]. Available: https://www.k1-met.com/fileadmin/user_upload/Publications/Journal_articles_open_access/Wartha_et_al_2020.pdf

[19] O. Colin, "Simulations aux grandes échelles de la combustion turbulente prémélangée dans les statoréacteurs," 2000.

[20] O. Colin, F. Ducros, D. Veynante, and T. Poinsot, "A thickened flame model for large eddy simulations of turbulent premixed combustion," *Physics of Fluids*, vol. 12, no. 7, pp. 1843–1863, 2000.

[21] A. Ballotti, S. Castellani, and A. Andreini, "A Dynamic Thickening Strategy for High-Fidelity Computational Fluid Dynamics Analyses of Multi-Regime Combustion," 2024. [Online]. Available: http://asmedigitalcollection.asme.org/gasturbinespower/article-pdf/146/12/121010/7374751/gtp_146_12_121010.pdf?casa_token=d7I3UnjK_OMAAAAA:Z3i7fx3aDg0LHR_SGT2vI0YsCjpdo2trvJ0UXwx0e0Z4qGWpodGgdYv8e-C-EI7_vMATIO8

[22] T. Capurso, D. Laera, E. Riber, and B. Cuenot, "NOx pathways in lean partially premixed swirling H2-air turbulent flame," *Combustion and Flame*, vol. 248, p. 112581, 2 2023.

[23] T. G. Reichel and C. O. Paschereit, "Interaction mechanisms of fuel momentum with flashback limits in lean-premixed combustion of hydrogen," *International Journal of Hydrogen Energy*, vol. 42, no. 7, pp. 4518–4529, 2 2017.

[24] H. A. H. Sehole, I. Morev, A. A. Rintanen, Z. A. A. Shahin, P. Tamadonfar, S. Karimkashi, A. Wehrfritz, and V. Vuorinen, "Dlbfoam: An Efficient Open-Source Cfd Suite for Hydrogen Combustion with Enhanced Diffusion Models," 2024. [Online]. Available: https://papers.ssrn.com/abstract=4997744

[25] "Accurate 3D Mesh Generation Software for CFD Simulations." [Online]. Available: https://ennova-cfd.com/

Study questions

- Which type of turbulence chemistry interaction models are available in OpenFOAM?
- How to implement the DTF after compiling it?
- What is the purpose of the turbulence chemistry model and why are they required?
- What does the laminar flame speed function do?
- What is the purpose of calculating the flame sensor, thickening factor, and efficiency function?
- How is DTF, as a TCI model, different from PaSR?
- How does a turbulence-chemistry interaction model access information and functionality in the chemistry model?

Appendix A

Developed Codes

Appendix A consists of the code that was developed during this implementation. The complete code you can download from , visit https://github.com/hahspk/DTF

A.1 Dynamic Thickened Flame Header, `DTF.H`

src/combustionModels/DTF/DTF.H

```
/*---------------------------------------------------------------------------*\
  =========                 |
  \\      /  F ield         | OpenFOAM: The Open Source CFD Toolbox
   \\    /   O peration     | Website:  https://openfoam.org
    \\  /    A nd           | Copyright (C) 2011-2020 OpenFOAM Foundation
     \\/     M anipulation  |
-------------------------------------------------------------------------------
License
    This file is part of OpenFOAM.

    OpenFOAM is free software: you can redistribute it and/or modify it
    under the terms of the GNU General Public License as published by
    the Free Software Foundation, either version 3 of the License, or
    (at your option) any later version.

    OpenFOAM is distributed in the hope that it will be useful, but WITHOUT
    ANY WARRANTY; without even the implied warranty of MERCHANTABILITY or
    FITNESS FOR A PARTICULAR PURPOSE.  See the GNU General Public License
    for more details.

    You should have received a copy of the GNU General Public License
    along with OpenFOAM.  If not, see <http://www.gnu.org/licenses/>.

Class
    Foam::combustionModels::DTF

Description
    Partially stirred reactor turbulent combustion model.

    This model calculates a finite rate, based on both turbulence and chemistry
    time scales.  Depending on mesh resolution, the Cmix parameter can be used
    to scale the turbulence mixing time scale.

SourceFiles
    DTF.C

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

#ifndef DTF_H
#define DTF_H
```

```
#include "laminar.H"
#include "flameSensor.H"
#include "efficiencyFunction.H"
#include "autoPtr.H"
//#include "basicChemistryModel.H"


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

namespace Foam
{
namespace combustionModels
{

/*---------------------------------------------------------------------------*\
                           Class DTF Declaration
\*---------------------------------------------------------------------------*/

class DTF
:
    public laminar
{
    // Private Data
        scalar Fmax_;
        volScalarField F_, EF_,tRR_, D_,Fs_;
        autoPtr<flameSensor> flameSensor_;
        autoPtr<efficiencyFunction> efficiencyFunction_;
        scalar deltaL_;
        scalar SL_;
        scalar upperLimit_;
        scalar lowerLimit_;
        scalar filters_;
    protected:
        // Protected Data
        //- Pointer to chemistry model
        autoPtr<basicChemistryModel> chemistryPtr_;
    public:

    //- Runtime type information
    TypeName("DTF");


    // Constructors

        //- Construct from components
        DTF
        (
            const word& modelType,
            const fluidReactionThermo& thermo,
            const compressibleMomentumTransportModel& turb,
            const word& combustionProperties
        );

        //- Disallow default bitwise copy construction
        DTF(const DTF&);


        //- Destructor
        virtual ~DTF();


    // Member Functions

        //- Correct combustion rate
        virtual void correct();

        void flameSpeed();
```

```
        //- Fuel consumption rate matrix.
        virtual tmp<fvScalarMatrix> R(volScalarField& Y) const;

        //- Heat release rate [kg/m/s^3]
        virtual tmp<volScalarField> Qdot() const;

        scalar deltaL() const { return deltaL_; }
        scalar SL() const { return SL_; }
        scalar filters() const{return filters_;}

        const volScalarField& F() const{return F_;}


        //- Update properties from given dictionary
        virtual bool read();


    // Member Operators

        //- Disallow default bitwise assignment
        void operator=(const DTF&) = delete;
};


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

} // End namespace combustionModels
} // End namespace Foam

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

#endif

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

A.2 Dynamic Thickened Flame Constructor, DTF.C

src/combustionModels/DTF/DTF.C

```
/*---------------------------------------------------------------------------*\
  =========                 |
  \\      /  F ield         | OpenFOAM: The Open Source CFD Toolbox
   \\    /   O peration     | Website:  https://openfoam.org
    \\  /    A nd           | Copyright (C) 2011-2021 OpenFOAM Foundation
     \\/     M anipulation  |
-------------------------------------------------------------------------------
License
    This file is part of OpenFOAM.

    OpenFOAM is free software: you can redistribute it and/or modify it
    under the terms of the GNU General Public License as published by
    the Free Software Foundation, either version 3 of the License, or
    (at your option) any later version.

    OpenFOAM is distributed in the hope that it will be useful, but WITHOUT
    ANY WARRANTY; without even the implied warranty of MERCHANTABILITY or
    FITNESS FOR A PARTICULAR PURPOSE.  See the GNU General Public License
    for more details.

    You should have received a copy of the GNU General Public License
    along with OpenFOAM.  If not, see <http://www.gnu.org/licenses/>.

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

#include "DTF.H"
```

```
#include "addToRunTimeSelectionTable.H"
#include "fvcSmooth.H"

// * * * * * * * * * * * * * * * Static Data Members * * * * * * * * * * * * * * * //

namespace Foam
{
namespace combustionModels
{
    defineTypeNameAndDebug(DTF, 0);
    addToRunTimeSelectionTable(combustionModel, DTF, dictionary);
}
}


// * * * * * * * * * * * * * * * * * Constructors  * * * * * * * * * * * * * * * * * //

Foam::combustionModels::DTF::DTF
(
    const word& modelType,
    const fluidReactionThermo& thermo,
    const compressibleMomentumTransportModel& turb,
    const word& combustionProperties
)
:
    laminar(modelType, thermo, turb, combustionProperties),
    Fmax_(this->coeffs().template lookup<scalar>("Fmax")),
    F_
     (
         IOobject
         (
             "F",
             //this->mesh().time().name(),
             this->mesh().time().timeName(),
             this->mesh(),
             IOobject::NO_READ,
             IOobject::NO_WRITE
         ),
         this->mesh(),
         dimensionedScalar(dimless, 1)
     ),
    EF_
     (
         IOobject
         (
             "EF",
             //this->mesh().time().name(),
             this->mesh().time().timeName(),
             this->mesh(),
             IOobject::NO_READ,
             IOobject::NO_WRITE
         ),
         this->mesh(),
         dimensionedScalar(dimless, 1)
     ),
    tRR_
    (
        IOobject
        (
            "tRR",
            mesh_.time().timeName(),
            mesh_,
            IOobject::NO_READ,
            IOobject::NO_WRITE
        ),
        mesh_,
        dimensionedScalar("tRR", dimMass/dimVolume/dimTime, Zero)
    ),
```

```
    D_
    (
        IOobject
        (
            "D",
            mesh_.time().timeName(),
            mesh_,
            IOobject::NO_READ,
            IOobject::NO_WRITE
        ),
        mesh_,
        dimensionedScalar("D", dimLength*dimLength/dimTime, Zero)
    ),
       Fs_
     (
         IOobject
         (
             "Fs",
             //this->mesh().time().name(),
             this->mesh().time().timeName(),
             this->mesh(),
             IOobject::NO_READ,
             IOobject::NO_WRITE
         ),
         this->mesh(),
         dimensionedScalar(dimless, 1)
     ),
    flameSensor_(flameSensor::New(coeffs(),turb.mesh())),
    efficiencyFunction_(efficiencyFunction::New(turb.mesh(),coeffs(),turb,Fmax_, *this)),
    deltaL_(0.001),
    SL_(3),
    upperLimit_(this->coeffs().template lookup<scalar>("upperLimit")),
    lowerLimit_(this->coeffs().template lookup<scalar>("lowerLimit")),
    filters_(this->coeffs().template lookup<scalar>("filters")),
    chemistryPtr_(basicChemistryModel::New(thermo))
{}

// * * * * * * * * * * * * * * * * Destructor  * * * * * * * * * * * * * * * * //

Foam::combustionModels::DTF::~DTF()
{}

// * * * * * * * * * * * * * * * * Member Functions  * * * * * * * * * * * * * * * * //

void Foam::combustionModels::DTF::correct()
{
    laminar::correct();
    flameSensor_->correct();
    efficiencyFunction_->correct();
    ////////////////////////////////////////////////////////////////////////
    ///FlameSPeed Switch, by default it is true. Set to False incase of cold (non-reactive Flows)
    const bool FlameSpeed = this->coeffs().lookupOrDefault("FlameSpeed", true);
    if (FlameSpeed)
    {
        flameSpeed();
    }
    ////////////////////////Cell_Size_Implementation////////////////////////
    //loop will calculate the dyanmic thickened factor according to the cell size and flame thickness.
    forAll(F_, celli) {
    //Cell Size
    scalar delta_g = pow(mesh_.V()[celli], 1.0 / static_cast<scalar>(mesh_.nGeometricD()));
    if (deltaL_ > SMALL) {
        //thickening Factor
        scalar Fvalue = (delta_g* this->coeffs().template lookup<scalar>("N") / (deltaL_ ));
        //switch to choose the minimum of thickening factor, between calculated thickening factor and
     user defined maximum value.
        Fs_[celli] = min(Fvalue, Fmax_);
```

```
        scalar sensorValue = flameSensor_->S()[celli];
        //Applying thickening to the flame
        F_[celli] = 1 + (Fs_[celli] - 1) * sensorValue;
        //check if flame thickening is less then 1.
        if (F_[celli] < 1) {
            F_[celli] = 1;
        }

        ////////////////////////Log////////////////////////////////////
        Info << "Maximum Thickening Factor, F_: " << Foam::gMax(tRR_) << endl;
        Info << "Minimum Thickening Factor, F_: " << Foam::gMin(tRR_) << endl;

        } else {
            F_[celli] = 1.0;
        ////////////////////////////log/////////////////////////////////////////
        if (celli % 100 == 0)  // Progress indicator and debugging every 100 cells
        {
            Info << "Cell " << celli << ": deltaL_ too small, set F_[" << celli << "] to " << "Fmin"
     << endl;
        }
        }
    }
        Info << "Maximum Thickening Factor, Fs_: " << Foam::gMax(Fs_) << endl;
        Info << "Minimum Thickening Factor, Fs_: " << Foam::gMin(Fs_) << endl;
        Info << "Maximum Thickening Factor, F_: " << Foam::gMax(F_) << endl;
        Info << "Minimum Thickening Factor, F_: " << Foam::gMin(F_) << endl;
    ////////////////////////Efficiency into Thickening Factor//////////
    forAll(EF_, celli) {
    EF_[celli] = efficiencyFunction_->E()[celli] * F_[celli];
    }
        Info << "Maximum Efficiency Factor, E_: " << Foam::gMax(efficiencyFunction_->E()) << endl;
        Info << "Minimum Efficiency Factor, E_: " << Foam::gMin(efficiencyFunction_->E()) << endl;
}


Foam::tmp<Foam::fvScalarMatrix>
Foam::combustionModels::DTF::R(volScalarField& Y) const
{
    return efficiencyFunction_->E()/F_*laminar::R(Y);
}

Foam::tmp<Foam::volScalarField>
Foam::combustionModels::DTF::Qdot() const
{
    return volScalarField::New
    (
        this->thermo().phasePropertyName(typeName + ":Qdot"),
        efficiencyFunction_->E()/F_*laminar::Qdot()
    );
}


void Foam::combustionModels::DTF::flameSpeed()
{
    // get number of species
    const label nSpecie = chemistryPtr_->nSpecie();
    Info << "Number of species: " << nSpecie << endl;

    //get number of reactions
    const label nReaction = chemistryPtr_->nReaction();
    Info << "Number of reactions: " << nReaction << endl;

    ////////////////////////// Reaction Rate Calculations ////////////////////

    // Reset reactionRate for all cells to zerro
    forAll(tRR_.internalField(), celli)
    {
        tRR_[celli] = 0.0;
```

```
    }

    // calculate the reaction rate for all the recations and species
    for (label ri = 0; ri < nReaction; ++ri)
    {
        for (label si=0; si<nSpecie; si++)
        {
        volScalarField::Internal RR = Foam::mag(chemistryPtr_->calculateRR(ri, si));
        forAll(RR, celli)
        {
            if (!std::isnan(F_[celli]) && F_[celli] > SMALL)
            {
                tRR_[celli] +=  RR[celli];
            }
        }
        }
    }
    volScalarField smoothed_tRR = tRR_; // Initialize smoothed field with raw tRR_

    //smoothing filter
    fvc::smooth(smoothed_tRR, filters_);
    tRR_ = smoothed_tRR;
    ////////////////////////// Log //////////////////////////
    dimensionedScalar maxSumRR_(tRR_.dimensions(),gMax(tRR_));
    Info << "Maximum reaction rate, maxSumRR_: " << maxSumRR_ << endl;
    Info << "Minimum reaction rate, minSumRR_: " << Foam::gMin(tRR_) << endl;
    ////////////////////////////////////////////////////////////////////////////
    //Flame Sensor
    volScalarField q_sensor = flameSensor_->S();
    scalar sumReactionRate = 0.0;
    scalar count = 0.0;
    Info << "Internal field size: " << q_sensor.internalField().size() << endl;
    //Extracting values reaction rates between the set values of flame sensor
    forAll(q_sensor.internalField(), celli)
    {
        if (q_sensor[celli] >= lowerLimit_ && q_sensor[celli] <= upperLimit_)
        {
            // Summing reaction rates and counting cells
            sumReactionRate += tRR_[celli];
            count++;
        }
    }
    // Parallel reduction for sum and count
    scalar globalSumReactionRate = sumReactionRate;
    scalar globalCount = count;
    reduce(globalSumReactionRate, sumOp<scalar>());
    reduce(globalCount, sumOp<scalar>());

    // Compute average reaction rate across all processors
    scalar avgReactionRate = (globalCount > 0) ? (globalSumReactionRate / globalCount) : 0.0;

    Info << "Average reaction rate (based on sensor): " << avgReactionRate << endl;

    ////////////////////////// Diffusion Calculations //////////////////////////
    const volScalarField& rhoField = Foam::combustionModel::thermo_.rho();
    const volScalarField& kappaField = Foam::combustionModel::thermo_.kappa();
    const volScalarField& CpField = Foam::combustionModel::thermo_.Cp();

    forAll(this->mesh().V(), celli)
    {
        scalar rho = rhoField[celli];
        scalar lambda = kappaField[celli];
        scalar cp = CpField[celli];

        if (rho > SMALL && cp > SMALL)
        {
            D_[celli] =  lambda / ((rho ) * (cp));
        }
```

```
        else
        {
            D_[celli] = 0;
        }
    }
        Info << "Maximum Diffusion, D_:" << Foam::gMax(D_) << endl;
        Info << "Minimum Diffusion, D_:" << Foam::gMin(D_) << endl;

        //Smootinh Filter
        volScalarField smoothed_D = D_;
        fvc::smooth(smoothed_D, filters_);
        D_ = smoothed_D;
        scalar maxD = Foam::gMax(D_);
        Info << "Maximum Diffusion: smoothed_D" << maxD << endl;
        Info << "Minimum Diffusion: smoothed_D" << Foam::gMin(D_) << endl;

        scalar sumD = 0.0;
        scalar countD = 0.0;
        //Extracting Values of diffusion between the set values of flame sensor
        forAll(q_sensor.internalField(), celli)
        {
            if (q_sensor[celli] >= lowerLimit_ && q_sensor[celli] <= upperLimit_)
            {
                sumD += D_[celli];
                countD++;
            }
        }

        // Parallel reduction for sumD and countD
        scalar globalSumD = sumD;
        scalar globalCountD = countD;
        reduce(globalSumD, sumOp<scalar>());
        reduce(globalCountD, sumOp<scalar>());

        // Compute average D across all processors
        scalar avgD = (globalCountD > 0) ? (globalSumD / globalCountD) : 0.0;

        Info << "Average Diffusion (based on sensor): " << avgD << endl;

        ////////////////////// SL_ and deltaL_ Calculation //////////////////////
        Info << "Starting Lamianr flame speed (SL_) and Laminar flame thickness (deltaL_) calculation
    ..." << endl;

            if ( avgReactionRate > SMALL && avgD > SMALL )
            {
                SL_ = sqrt(avgReactionRate * avgD);
                Info << "Laminar Flame Speed: " << SL_ << endl;

                deltaL_ = avgD /SL_;

                Info << "Laminar Flame thickness: " << SL_ << endl;

                if (SL_ > 50)
                {
                    Info <<  " SL_ value " << SL_ << " exceeds 50, clamping to 50." << endl;
                    SL_ = 50;
                }

                if (deltaL_ > 10)
                {
                    Info << "deltaL_ value " << deltaL_ << " exceeds 10, clamping to 10." << endl;
                    deltaL_ = 10;
                }
            }
            else
            {
                SL_ = this->coeffs().template lookup<scalar>("SL");
                deltaL_ = this->coeffs().template lookup<scalar>("deltaL");
```

```
            }
        Info << "Completed Lamianr flame speed  (SL_) and Laminar flame thickness (deltaL_)
    calculation." << endl;
}


bool Foam::combustionModels::DTF::read()
{
    if (laminar::read())
    {
        this->coeffs().lookup("Fmax") >> Fmax_;
        return true;
    }
    else
    {
        return false;
    }
}


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

A.3 Efficiency Function: Colin Header, colin.H

src/combustionModels/EfficiencyFunctions/ColinEfficiencyFunction/colin.H

```
#ifndef colin_H
#define colin_H

#include "efficiencyFunction.H"
#include "simpleFilter.H"
#include "momentumTransportModel.H"

namespace Foam
{
namespace efficiencyFunctionModels
{

/*---------------------------------------------------------------------------*\
                        Class colin Declaration
\*---------------------------------------------------------------------------*/

class colin
:
    public efficiencyFunction
{
    // Private Data
    dictionary coeffsDict_;
    scalar alpha_;
    const Foam::combustionModels::DTF& dtfModel_;
    volScalarField uPrime_;
   simpleFilter sFilter_;


public:

    //- Runtime type information
    TypeName("colin");

    // Constructors
    colin
    (
        const dictionary&,
        const fvMesh&,
        const compressibleMomentumTransportModel& turb,
```

```
        scalar F,
        const Foam::combustionModels::DTF& dtfModel
    );

    // Disallow default bitwise copy construction
    colin(const colin&) = delete;

    // Destructor
    virtual ~colin();

/*
    //- Access function to filter width
    inline const volScalarField& delta() const
        {
            return delta();
        }
*/

    // Correct function
    virtual void correct();

    // Disallow default bitwise assignment
    void operator=(const colin&) = delete;
};

} // End namespace efficiencyFunctionModels
} // End namespace Foam

#endif
```

A.4 Efficiency Function Colin Constructor, colin.C

src/combustionModels/EfficiencyFunctions/ColinEfficiencyFunction/colin.C

```
#include "colin.H"
#include "addToRunTimeSelectionTable.H"
#include "fvcCurl.H"
#include "fvcLaplacian.H"
#include "DTF.H"

namespace Foam
{
namespace efficiencyFunctionModels
{
    defineTypeNameAndDebug(colin, 0);

    addToRunTimeSelectionTable
    (
        efficiencyFunction,
        colin,
        dictionary
    );

} // namespace efficiencyFunctionModels
} // namespace Foam

// Constructors
Foam::efficiencyFunctionModels::colin::colin
(
    const dictionary& dict,
    const fvMesh& mesh,
    const compressibleMomentumTransportModel& turb,
    scalar Fmax,
    const Foam::combustionModels::DTF& dtfModel
)
```

```
:
    efficiencyFunction(dict, mesh, turb, Fmax, dtfModel),
    coeffsDict_(dict.subDict("colinCoeffs")),
    alpha_(coeffsDict_.lookup<scalar>("alpha")),
    dtfModel_(dtfModel),
    uPrime_
    (
        IOobject
        (
            "uPrime",
            mesh.time().timeName(),
            mesh,
            IOobject::NO_READ,
            IOobject::AUTO_WRITE
        ),
        mesh,
        dimensionedScalar("", dimLength/dimTime, 0.0)
    ),
    sFilter_(mesh_)
{}


void Foam::efficiencyFunctionModels::colin::correct() {

    scalar delta_L = dtfModel_.deltaL();
    scalar SL_ = dtfModel_.SL();
    scalar Filters_ = dtfModel_.filters();
    const volScalarField& delta_ = mesh_.lookupObject<volScalarField>("delta");
    const volScalarField& Fc_ = dtfModel_.F();



    // Check validity of inputs
    if (delta_L <= 0) {
        FatalErrorInFunction << "Invalid flame thickness (delta_L): " << delta_L << exit(FatalError);
    }
    if (SL_ <= 0) {
        FatalErrorInFunction << "Invalid flame speed (SL_): " << SL_ << exit(FatalError);
    }

    Info << "Flame Thickness (delta_L): " << delta_L << endl;
    Info << "Flame Speed (SL_): " << SL_ << endl;

    // Calculate uPrime_ using filtered U field
    volVectorField U_hat(turb_.U());
    Info << "Uhat_ computed successfully." << endl;

    for (int i = 0; i < Filters_; i++) {
        U_hat = sFilter_(U_hat);
    }

    // Ensure delta_ is valid
    if (delta_.internalField().size() == 0) {
        FatalErrorInFunction << "delta_ field is empty or uninitialized." << exit(FatalError);
    }

    uPrime_ = 2.0 * mag(pow(delta_, 3) * fvc::laplacian(fvc::curl(U_hat)));




    // Calculate the efficiency
    forAll(E_, celli) {
        scalar delta_e = Fc_[celli] * delta_L;

        if (delta_e <= 0) {
            WarningInFunction << "Invalid delta_e at cell " << celli << endl;
```

```
            E_[celli] = 1.0; // Assign minimum efficiency to avoid invalid calculations
            continue;
        }

        scalar delta_r = delta_e / delta_L;
        scalar delta_R = delta_e / (delta_L * Fc_[celli]);

        if (delta_r <= 0 || delta_R <= 0) {
            FatalErrorInFunction << "Invalid delta_r or delta_R values." << exit(FatalError);
        }

        scalar up_SL_r = (uPrime_[celli] * delta_e) / SL_;

        if (up_SL_r <= 0) {
            WarningInFunction << "Invalid up_SL_r at cell " << celli << endl;
            E_[celli] = 1.0; // Assign minimum efficiency to avoid invalid calculations
            continue;
        }

        scalar numerator = 1.0 + alpha_ * 0.75 * exp(-1.2 / pow(up_SL_r, 0.3)) * pow(delta_r, 2.0 /
    3.0) * up_SL_r;
        scalar denominator = 1.0 + alpha_ * 0.75 * exp(-1.2 / pow(up_SL_r, 0.3)) * pow(delta_R, 2.0 /
    3.0) * up_SL_r;

        if (denominator <= 0) {
            WarningInFunction << "Invalid denominator at cell " << celli << endl;
            E_[celli] = 1.0; // Assign minimum efficiency to avoid division by zero
            continue;
        }

        scalar efficiency = numerator / denominator;

        // Enforce the condition E >= 1
        E_[celli] = max(efficiency, 1.0);
    }
}

// Destructor
Foam::efficiencyFunctionModels::colin::~colin()
{}
```

Appendix B

Case 01 Setup

B.1 Allclean and Allrun Scripts

Listing B.1: `Allrun`

```
#!/bin/bash
#------------------------------------------------------------------------------
blockMesh > log.blockMesh
cp -rp orig_0 0
setFields -dict setFieldsDict > log.setFields
setFields -dict setFieldsDict2 > log.setFields2
decomposePar -force > log.decomposePar
mpirun -np 4 reactingFoam -parallel > log.run
reconstructPar > log.reconstructPar
#------------------------------------------------------------------------------
```

Listing B.2: `Allclean`

```
#!/bin/bash

rm -rf 0
rm -r constant/polyMesh
rm -r 0.*
rm -r processor*
rm log.*
rm -f TDAC
rm -rf dynamicCode
```

B.2 `0` Directory

Listing B.3: 0/U

```
/*--------------------------------*- C++ -*----------------------------------*\
  =========                 |
  \\      /  F ield         | OpenFOAM: The Open Source CFD Toolbox
   \\    /   O peration     | Website:  https://openfoam.org
    \\  /    A nd           | Version:  10_latest
     \\/     M anipulation  |
\*---------------------------------------------------------------------------*/
FoamFile
{
    format      binary;
    class       volVectorField;
    location    "0";
    object      U;
}
```

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

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

internalField   uniform (0.026 0 0);

boundaryField
{
    symm1
    {
        type            symmetryPlane;
    }
    inlet
    {
        type            fixedValue;
        value           uniform (0.026 0 0);
    }
    outlet
    {
        type            fixedValue;
        value           uniform (0 0 0);
    }
    symm2
    {
        type            symmetryPlane;
    }
    frontAndBack
    {
        type            zeroGradient;
    }
}


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

Listing B.4: 0/p

```
/*--------------------------------*- C++ -*----------------------------------*\
  =========                 |
  \\      /  F ield         | OpenFOAM: The Open Source CFD Toolbox
   \\    /   O peration     | Website:  https://openfoam.org
    \\  /    A nd           | Version:  10
     \\/     M anipulation  |
\*---------------------------------------------------------------------------*/
FoamFile
{
    format      ascii;
    class       volScalarField;
    location    "0";
    object      p;
}
// * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * //

dimensions      [1 -1 -2 0 0 0 0];

internalField   uniform 101325;

boundaryField
{
    inlet
    {
        type            zeroGradient;
    }
    outlet
    {
        type            zeroGradient;
    }
    symm1
```

```
    {
        type            symmetryPlane;
    }
    symm2
    {
        type            symmetryPlane;
    }
    frontAndBack
    {
        type            zeroGradient;
    }
}


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

Listing B.5: 0/T

```
/*--------------------------------*- C++ -*----------------------------------*\
  =========                 |
  \\      /  F ield         | OpenFOAM: The Open Source CFD Toolbox
   \\    /   O peration     | Website:  https://openfoam.org
    \\  /    A nd           | Version:  10
     \\/     M anipulation  |
\*---------------------------------------------------------------------------*/
FoamFile
{
    format      ascii;
    class       volScalarField;
    location    "0";
    object      T;
}
// * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * //

dimensions      [0 0 0 1 0 0 0];

internalField   uniform 300;

boundaryField
{
    inlet
    {
        type            fixedValue;
        value           uniform 300;
    }
    outlet
    {
       type            zeroGradient;
    }
    symm1
    {
        type            symmetryPlane;
    }
    symm2
    {
        type            symmetryPlane;
    }
    frontAndBack
    {
        type            zeroGradient;
    }
}


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

Listing B.6: 0/H2

```
/*--------------------------------*- C++ -*----------------------------------*\
  =========                 |
  \\      /  F ield         | OpenFOAM: The Open Source CFD Toolbox
   \\    /   O peration     | Website:  https://openfoam.org
    \\  /    A nd           | Version:  10_latest
     \\/     M anipulation  |
\*---------------------------------------------------------------------------*/
FoamFile
{
    format      binary;
    class       volScalarField;
    location    "0";
    object      H2;
}
// * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * //

dimensions      [0 0 0 0 0 0 0];

internalField   uniform 0.014468;

boundaryField
{
    symm1
    {
        type            symmetryPlane;
    }
    inlet
    {
        type            fixedValue;
        value           uniform 0.014468;
    }
    outlet
    {
        type            zeroGradient;
    }
    symm2
    {
        type            symmetryPlane;
    }
    frontAndBack
    {
        type            zeroGradient;
    }
}


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

Listing B.7: 0/O2

```
/*--------------------------------*- C++ -*----------------------------------*\
  =========                 |
  \\      /  F ield         | OpenFOAM: The Open Source CFD Toolbox
   \\    /   O peration     | Website:  https://openfoam.org
    \\  /    A nd           | Version:  10_latest
     \\/     M anipulation  |
\*---------------------------------------------------------------------------*/
FoamFile
{
    format      binary;
    class       volScalarField;
    location    "0";
    object      O2;
}
// * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * //

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

```
18
19 internalField   uniform 0.229629;
20
21 boundaryField
22 {
23     symm1
24     {
25         type            symmetryPlane;
26     }
27     inlet
28     {
29         type            fixedValue;
30         value           uniform 0.229629;
31     }
32     outlet
33     {
34         type            zeroGradient;
35     }
36     symm2
37     {
38         type            symmetryPlane;
39     }
40     frontAndBack
41     {
42         type            zeroGradient;
43     }
44 }
45
46
47 // ************************************************************************* //
```

Listing B.8: 0/N2

```
1  /*--------------------------------*- C++ -*----------------------------------*\
2    =========                 |
3    \\      /  F ield         | OpenFOAM: The Open Source CFD Toolbox
4     \\    /   O peration     | Website:  https://openfoam.org
5      \\  /    A nd           | Version:  10_latest
6       \\/     M anipulation  |
7  \*---------------------------------------------------------------------------*/
8  FoamFile
9  {
10     format      binary;
11     class       volScalarField;
12     location    "0";
13     object      N2;
14 }
15 // * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * //
16
17 dimensions      [0 0 0 0 0 0 0];
18
19 internalField   uniform 0.755903;
20
21 boundaryField
22 {
23     symm1
24     {
25         type            symmetryPlane;
26     }
27     inlet
28     {
29         type            fixedValue;
30         value           uniform 0.755903;
31     }
32     outlet
33     {
34         type            zeroGradient;
35     }
```

```
    symm2
    {
        type            symmetryPlane;
    }
    frontAndBack
    {
        type            zeroGradient;
    }
}


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

Listing B.9: 0/alphat

```
/*--------------------------------*- C++ -*----------------------------------*\
  =========                 |
  \\      /  F ield         | OpenFOAM: The Open Source CFD Toolbox
   \\    /   O peration     | Website:  https://openfoam.org
    \\  /    A nd           | Version:  10
     \\/     M anipulation  |
\*---------------------------------------------------------------------------*/
FoamFile
{
    format      ascii;
    class       volScalarField;
    location    "0";
    object      alphat;
}
// * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * //

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

internalField   uniform 0;

boundaryField
{
    inlet
    {
        type            fixedValue;
        value           uniform 0;
    }
    outlet
    {
       type            zeroGradient;
    }
    symm1
    {
        type            symmetryPlane;
    }
    symm2
    {
        type            symmetryPlane;
    }
    frontAndBack
    {
        type            zeroGradient;
    }
}


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

Listing B.10: 0/nut

```
/*--------------------------------*- C++ -*----------------------------------*\
  =========                 |
```

```
  \\      /  F ield         | OpenFOAM: The Open Source CFD Toolbox
   \\    /   O peration     | Website:  https://openfoam.org
    \\  /    A nd           | Version:  10
     \\/     M anipulation  |
\*---------------------------------------------------------------------------*/
FoamFile
{
    format      ascii;
    class       volScalarField;
    location    "0";
    object      nut;
}
// * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * //

dimensions      [0 2 -1 0 0 0 0];

internalField   uniform 0;

boundaryField
{
    inlet
    {
        type            fixedValue;
        value           uniform 0;
    }
    outlet
    {
       type            zeroGradient;
    }
    symm1
    {
        type            symmetryPlane;
    }
    symm2
    {
        type            symmetryPlane;
    }
    frontAndBack
    {
        type            zeroGradient;
    }
}


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

B.3 constant Directory

Listing B.11: constant/chemistryProperties

```
/*--------------------------------*- C++ -*----------------------------------*\
  =========                 |
  \\      /  F ield         | OpenFOAM: The Open Source CFD Toolbox
   \\    /   O peration     | Website:  https://openfoam.org
    \\  /    A nd           | Version:  10
     \\/     M anipulation  |
\*---------------------------------------------------------------------------*/
FoamFile
{
    format      ascii;
    class       dictionary;
    location    "constant";
    object      chemistryProperties;
}
// * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * //
```

```
16
17 //#includeEtc "caseDicts/solvers/chemistry/TDAC/chemistryProperties.cfg"
18
19 chemistryType
20 {
21     solver            ode;
22 }
23
24
25 chemistry       on;
26
27 initialChemicalTimeStep 1e-07;
28
29
30 loadbalancing
31 {
32     active true;
33     log true;
34 }
35
36
37
38
39 odeCoeffs
40 {
41     solver          seulex;
42     eps             0.05;
43 }
44
45
46 tabulation
47 {
48     // Activate tabulation
49     active      on;
50
51     // Switch logging of the tabulation statistics and performance
52     log         on;
53
54     printProportion    off;
55
56     printNumRetrieve   off;
57
58     // Tolerance used for retrieve and grow
59     tolerance   1e-2;
60
61     // ISAT is the only method currently available
62     method    ISAT;
63
64     // Scale factors used in the definition of the ellipsoid of accuracy
65     scaleFactor
66     {
67         otherSpecies 1;
68         Temperature  2500;
69         Pressure     1e15;
70     deltaT      5000;
71     }
72
73     // Maximum number of leafs stored in the binary tree
74     maxNLeafs  2000;
75
76     // Maximum life time of the leafs (in time steps) used in unsteady
77     // simulations to force renewal of the stored chemPoints and keep the tree
78     // small
79     chPMaxLifeTime 100;
80
81     // Maximum number of growth allowed on a chemPoint to avoid distorted
82     // chemPoints
83     maxGrowth  10;
```

```

    // Number of time steps between analysis of the tree to remove old
    // chemPoints or try to balance it
    checkEntireTreeInterval  5;

    // Parameters used to decide whether to balance or not if the tree's depth
    // is larger than maxDepthFactor*log2(nLeafs) then balance the tree
    maxDepthFactor   2;

    // Try to balance the tree only if the size of the tree is greater
    minBalanceThreshold 30;

    // Activate the use of a MRU (most recently used) list
    MRURetrieve false;

    // Maximum size of the MRU list
    maxMRUSize 0;

    // Allow to grow points
    growPoints  true;

    // When mechanism reduction is used, new dimensions might be added
    // maxNumNewDim set the maximum number of new dimensions added during a
    // growth
    maxNumNewDim 10;

}

#include "reactionsCap"

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

Listing B.12: constant/combustionProperties

```
/*--------------------------------*- C++ -*----------------------------------*\
  =========                 |
  \\      /  F ield         | OpenFOAM: The Open Source CFD Toolbox
   \\    /   O peration     | Website:  https://openfoam.org
    \\  /    A nd           | Version:  10
     \\/     M anipulation  |
\*---------------------------------------------------------------------------*/
FoamFile
{
    format      ascii;
    class       dictionary;
    location    "constant";
    object      combustionProperties;
}
// * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * //


combustionModel  DTF;

DTFCoeffs
{

//////////////////Flame_Speed//////////////////////////
     fuel
     {
     fuelSpecie H2;
     }
//////////////////////////Efficiency_Functions//////////////
    //efficiencyFunction none;
    //efficiencyFunction powerLaw;
    efficiencyFunction colin;


    powerLawCoeffs
```

```
35      {
36      beta 0.5;
37      n_filters 5;  // Number of filtering operations
38      }
39
40
41      colinCoeffs
42      {
43      alpha 0.02;
44      n_filters 10;  // Number of filtering operations
45      }
46  ////////////////////DTF_settings//////////////////////////
47      Fmax          11;           // Maximum thickening factor
48      N             15;           // Number of Cells
49      deltaL    0.001;
50      SL            3;
51
52  /////////////Flame_Sensor_Settings///////////////////////
53      //flameSensor none;
54      flameSensor tanh;
55
56      tanhCoeffs
57      {
58      beta 100;
59      n_filters 10;
60      }
61  ////////////////////////////////////////////////////////
62  }
63
64
65
66  // ************************************************************************* //
```

Listing B.13: constant/thermophysicalProperties

```
1   /*--------------------------------*- C++ -*----------------------------------*\
2     =========                 |
3     \\      /  F ield         | OpenFOAM: The Open Source CFD Toolbox
4      \\    /   O peration     | Website:  https://openfoam.org
5       \\  /    A nd           | Version:  9
6        \\/     M anipulation  |
7   \*---------------------------------------------------------------------------*/
8   FoamFile
9   {
10      format      ascii;
11      class       dictionary;
12      location    "constant";
13      object      thermophysicalProperties;
14  }
15  // * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * //
16
17  thermoType
18  {
19
20      type            hePsiThermo;
21      mixture         multiComponentMixture; //coefficientWilkeMultiComponentMixture; //
        multiComponentMixture;
22      transport       sutherland;
23      thermo          janaf;
24      energy          sensibleEnthalpy;
25      equationOfState perfectGas;
26      specie          specie;
27  }
28
29  defaultSpecie N2;
30  //inertSpecie N2;
31  #include "thermo.compressibleGasCap"
32
```

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

Listing B.14: constant/thermophysicalTransport

```
/*--------------------------------*- C++ -*----------------------------------*\
  =========                 |
  \\      /  F ield         | OpenFOAM: The Open Source CFD Toolbox
   \\    /   O peration     | Website:  https://openfoam.org
    \\  /    A nd           | Version:  10
     \\/     M anipulation  |
\*---------------------------------------------------------------------------*/
FoamFile
{
    version         2;
    format          ascii;
    class           dictionary;
    location        "constant";
    object          thermophysicalTransport;
}
// * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * //

    LES
    {
        model           DTFunityLewisEddyDiffusivity;
        Prt             0.85;
    }


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

B.4 system Directory

Listing B.15: system/blockMeshDict

```
/*--------------------------------*- C++ -*----------------------------------*\
  =========                 |
  \\      /  F ield         | OpenFOAM: The Open Source CFD Toolbox
   \\    /   O peration     | Website:  https://openfoam.org
    \\  /    A nd           | Version:  10
     \\/     M anipulation  |
\*---------------------------------------------------------------------------*/
FoamFile
{
    format      ascii;
    class       dictionary;
    object      blockMeshDict;
}
// * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * //

convertToMeters 0.1;

vertices
(
    (0 0 0)
    (1 0 0)
    (1 1 0)
    (0 1 0)
    (0 0 1)
    (1 0 1)
    (1 1 1)
    (0 1 1)
);

blocks
(
```

```
    hex (0 1 2 3 4 5 6 7) (100 100 100) simpleGrading (1 1 1)
);

boundary
(
    symm1
    {
        type symmetryPlane;
        faces
        (
            (3 7 6 2)
        );
    }
    inlet
    {
        type patch;
        faces
        (
            (0 4 7 3)
        );
    }
    outlet
    {
        type patch;
        faces
        (
            (2 6 5 1)
        );
    }
    symm2
    {
        type symmetryPlane;
        faces
        (
            (1 5 4 0)
        );
    }
    frontAndBack
    {
        type wall;
        faces
        (
            (0 3 2 1)
            (4 5 6 7)
        );
    }
);


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

Listing B.16: system/controlDict

```
/*--------------------------------*- C++ -*----------------------------------*\
  =========                 |
  \\      /  F ield         | OpenFOAM: The Open Source CFD Toolbox
   \\    /   O peration     | Website:  https://openfoam.org
    \\  /    A nd           | Version:  9
     \\/     M anipulation  |
\*---------------------------------------------------------------------------*/
FoamFile
{
    version     2.0;
    format      ascii;
    class       dictionary;
    object      controlDict;
}
// * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * //
```

```

application     reactingFoam;

startFrom       startTime;

startTime       0;

stopAt          endTime;

endTime         0.01;

deltaT          1e-6;

writeControl    timeStep;

writeInterval   1000;

purgeWrite      3;

writeFormat     binary;

writePrecision  8;

writeCompression off;

timeFormat      general;

timePrecision   6;

runTimeModifiable true;

OptimisationSwitches
{
    fileHandler collated;
    maxThreadFileBufferSize 5e9;
}


libs
(
    "libDTFcombustion.so"
    "libDTFtransport.so"
);


functions
{
     #includeFunc cp
}
// ************************************************************************* //
```

Listing B.17: system/fvSchemes

```
/*--------------------------------*- C++ -*----------------------------------*\
  =========                 |
  \\      /  F ield         | OpenFOAM: The Open Source CFD Toolbox
   \\    /   O peration     | Website:  https://openfoam.org
    \\  /    A nd           | Version:  10
     \\/     M anipulation  |
\*---------------------------------------------------------------------------*/
FoamFile
{
    format      ascii;
    class       dictionary;
    object      fvSchemes;
}
// * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * //

```

```
ddtSchemes
{
    default         backward;
}

gradSchemes
{
    default         Gauss linear;

    limited         cellLimited Gauss linear 1;
    grad(U)         $limited;
    grad(k)         $limited;
    grad(omega)     $limited;
}

divSchemes
{

     default             none;

    div(phi,U)          Gauss limitedLinearV 1;
    div(phi,Yi)         Gauss limitedLinear01 1;
    div(phi,h)          Gauss limitedLinear 1;
    div(phi,K)          Gauss limitedLinear 1;
    div(phid,p)         Gauss limitedLinear 1;
    div(phi,omega)    Gauss limitedLinear 1;
    div(phi,Yi_h)       Gauss limitedLinear01 1;
    div(phi,k)          Gauss limitedLinear 1;
    div(((rho*nuEff)*dev2(T(grad(U)))))     Gauss linear;
    div(heatFluxCorr) Gauss linear;
}

laplacianSchemes
{
       default         Gauss linear corrected;
}

interpolationSchemes
{
    default         linear;
}

snGradSchemes
{
    default         corrected;
}

wallDist
{
    method meshWave;
}

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

Listing B.18: system/fvSolution

```
/*--------------------------------*- C++ -*----------------------------------*\
  =========                 |
  \\      /  F ield         | OpenFOAM: The Open Source CFD Toolbox
   \\    /   O peration     | Website:  https://openfoam.org
    \\  /    A nd           | Version:  10
     \\/     M anipulation  |
\*---------------------------------------------------------------------------*/
FoamFile
{
    format      ascii;
    class       dictionary;
    object      fvSolution;
```

```
}
// * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * //

solvers
{


    "rho.*"
    {
        solver          diagonal;
    }



    p
    {
        solver          GAMG;
        smoother        GaussSeidel;
        tolerance       1e-6;
        relTol          0.01;
    }

    pFinal
    {
        $p;
        relTol          0;
    }


    "(U|h|k|omega)"
    {
        solver          PBiCGStab;
        preconditioner  DILU;
        tolerance       1e-6;
        relTol          0.1;
    }

    "(U|h|k|omega)Final"
    {
        $U;
    }

    "Yi.*"
    {
        solver          PBiCG;
        preconditioner  DILU;
        tolerance       1e-8;
        relTol          0.1;
    }
}

PIMPLE
{
    momentumPredictor yes;
    nOuterCorrectors 2;
    nCorrectors     3;
    nNonOrthogonalCorrectors 3;
}


relaxationFactors
{

    fields
    {
    p  0.25;
        rho 0.25;
    }
```

```
81      equations
82      {
83          ".*"        0.5;
84      }
85  }
86  // ************************************************************************* //
```