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

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

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

1 Introduction 7
1.1 Motivation and Background . . . 7
1.2 Report Outline . . . 8

2 Theory 9
2.1 Governing Equations . . . 9
2.2 Diffusion . . . 9
2.3 Reaction Rate . . . 10
2.3.1 Laminar Model . . . 10
2.3.2 Partially Stirred Reactor (PaSR) . . . 10
2.3.3 Dynamic Thickened Flame (DTF) . . . 11
2.3.3.1 Thickening Factor . . . 11
2.3.3.2 Flame Sensor . . . 11
2.3.3.3 Efficiency Function . . . 11
2.3.3.4 Laminar Flame Speed . . . 12

3 PaSR: OpenFOAM Implementation 13
3.1 Registration . . . 14
3.2 Constructor . . . 14
3.3 Destructor . . . 16
3.4 Member Functions . . . 16
3.4.1 `correct()` . . . 16
3.4.2 `R(volScalarField& Y)` . . . 17
3.4.3 `Qdot()` . . . 17
3.4.4 `read()` . . . 17

4 DTF: Implementation 18
4.1 Introduction to `PaSR` & `TFM` . . . 18
4.2 Introduction to `TFM` & `DTF` . . . 19
4.3 Registration . . . 20
4.4 Constructor . . . 20
4.4.1 Update the `DTF.H` . . . 20
4.4.2 Update the `DTF.C` . . . 21
4.5 Member Functions . . . 22
4.5.1 Implementation of the `flameSpeed()` Function . . . 22
4.5.2 Update Thickening Factor . . . 26
4.5.3 Efficiency Function Implementation in `DTF` . . . 27
4.5.3.1 Adding Getter Functions to `DTF.H` . . . 27
4.5.3.2 Update the Efficiency Function Constructor `DTF.C` . . . 27
4.5.3.3 Update `efficiencyFunction.H` . . . 27
4.5.3.4 Update `efficiencyFunction.C` . . . 28
4.5.3.5 Update `efficiencyFunctionNew.C` . . . 28
4.5.3.6 Colin Efficiency Function Implementation . . . 29

4.5.4 `R(volScalarField& Y)` 30
4.5.5 `Qdot()` 30
4.6 `DTFThermophysicalTransportModels` 31

5 Test Cases and Results 32
5.1 Case 01: 3D Freely Propagating Hydrogen Flame 32
5.1.1 Results 34
5.2 Case 02: Turbulent Hydrogen Flame 34
5.2.1 Validation 35

A Developed Codes 43
A.1 Dynamic Thickened Flame Header, `DTF.H` 43
A.2 Dynamic Thickened Flame Constructor, `DTF.C` 45
A.3 Efficiency Function: Colin Header, `colin.H` 51
A.4 Efficiency Function Colin Constructor, `colin.C` 52

B Case 01 Setup 55
B.1 Allclean and Allrun Scripts 55
B.2 `0` Directory 55
B.3 `constant` Directory 61
B.4 `system` Directory 65

Nomenclature

Acronyms

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

English symbols

$\vec{J}_i$	Di usive ux of species i	kg=m^2=s
$\vec{v}$	Velocity vector	m=s
c_p	Speci c heat capacity at constant pressure	J=(kg K)
c_{ms}	Wrinkling factor model constant	
D	Thermal di usivity	m^2=s
D_i	Di usion coe cient for species i	m^2=s
D_{iF}	Modi ed di usion coe cient (DTF model)	m^2=s
E	E ciency function	
F	Flame thickening factor	
F_s	Dynamically calculated thickening factor	
F_{max}	Maximum thickening factor	
h	Heat release rate	J=s
h_{max}	Maximum heat release rate	J=s
Le_i	Lewis number for species i	
R_{iF}	Modi ed reaction rate (DTF model)	kg =m^3=s
Re_t	Turbulent Reynolds number	
S_l^0	Premixed laminar ame speed	m=s
u^0	Subgrid-scale velocity uctuations	m=s
Y_i	Mass fraction of species i	

Greek symbols

	Adjustment factor for e ciency function	
	Scaling parameter for ame sensor	
	Filter width	m
$_e$	Test lter width	m
$_g$	Geometric cell size	m
$_l$	Laminar ame thickness	m
$_l^0$	Unthickened laminar ame thickness	m
$_l^1$	Thickened ame thickness	m
	Turbulent dissipation rate	m^2=s^3
	Stretch function	
	Reacting volume fraction	
	Thermal conductivity	W =(m K)
$_e$	E ective viscosity	kg =(m s)

	Density	$kg=m^3$
$_c$	Chemical time scale	s
$_k$	Mixing time scale	s
	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 lter width can be reduced, approaching the Kolmogorov scales [4], which decreases the in uence of unresolved scales and reduces reliance on sub-grid models. However, as long as turbulence remains unresolved, sub-grid turbulence models are required. LES o ers 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 di erent sizes break down into smaller ones, strain and shear at their interfaces enhance mixing and molecular di usion [7]. In the presence of a ame, heat and radicals di use 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 di erent 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 e ect of turbulent uctuations on kinetic rates is neglected, and reaction rates are determined by general nite-rate chemistry directly.
- Eddy-Dissipation Concept (EDC) : It assumes that reactions occur in regions of the ow 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 ames in OpenFOAM. The PaSR is a model used in LES to approximate the combustion process. It helps simulate how the turbulent uid 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 ltered 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 ame 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 ame 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 modi cations to the species transport equation, Adjustment of the reaction source term by multiplying it with e ciency function (E) and divided by thickening factor (F). Modi cation of the di usion coe cient by multiplying it with an e ciency 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 di erence is represented in Table 1.1.

Table 1.1: Comparison of DTF with other TCI Models

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

From Table 1.1, the Laminar model is primarily used for non-premixed combustion in OpenFOAM, though it can also be applied to premixed ames in Direct Numerical Simulation (DNS). Similarly, the functionality of the Eddy-Dissipation Concept (EDC) and Partially Stirred Reactor (PaSR) models for di erent ow 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 ows.

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 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 di usion coe cient.

1.2 Report Outline

In Chapter 2, the required theory related to the species transport equation is given, and further it explains the di usion coe cient and reaction rate source term calculation using di erent 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

$$\frac{\partial}{\partial t}(\rho Y_i) + \nabla \cdot (\rho \vec{v} Y_i) = -\nabla \cdot \vec{j}_i + R_i; \tag{2.1}$$

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

2.2 Diffusion

The diffusive flux $\vec{j}_i$ 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

$$\vec{j}_i = -\rho D_i \nabla Y_i; \tag{2.2}$$

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 usingunityLewisEddyDiffusivity 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 di erent 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 un ltered reaction rate $\overline{R_i} = R_i$. This assumption is valid for laminar ows or turbulent ows where the computational grid is ne 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 modi es the reaction source term by kappa ( ), a reacting volume fraction based on the ratio of chemical and turbulent mixing timescales. The reacting source term is de ned as

$$R_i = \quad R_i(Y_i); \tag{2.5}$$

where is

$$= \frac{c}{c + k}; \tag{2.6}$$

where R_i is the reaction rate of speciei, Y_i is the specie that passes to reaction rate, $_c$ is the chemical time scale, and $_k$ represents the turbulent mixing time scale. The $_k$ is calculated as the Kolmogorov time scale, it is de ned in PaSR [16] at line 93

$$_k = C_{mix} \sqrt{\frac{e}{}}; \tag{2.7}$$

where C_{mix} is the constant value provided by the user, default is 1, $_e$ is the e ective viscosity accounting for turbulence, and is the turbulent dissipation rate. The chemical time scale $_c$ which is implemented in OpenFOAM de ned in chemistryModel [17, 18] at line 446 is derived based on reaction kinetics

$$_c = \frac{c_{tot}}{\sum_{j=1}^{n_R} \sum_{i=1}^{N_{s;RHS}} {}_{i;j} k_{f;j}}; \tag{2.8}$$

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

$$= \begin{cases} 1.0 & \text{if } _k < \text{small}; \\ \frac{c}{c + k} & \text{otherwise}. \end{cases} \tag{2.9}$$

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

2.3.3 Dynamic Thickened Flame (DTF)

The DTF model is based on the thickened ame (TF) approach as described by Legier et al. [13], which is traditionally used to make ames resolvable on coarse grids in large eddy simulations (LES). In the TF approach, the ame thickness is arti cially increased by a factor F while maintaining the correct ame propagation speed. This is achieved by multiplying the species and thermal di usion coe cients 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

$$\frac{@(Y_i)}{@t} + r \quad (\vec{v} Y_i) = r \quad (D_{iF} r Y_i) \quad R_{iF} ; \tag{2.10}$$

where, is density, $\vec{v}$ is velcoity, D_{iF} is modi ed di usion coe cient scaled by the thickening factor (F), and R_{iF} is reaction rate source term. D_{iF} and R_{iF} are represented as

$$D_{iF} \; ! \; DF; \quad R_{iF} \; ! \; \frac{1}{F} R_i : \tag{2.11}$$

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 ame front and activates the thickening only within reaction zones. The thickening factor (F) is de ned as

$$F = 1 + (\min(F_{max} ; F_s) \quad 1) \; ; \tag{2.12}$$

where 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 is based on the heat release rate and is de ned as

$$= \tanh(\frac{h}{h_{max}}); \tag{2.13}$$

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

2.3.3.3 E ciency Function

This dynamic adjustment ensures that the ame remains resolvable in the grid while preserving accurate mixing and di usion characteristics in non-reactive regions. To account for the e ects of unresolved turbulence, the DTF model uses an e ciency function (E), which modi es the di usion coe cients and reaction rate in Eq. 2.14 as

$$D_{iEF} \; ! \; EFD; \quad R_{iEF} \; ! \; \frac{E}{F} R_i ; \tag{2.14}$$

where D_{iEF} is the modi ed di usion coe cient, R_{iEF} is the modi ed reaction rate source term, F is the thickening factor, and E is the e ciency function. The e ciency function E is expressed as the ratio between the wrinkling factor of the real ame $_l = {}^0_l$ and that of the thickened ame with thickness, ${}^1_l = F \, {}^0_l$ as

$$E = \frac{j_{l = {}^0_l}}{j_{l = {}^1_l}} \quad 1; \tag{2.15}$$

where $_l$ is ame thickness, ${}^0_l$ is the laminar ame speed, and ${}^1_l$ is the turbulent ame thickening. To estimate a dimensionless wrinkling factor , which is proposed by Colin et. al. [19, 20] de ned as

the ame surface to its projection in the propagating direction. It depends on velocity uctuations ($u^0{}_e$) and ame parameters, which include laminar ame speed (^0_l), laminar ame thickness ($_l$) turbulent ame thickening (^1_l) and thickening Factor (F) as

$$= 1 + \quad \frac{u^0{}_e}{S^0_l} \quad \frac{{}_e}{{}_l}; \frac{u^0{}_e}{S^0_l} \quad ; \tag{2.16}$$

where

$$= \frac{2 \ln 2}{3c_{ms} \quad \sqrt{Re_t} \quad 1}; \quad c_{ms} = 0:28; \tag{2.17}$$

and is a dimensionless stretch function

$$\frac{{}_e}{{}_l}; \frac{u^0{}_e}{S^0_l} \quad 0:75 \exp \quad 1:2 \quad \frac{u^0{}_e}{S^0_l} \quad {}^{0:3} \# \quad \frac{{}_e}{{}_l} \quad {}^{2=3} : \tag{2.18}$$

The test lter scale $_e$ is chosen such that $_e = \ ^1_l = F \ ^0_l > \ x$, which ensures the proper ame resolution. The as desired in Eq. 2.17 is not implemented in the present implementation. 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 ame front propagates normal to itself and relative to the ow into the unburnt mixture. This velocity is called the laminar ame speed [21]. It is proportional to the thermal di usivity (D) and the reaction rate (R). Mathematically, the laminar ame speed is expressed as

$$S^0_l = \sqrt{DR}; \tag{2.19}$$

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

$$D = \frac{}{c_p}; \tag{2.20}$$

where is the thermal conductivity, is the density, and c_p is the speci c heat capacity. The laminar ame thickness ($_L$) is proportional to $\frac{D}{S^0_l}$ and is expressed as

$$_l = \frac{D}{S^0_l}: \tag{2.21}$$

To simulate the laminar ame on a coarse mesh without altering its speed, the ame can be arti cially thickened by proportionally increasing the di usivity (D) and decreasing the reaction rate ($1=R$). The dynamic thickening factor (F_s) is de ned as

$$F_s = \frac{N}{{}_l}; \tag{2.22}$$

where is the grid size, $_L$ is the laminar ame thickness, and N is the user-speci ed number of points in the ame. The grid size ( ) is calculated as

$$= \quad V^{\frac{1}{N_d}}; \tag{2.23}$$

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 ( ) 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 coefficient, 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 con gurable and e cient 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 eld 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) de nition 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 modi cations to the di usion coe cient, 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 speci cally 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 eld. This kappa eld is then used as a parameter to modify the reaction rate source term function R (foam::combustionModels::PaSR::R) as return kappa_*laminar::R(Y); , which modi es 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 (Ξ)	Power-law	Colin
Flame Sensor (S)	Based on heat release	Optimized TFM sensor
Thermophysical Transport Models	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 modi cations to the TFMFoamdirectories and les to set up the DTFframework. Update the le 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 forDTFis set up, the next step is to modify theDTFcode. In the implementation process ofDTF, the report will try to connect with PaSRimplementation in Chapter 3. Follow the steps below to update the code accordingly.

4.3 Registration

During the process of renaming the les and updating le content, the TFMregistration 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_), di usivity (D_) and thickening factor (Fs_) after the rst 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 rst 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, de ne the member function for ame speed calculations in the member function section.

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

```
void flameSpeed();
```

4.4.2 Update the DTF.C

Initialize the newly added members in the header le. 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", E ciency Function Initialization

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

Here is the initialization of new variables, Listing 4.9 is the representation ofvolScalarField and Listing 4.10 representsscalar .

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 e ciency function (E). Speci cally, it computes the laminar ame speed (Eq. 2.19) and the laminar ame 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 de ned 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, ameSpeed(); Implementation

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

The laminar ame speed (S_l^0) is proportional to the reaction rate and thermal di usivity. 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, ameSpeed(); 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, ameSpeed(); 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 lter. To use the smoothing lter, include the header le "fvcSmooth.H" along with the other header les at the top of the DTF.C le. The implementation of the smoothing lter is shown in Listing 4.15.

Listing 4.15: src/combustionModels/DTF/DTF.C, ameSpeed(); 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 ame front. Listing 4.16 demonstrates how the reaction rate is extracted and calculated across all processors.

Listing 4.16: src/combustionModels/DTF/DTF.C, ameSpeed(); 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 di usivity (D) is calculated, as described in Eq.2.3. While the reaction rate calculation uses thecalculateRR function implemented in the basicChemistryModel class, the thermal di usivity calculation accesses the values of thermal conductivity ( ), density ( ), and heat capacity (C_p) through the thermo_ pointer from the fluidReactionThermo class. Afterward, the smoothing lter and extraction of di usion values from the reaction front follow the same procedure as the reaction rate tRR_ calculation. The thermal di usivity calculations are shown in Listing 4.17.

Listing 4.17: src/combustionModels/DTF/DTF.C, ameSpeed(); Thermal Di usivity 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 di usivity, the laminar ame speed (S_l^0) and laminar ame thickness ($_l$) are computed inside the flameSpeed() function, as shown in Listing 4.18.

Listing 4.18: src/combustionModels/DTF/DTF.C, ameSpeed(); 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 e ciency (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 e ciency-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 e ciency function (E). The e ciency function (E) requires the laminar ame speed (S_l^0), the laminar ame thickness ($_l$), and the thickening factor (F).

4.5.3 E ciency Function Implementation in DTF

Previously, the e ciency function (E) was implemented outside the TFM class. To make the e ciency function dynamic, there are several ways to introduce the variables calculated inside the DTF class to the e ciency function class. One approach is through getter functions. In this implementation, the required parameters are introduced to the e ciency function (E) via getter functions. This requires modi cations to the header and constructor les 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 le 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 E ciency Function Constructor DTF.C

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

Listing 4.22: "src/combustionModels/DTF/DTF.C", E ciency 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/E ciencyFunctions/e ciencyFunction/e ciencyFunction.H", DTF Member

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

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

Listing 4.24: "src/combustionModels/E ciencyFunctions/e ciencyFunction/e ciencyFunction.H", E ciency 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", E ciency 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/E ciencyFunctions/e ciencyFunction/e ciencyFunction.H", E ciency 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 E ciency Function Implementation

Now, implement the Colin e ciency function inside the EfficiencyFunction directory. Create the folder ColinEfficiencyFunction , and then create two les, 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 theColin e ciency is implemented in the colin.C le. The complete content of colin.H and colin.C can be found in Appendix A. The Colin e ciency 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/E ciencyFunctions/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 DTFcombustion 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
    (
```

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

4.6 DTFThermophysicalTransportModels

The modi cation of the di usion coe cient requires scaling the mass di usion coe cient and thermal di usion coe cient. These modi cations of the di usion coe cient are implemented in TFMand DTF will use the same implementation to modify the di usion coe cient. The ThermophysicalTransportModels code is available insrc directory of the provided code. The modi cation done to the di usivity term is represented in Listing 4.30.

Listing 4.30: "src/ThermophysicalTransportModels/turbulence/DTFunityLewisEddyDi usivity/ DTFunityLewisEddyDi usivity.H",Modi cation to Di usivity

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

Now, change the folder toDTFmain folder, where it contains the Allwmake and Allwclean les 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 di erent levels of complexity and computational resource requirements. The details of the test cases are as follows:

- Case 01: This is a simpli ed test case to run simulation on a personal computer. It involves the simulation of a 3D freely propagating hydrogen ame using DTF.
- Case 02: This case focuses on a turbulent hydrogen ame 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 les to run the cases. The step-by-step process is to rst 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 ow moves along the x-direction. The boundary conditions include inlet and outlet along the ow direction. The top and bottom of the domain are de ned as symmetryPlane and are labeled symm1 and symm2 respectively. The left and right sides, named frontAndBack , are modeled as zeroGradient walls. The ow is initialized with a xed velocity of 0 :026 m/s at a temperature of 300 K. The composition of the ow 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 le, where the combustion model name and other required coe cients are speci ed, 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 coe cients 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 di usivity values.
- upperLimit : The upper threshold for the flameSensor .
- filters : Speci es the number of lters used to smooth uctuations in reaction rate, di usivity, and velocity.

The flameSensor o ers two options: none and tanh . If tanh is selected, additional parameters beta and n_filters must be provided. Similarly, the e ciency function can be set to either none or colin . If colin is chosen, the alpha parameter must be speci ed. The con guration 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 speci ed at the end of the controlDict le. 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 les, including chemistryProperties and thermoPhyscialProperties , along with other case setup les, are listed in Appendix B.

5.1.1 Results

Fig. 5.2 represents the ame 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 di erent instants, showing the temperature eld 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