Cite as: Chen, W.: Radiative heat transfer in OpenFOAM and its non-grey implementation. In Proceedings of CFD with OpenSource Software, 2023, Edited by Nilsson. H., http://dx.doi.org/10.17196/OS_CFD#YEAR_2023

CFD with OpenSource software

A course at Chalmers University of Technology
Taught by Håkan Nilsson

Radiative heat transfer in OpenFOAM and its non-grey implementation

Developed for OpenFOAM-v2112

Author:
Wei Chen
Shanghai Jiao Tong University
duanwu.chen@sjtu.edu.cn

Peer reviewed by:
Tao Ren
Saeed Salehi
Chit Yan Toe

January 14, 2024

Learning outcomes

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

The reader will learn:

How to use it:

- How to use the radiation model in OpenFOAM with a focus on the combustion application.
- How to choose the radiation model in OpenFOAM.

The theory of it:

- The theory of radiative heat transfer.
- The theory of the radiative transfer equation (RTE) solution methods.
- The theory of the spectral models.

How it is implemented:

- How the radiation model is implemented in OpenFOAM.
- How the `greyMeanAbsorptionEmission` model cooperates with the RTE solver.

How to modify it:

- How to model the non-grey radiative heat transfer in OpenFOAM.

Prerequisites

The reader is expected to know the following in order to get maximum benefit out of this report:

- How to run standard document tutorials like `SandiaD_LTS` tutorial.
- Radiative Heat Transfer, Book by M. F. Modest and S. Mazumder
- How to customize a solver and do top-level application programming.

Contents

Nomenclature

Acronyms

CFD	Computational Fluid Dynamics
DOM	Discrete Ordinates Method
FSCK	Full Spectral Correlated-k Distribution
LBL	Line-By-Line
P1	Spherical Harmonics Method of Order 1
PDE	Partial Differential Equation
PN	Spherical Harmonics Method of Order n
RTE	Radiative Transfer Equation
SLW	Spectral-line-based Weighted Sum of Grey Gases
UDF	User-Defined Function

English symbols

$\nabla \cdot q$	Radiative heat source	W/m^3
a	Non-grey stretching factor in the FSCK model	m^{-1}
G	Incident radiation	W/m^2
g	Quadrature points in the FSCK model	
I	Radiative intensity	W/sr/m^2
k	k value in the FSCK model	m^{-1}
P	Associated Legendre polynomial	
p	Pressure	Pa
T	Temperature	K
w	Weighting factor in the FSCK model	
x_i	Species' volume fraction	mol/mol

Greek symbols

β_η	Extinction coefficient	
$\delta_{i,j}$	Kronecker Tensor	
ϵ	Emissivity	
Γ	Diffusivity of P1 Equation	m
κ_η	Absorption coefficient	m^{-1}
Φ_η	Phase function	
σ	Stefan-Boltzmann constant	5.67×10^{-8}W/(m^{2}K^4)
$\sigma_{s\eta}$	Scattering coefficient	

Subscripts

ref	reference

Chapter 1

Introduction

Radiative heat transfer plays a crucial role in a multitude of engineering and scientific applications, ranging from combustion processes [1, 2, 3] and heating, ventilation, and air conditioning (HVAC) systems [4, 5] to atmospheric studies [6, 7, 8] and astrophysics [9]. In these scenarios, understanding and accurately predicting the behavior of radiative heat transfer is vital for effective design and analysis of the reseraching objects. This report presents a brief exploration of radiative heat transfer within the realm of computational fluid dynamics (CFD), focusing on the integration and validation of advanced radiative transfer models in OpenFOAM.

1.1 Background

Radiative heat transfer, a mechanism of energy transfer through electromagnetic radiation, is distinct from conduction and convection as it does not require a medium and can occur in a vacuum. Its significance is especially pronounced at high temperatures, where radiation becomes the dominant mode of heat transfer. In participating media, defined as media that can emit, absorb, and scatter radiation, the complexity of modeling radiative heat transfer increases substantially.

The challenge in simulating radiative heat transfer in participating media lies in the inherent high dimensionality of the Radiative Transfer Equation (RTE). Traditional CFD solvers, often optimized for three-dimensional problems, may struggle with the intricacies of the RTE. Moreover, the variability of the absorption coefficient across different wavelengths adds another layer of complexity, necessitating advanced modeling techniques.

Modeling radiative heat transfer within the CFD context can be divided into two main parts: solving the RTE and decreasing the RTE solution times in the wavenumber dimension. The models for the former are referred to as RTE solvers, and for the latter as spectral models. This study focuses on spectral models, which could enable us to address the problem with a limited number of RTE evaluations. They are discussed in detail in Chapter 2.

Modest and his research group [10, 11, 12, 13, 14] have developed a compact radiative heat transfer library in OpenFOAM 2.2x. This library has been extensively tested, validated, and utilized in various publications. However, their code, developed in an older version of OpenFOAM and not open-source, requires modifications on the solvers' source code if radiative heat transfer is to be included, which largely limits its applicability. Recently, Sun et al. [15] have implemented the weighted-sum-of-grey-gases (WSGG) model within the OpenFOAM framework. However, this model is designed for the homogeneous mixture and may be inaccurate in non-homogeneous applications.

Guo et al. [16] implemented one of the state-of-the-art spectral models, the full-spectrum correlated k distribution (FSCK), in ANSYS Fluent 16.0 using the user-defined function. However, limitations in ANSYS Fluent user-defined functions and the corresponding macro posed challenges in accurately implementing the model and in handling boundary conditions. Recently, Tao Ren's research group [17] implemented this model in ANSYS Fluent 2022, encountering similar limitations. However, as a commercial software, ANSYS Fluent is not open-source and requires a license to use. The close-source nature makes model development difficult or even limited in this software.

1.2 Objectives

This report is structured to guide the reader through several aspects of radiative heat transfer within the context of CFD and implement an advanced spectral model, namely FSCK, into OpenFOAM.

Initially, the focus is on explaining the fundamentals of radiative heat transfer. This includes an exploration of the underlying principles and mechanisms that govern this mode of heat transfer, with particular emphasis on its relevance and application in CFD scenarios. Following this foundational understanding, the report delves into the specifics of OpenFOAM's built-in radiation models. This segment aims to provide an in-depth look at the existing models within this open-source CFD toolbox, detailing their features, capabilities, and typical use cases. The next objective is to show the process of utilizing the radiation models in OpenFOAM. This includes a step-by-step guide on how to implement these models in various CFD simulations, highlighting the procedural and technical aspects of their application.

Building upon the existing framework, the report introduces and detail the integration of the spectral model FSCK into the OpenFOAM library. This section offers a comprehensive explanation of the model, its theoretical basis, implementation strategies, and expected enhancements in simulation accuracy and efficiency. Lastly, the report aims to validate the newly implemented model through multiple cases. This validation process involves comparing the performance and results of the newly implemented FSCK model with those of the pre-existing in-house codes in OpenFOAM and ANSYS Fluent. The comparison is based on various metrics and simulation scenarios to ensure a robust evaluation of the new model's accuracy and validity.

In essence, this report endeavors to provide an overview of radiative heat transfer modeling in OpenFOAM, from basic principles to advanced model implementation and validation.

1.3 Report Structure

This report is organized into six chapters, each focusing on a distinct aspect of radiative heat transfer and its modeling in OpenFOAM.

Chapter 1: Introduction. This chapter sets the foundation for the report by outlining its purpose and scope. It introduces the main themes and objectives, providing the reader with a clear understanding of what the report aims to achieve and the significance of its content in the field of CFD and radiative heat transfer.

Chapter 2: Fundamentals of radiative heat transfer. Here, the fundamental concepts and principles governing radiative heat transfer are discussed. This chapter is essential for establishing a theoretical basis that supports the understanding of the models and simulations discussed in later chapters.

Chapter 3: Existing radiation models in OpenFOAM. In this chapter, the focus shifts to a detailed examination of the radiation models currently available in OpenFOAM. It includes a walk through of the source code and a practical tutorial section that guides the reader through the process of utilizing these existing models in CFD simulations.

Chapter 4: Developing a non-grey radiation model in OpenFOAM. This chapter delves into the integration of the FSCK model into OpenFOAM. It provides the implementation of the FSCK model and the nongrey P1 model, enhancing the capabilities of OpenFOAM in handling complex radiative heat transfer scenarios.

Chapter 5: Model validation. The fifth chapter presents a comprehensive validation of the newly implemented model. It compares the results of the newly developed code for the FSCK model with two existing codes, offering a critical assessment of its accuracy and reliability.

Chapter 6: Conclusion and future work. The final chapter concludes the report by summarizing the key findings and contributions. It also discusses potential avenues for future research, suggesting ways to build upon the work presented in this report and exploring emerging trends and challenges in the field.

Through these chapters, the report aims to provide a thorough and insightful exploration of radiative heat transfer modeling in OpenFOAM, from basic principles to advanced implementations and validations.

Chapter 2

Fundamentals of radiative heat transfer

This chapter introduces key concepts and methodologies in radiative heat transfer relevant to the subsequent analysis and discussions. It aims to provide readers with essential terminologies and principles that form the foundation of the computational approaches used in this study.

2.1 Radiative heat transfer in participating media

The maximum emissive power of an object is a function of its surface temperature, raised to the fourth power. This relationship involves the Stefan-Boltzmann constant, σ, which is $5.67 \times 10^{-8}\mathrm{W/(m^2K^4)}$ [18]. Typically, the radiative heat transfer between surfaces is modeled using the concept of a view factor, as implemented in the `viewFactor` function in OpenFOAM. However, in participating media, where the medium itself can emit and absorb radiation, this approach is insufficient. Thus, we introduce the Radiative Transfer Equation (RTE) to model these interactions.

2.1.1 Radiative transfer equation

The RTE, essential in modeling radiative heat transfer in participating media, is expressed as follows [19]:

$$\frac{\mathrm{d}I_\eta}{\mathrm{d}s} = \hat{\mathbf{s}} \cdot \nabla I_\eta = \kappa_\eta I_{b\eta} - \beta_\eta I_\eta + \frac{\sigma_{s\eta}}{4\pi} \int_{4\pi} I_\eta(\hat{\mathbf{s}}_i)\Phi_\eta(\hat{\mathbf{s}}_i, \hat{\mathbf{s}})\mathrm{d}\Omega_i, \tag{2.1}$$

where I_η denotes the spectral intensity of radiation in direction $\hat{\mathbf{s}}$ at wavenumber η. The term κ_η, representing the absorption coefficient, quantifies the medium's propensity to absorb radiation at wavenumber η. The extinction coefficient, $\beta_\eta = \kappa_\eta + \sigma_{s\eta}$, accounts for the total diminishment of radiation due to both absorption and scattering. Here, $\sigma_{s\eta}$ is the scattering coefficient, indicating the proportion of radiation being scattered at wavenumber η. The phase function Φ_η describes the angular distribution of scattering, effectively representing the probability of radiation changing direction from $\hat{\mathbf{s}}_i$ to $\hat{\mathbf{s}}$ upon scattering.

The first term on the right-hand side of the equation corresponds to the emission of radiation, which is the energy radiated by the medium. The second term represents the absorption of radiation, reflecting the medium's attenuation of radiative intensity. The third term is the scattering of radiation, a mechanism where radiation deviates from its path due to interactions within the medium. This scattering term is integrated over all possible directions $\hat{\mathbf{s}}_i$, with each contribution weighted by the phase function Φ_η to account for the directional dependence of scattering. Additionally, the term is modulated by the scattering coefficient $\sigma_{s\eta}$, which quantifies the rate of scattering from $\hat{\mathbf{s}}_i$ into any other direction.

2.1.2 Challenges in solving the RTE

Solving the Radiative Transfer Equation (RTE) presents three primary challenges: high dimensionality, and the significant variability of the absorption coefficient across different wavenumbers.

The high dimensionality of the RTE arises from the directional dependence of radiation intensity. Since the intensity varies with both spatial position and radiation direction, the RTE essentially becomes a Partial Differential Equation (PDE) in both spatial and directional domains. This multidimensional nature significantly complicates the analytical and numerical solutions.

These aspects—non-linearity and high dimensionality—present significant challenges to conventional CFD solvers, which are typically optimized for linear or less complex non-linear systems. Consequently, direct application of standard CFD techniques to solve the RTE may not be feasible.

Additionally, the variability in the absorption coefficient, which depends on the radiation wavenumber, adds another layer of complexity. As illustrated in Fig. 2.1, the absorption coefficient varies significantly across different wavenumbers. This variation necessitates solving the RTE as a PDE in the wavenumber domain as well. Without simplifying assumptions or approximations, it would require solving the RTE independently for each wavenumber, a task that is computationally intensive and often impractical in CFD computations.

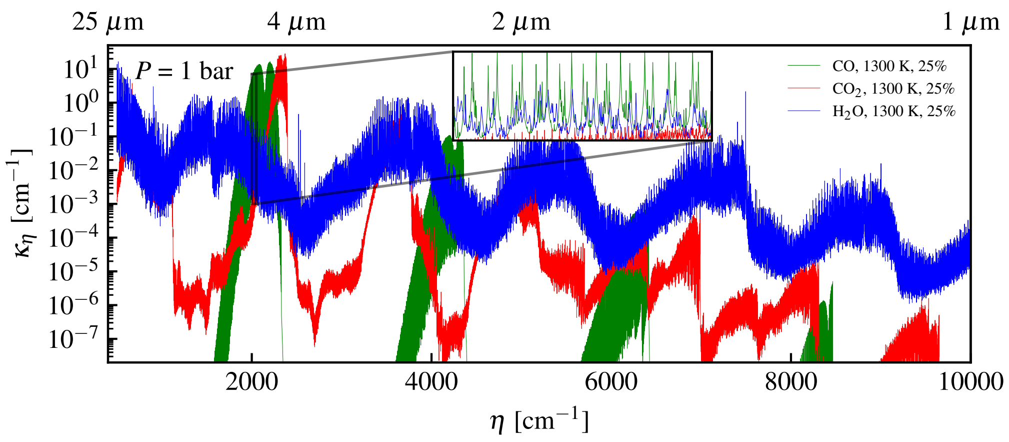

Figure 2.1: Absorption coefficient of H_2O, CO_2, and CO at 1 bar, 2400 K, and 25% volume fraction.

2.2 Solution Methods for the radiative heat transfer

Modeling radiative heat transfer mainly consists of two parts, solving the RTE and ensuring minimal times of RTE solving. The former one represents the RTE solver and the latter one represents the spectral model. Solving the Radiative Transfer Equation (RTE) is typically approached through two principal methodologies: statistical or deterministic models. The statistical method, particularly the Photon Monte Carlo Method, is highly regarded for its precision but is also known for being computationally intensive, often serving as a benchmark for other methods [20, 21]. This study, however, primarily focuses on deterministic models, which comprise two key components: the RTE solver and the spectral model. The RTE solver tackles the equation directly, utilizing specific algorithms and techniques for efficient computation. On the other hand, the spectral model plays an indispensable role in reformulating the RTE solving process within the wavenumber domain. This approach significantly reduces the computational burden, transforming what would typically require millions of RTE evaluations at each wavenumber into a process that necessitates only a few such calls. These two models will be discussed in detail in the following parts.

2.2.1 RTE Solver

The objective of numerical RTE solvers is to mitigate the strong directional dependency characteristic of the RTE, transforming it into a single or a set of Partial Differential Equations (PDEs) with only spatial dependence. This goal is achievable through two primary methods: the discrete ordinates method (DOM) [22, 23] and the spherical harmonics method (PN) [14, 24]. The DOM, implemented as `fvDOM` in OpenFOAM, discretizes the radiation intensity into a finite number of discrete directions. This discretization allows the integral term in the RTE to be approximated by summation, effectively converting the RTE into a set of PDEs solvable by the Finite Volume Method (FVM).

Conversely, the PN method, a type of spectral method, expands the radiation intensity into a series of spherical harmonics. The orthogonality of these functions simplifies the RTE, leading to its transformation into a single PDE or a set of PDEs, contingent on the expansion order. Due to the mathematical intricacies of PN, most CFD software, including OpenFOAM's `P1`, typically implement only the first-order PN method.

For optically thick scenarios, where the absorption coefficient is high and the spatial scale is large, resulting in gradual spatial variation in intensity, low-order methods like P1 and DOM can accurately predict radiative heat transfer at a limited computational cost. The term "optically thick" refers to media in which radiation is significantly absorbed or scattered, reducing its transmission distance. In such cases, the P1 method, involving only one PDE solution, is particularly advantageous, as seen in applications like combustion simulation. However, in optically thin situations—where radiation travels further due to lower absorption—higher-order RTE solvers are required. Yet, these higher-order PN methods often encounter numerical oscillation issues, presenting a challenge in accurately modeling radiative transfer in such conditions [25].

2.2.2 Spectral Model

Spectral models in radiative heat transfer can be broadly categorized into four types: Line-by-Line model (LBL), narrow-band model (NB), wide-band model (WB), and global model (GM) [26]. The LBL model, true to its name, solves the RTE for each spectral line, yielding highly accurate results at the cost of significant computational resources, often serving as a benchmark. The NB and WB models, however, are becoming less favored in CFD applications for heat transfer due to their respective drawbacks in either efficiency or accuracy.

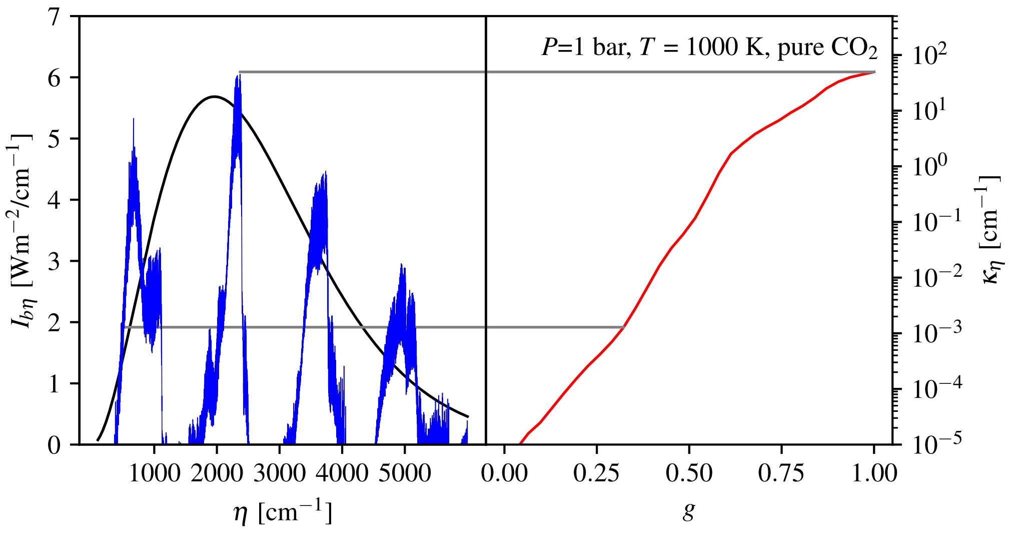

Figure 2.2: Illustration of the FSCK model

In CFD contexts, the global model is typically preferred, particularly for applications focused on spectrally integrated total radiative heat flux. One of the most widely used global models is the weighted-sum-of-grey gases model [27]. This model simplifies the problem by approximating non-grey gases with a set of grey gases, each having a constant absorption coefficient. Independent calculations for each grey gas are summed to estimate the total radiative heat flux. Despite its simplicity and reasonable accuracy, making it a choice in commercial CFD software like ANSYS Fluent and ANSYS CFX, the model's foundational principle lacks rigor, which can lead to inaccuracies, especially in non-homogeneous applications.

Enhancing the global model approach, Denison et al. [28, 29] introduced the spectral-line-based weighted sum of grey gases (SLW) model, which operates similarly to the WSGG model but utilizes a high-resolution spectral database to calculate the weights of the grey gases. Ongoing refinements by researchers [30, 31] have made the SLW model one of the state-of-the-art spectral models for CFD applications.

Another advanced model is the full-spectrum correlated k distribution model (FSCK), developed by Modest and his team [13, 32, 33, 34]. As illustrated in Fig. 2.2, this model reorders the spectrum based on the absorption coefficient and Planck function. Mathematical induction allows the RTE to be solved just a few times, with the number of solutions depending on desired accuracy levels.

After discussing the FSCK and SLW models, it is noteworthy to mention that these two state-of-the-art spectral models are mathematically equivalent. This equivalence, as detailed in [13], suggests that despite their different methodological approaches, both models converge to similar theoretical foundations when applied to solving the RTE

2.3 Radiative heat transfer models in this study

In this project, we have implemented the non-grey P1 model and the Full-Spectrum Correlated k (FSCK) spectral model into OpenFOAM. This section is dedicated to discussing these models, with an emphasis on their practical implementation aspects within the OpenFOAM framework. While the focus here is on implementation strategies and computational considerations, a comprehensive exploration of the theoretical foundations of these models can be found in Chapters 15 and 19 of "Radiative Heat Transfer" by Michael Modest [14, 13]. The decision to implement these particular models was driven by their relevance and applicability to the specific challenges and objectives of our study, which will be detailed in the following sections.

2.3.1 The P1 model

The P1 model is a subset of the PN method, which expands the radiation intensity into spherical harmonics in the first order. The spherical harmonics can be written as,

$$Y_l^m(\theta,\psi) = \frac{(-1)^l}{2^l l!}\sqrt{\frac{(2l+1)(l+m)!}{4\pi(l-m)!}}\mathrm{e}^{im\psi}P_l^m(\cos\theta), \tag{2.2}$$

where θ is the polar angle, ψ is the azimuthal angle, l is the order of the spherical harmonics, m is the degree of the spherical harmonics, and P_l^m is the associated Legendre polynomial. The polynomial is given by,

$$P_n^m(\mu) = (-1)^m \frac{(1-\mu^2)^{|m|/2}}{2^n n!}\frac{\mathrm{d}^{\mathrm{n+|m}}}{\mathrm{d}\mu^{n+|m|}}(\mu^2-1)^n. \tag{2.3}$$

The Legendre polynomial has two important properties: orthogonality and recursion relation. For the orthogonality, we have,

$$\int_{-1}^{1} P_l(\mu)P_m(\mu)\mathrm{d}\mu = \frac{2\delta_{lm}}{2m+1} = \begin{cases} 0 & \text{for } m \neq l, \\ \dfrac{2}{2m+1} & \text{for } m = l, \end{cases} \tag{2.4}$$

where δ_{ij} is the Kronecker Tensor. For the recursion relation, we have,

$$(2l+1)\mu P_l(\mu) = lP_{l-1}(\mu) + (l+1)P_{l+1}(\mu), \tag{2.5}$$

where $P_0(x) = 1$ and $P_1(x) = x$ are the first two Legendre polynomials. These two properties are very important in the equation simplification but are not discussed in this report. By expressing the radiative intensity in terms of spherical harmonics,

$$I(\mathbf{r}, \hat{\mathbf{s}}) = \sum_{l=0}^{\infty} \sum_{m=-l}^{l} I_l^m(\mathbf{r}) Y_l^m(\hat{\mathbf{s}}), \tag{2.6}$$

where the $I_l^m(\mathbf{r})$ are position-dependent coefficients, the directional dependence of the RTE can be eliminated.

The first-order expansion of the radiative intensity is given by,

$$I(\mathbf{r}, \theta, \psi) = I_0^0 + I_1^0 \cos\theta - I_1^{-1} \sin\theta \sin\psi - I_1^1 \sin\theta \cos\psi. \tag{2.7}$$

By defining the incident radiation as the integration of the radiative intensity over the hole sphere,

$$G(\mathbf{r}) = \int_{4\pi} I(\mathbf{r}, \hat{\mathbf{s}}) \mathrm{d}\Omega, \tag{2.8}$$

the approximated RTE can be written as,

$$\nabla \cdot (\Gamma \nabla G) - aG = -4\epsilon\sigma T^4 - E, \tag{2.9}$$

with some mathematical manipulation, where $\Gamma = \frac{1}{3a+\sigma_s+a_0}$ is the diffusivity of the equation, a is the absorption coefficient, σ_s is the linear scattering factor, ϵ is the emission coefficient, and$E$ is the emision coefficient. The a_0 in Γ is a small value to make sure the division by zero error will not happen. Eq. (2.9) is formulated in OpenFOAM as below,

Formulation of the P1 model in OpenFOAM (`$FOAM_RADIATION/radiationModels/P1/P1.C`)

```
void Foam::radiation::nonGreyP1::calculate()
{
    absorptionEmission_->correct(G_, Gg_);

    const dimensionedScalar a0("a0", a_.dimensions(), ROOTVSMALL);
    // ...
    solve
    (
        fvm::laplacian(gamma, G_)
      - fvm::Sp(a_, G_)
      ==
      - 4.0*(e_*physicoChemical::sigma*pow4(T_)) - E_
    );
    // ...
}
```

It will be discussed in detail in the following sections. For the `FOAM_RADIATION`, it is defined as `export FOAM_RADIATION=$FOAM_SRC/thermophysicalModels/radiation`. This system variable is used in the following sections.

The generalized boundary condition for the PN model is the Marshak's boundary condition, which is given by,

$$\int_{\hat{\mathbf{n}}\cdot\hat{\mathbf{s}}>0} I(\mathbf{r}_w, \hat{\mathbf{s}}) \bar{Y}_{2i-1}^m(\hat{\mathbf{s}}) \mathrm{d}\Omega = \int_{\hat{\mathbf{n}}\cdot\hat{\mathbf{s}}>0} I_w(\hat{\mathbf{s}}) \bar{Y}_{2i-1}^m(\hat{\mathbf{s}}) \mathrm{d}\Omega, \quad i = 1, 2, \ldots \frac{1}{2}(N+1), \tag{2.10}$$

where the $\bar{Y}_{2i-1}^m(\hat{\mathbf{s}})$ are expressed in terms of a local coordinate system. That can be simplified as,

$$-\frac{2(2-\epsilon)}{\epsilon} \Gamma(\hat{\mathbf{n}} \cdot \nabla G) + G = 4\sigma T^4, \tag{2.11}$$

for the P1 approximation. This boundary conditions have two implementations in OpenFOAM, namely

1. `MarshakRadiationFvPatchScalarField`
2. `MarshakRadiationFixedTemperatureFvPatchScalarField`

The first one is for the general case, where the temperature is not fixed. The second one is for the case where the temperature is fixed. Both of them inherit from the `mixedFvPatchScalarField` class, which is a mixed boundary condition. Taking the first one as an example, the `updateCoeffs` function is shown below.

Formulation of the Marshak boundary condition in OpenFOAM

```
void Foam::radiation::MarshakRadiationFvPatchScalarField::updateCoeffs()
{
    // ...
    // Re-calc reference value
    refValue() = 4.0*constant::physicoChemical::sigma.value()*pow4(Tp);

    // ...

    // Set value fraction
    valueFraction() = 1.0/(1.0 + gamma*patch().deltaCoeffs()/Ep);

    // ...

    mixedFvPatchScalarField::updateCoeffs();
}
```

2.3.2 The full-spectrum correlated *k*-distribution Model

The full-spectrum correlated k-distribution (FSCK) model transfers the RTE from wavenumber space into the so-called g space, which can largely reduce the times of RTE evaluation. Detailed induction and procedure can be found in the reference [13]. For the implementation, it is pretty simple, it involves replacing the standard absorption coefficient a in the RTE with the k value from the FSCK model. Similarly, the emission coefficient e is substituted with the non-grey stretching coefficient (also denoted as a) from the FSCK model, adapting the RTE for FSCK computations.

This adaptation enables solving the RTE at designated FSCK quadrature points, selected using the Gauss-Chebyshev integration method for optimized spectral integral computation. At each quadrature point, the incident radiation (G_{g_n}) is calculated and then aggregated, with each G_{g_n} weighted by a factor w_n. This process is expressed as:

$$\mathrm{RTE}(k, a, G_{g_n}) = 0, \quad (n \in [1, nq]), \tag{2.12}$$

$$G = \sum_{n=1}^{nq} w_n G_{g_n}, \tag{2.13}$$

where nq is the number of FSCK quadrature points. This technique effectively captures the overall radiative behavior, enhancing computational efficiency.

The source term in the energy equation is:

$$\nabla \cdot q = \sum_{n=1}^{nq} w_n \nabla \cdot q_{g_n} = \sum_{n=1}^{nq} w_n (4\pi a I_b - G_{g_n}). \tag{2.14}$$

The k values, which depend on pressure p, temperature T, reference temperature T_{ref}, and species' volume fraction $\mathbf{x}$, are computationally intensive to calculate. To reduce this cost, Wang et al. developed a 12.45 GB FSCK lookup table for interpolating k values [12, 10, 35]. Later, Zhou et al. introduced a machine learning-based FSCK model, significantly reducing the model size to less than 50 MB [33, 32]. In this study, we utilize Zhou et al.'s machine learning-based model for predicting k values. The a array can be derived from the k array.

The reference temperature mentioned above can be derived from the following implicit equations,

$$\mathbf{x}_{\mathrm{ref}} = \frac{1}{V}\int_V \mathbf{x}\mathrm{d}V \tag{2.15}$$

$$k(p, T, T_{\mathrm{ref}}, \mathbf{x}_{\mathrm{ref}})I_b(T_{\mathrm{ref}}) = \frac{1}{V}\int_V k(p, T, T, \mathbf{x})I_b(T)\mathrm{d}V. \tag{2.16}$$

Chapter 3

Existing radiation models in OpenFOAM

This chapter provides an in-depth exploration of the implementation and application of radiation models within OpenFOAM. The discussion encompasses the integration of the radiation model as source terms in energy equations, detailing both the structure and functionality of the model. Special emphasis is placed on the `radiationModel` class and its subclasses, focusing on their roles in solving the RTE and calculating relevant source terms. Furthermore, practical aspects concerning the usage of the radiation model are elaborated, including configuration details and example applications. This chapter may contain many code snippets. If you can not find them, please use the `grep -r` command under the `$FOAM_SRC/thermophysicalModels/radiation` to find the corresponding files.

3.1 Overview

OpenFOAM currently does not have an individual radiation solver. Instead, the radiation model is implemented as a source term for solver's energy equation in `fvOptions`, as shown below.

Energy equation in the `reactingFoam` solver

```
    fvScalarMatrix EEqn
    (
        fvm::ddt(rho, he) + mvConvection->fvmDiv(phi, he)
        // ...
     ==
        Qdot
      + fvOptions(rho, he)
    );
```

Then the `radiation` class is defined in `radiation.H`, which inherits from `fv:option`.

The definition of `radiation` class (`$FOAM_RADIATION/fvOptions/radiation/radiation.C`)

```
    class radiation
    :
        public fv::option
    {
        // Private Data
        // ..
    };
```

However, this class is just a wrapper of the radiation model. The latter one is constructed in the constructor of `radiation`.

The constructor of `radiation` class

```
Foam::fv::radiation::radiation
```

```
(
    // ...
)
:
    fv::option(sourceName, modelType, dict, mesh)
{
    // ...
    radiation_ = Foam::radiation::radiationModel::New(thermo.T());
}
```

The `radiation` model overrides the `addSup` method in `fv::option`, which is called in the `fvOptions` method in the energy equation and serves as the source term. In `addSup` method, the `correct` is called to solve the RTE and the `Sh` method in `radiationModel` is called to calculate the source term.

The `addSup` method in `radiation` class

```
void Foam::fv::radiation::addSup
(
    const volScalarField& rho,
    fvMatrix<scalar>& eqn,
    const label fieldi
)
{
    const auto& thermo = mesh_.lookupObject<basicThermo>(basicThermo::dictName);

    radiation_->correct();

    eqn += radiation_->Sh(thermo, eqn.psi());
}
```

3.2 Detailed analysis

Following the overview of the radiation model's integration in OpenFOAM as a source term for energy equations, this section goes deeper into the specifics of the `radiationModel` class. It explores the functionalities and interactions of `radiationModel` with its essential subclasses.

3.2.1 The `radiationModel` class: RTE solver

The `radiationModel` class is a fundamental component in OpenFOAM's radiation model framework, involving tasks such as spectral coefficient evaluations, solving the Radiative Transfer Equation (RTE), and calculating source terms. This class interfaces with three essential radiation sub-models: `absorptionEmissionModel`, `scatterModel`, and `sootModel`. The `absorptionEmissionModel` calculates absorption and emission coefficients and the emission contribution. The `scatterModel` is responsible for computing the scattering coefficient, while the `sootModel` predicts the spatial distribution of soot.

The RTE is solved in the `correct` method, by calling the virtual method `calculate`. The source term is calculated in the `Sh` or the `ST` methods depending on the energy equation, which in turn both call two virtual members, `Ru` and `Rp`. As with other CFD solvers, linearization of the source term is necessary, hence the use of `Ru` and `Rp`. These methods should be overloaded by subclasses of `radiationModel`.

The `Sh` method in `radiationModel` class

```
Foam::tmp<Foam::fvScalarMatrix> Foam::radiation::radiationModel::Sh
(
    const basicThermo& thermo,
    const volScalarField& he
) const
{
    const volScalarField Cpv(thermo.Cpv());
    const volScalarField T3(pow3(T_));

    return
    (
        Ru()
        - fvm::Sp(4.0*Rp()*T3/Cpv, he)
        - Rp()*T3*(T_ - 4.0*he/Cpv)
    );
}
```

`radiationModel` serves as a base class and has six subclasses, representing different RTE solver models in OpenFOAM:

- `fvDOM`
- `noRadiation`
- `opaqueSolid`
- `P1`
- `solarLoad`
- `viewFactor`

Taking `P1` as an example, the `calculate` method in `P1` initially utilizes the `absorptionEmission_` object to obtain absorption and emission coefficients, and the emission contribution. This object is a pointer to the `absorptionEmissionModel`. The scattering coefficient is then obtained from the pointer object `scatter_`.

Getting spectral coefficient in `P1::calculate() method`

```
void Foam::radiation::P1::calculate()
{
    a_ = absorptionEmission_->a();
    e_ = absorptionEmission_->e();
    E_ = absorptionEmission_->E();
    const volScalarField sigmaEff(scatter_->sigmaEff());
    // ...
```

Subsequently, `gamma` is calculated, incorporating a minimal value `a0` to prevent division by zero errors.

Defining equation parameter in `P1::calculate() method`

```
    // ...
    const dimensionedScalar a0("a0", a_.dimensions(), ROOTVSMALL);

    // Construct diffusion
    const volScalarField gamma
    (
        IOobject
        (
            "gammaRad",
            G_.mesh().time().timeName(),
            G_.mesh(),
            IOobject::NO_READ,
            IOobject::NO_WRITE
        ),
        1.0/(3.0*a_ + sigmaEff + a0)
    );
    // ...
```

Equation (2.9) is then formulated and solved.

Solving equation in `P1::calculate() method`

```
    // ...
    // Solve G transport equation
    solve
    (
        fvm::laplacian(gamma, G_)
      - fvm::Sp(a_, G_)
     ==
      - 4.0*(e_*physicoChemical::sigma*pow4(T_)) - E_
    );
    // ...
```

Radiative heat flux on boundaries is calculated at the conclusion of the `calculate` method.

Getting radiative heat flux on boundaries in `P1::calculate() method`

```
    // ...
    // Calculate radiative heat flux on boundaries.
    volScalarField::Boundary& qrBf = qr_.boundaryFieldRef();
    const volScalarField::Boundary& GBf = G_.boundaryField();
    const volScalarField::Boundary& gammaBf = gamma.boundaryField();

    forAll(mesh_.boundaryMesh(), patchi)
    {
        if (!GBf[patchi].coupled())
        {
            qrBf[patchi] = -gammaBf[patchi]*GBf[patchi].snGrad();
        }
    }
}
```

The `Rp()` method is overloaded to represent $4e\sigma$, and `Ru()` is formulated as $aG - E$, indicating the source terms for the energy equation.

3.2.2 The `absorptionEmissionModel` class: spectral model

The `absorptionEmissionModel` class serves as the foundational class for spectral models in OpenFOAM. This class is simple, providing virtual methods that can be overloaded to acquire or adjust spectral coefficients. It has eight subclasses, each tailored for specific spectral modeling scenarios.

One simple subclass is `greyMeanAbsorptionEmission`. This class operates under the assumption that the participating gases are grey, meaning their absorption coefficients are constant. The absorption coefficient is modeled as a temperature-dependent polynomial, and the emission coefficient is set equal to the absorption coefficient.

The code snippet below illustrates the process of evaluating the absorption coefficient in the `greyMeanAbsorptionEmission` class:

Absorption Coefficient Evaluation in `greyMeanAbsorptionEmission` class

```
Foam::tmp<Foam::volScalarField>
Foam::radiation::greyMeanAbsorptionEmission::aCont(const label bandI) const
{
    // ...

    scalar Ti = T[celli];
    // Adjust for negative temperature exponents
    if (coeffs_[n].invTemp())
    {
        Ti = 1.0/T[celli];
    }
    // Polynomial calculation for absorption coefficient
    a[celli] +=
        Xipi
       *(
            ((((b[5]*Ti + b[4])*Ti + b[3])*Ti + b[2])*Ti + b[1])*Ti
          + b[0]
        );
    }
}
ta.ref().correctBoundaryConditions();
return ta;
```

3.3 Usage of radiation models

There is only one case in the OpenFOAM tutorial that uses the radiation model, which is the `SandiaD_LTS` case. However, the radiation model can be coupled with any solver that has an energy equation. To use the model, one may need first create a `fvOptions` file under the `constant` folder,

`fvOptions` dictionary for radiation model

```
/*--------------------------------*- C++ -*----------------------------------*\
| =========                 |                                                 |
| \\      /  F ield         | OpenFOAM: The Open Source CFD Toolbox           |
|  \\    /   O peration     | Version:  v2112                                 |
|   \\  /    A nd           | Website:  www.openfoam.com                      |
|    \\/     M anipulation  |                                                 |
\*---------------------------------------------------------------------------*/
FoamFile
{
    version     2.0;
    format      ascii;
    class       dictionary;
    object      fvOptions;
}
// * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * //

radiation
{
    type            radiation;
    libs (radiationModels);
}


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

Then, `radiationProperties` and `boundaryRadiationProperties` are required to specify the radiation model's properties. The `radiationProperties` is a dictionary that specifies the RTE solver and the spectral model under the `constant` folder, while the `boundaryRadiationProperties` is a dictionary that specifies the radiation model's properties on the boundary. The `radiationProperties` is defined as below, which shows how to select P1 as the RTE solver and the grey mean absorption emission model as the spectral model. The absorption coefficient for each gas is the function of temperature and is described as a polynomial.

`radiationProperties` dictionary for radiation model

```
/*--------------------------------*- C++ -*----------------------------------*\
| =========                 |                                                 |
| \\      /  F ield         | OpenFOAM: The Open Source CFD Toolbox           |
|  \\    /   O peration     | Version:  v2112                                 |
|   \\  /    A nd           | Website:  www.openfoam.com                      |
|    \\/     M anipulation  |                                                 |
\*---------------------------------------------------------------------------*/
FoamFile
{
    version     2.0;
    format      ascii;
    class       dictionary;
    object      radiationProperties;
}
// * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * //
```

```

radiation on;

radiationModel  P1;

P1Coeffs
{
    C               C [0 0 0 0 0 0 0] 0;
}

// Number of flow iterations per radiation iteration
solverFreq 1;

absorptionEmissionModel greyMeanAbsorptionEmission;

greyMeanAbsorptionEmissionCoeffs
{
    lookUpTableFileName      none;

    EhrrCoeff                0.0;

    CO2
    {
        Tcommon         200;   //Common Temp
        invTemp         true;   //Is the polynomio using inverse temperature.
        Tlow            200;   //Low Temp
        Thigh           2500;  //High Temp

        loTcoeffs       //coefss for T < Tcommon
        (
            0           //  a0            +
            0           //  a1*T          +
            0           //  a2*T^(+/-)2   +
            0           //  a3*T^(+/-)3   +
            0           //  a4*T^(+/-)4   +
            0           //  a5*T^(+/-)5   +
        );
        hiTcoeffs        //coefss for T > Tcommon
        (
            18.741
            -121.31e3
            273.5e6
            -194.05e9
            56.31e12
            -5.8169e15
        );

    }

    // Rest of the gas...
}

scatterModel    none;

sootModel       none;

transmissivityModel none;


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

```

For the `boundaryRadiationProperties`, a `lookup` model is selected, which assumes the boundary is grey. This assumption can be taken in most of the combustion cases.

`boundaryRadiationProperties` dictionary for radiation model

```
/*--------------------------------*- C++ -*----------------------------------*\
| =========                 |                                                 |
| \\      /  F ield         | OpenFOAM: The Open Source CFD Toolbox           |
|  \\    /   O peration     | Version:  v2112                                 |
|   \\  /    A nd           | Website:  www.openfoam.com                      |
|    \\/     M anipulation  |                                                 |
\*---------------------------------------------------------------------------*/
FoamFile
{
    version     2.0;
    format      ascii;
    class       dictionary;
    object      boundaryRadiationProperties;
}
// * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * //

".*"
{
    type            lookup;
    emissivity      1;
    absorptivity    0;
}


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

Chapter 4

Developing a non-grey radiation model in OpenFOAM

As outlined in the preceding chapter, the grey gas model provides a simplified approach for modeling radiative heat transfer, and its implementation in OpenFOAM is well-established. However, this model's accuracy is significantly limited by the assumption of a constant absorption coefficient across different wavenumbers. This grey global spectral model can sometimes lead to greater errors than those incurred by neglecting radiative heat transfer altogether. Consequently, developing a non-grey radiation model within OpenFOAM is crucial.

In this study, the FSCK model is implemented in OpenFOAM. Firstly, the reference temperature is determined based on the current computational field. Subsequently, the k and a values are calculated considering the pressure, temperature, reference, and species' volume fraction, as per Zhou's model [32]. Additionally, the existing `P1` model in OpenFOAM, a grey RTE solver as described in Subsection. 3.2.1, solves the RTE only once per iteration. In contrast, a non-grey RTE solver requires multiple RTE solutions per iteration. Hence, there is also a need to develop a non-grey P1 model. Similarly, a non-grey version of the Marshak boundary condition, used by the P1 model, should be developed.

To integrate with the existing OpenFOAM framework, a support class named `fsckMLP` is created. This class calculates the reference temperature T_{ref}, computes k and a values, and stores the k and a fields. The singleton pattern is employed to facilitate interaction with the rest of the model.

To summarize, the following parts have been developed in this study:

1. `fsckMLP`: Performs calculations for the FSCK model and stores fields related to it.
2. `nonGreyMeanAbsorptionEmission`: A non-grey version of the `meanAbsorptionEmission` class.
3. `nonGreyP1`: A non-grey version of the `P1` class.
4. `nonGreyMarshakRadiationFvPatchScalarField`: A non-grey version of the Marshak boundary condition.
5. `nonGreyMarshakRadiationFixedTemperatureFvPatchScalarField`: A non-grey version of the Marshak boundary condition with fixed temperature.
6. Rest classes and functions for the FSCK model, like deriving the a array from k array etc.

These classes are discussed in detail in the following sections.

4.1 Developing the FSCK model

4.1.1 Obtaining the FSCK parameters

Zhou et al. [32, 33] employed three multilayer perceptrons (MLPs) to predict the FSCK parameters across varying pressure ranges. In our study, an `MLP` class was developed to implement the MLP model. This class utilizes a third-party, header-only JSON library [36] for reading the model parameters. These parameters are loaded within the constructor of the `MLP` class. Subsequently, several methods are defined: `normalize_input` for input normalization, `forward` for conducting the forward pass, `denormalize_output` for output denormalization, and `preprocess` and `postprocess` for pre- and post-processing steps, respectively. Furthermore, an `MLPManager` class, acting as a wrapper for the three `MLP` instances, was implemented. This class enables the prediction of FSCK parameters across the entire pressure spectrum through a singular method invocation, `get_prediction`.

The computation and storage of the FSCK parameters, namely k and a, are managed within the `fsckMLP` class. In this class's constructor, an object of `MLPManager` is constructed. Then the array for g and the weight array for w are computed using the `quadgen2` function. Following this, two `PtrLists` of `volScalarField`, named `ki_` and `ai_`, are initialized through the `initKA` method, as described below, used to store the k and a value across different quadrature points.

Initialzing fields for k and a

```
void Foam::radiation::fsckMLP::initKA()
{
    ki_.setSize(nBands_);
    ai_.setSize(nBands_);
    for (label i=0; i<nBands_; i++)
    {
        Info << "Initialzing k and a for band" << i << endl;
        ki_.set(i, new volScalarField
        (
            IOobject
            (
                "k" + std::to_string(i),
                mesh_.time().timeName(),
                mesh_,
                IOobject::NO_READ,
                IOobject::AUTO_WRITE
            ),
            mesh_,
            dimensionedScalar(dimless/dimLength, ROOTVSMALL)
        ));
        ai_.set(i, new volScalarField
        (
            IOobject
            (
                "a" + std::to_string(i),
                mesh_.time().timeName(),
                mesh_,
                IOobject::NO_READ,
                IOobject::AUTO_WRITE
            ),
            mesh_,
            dimensionedScalar(dimless/dimLength, ROOTVSMALL)
        ));
    }
}
```

The reference temperature computation is implemented by the `updateTref` method. This process involves iterating over the fields to determine the average values of $\mathbf{x}$ and $k(p, T, T, \mathbf{x})I_b(T)$, under Eqs (2.15) and (2.16). Since OpenFOAM stores the species density by the mass fractions, the corresponding volume fractions are supposed to be computed. Subsequently, Equation (2.16) is solved using an implicit approach.

Caculating volume fractions

```
    scalar invWt = 0.0;
    scalar xco2_cell, xh2o_cell, xco_cell;
    forAll(mixture.Y(), s)
    {
        invWt += mixture.Y(s)[celli]/mixture.W(s);
    }
    xco2_cell = mixture.Y("CO2")[celli]/(mixture.W(mixture.species()["CO2"])*invWt);
    xh2o_cell = mixture.Y("H2O")[celli]/(mixture.W(mixture.species()["H2O"])*invWt);
    xco_cell = mixture.Y("CO")[celli]/(mixture.W(mixture.species()["CO"])*invWt);

    xco2 = xco2 + xco2_cell * mesh_.cellVolumes()[celli];
    xh2o = xh2o + xh2o_cell * mesh_.cellVolumes()[celli];
    xco = xco + xco_cell * mesh_.cellVolumes()[celli];
```

It is noteworthy that `reduce` function should be called at the end of the loop to ensure that the values of `xco2`, `xh2o`, `xco`, etc. are consistent across all processors make the model can be ran parallelly.

To update the k and a values for each quadrature point, the `updateKA` method is invoked. The loop in fields is performed to iterate over all internal cells at first. In the loop, the cell's pressure, temperature and volume fraction are obtained. Then the `MLPManager` instance is used to predict k array with a size of 32. And then the a array is calculated using the `afun` function. Depending on the user's demand on the number of quadrature points, k and a array would be interpolated to the corresponding size. Finally, the k and a values are stored in the volScalarField array.

Update k and a for internal field

```
    forAll(T, celli)
    {
        //...
        double a[NqDB];
        afun(gNqDB, k_Tref.data(), k_T.data(), wNqDB, a);
        double k_new[nBands_], a_new[nBands_];
        simple_interp(NqDB, nBands_, gNqDB, k_Tref.data(), gNq, k_new);
        simple_interp(NqDB, nBands_, gNqDB, a, gNq, a_new);

        forAll(ki_, bandI)
        {
            ki_[bandI][celli] = k_new[bandI]*100.0;
            ai_[bandI][celli] = a_new[bandI];
        }
    }
```

We should also obtain the FSCK parameters for boundary cells.

Update k and a for boundaries

```
    forAll(T.boundaryField(), patchi)
    {
        forAll (T.boundaryField()[patchi], facei)
        {
            // ...

            forAll(ai_, bandI)
            {
                ai_[bandI].boundaryFieldRef()[patchi][facei] = a_new[bandI];
                ki_[bandI].boundaryFieldRef()[patchi][facei] = k_new[bandI]*100.0;
            }
        }
    }
```

We must use `T.boundaryField()`, or a comparable method, for iterating over each patch, rather than employing `mesh_.boundary()` directly. This approach is essential to avoid accessing boundaries with the `empty` type, which would otherwise result in an error.

4.1.2 Providing the FSCK parameters

The `nonGreyMeanAbsorptionEmission` class is implemented to build a bridge between `fsckMLP` and `nonGreyP1`. In the constructor of this class, the number of bands is obtained from the dictionary as follows.

Getting the number of bands

```
    // read the number of bands
    if(coeffsDict_.found("nBands"))
    {
        nBands_ = coeffsDict_.get<label>("nBands");
    }
    else
    {
        nBands_ = 4;
    }
    Info << "nBands: " << nBands_ << endl;
```

In the `correct` method, the `fsckMLP` instance is obtained through the singleton pattern. Then the `updateTref` and `updateKA` methods are invoked to update the reference temperature. Finally, the `k` and `a` fields are updated. Since the `correct` method is a virtual method from the parent class, it has to take two inputs. However, these two inputs are not used in the `correct` method of `nonGreyMeanAbsorptionEmission`, so they are named as `dummy` and `dummyj`.

Updating the k and a fields

```
void Foam::radiation::nonGreyMeanAbsorptionEmission::correct
    (
        volScalarField& dummy,
        PtrList<volScalarField>& dummyj
    ) const
{
    Foam::radiation::fsckMLP* fsck = Foam::radiation::fsckMLP::getInstance();
    fsck->updateTref();
    fsck->updateKA();
}
```

After that, the k and a value at different quadrature points can be obtained by overloading the the existing `aCont` and `eCont` methods.

Providing k and a values

```
Foam::tmp<Foam::volScalarField>
Foam::radiation::nonGreyMeanAbsorptionEmission::aCont(const label bandI) const
{
    Foam::radiation::fsckMLP* fsck = Foam::radiation::fsckMLP::getInstance();
    return fsck->get_k(bandI);
}


Foam::tmp<Foam::volScalarField>
Foam::radiation::nonGreyMeanAbsorptionEmission::eCont(const label bandI) const
{
    Foam::radiation::fsckMLP* fsck = Foam::radiation::fsckMLP::getInstance();
    return fsck->get_a(bandI);
}
```

4.2 Developing the non-grey P1 model

4.2.1 Solving the RTE

At the beginning of the `calculate` method, the `correct` method within the `absorptionEmission_` object is called to update the k and a fields. The loop of solving the RTE on different quadrature points is performed. In every iteration, the spectral parameters are obtained from this object by invoking the `a`, `e`, and `E` methods. The diffusion factor Γ is derived analogously to the grey P1 model, but the non-grey model uses different spectral parameters on different quadrature points. Following this, the P1 equation is formulated, for the incident radiation at various quadrature points. The `solve` method is then employed to solve the P1 equation.

In the context of Marshak's boundary condition integrated with the non-grey P1 model, a non-grey stretching factor is multiplied by the reference value, as illustrated below, to fulfill the FSCK model's requirement.

Non-grey Marshak boundary condition

```
    fsckMLP* fsck = fsckMLP::getInstance();

    const word& aName_("a" + std::to_string(fsck->getBandI()));

    // Nongrey streching factor field
    const scalarField& ap =
        patch().lookupPatchField<volScalarField, scalar>(aName_);

    // Temperature field
    const scalarField& Tp =
        patch().lookupPatchField<volScalarField, scalar>(TName_);

    // Re-calc reference value
    refValue() = 4.0*constant::physicoChemical::sigma.value()*pow4(Tp)*ap;
```

4.2.2 Collecting the results

The incident radiation in the whole spectrum `G_` is collected by the summation of incident radiation on each quadrature point with a weight factor w_n as described in Eq. (2.13).

Collecting the incident radiation

```
    forAll(Gg_, bandI)
    {
        G_ += Gg_[bandI]*wNq[bandI];
        forAll(mesh_.boundaryMesh(), patchi)
        {
            G_.boundaryFieldRef()[patchi] == G_.boundaryFieldRef()[patchi] +
                Gg_[bandI].boundaryField()[patchi]*wNq[bandI];
        }
    }
```

Collecting the source term components follows a similar pattern to that of the incident radiation. For the source term component for T^4, we implement,

Collecting the source term component for T^4

```
Foam::tmp<Foam::volScalarField> Foam::radiation::nonGreyP1::Rp() const
{
    tmp<volScalarField> tRp
    (
        new volScalarField
        (
            IOobject
            (
                "Rp",
                mesh_.time().timeName(),
                mesh_,
                IOobject::NO_READ,
                IOobject::NO_WRITE
            ),
            4.0*absorptionEmission_->eCont()*physicoChemical::sigma*0.0
        )
    );

    fsckMLP* fsck = fsckMLP::getInstance();

    forAll(Gg_, bandI)
    {
        tRp = tRp + fsck->getwNq()[bandI] * absorptionEmission_->a(bandI) *
                4*physicoChemical::sigma * absorptionEmission_->e(bandI) *
                dimensionedScalar(dimLength, 1);
    }

    return tRp;
}
```

For the constant source term component, we implement,

Collecting the constant source term component

```
Foam::tmp<Foam::DimensionedField<Foam::scalar, Foam::volMesh>>
Foam::radiation::nonGreyP1::Ru() const
{
    tmp<DimensionedField<scalar, volMesh>> tRu
    (
        new volScalarField
        (
            IOobject
            (
                "Ru",
                mesh_.time().timeName(),
                mesh_,
                IOobject::NO_READ,
                IOobject::NO_WRITE
            ),
            mesh_,
            dimensionedScalar(dimMass/dimLength/pow3(dimTime), Zero)
        )
    );

    fsckMLP* fsck = fsckMLP::getInstance();

    forAll(Gg_, bandI)
    {
        DimensionedField<scalar, volMesh> tRui
        (
            fsck->getwNq()[bandI] * absorptionEmission_->a(bandI) *
                (
                    Gg_[bandI]// -
                    // 4*physicoChemical::sigma *
                    // absorptionEmission_->e(bandI) * pow4(T_) *
                    // dimensionedScalar(dimLength, 1)
                )
        );
        tRu = tRu + tRui;
    }

    return tRu;
}
```

Chapter 5

Model validation

In this chapter, the model's validity is assessed by comparing its results with those obtained from two distinct codes. Modest's research group has developed a compact radiative heat transfer model in OpenFOAM 2.2x. This model has undergone extensive verification, as documented in several publications [10, 11, 12, 13, 14]. However, they develop the radiative heat transfer model in their framework. To couple with the existing solver in OpenFOAM, modification on the source code of the solver is necessary. Ren's research group [17] has successfully implemented an in-house FSCK model within ANSYS Fluent, which has been validated against Modest's code. However, due to limitations in ANSYS Fluent's user-defined function (UDF), the boundary treatment in Ren's model is not entirely accurate, potentially leading to errors in highly non-homogeneous cases. The subsequent sections present a comparative analysis of the results from these three codes to ascertain the accuracy of the newly developed model.

All simulations discussed herein are termed as 'snapshot simulations'. This approach means solving the RTE once, under fixed conditions of temperature, pressure, and species volume fractions, which remain constant over time. The OpenFOAM solver `thermoFoam` is employed.

5.1 1D slab

The model's initial validation is conducted using a one-dimensional (1D) homogeneous slab. This slab, measuring 1 m in length, encompasses a uniformly distributed internal temperature field of 1200 K. It contains a mixture of gases, consisting of 27.32% CO_2, 3.477% CO, and 6.298% H_2O, at a pressure of 1 bar. To assess the model's performance under varying conditions, two subcases are simulated, each characterized by distinct wall temperatures of 300 K and 600 K, respectively. The comparative results, as demonstrated in Fig. 5.1, are derived from simulations conducted with the newly developed code (hereafter 'new code') and the established OpenFOAM code developed by Modest's group. The simulations correspond to wall temperatures set at 300 K and 600 K. It is evident that the computational outcomes from both codes are identical for the incident radiation and the radiative heat source terms across the two cases. Notably, the case with the elevated wall temperature of 1200 K results in a higher level of incident radiation. However, the variations in the radiative heat source term for both cases are minimal and can be considered negligible.

Temperature distribution and species' volume fractions are prescribed following Gaussian distri-

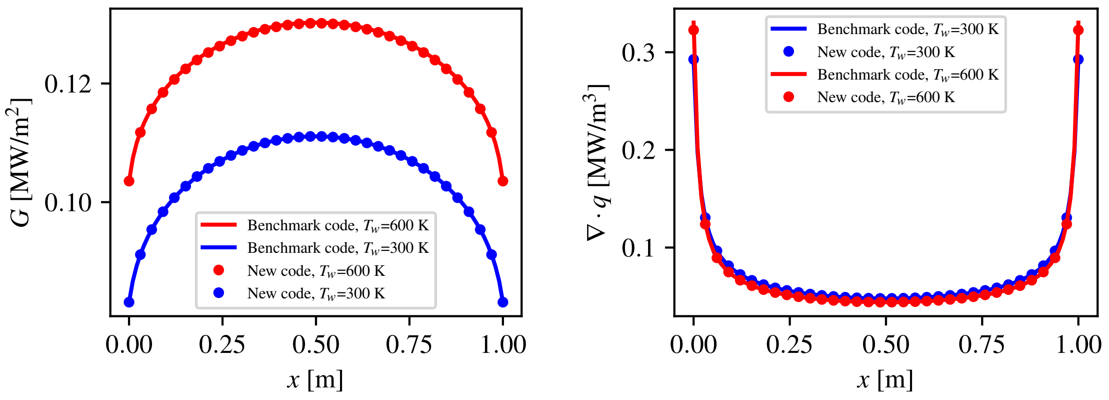

Figure 5.1: Incident radiation (left) and radiative heat source (right) for 1D homogeneous slab

butions, given by:

$$T = \left(1600 \exp\left(-\frac{(x-0.2)^2}{0.3^2}\right) + 400\right) \text{ K}, \tag{5.1}$$

$$x_{\mathrm{CO_2}} = 0.15 \exp\left(-\frac{(x-0.2)^2}{0.3^2}\right), \tag{5.2}$$

$$x_{\mathrm{H_2O}} = 0.15 \exp\left(-\frac{(x-0.2)^2}{0.3^2}\right), \tag{5.3}$$

$$x_{\mathrm{CO}} = 0.075 \exp\left(-\frac{(x-0.2)^2}{0.3^2}\right). \tag{5.4}$$

Fig. 5.2 illustrates the incident radiation and radiative heat source terms as predicted by the new code (denoted with dots) and the benchmark code (denoted with lines) under varying pressure conditions, ranging from 0.5 bar to 20 bar. The disparity in predictions made by the two models is minimal.

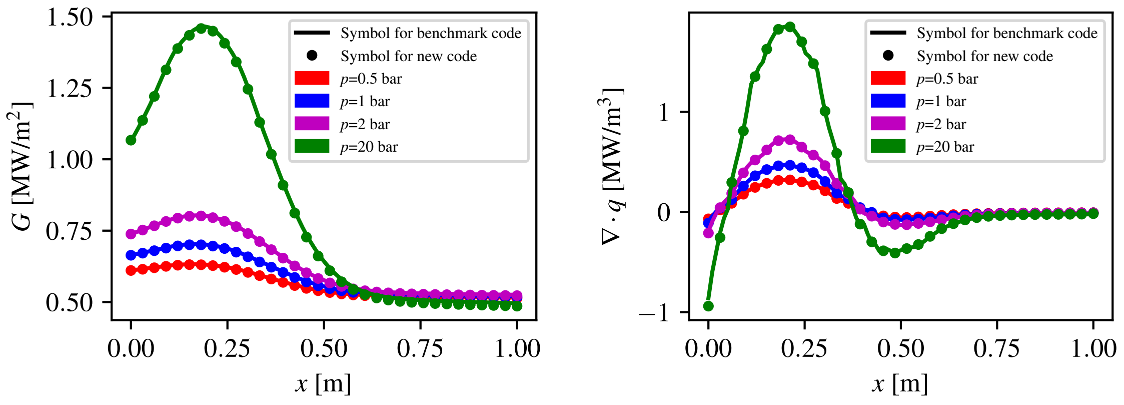

Figure 5.2: Incident radiation (left) and radiative heat source (right) for a 1D non-homogeneous slab at different pressures.

To sum up, the validation cases in this section demonstrate the correctness of the newly developed model. The model is capable of predicting the radiative heat transfer in a 1D slab with non- or homogeneous cases with varying pressure, temperature, and species' volume fractions. The results from the new code are in good agreement with those from the benchmark code.

5.2 2D square enclosure

A 2D homogeneous square enclosure, with dimensions of 1 m × 1 m, is established to evaluate the performance of the newly developed code against the in-house FSCK model implemented through the ANSYS Fluent User-Defined Function (UDF). The internal conditions of the enclosure are set to a pressure of 1 bar and a temperature of 1200 K, with a gas composition of 30% CO_2, 20% CO, and 50% H_2O. The wall is set to be cold and black (0 K, blackbody). The results obtained from the newly developed code are presented in Fig. 5.3, while a comparative analysis with the results from the existing Fluent UDF is illustrated in Fig. 5.4.

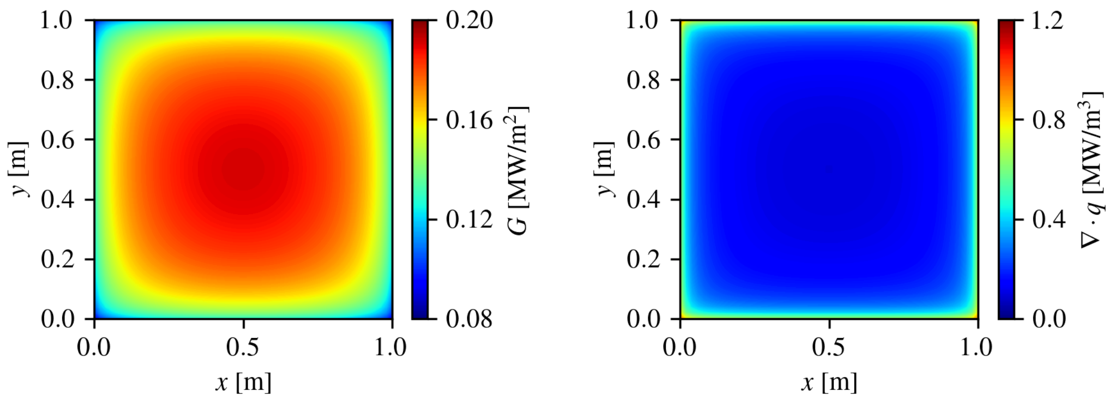

Figure 5.3: Incident radiation (left) and radiative heat source (right) for 2D homogeneous enclosure predicted by the new code

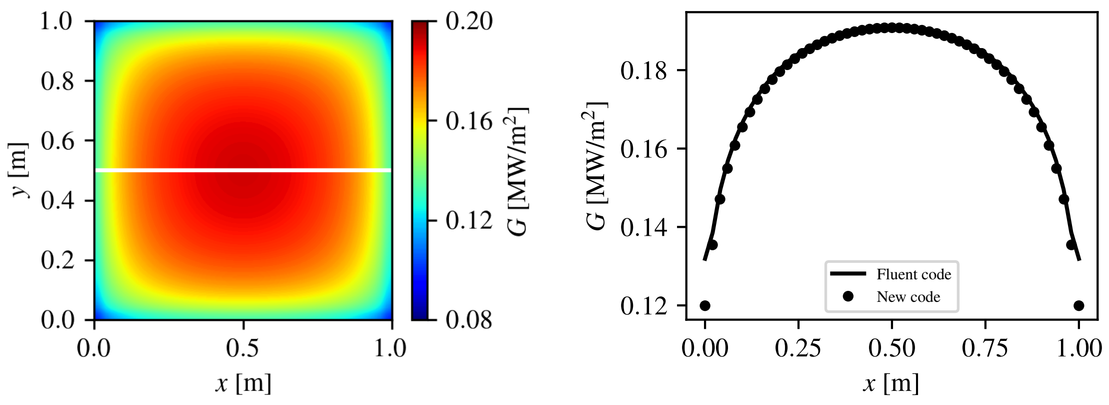

Figure 5.4: Incident radiation for 2D homogeneous enclosure predicted by the in-house UDF code (left) and the comparison with the new code sampling at the middle white line (right)

In the non-homogeneous two-dimensional (2D) case, the same square enclosure of dimensions 1 m × 1 m is employed. The internal pressure is maintained at 1 bar, while the temperature and species concentration distributions are governed by a spatially varying profile. This profile is defined by the equation:

$$\phi(x,y) = \max\left(0, 1 - \frac{\|\mathbf{pos}(x,y) - \mathbf{c}\|}{r}\right), \tag{5.5}$$

where $\phi(x,y)$ represents the scaling factor for temperature and concentrations, $\mathbf{pos}(x,y)$ is the position vector within the enclosure, $\mathbf{c} = (0.5, 0.5)$ m is the center of the distribution, and $r = 0.5$ m is the radius of influence. Accordingly, the temperature varies from 0 to 1200 K, CO_2 concentration from 0 to 0.1, H_2O from 0 to 0.05, and CO from 0 to 0.03, all scaled by $\phi(x,y)$. This spatial distribution introduces a gradient in both thermal and compositional fields, providing a more complex scenario for the evaluation of the computational model.

The results obtained from the new code for the 2D nonhomogeneous enclosure case are depicted in Fig. 5.5, showcasing the incident radiation and radiative heat source distributions. For comparative analysis, results from the existing in-house UDF code are presented in the subsequent figure, highlighting the incident radiation predicted by the UDF and a detailed comparison along a specific line within the enclosure, as shown in Fig 5.6.

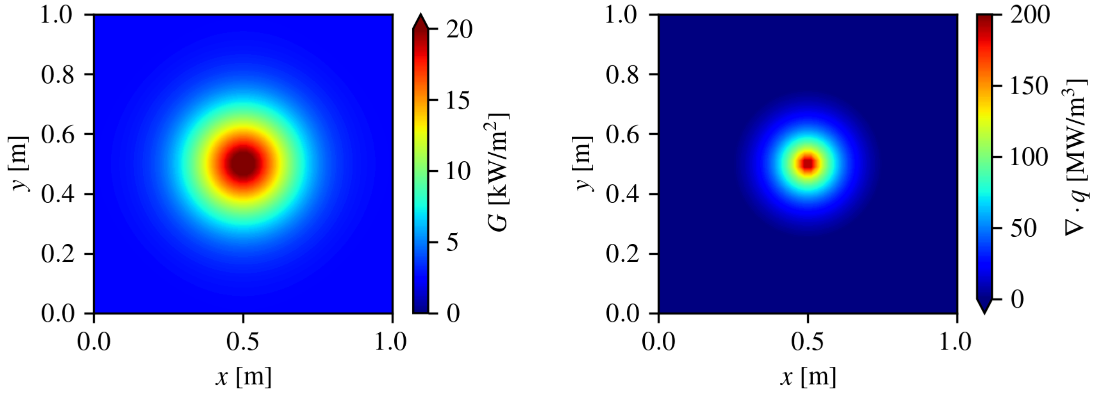

Figure 5.5: Incident radiation (left) and radiative heat source (right) for 2D nonhomogeneous enclosure predicted by the new code

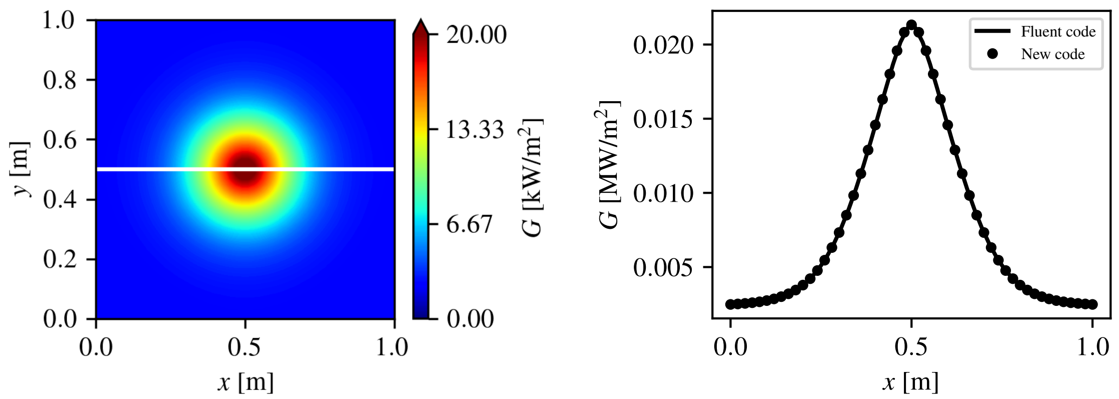

Figure 5.6: Incident radiation for 2D nonhomogeneous enclosure predicted by the in-house UDF code (left) and the comparison with the new code sampling at the middle white line (right)

It is observed that in the non-homogeneous case with a cold wall, the predictions made by the newly developed code demonstrate high consistency with the results obtained using the in-house

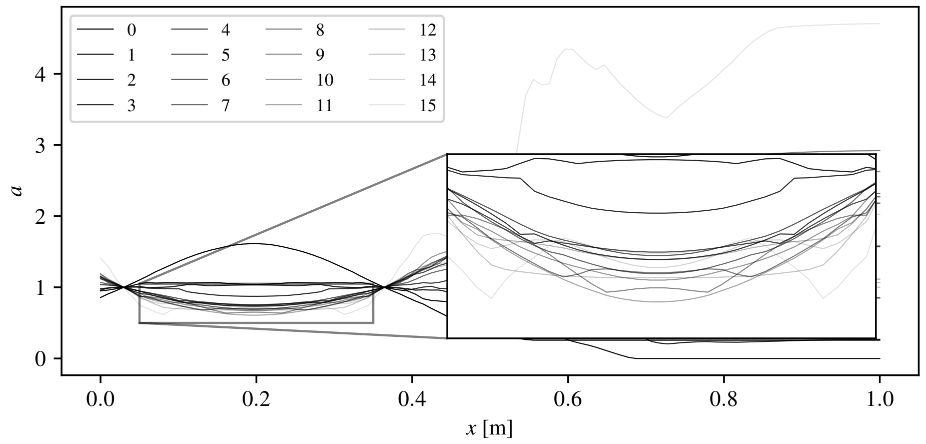

Figure 5.7: Variation of the non-grey stretching factor a in 1D nonhomogeneous cases at 1 bar (Number in the legend denotes the quadrature index).

ANSYS Fluent User-Defined Function (UDF). However, in the homogeneous case with a strong temperature gradient from the internal field to the boundary (1200 K to 0 K), the incident radiation predictions near the boundary exhibit slight discrepancies between the two codes. This variation is attributed to the UDF's lack of a suitable macro for boundary condition treatment, leading to minor differences in the incident radiation near the boundary (see Fig. 5.4), which means that the non-grey stretching factor at the boundary in the UDF code is constrained to be a constant value of 1.

5.3 Discussion

It is noteworthy that the radiative heat loss in the 1D nonhomogeneous case exhibits minor oscillations, as illustrated in Fig. 5.2. This phenomenon is not attributable to stability issues with the numerical solver. Instead, it arises from the method employed to derive a values from k values, which is an approximation technique and may introduce additional errors. Figure 5.7 clearly demonstrates that the a values themselves oscillate spatially, particularly at larger quadrature points, which is not usual.

5.4 Summary

This chapter conducted several validation cases to assess the accuracy of the newly developed model. I compared the results from my new code with those from existing codes, specifically those developed by Modest's group (in OpenFOAM 2.2x) and Ren's group (in ANSYS Fluent UDF). This comparison demonstrates that my code effectively implements the FSCK model across non-homogeneous and homogeneous scenarios at varying pressures, temperatures, and species volume fractions. The results from my new code show strong agreement with the benchmark results from Modest's code in OpenFOAM 2.2x.

One significant advantage of my newly developed code is its ability to integrate with any OpenFOAM solver or third-party solver that includes thermodynamic models, without requiring modifications to the source code of the solver. This feature offers a distinct benefit over the model from Modest's group. Additionally, the flexibility of the OpenFOAM framework allows my code to accurately handle boundary conditions. In contrast, Tao's group's model (in ANSYS Fluent UDF) can only set the non-grey stretching factor as a constant value of 1 at the boundaries, which may bring additional error.

Chapter 6

Conclusion and future work

In this report, I have explored the fundamentals of radiative heat transfer and its modeling in OpenFOAM. I have discussed the underlying principles and mechanisms that govern this mode of heat transfer, with particular emphasis on its relevance and application in CFD scenarios. I have also examined the radiation models currently available in OpenFOAM, detailing their features, capabilities, and typical use cases. Furthermore, I have provided a step-by-step guide on how to implement these models in various CFD simulations. Building upon the existing framework, I have introduced and detailed the integration of an advanced spectral model - the full-spectrum correlated k distribution model - into the OpenFOAM library. I have provided an in-depth explanation of the model, its theoretical basis, implementation strategies, and expected enhancements in simulation accuracy and efficiency. Lastly, I have validated the newly implemented model through multiple cases, comparing its performance and results with those of the pre-existing in-house codes for OpenFOAM and ANSYS Fluent. The comparison was based on various metrics and simulation scenarios to ensure a robust evaluation of the new model's accuracy and validity.

Future research will focus on several key areas to enhance the FSCK model's performance and applicability. Firstly, the development of Multilayer Perceptrons (MLPs) to accurately predict both k and a values is a priority. This initiative aims to address the errors in calculating a values from correlation patterns, a challenge identified in Section 5.3. By employing advanced machine learning techniques, I expect a significant improvement in the predictive accuracy of these parameters. Secondly, it is imperative to improve the newly implemented FSCK model's compatibility with other radiation solvers in OpenFOAM. Currently, the model integrates exclusively with the `nonGreyP1` solver and is not compatible with the built-in `fvDOM`. Enhancing this compatibility will broaden the model's utility across various simulation scenarios within OpenFOAM. Lastly, applying the FSCK model to more practical cases, such as combustion scenarios, is essential. This application will include comparing simulation outcomes with experimental data to validate the model's efficacy in real-world settings, thereby reinforcing its value in addressing practical engineering challenges.

Bibliography

[1] T. Ren, M. F. Modest, and S. Roy, "Monte Carlo Simulation for Radiative Transfer in a High-Pressure Industrial Gas Turbine Combustion Chamber," *Journal of Engineering for Gas Turbines and Power*, vol. 140, p. 051503, May 2018.

[2] T. Ren and M. F. Modest, "Line-by-Line Random-Number Database for Monte Carlo Simulations of Radiation in Combustion System," *Journal of Heat Transfer*, vol. 141, p. 022701, Feb. 2019.

[3] F. Liu, J.-L. Consalvi, and F. Nmira, "The importance of accurately modelling soot and radiation coupling in laminar and laboratory-scale turbulent diffusion flames," *Combustion and Flame*, p. 112573, Dec. 2022.

[4] M. Sheng, H. Pan, D. Xu, and D. Zhao, "Characterization and performance enhancement of radiative cooling on circular surfaces," *Renewable and Sustainable Energy Reviews*, vol. 188, p. 113782, Dec. 2023.

[5] Y. Lei, Y. He, X. Li, Y. Tian, X. Xiang, and C. Feng, "Experimental comparison on the performance of radiative, reflective and evaporative cooling in extremely hot climate: A case study in Chongqing, China," *Sustainable Cities and Society*, vol. 100, p. 105023, Jan. 2024.

[6] Z. Zhao, F. Xie, T. Ren, and C. Zhao, "Atmospheric CO2 retrieval from satellite spectral measurements by a two-step machine learning approach," *Journal of Quantitative Spectroscopy and Radiative Transfer*, vol. 278, p. 108006, Feb. 2022.

[7] F. Xie, T. Ren, Z. Zhao, and C. Zhao, "A machine learning based line-by-line absorption coefficient model for the application of atmospheric carbon dioxide remote sensing," *Journal of Quantitative Spectroscopy and Radiative Transfer*, p. 108441, Nov. 2022.

[8] T. E. Taylor, C. W. O'Dell, D. Baker, C. Bruegge, A. Chang, L. Chapsky, A. Chatterjee, C. Cheng, F. Chevallier, D. Crisp, L. Dang, B. Drouin, A. Eldering, L. Feng, B. Fisher, D. Fu, M. Gunson, V. Haemmerle, G. R. Keller, M. Kiel, L. Kuai, T. Kurosu, A. Lambert, J. Laughner, R. Lee, J. Liu, L. Mandrake, Y. Marchetti, G. McGarragh, A. Merrelli, R. R. Nelson, G. Osterman, F. Oyafuso, P. I. Palmer, V. H. Payne, R. Rosenberg, P. Somkuti, G. Spiers, C. To, P. O. Wennberg, S. Yu, and J. Zong, "Evaluating the consistency between OCO-2 and OCO-3 XCO_2 estimates derived from the NASA ACOS version 10 retrieval algorithm," *Atmospheric Measurement Techniques Discussions*, pp. 1–61, Feb. 2023. Publisher: Copernicus GmbH.

[9] W. Steffen and N. Koning, "Hybrid polygon and hydrodynamic nebula modeling with multi-waveband radiation transfer in astrophysics," *Astronomy and Computing*, vol. 20, pp. 87–96, July 2017.

[10] C. Wang, M. F. Modest, and B. He, "Full-spectrum k-distribution look-up table for nonhomogeneous gas–soot mixtures," *Journal of Quantitative Spectroscopy and Radiative Transfer*, vol. 176, pp. 129–136, June 2016.

[11] W. Ge, M. F. Modest, and S. P. Roy, "Development of high-order PN models for radiative heat transfer in special geometries and boundary conditions," *Journal of Quantitative Spectroscopy and Radiative Transfer*, vol. 172, pp. 98–109, Mar. 2016.

[12] C. Wang, W. Ge, M. F. Modest, and B. He, "A full-spectrum k-distribution look-up table for radiative transfer in nonhomogeneous gaseous media," *Journal of Quantitative Spectroscopy and Radiative Transfer*, vol. 168, pp. 46–56, Jan. 2016.

[13] M. F. Modest and S. Mazumder, "Solution Methods for Nongrey Extinction Coefficients," in *Radiative Heat Transfer*, pp. 657–736, Elsevier, 2022.

[14] M. F. Modest and S. Mazumder, "The Method of Spherical Harmonics (P N -Approximation)," in *Radiative Heat Transfer*, pp. 513–561, Elsevier, 2022.

[15] Y. Sun, H. Shen, S. Zheng, and L. Jiang, "A hybrid non-gray gas radiation heat transfer solver based on OpenFOAM," *Journal of Quantitative Spectroscopy and Radiative Transfer*, vol. 281, p. 108105, Apr. 2022.

[16] J. Guo, X. Li, X. Huang, Z. Liu, and C. Zheng, "A full spectrum k-distribution based weighted-sum-of-grey-gases model for oxy-fuel combustion," *International Journal of Heat and Mass Transfer*, vol. 90, pp. 218–226, Nov. 2015.

[17] W. Long, W. Liang, Z. Qi, C. Jun, and R. Tao, "Development and validation of a full-spectrum correlated k-distribution radiation model for co$_2$-h$_2$o-co-particulate matter mixtures in ansys-fluent," *Journal of Power Engineering*, 2023. Accepted. Original publication in Chinese.

[18] F. P. Incropera, D. P. DeWitt, T. L. Bergman, and A. S. Lavine, eds., *Principles of heat and mass transfer.* Hoboken, NJ: Wiley, 7. ed., international student version ed., 2013.

[19] M. F. Modest and S. Mazumder, "The Radiative Transfer Equation in Participating Media (RTE)," in *Radiative Heat Transfer*, pp. 285–309, Elsevier, 2022.

[20] T. Ren, M. F. Modest, and S. Roy, "Monte Carlo Simulation for Radiative Transfer in a High-Pressure Industrial Gas Turbine Combustion Chamber," *Journal of Engineering for Gas Turbines and Power*, vol. 140, p. 051503, May 2018.

[21] M. F. Modest and S. Mazumder, "The Monte Carlo Method for Participating Media," in *Radiative Heat Transfer*, pp. 737–773, Elsevier, 2022.

[22] J. S. Truelove, "Discrete-Ordinate Solutions of the Radiation Transport Equation," *Journal of Heat Transfer*, vol. 109, no. 4, pp. 1048–1051, 1987.

[23] W. A. Fiveland, "Discrete-Ordinates Solutions of the Radiative Transport Equation for Rectangular Enclosures," *Journal of Heat Transfer*, vol. 106, no. 4, pp. 699–706, 1984.

[24] M. F. Modest and G. Pal, "Advanced Differential Approximation Formulation of the PN Method for Radiative Transfer," in *HT2009*, (Volume 1: Heat Transfer in Energy Systems; Thermophysical Properties; Heat Transfer Equipment; Heat Transfer in Electronic Equipment), pp. 233–240, July 2009.

[25] W. Ge, C. David, M. F. Modest, R. Sankaran, and S. P. Roy, "Comparison of spherical harmonics method and discrete ordinates method for radiative transfer in a turbulent jet flame," *Journal of Quantitative Spectroscopy and Radiative Transfer*, vol. 296, p. 108459, Feb. 2023.

[26] M. F. Modest and S. Mazumder, "Radiative Properties of Molecular Gases," in *Radiative Heat Transfer*, pp. 311–399, Elsevier, 2022.

[27] H. C. Hottel, "Radiant heat transmission," *WH McAdams. Heat Transmission*, 1954. Publisher: McGraw-Hill Book Company, New York.

[28] M. K. Denison and B. W. Webb, "A Spectral Line-Based Weighted-Sum-of-Grey-Gases Model for Arbitrary RTE Solvers," *Journal of Heat Transfer*, vol. 115, pp. 1004–1012, Nov. 1993.

[29] B. W. Webb, V. P. Solovjov, and F. André, "Chapter Four - The spectral line weighted-sum-of-grey-gases (SLW) model for prediction of radiative transfer in molecular gases," in *Advances in Heat Transfer* (E. M. Sparrow, J. P. Abraham, J. M. Gorman, and W. J. Minkowycz, eds.), vol. 51 of *Advances in Heat Transfer*, pp. 207–298, Elsevier, 2019. ISSN: 0065-2717.

[30] V. P. Solovjov, D. Lemonnier, and B. W. Webb, "The SLW-1 model for efficient prediction of radiative transfer in high temperature gases," *Journal of Quantitative Spectroscopy and Radiative Transfer*, vol. 112, pp. 1205–1212, May 2011.

[31] F. André, V. Solovjov, and B. Webb, "The ω-absorption line distribution function for rank correlated SLW model prediction of radiative transfer in non-uniform gases," *Journal of Quantitative Spectroscopy and Radiative Transfer*, vol. 280, 2022.

[32] Y. Zhou, C. Wang, T. Ren, and C. Zhao, "A machine learning based full-spectrum correlated k-distribution model for nonhomogeneous gas-soot mixtures," *Journal of Quantitative Spectroscopy and Radiative Transfer*, vol. 268, p. 107628, July 2021.

[33] Y. Zhou, C. Wang, and T. Ren, "A machine learning based efficient and compact full-spectrum correlated k-distribution model," *Journal of Quantitative Spectroscopy and Radiative Transfer*, vol. 254, p. 107199, Oct. 2020.

[34] S. Zheng, R. Sui, Y. Yang, Y. Sun, H. Zhou, and Q. Lu, "An improved full-spectrum correlated-k-distribution model for non-grey radiative heat transfer in combustion gas mixtures," *International Communications in Heat and Mass Transfer*, vol. 114, p. 104566, May 2020.

[35] C. Wang, B. He, M. F. Modest, and T. Ren, "Efficient full-spectrum correlated-k-distribution look-up table," *Journal of Quantitative Spectroscopy and Radiative Transfer*, vol. 219, pp. 108–116, Nov. 2018.

[36] N. Lohmann, "JSON for Modern C++," Nov. 2023.

Study questions

How to use it:

- Which RTE solver is preferred for combustion when the case is sensitive to the computational cost?
- How to use the existing radiation model in OpenFOAM?

The theory of it:

- What is the optically thin? What is the optically thick?
- Is combustion optically thin or thick?
- When will the P1 model be inaccurate?

How it is implemented:

- Why there is no stand-alone radiative heat transfer solver in OpenFOAM?
- Why is the radiative heat transfer model coupled with the energy equation?

How to modify it:

- What is the meaning of `nonGrey` in the `nonGreyP1` model?
- What is the difference between `nonGreyP1` and `P1` in the governing equation (hint: the FSCK model)?
- Why can't we use the `mesh_.boundary()` to iterate through the boundary patches to set values for some fields?

Appendix A

Guide for this model

Before running the Allwmake script to compile the model, it is necessary to modify the last line of EXE_INC and LIB_LIBS in the options file of radiation. This modification should reflect the correct path to the source code of fsckMLPModel and the compiled library. Subsequently, execute the Allwmake script to initially compile the pure cpp library libmlpmanager.so under fsckMLPModel. Following this, compile the OpenFOAM library libmyradiationModels.so under radiation.

The process of integrating the newly developed non-grey model with the existing OpenFOAM solvers involves a few key steps. Initially, create a radiationProperties file within the constant folder of your project.

radiationProperties for the non-grey model

```
/*--------------------------------*- C++ -*----------------------------------*\
| =========                 |                                                 |
| \\      /  F ield         | OpenFOAM: The Open Source CFD Toolbox           |
|  \\    /   O peration     | Version:  v2112                                 |
|   \\  /    A nd           | Website:  www.openfoam.com                      |
|    \\/     M anipulation  |                                                 |
\*---------------------------------------------------------------------------*/
FoamFile
{
    version     2.0;
    format      ascii;
    class       dictionary;
    object      radiationProperties;
}
// * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * //

radiation on;

radiationModel  nonGreyP1;

nonGreyP1Coeffs
{
    C               C [0 0 0 0 0 0 0] 0;
}

// Number of flow iterations per radiation iteration
solverFreq 1;

absorptionEmissionModel nonGreyMeanAbsorptionEmission;

nonGreyMeanAbsorptionEmissionCoeffs
{
    nBands                  16;    // Number of bands
    EhrrCoeff                0.0;
}

scatterModel    none;
```

```

sootModel       none;

transmissivityModel none;


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

Adjust the `nBands` parameter as per the accuracy requirements of the model. The default setting is 16, but it can be configured to 4, 8, 16, or 32, depending on the desired level of precision.

For the non-grey model, a `boundaryRadiationProperties` file is also necessary. Below is an example of the `boundaryRadiationProperties` configuration for the non-grey model.

`boundaryRadiationProperties` for the non-grey model

```
/*--------------------------------*- C++ -*----------------------------------*\
| =========                 |                                                 |
| \\      /  F ield         | OpenFOAM: The Open Source CFD Toolbox           |
|  \\    /   O peration     | Version:  v2112                                 |
|   \\  /    A nd           | Website:  www.openfoam.com                      |
|    \\/     M anipulation  |                                                 |
\*---------------------------------------------------------------------------*/
FoamFile
{
    version     2.0;
    format      ascii;
    class       dictionary;
    object      boundaryRadiationProperties;
}
// * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * //

".*"
{
    type            nonGreyLookup;
    emissivity      1;
    absorptivity    1;
}


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

Additionally, it is essential to specify both the initial and boundary values for the incident radiation field, as demonstrated in the following sections.

`radiationProperties` for the non-grey model

```
/*--------------------------------*- C++ -*----------------------------------* \
| =========                 |                                                 |
| \\      /  F ield         | OpenFOAM: The Open Source CFD Toolbox           |
|  \\    /   O peration     | Version:  v2112                                 |
|   \\  /    A nd           | Website:  www.openfoam.com                      |
|    \\/     M anipulation  |                                                 |
\*---------------------------------------------------------------------------*/
FoamFile
{
    version     2.0;
    format      ascii;
    class       volScalarField;
    object      G;
}
// * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * //

dimensions      [1 0 -3 0 0 0 0];

internalField   uniform 0;

boundaryField
{
    ".*"
    {
        type            MarshakRadiation;
        T               T;
        value           uniform 0;
    }

    frontAndBack
    {
        type            empty;
    }
}


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

The recommended solver settings for managing the incident radiation are largely in line with those provided in the standard OpenFOAM tutorials, as detailed below.

`fvSolution` for the non-grey model

```
    "(G.*)"
    {
        solver          PCG;
        preconditioner  DIC;
        tolerance       1e-9;
        relTol          0.0;
    }
```

This model is designed to be compatible with any OpenFOAM solver that includes a predefined energy equation. Don't forget to include the library in your `controlDict` file.

Appendix B

Validation cases

Several cases are set up to validate the newly implemented FSCK model with the results run by two existing codes, as mentioned in Chapter 5. They are also handed in as support files. There are five folders.

1. `1d_homo`: 1D homogeneous case
2. `1d_nonhomo`: 1D nonhomogeneous case
3. `2d_homo`: 2D homogeneous case
4. `2d_nonhomo`: 2D nonhomogeneous case
5. `validation_data`: Results run by the existing codes

Two Python scripts, `plot.py` and `contour.py`, are also given to plot the results. You may need `numpy`, `matplotlib`, and `pandas` to run these scripts. A proper `conda` environment is recommended.

To run the simulation and plot the results, you just need to execute `Allrun` script. The plotted images are stored under the `img` folder. You can also use the `Allclean` script to clean the case.

Appendix C

Developed codes

C.1 FSCK model

C.1.1 MLPManager class

MLPManager.h

```cpp
#ifndef MLPMANAGER_H
#define MLPMANAGER_H

#include "mlp.h"
#include <vector>
#include <memory>

class MLPManager {
public:
    MLPManager(); // Constructor that loads all MLPs
    std::vector<double> get_prediction(const std::vector<double>& input); // Public method to get
     predictions

private:
    std::vector<std::unique_ptr<MLP>> mlps; // Vector of unique_ptr to MLPs
    void load_all_mlps(std::vector<std::string> mlp_names); // Private method to load all MLPs
};

#endif // MLPMANAGER_H
```

MLPManager.cpp

```cpp
#include "MLPManager.h"
#include <stdexcept>
#include <dlfcn.h>
#include <string>

std::string GetLibraryPath() {
    Dl_info dl_info;
    dladdr((void*)&GetLibraryPath, &dl_info);
    std::string fullPath(dl_info.dli_fname);

    size_t pos = fullPath.find_last_of('/');
    if (pos != std::string::npos) {
        return fullPath.substr(0, pos);
    } else {
        return fullPath;
    }
}

// Constructor implementation
MLPManager::MLPManager() {
    std::string dl_path = GetLibraryPath();
    std::vector<std::string> mlp_names =
                {dl_path + "/../../parameter/mlp_0.json",
                dl_path + "/../../parameter/mlp_1.json",
                dl_path + "/../../parameter/mlp_2.json"};
    load_all_mlps(mlp_names); // Load all MLPs when the object is created
}

// load_all_mlps method implementation
void MLPManager::load_all_mlps(std::vector<std::string> mlp_names) {
    for(int i = 0; i < 3; i++){
        std::cout << "Loading MLP parameter from: " << mlp_names[i] << std::endl;
        mlps.emplace_back(std::unique_ptr<MLP>(new MLP(mlp_names[i])));
    }
}

// get_prediction method implementation
std::vector<double> MLPManager::get_prediction(const std::vector<double>& input) {
    if (input.empty()) {
        throw std::invalid_argument("Input vector is empty.");
    }

    double value = input[0];
    MLP* mlp = nullptr;

    if (value >= 0.0 && value < 1.0) {
        mlp = mlps[0].get();
    } else if (value >= 1.0 && value < 10.0) {
        mlp = mlps[1].get();
    } else if (value >= 10.0 && value < 80.0) {
        mlp = mlps[2].get();
    } else {
        throw std::out_of_range("Input[0] is out of the expected range.");
    }

    return mlp->predict(input);
}
```

C.1.2 MLP class

mlp.h

```
#ifndef MLP_H
#define MLP_H

#include <iostream>
#include <fstream>
#include <vector>
#include <string>
#include "json.hpp"

using json = nlohmann::json;
using std::vector;

class MLP {
private:
    struct Layer {
        vector<vector<double>> weights;
        vector<double> bias;
        std::string activation;
        int units;
    };

    vector<Layer> layers;
    vector<double> input_mean;
    vector<double> input_std;
    vector<double> output_mean;
    vector<double> output_std;

    // Activation function
    inline double relu(double x) {
        return x > 0 ? x : 0;
    };

    // Forward pass
    vector<double> forward(const vector<double>& input);

    // Function to normalize input
    vector<double> normalize_input(const vector<double>& input);

    // Function to denormalize output
    vector<double> denormalize_output(const vector<double>& output);

    // Preprocessing
    vector<double> preprocess(const vector<double>& input);

    // Postprocessing
    vector<double> postprocess(const vector<double>& output);

public:
    MLP(const std::string& json_file);

    // Predict function
    vector<double> predict(const vector<double>& input);
};

#endif
```

mlp.cpp

```
#include <iostream>
#include <fstream>
#include <vector>
#include <string>
#include "json.hpp"
#include "mlp.h"

using json = nlohmann::json;
using std::vector;

vector<double> MLP::forward(const vector<double>& input) {
    vector<double> a = input;
    for (auto& layer : layers) {
        vector<double> z(layer.units, 0.0);
        for (int i = 0; i < layer.units; ++i) {
            for (size_t j = 0; j < a.size(); ++j) {
                z[i] += a[j] * layer.weights[j][i];
            }
            z[i] += layer.bias[i];
            if (layer.activation == "relu") {
                z[i] = relu(z[i]);
            }
        }
        a = z;
    }
    return a;
}

// Normalized input
vector<double> MLP::normalize_input(const vector<double>& input) {
    vector<double> normalized(input.size());
    for (size_t i = 0; i < input.size(); ++i) {
        normalized[i] = (input[i] - input_mean[i]) / input_std[i];
    }
    return normalized;
}

// Denormalize output
vector<double> MLP::denormalize_output(const vector<double>& output) {
    vector<double> denormalized(output.size());
    for (size_t i = 0; i < output.size(); ++i) {
        denormalized[i] = output[i] * output_std[i] + output_mean[i];
    }
    return denormalized;
}

// Preprocessing
vector<double> MLP::preprocess(const vector<double>& input) {
    vector<double> preprocessed(input.size());
    // perform log10 on the last element
    preprocessed = input;
    if(input.back()==0.0) {
        preprocessed.back() = -10.0;
    } else {
        preprocessed.back() = log10(input.back());
    }
    return preprocessed;
}

// Postprocessing
vector<double> MLP::postprocess(const vector<double>& output) {
    vector<double> postprocessed(output.size());
    // perform 10^x on each element
    for (size_t i = 0; i < output.size(); ++i) {
        postprocessed[i] = pow(10, output[i]);
    }
    return postprocessed;
}

// Constructor
MLP::MLP(const std::string& json_file) {
    std::ifstream file(json_file);
    if (!file.is_open()) {
        throw std::runtime_error("Unable to open JSON file.");
    }

    json j;
    file >> j;

    int n_layers = j["n_layers"];
    for (int i = 0; i < n_layers; ++i) {
        std::string layer_key = "layer_" + std::to_string(i);
        auto& json_layer = j[layer_key];

        Layer layer;
        layer.units = json_layer["units"];
        layer.activation = json_layer["activation"];

        // Weights
        auto& json_weights = json_layer["weights"];
        for (auto& w : json_weights) {
            layer.weights.push_back(w.get<vector<double>>());
        }

        // Bias
        layer.bias = json_layer["bias"].get<vector<double>>();

        layers.push_back(layer);
    }

    // Normalization parameters
    input_mean = j["input_mean"].get<vector<double>>();
    input_std = j["input_std"].get<vector<double>>();
    output_mean = j["output_mean"].get<vector<double>>();
    output_std = j["output_std"].get<vector<double>>();
}

// Predict function
vector<double> MLP::predict(const vector<double>& input) {
    auto preprocessed = preprocess(input);
    auto normalized = normalize_input(preprocessed);
    auto raw_output = forward(normalized);
    auto denomalized = denormalize_output(raw_output);
    return postprocess(denomalized);
}
```

C.1.3 fsckMLP class

fsckMLP.H

```
/*---------------------------------------------------------------------------*\
I will use singleton pattern to make sure that only one instance of the class
is created. Some common information like reference state can be shared among
RTE solver (P1), absorptionEmissionModel and boundary radiation properties.
\*---------------------------------------------------------------------------*/

#ifndef fsckMLP_H
#define fsckMLP_H

#include "volFields.H"
#include "fluidThermo.H"
#include "MLPManager.h"

namespace Foam
{
namespace radiation
{
class fsckMLP
{
private:
        static fsckMLP* instance;

        // number of selected bands
        label nBands_;

        // reference state
        scalar Tref_;
        scalar xco2_ref_;
        scalar xh2o_ref_;
        scalar xco_ref_;

        const fvMesh& mesh_;

        //- SLG thermo package
        const fluidThermo& thermo_;

        //- list of absorption ceofficient k
        PtrList<volScalarField> ki_;

        //- list of a
        PtrList<volScalarField> ai_;
        DimensionedField<scalar, volMesh> writeRu_;
        volScalarField writeRp_;
        label bandI_;           // Just for Marshak

        double* gNqDB;
        double* wNqDB;
        double* gNq;
        double* wNq;

        MLPManager mlpManager_;

        fsckMLP(label nBands, const fvMesh& mesh);

        void initKA();
public:
        static fsckMLP* getInstance();
        static fsckMLP* getInstance(label nBands, const fvMesh& mesh);
        label getNBands();
        void updateTref();
        void updateKA();
        tmp<Foam::volScalarField> get_k(const label bandI);
        tmp<Foam::volScalarField> get_a(const label bandI);
        scalar getTref();
        double* getwNq();
        void updateRu(const DimensionedField<scalar, volMesh>& Ru);
        void updateRp(const volScalarField Rp);

        void setBandI(const label bandI);
        label getBandI();
};

} // namespace radiation
} // namespace Foam

#endif
```

fsckMLP.C

```
#include "fsckMLP.H"
#include "MLPManager.h"
#include "kpl.h"
#include "basicSpecieMixture.H"
#include "extrapolatedCalculatedFvPatchFields.H"
#include "support_func.h"

Foam::radiation::fsckMLP* Foam::radiation::fsckMLP::instance = nullptr;

Foam::radiation::fsckMLP::fsckMLP(label nBands, const fvMesh& mesh)
:
    nBands_(nBands),
    mesh_(mesh),
    thermo_(mesh.lookupObject<fluidThermo>(basicThermo::dictName)),
    writeRu_
    (
        IOobject
        (
            "debug_Ru",
            mesh_.time().timeName(),
            mesh_,
            IOobject::NO_READ,
            IOobject::AUTO_WRITE
        ),
        mesh_,
        dimensionedScalar(dimMass/dimLength/pow3(dimTime), Zero)
    ),

    writeRp_
    (
        IOobject
        (
            "debug_Rp",
            mesh_.time().timeName(),
            mesh_,
            IOobject::NO_READ,
            IOobject::AUTO_WRITE
        ),
        mesh_,
        dimensionedScalar(dimMass/dimLength/pow3(dimTime)/pow4(dimTemperature), Zero)
    ),
    bandI_(0),
    mlpManager_()
{
    quadgen2(NqDB, 2.0, &gNqDB, &wNqDB);
    quadgen2(nBands, 2.0, &gNq, &wNq);
    Info << "kdist Information" << endl;
    Info << "gNq :";
    for(int i = 0; i < nBands; i++)
    {
        Info << gNq[i] << "\t";
    }
    Info << endl << "wNq :";
    for(int i = 0; i < nBands; i++)
    {
        Info << wNq[i] << "\t";
    }
    Info << endl;
    initKA();
    Tref_ = 1000.0;
    xco2_ref_ = 0.0;
    xh2o_ref_ = 0.0;
    xco_ref_ = 0.0;
}

void Foam::radiation::fsckMLP::initKA()
{
    ki_.setSize(nBands_);
    ai_.setSize(nBands_);
    for (label i=0; i<nBands_; i++)
    {
        Info << "Initialzing k and a for band" << i << endl;
        ki_.set(i, new volScalarField
        (
            IOobject
            (
                "k" + std::to_string(i),
                mesh_.time().timeName(),
                mesh_,
                IOobject::NO_READ,
                IOobject::AUTO_WRITE
            ),
            mesh_,
            dimensionedScalar(dimless/dimLength, ROOTVSMALL)
        ));
        ai_.set(i, new volScalarField
        (
            IOobject
            (
                "a" + std::to_string(i),
                mesh_.time().timeName(),
                mesh_,
                IOobject::NO_READ,
                IOobject::AUTO_WRITE
            ),
            mesh_,
            dimensionedScalar(dimless/dimLength, ROOTVSMALL)
        ));
    }
}

Foam::radiation::fsckMLP* Foam::radiation::fsckMLP::getInstance()
{
    return instance;
}
```

```
Foam::radiation::fsckMLP* Foam::radiation::fsckMLP::getInstance(label nBands, const fvMesh& mesh)
{
    if (instance == nullptr)
    {
        instance = new fsckMLP(nBands, mesh);
    }
    return instance;
}

Foam::label Foam::radiation::fsckMLP::getNBands()
{
    return nBands_;
}

void Foam::radiation::fsckMLP::updateTref()
{
    const basicSpecieMixture& mixture =
        dynamic_cast<const basicSpecieMixture&>(thermo_);

    const volScalarField& T = thermo_.T();

    // Calculate the reference temperature field
    scalar volume = 0.0;

    scalar xco2 = 0.0;
    scalar xh2o = 0.0;
    scalar xco = 0.0;
    scalar Tmax = 0.0;
    scalar Tmin = 5000.0;
    scalar rhs = 0.0;

    forAll(T, celli)
    {
        scalar invWt = 0.0;
        scalar xco2_cell, xh2o_cell, xco_cell;
        forAll(mixture.Y(), s)
        {
            invWt += mixture.Y(s)[celli]/mixture.W(s);
        }
        xco2_cell = mixture.Y("CO2")[celli]/(mixture.W(mixture.species()["CO2"])*invWt);
        xh2o_cell = mixture.Y("H2O")[celli]/(mixture.W(mixture.species()["H2O"])*invWt);
        xco_cell = mixture.Y("CO")[celli]/(mixture.W(mixture.species()["CO"])*invWt);

        xco2 = xco2 + xco2_cell * mesh_.cellVolumes()[celli];
        xh2o = xh2o + xh2o_cell * mesh_.cellVolumes()[celli];
        xco = xco + xco_cell * mesh_.cellVolumes()[celli];

        // Begin the reference temperature part
        scalar Ti = T[celli];
        int f;
        if (Ti < kpT[0])
            f = 0;
        else if (Ti > kpT[125])
            f = 124;
        else
            f = static_cast<int>((Ti - 300) / 20);
        double wx = (Ti - kpT[f]) / 20;

        double kp[4] = {0.0};

        kp[0] = (wx * kpCO2[f + 1] + (1. - wx) * kpCO2[f]) * xco2_cell;
        kp[1] = (wx * kpH2O[f + 1] + (1. - wx) * kpH2O[f]) * xh2o_cell;
        kp[2] = (wx * kpCO[f + 1] + (1. - wx) * kpCO[f]) * xco_cell;
        kp[3] = (wx * kpSoot[f + 1] + (1. - wx) * kpSoot[f]) * 0.0;


        for (int i = 0; i < 4; i++)
        {
            rhs += mesh_.cellVolumes()[celli] * kp[i] * pow(Ti, 4.);
        }

        Tmax = max(Tmax, Ti);
        Tmin = min(Tmin, Ti);
        volume = volume + mesh_.cellVolumes()[celli];
    }

    reduce(xco2, sumOp<scalar>());
    reduce(xh2o, sumOp<scalar>());
    reduce(xco, sumOp<scalar>());
    reduce(volume, sumOp<scalar>());
    reduce(rhs, sumOp<scalar>());
    reduce(Tmax, maxOp<scalar>());
    reduce(Tmin, minOp<scalar>());
    xh2o = xh2o / volume;
    xco2 = xco2 / volume;
    xco = xco / volume;
    rhs = rhs / volume;

    Info << "Tmin, Tmax = " << Tmin << "\t" << Tmax << endl;


    // Set the integrated Tref value to the fsckMLP object
    Foam::radiation::fsckMLP* fsck = Foam::radiation::fsckMLP::getInstance();
    double xmref[4] = {xco2, xh2o, xco, 0.0};

    double Tref = 200.0;
    double kpref[4][126], kpdata[126];
    for (int i = 0; i < 126; i++)
    {
        kpref[0][i] = xmref[0] * kpCO2[i];
        kpref[1][i] = xmref[1] * kpH2O[i];
        kpref[2][i] = xmref[2] * kpCO[i];
        kpref[3][i] = xmref[3] * kpSoot[i];
    }
    int imax = round((Tmax - 300) / 20 + 1);
    int imin = max(2, round((Tmin - 300) / 20 + 1));
    int ibgn = min(126, max(2, imin - 10)) - 1;
    int iend = min(126, max(2, imax + 10)) - 1;
    double lhs[126] = { 0. };
    for (int id = ibgn; id < iend; id++)
    {
        for (int speciesid = 0; speciesid < 4; speciesid++)
            lhs[id] = lhs[id] + kpref[speciesid][id] * pow(kpT[id], 4.);
        if (lhs[id] >= rhs)
        {
            Tref = kpT[id - 1] + (kpT[id] - kpT[id - 1]) * (rhs - lhs[id - 1]) / (lhs[id] - lhs[id -
     1] + 1e-25);
            break;
        }
    }
    Tref_ = Tref;
    xco2_ref_ = xco2;
    xh2o_ref_ = xh2o;
    xco_ref_ = xco;
    Info << "Newly calculated Tref: " << Tref << endl <<
            "xco2: " << xco2 << endl <<
            "xh2o: " << xh2o << endl <<
            "xco: " << xco << endl;
}

void Foam::radiation::fsckMLP::updateKA()
{
    const basicSpecieMixture& mixture =
    dynamic_cast<const basicSpecieMixture&>(thermo_);
    const volScalarField& T = thermo_.T();
    const volScalarField& p = thermo_.p();

    forAll(T, celli)
    {
        scalar invWt = 0.0;
        scalar xco2_cell, xh2o_cell, xco_cell;
        forAll(mixture.Y(), s)
        {
            invWt += mixture.Y(s)[celli]/mixture.W(s);
        }
        xco2_cell = mixture.Y("CO2")[celli]/(mixture.W(mixture.species()["CO2"])*invWt);
        xh2o_cell = mixture.Y("H2O")[celli]/(mixture.W(mixture.species()["H2O"])*invWt);
        xco_cell = mixture.Y("CO")[celli]/(mixture.W(mixture.species()["CO"])*invWt);

        std::vector<double> input_Tref = {p[celli]/1e5, Tref_, T[celli], xco2_cell, xh2o_cell,
     xco_cell, 1e-10};
        std::vector<double> k_Tref = mlpManager_.get_prediction(input_Tref);
        std::vector<double> input_T = {p[celli]/1e5, T[celli], T[celli], xco2_cell, xh2o_cell,
     xco_cell, 1e-10};
        std::vector<double> k_T = mlpManager_.get_prediction(input_T);

        double a[NqDB];
        afun(gNqDB, k_Tref.data(), k_T.data(), wNqDB, a);
        double k_new[nBands_], a_new[nBands_];
        simple_interp(NqDB, nBands_, gNqDB, k_Tref.data(), gNq, k_new);
        simple_interp(NqDB, nBands_, gNqDB, a, gNq, a_new);

        forAll(ki_, bandI)
        {
            ki_[bandI][celli] = k_new[bandI]*100.0;
            ai_[bandI][celli] = a_new[bandI];
        }
    }
    forAll(T.boundaryField(), patchi)
    {
        forAll (T.boundaryField()[patchi], facei)
        {
            scalar invWt = 0.0;
            scalar xco2_cell, xh2o_cell, xco_cell;
            forAll(mixture.Y(), s)
            {
                invWt += mixture.Y(s).boundaryField()[patchi][facei]/mixture.W(s);
            }
            xco2_cell = mixture.Y("CO2").boundaryField()[patchi][facei]/(mixture.W(mixture.species()["
     CO2"])*invWt);
            xh2o_cell = mixture.Y("H2O").boundaryField()[patchi][facei]/(mixture.W(mixture.species()["
     H2O"])*invWt);
            xco_cell = mixture.Y("CO").boundaryField()[patchi][facei]/(mixture.W(mixture.species()["CO
     "])*invWt);

            std::vector<double> input_Tref = {p.boundaryField()[patchi][facei]/1e5, Tref_, T.
     boundaryField()[patchi][facei], xco2_cell, xh2o_cell, xco_cell, 1e-10};
            std::vector<double> k_Tref = mlpManager_.get_prediction(input_Tref);
            std::vector<double> input_T = {p.boundaryField()[patchi][facei]/1e5, T.boundaryField()[
     patchi][facei], T.boundaryField()[patchi][facei], xco2_cell, xh2o_cell, xco_cell, 1e-10};
            std::vector<double> k_T = mlpManager_.get_prediction(input_T);

            double a[NqDB];
            afun(gNqDB, k_Tref.data(), k_T.data(), wNqDB, a);
            double k_new[nBands_], a_new[nBands_];
            simple_interp(NqDB, nBands_, gNqDB, k_Tref.data(), gNq, k_new);
            simple_interp(NqDB, nBands_, gNqDB, a, gNq, a_new);

            forAll(ai_, bandI)
            {
                ai_[bandI].boundaryFieldRef()[patchi][facei] = a_new[bandI];
                ki_[bandI].boundaryFieldRef()[patchi][facei] = k_new[bandI]*100.0;
            }
        }
    }
}

Foam::tmp<Foam::volScalarField> Foam::radiation::fsckMLP::get_k(const label bandI)
{
    ki_[bandI].correctBoundaryConditions();
    return ki_[bandI];
}

Foam::tmp<Foam::volScalarField> Foam::radiation::fsckMLP::get_a(const label bandI)
{
    ai_[bandI].correctBoundaryConditions();
    return ai_[bandI];
}

Foam::scalar Foam::radiation::fsckMLP::getTref()
{
    return Tref_;
}

double* Foam::radiation::fsckMLP::getwNq()
{
    return wNq;
}

void Foam::radiation::fsckMLP::updateRu(const DimensionedField<scalar, volMesh>& Ru)
{
    writeRu_ = Ru;
}

void Foam::radiation::fsckMLP::updateRp(const volScalarField Rp)
{
    writeRp_ = Rp;
}

void Foam::radiation::fsckMLP::setBandI(const label bandI)
{
    bandI_ = bandI;
}

Foam::label Foam::radiation::fsckMLP::getBandI()
{
    return bandI_;
}
```

C.1.4 nonGreyMeanAbsorptionEmission class

nonGreyMeanAbsorptionEmission.H

```
/*---------------------------------------------------------------------------*\
  =========                 |
  \\      /  F ield         | OpenFOAM: The Open Source CFD Toolbox
   \\    /   O peration     |
    \\  /    A nd           | www.openfoam.com
     \\/     M anipulation  |
-------------------------------------------------------------------------------
    Copyright (C) 2011-2016 OpenFOAM Foundation
-------------------------------------------------------------------------------
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::radiation::nonGreyMeanAbsorptionEmission

Group
    grpRadiationAbsorptionEmissionSubModels

Description
    nonGreyMeanAbsorptionEmission radiation absorption and emission coefficients
    for continuous phase

    The coefficients for the species in the Look up table have to be specified
    for use in moles x P [atm], i.e. (k[i] = species[i]*p*9.869231e-6).

    The coefficients for CO and soot or any other added are multiplied by the
    respective mass fraction being solved

    All the species in the dictionary need either to be in the look-up table or
    being solved. Conversely, all the species solved do not need to be included
    in the calculation of the absorption coefficient

    The names of the species in the absorption dictionary must match exactly the
    name in the look-up table or the name of the field being solved

    The look-up table ("speciesTable") file should be in constant

    i.e. dictionary
    \verbatim
        LookUpTableFileName     "speciesTable";

        EhrrCoeff       0.0;

        CO2
        {
            Tcommon     300.;   // Common Temp
            invTemp     true;   // Is the polynomial using inverse temperature?
            Tlow        300.;   // Low Temp
            Thigh       2500.;  // High Temp

            loTcoeffs           // coeffs for T < Tcommon
            (
                0               //  a0            +
                0               //  a1*T          +
                0               //  a2*T^(+/-)2   +
                0               //  a3*T^(+/-)3   +
                0               //  a4*T^(+/-)4   +
                0               //  a5*T^(+/-)5   +
            );
            hiTcoeffs           // coeffs for T > Tcommon
            (
                18.741
                -121.31e3
                273.5e6
                -194.05e9
                56.31e12
                -5.8169e15
            );
        }
    \endverbatim
```

```
SourceFiles
    nonGreyMeanAbsorptionEmission.C

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

#ifndef nonGreyMeanAbsorptionEmission_H
#define nonGreyMeanAbsorptionEmission_H

#include "interpolationLookUpTable.H"
#include "absorptionEmissionModel.H"
#include "HashTable.H"
#include "absorptionCoeffs.H"
#include "fluidThermo.H"

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

namespace Foam
{
namespace radiation
{

/*---------------------------------------------------------------------------*\
                 Class nonGreyMeanAbsorptionEmission Declaration
\*---------------------------------------------------------------------------*/

class nonGreyMeanAbsorptionEmission
:
    public absorptionEmissionModel
{
public:

    // Public data

        // Maximum number of species considered for absorptivity
        static const int nSpecies_ = 5;

        // Absorption Coefficients
        // absorptionCoeffs coeffs_[nSpecies_];


private:

    // Private data

        //- Absorption model dictionary
        dictionary coeffsDict_;
        label nBands_;

        //- SLG thermo package
        const fluidThermo& thermo_;

        //- Emission constant coefficient
        const scalar EhrrCoeff_;

public:

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


    // Constructors

        //- Construct from components
        nonGreyMeanAbsorptionEmission(const dictionary& dict, const fvMesh& mesh);


    //- Destructor
    virtual ~nonGreyMeanAbsorptionEmission();


    // Member Functions

        // Access

            // Absorption coefficient

                //- Absorption coefficient for continuous phase
                tmp<volScalarField> aCont(const label bandI = 0) const;


            // Emission coefficient

                //- Emission coefficient for continuous phase
                tmp<volScalarField> eCont(const label bandI = 0) const;


            // Emission contribution

                //- Emission contribution for continuous phase
                tmp<volScalarField> ECont(const label bandI = 0) const;

            //- Get reference temperature

            void correct
            (
                volScalarField& a,
                PtrList<volScalarField>& aj
            ) const;


    // Member Functions

        inline bool isGrey() const
        {
            return false;
        }

        inline label nBands() const
        {
            return nBands_;
        }
};


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

} // End namespace radiation
} // End namespace Foam

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

#endif

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

nonGreyMeanAbsorptionEmission.C

```
/*---------------------------------------------------------------------------*\
  =========                 |
  \\      /  F ield         | OpenFOAM: The Open Source CFD Toolbox
   \\    /   O peration     |
    \\  /    A nd           | www.openfoam.com
     \\/     M anipulation  |
-------------------------------------------------------------------------------
    Copyright (C) 2011-2017 OpenFOAM Foundation
    Copyright (C) 2020 OpenCFD Ltd.
-------------------------------------------------------------------------------
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 "nonGreyMeanAbsorptionEmission.H"
#include "addToRunTimeSelectionTable.H"
#include "unitConversion.H"
#include "extrapolatedCalculatedFvPatchFields.H"
#include "basicSpecieMixture.H"
#include "fsckMLP.H"
#include "MLPManager.h"
#include "kpl.h"

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

namespace Foam
{
    namespace radiation
    {
        defineTypeNameAndDebug(nonGreyMeanAbsorptionEmission, 0);

        addToRunTimeSelectionTable
        (
            absorptionEmissionModel,
            nonGreyMeanAbsorptionEmission,
            dictionary
        );
    }
}


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

Foam::radiation::nonGreyMeanAbsorptionEmission::nonGreyMeanAbsorptionEmission
(
    const dictionary& dict,
    const fvMesh& mesh
)
:
    absorptionEmissionModel(dict, mesh),
    coeffsDict_((dict.optionalSubDict(typeName + "Coeffs"))),
    nBands_(coeffsDict_.get<label>("nBands")),
    thermo_(mesh.lookupObject<fluidThermo>(basicThermo::dictName)),
    EhrrCoeff_(coeffsDict_.get<scalar>("EhrrCoeff"))
{
    if (!isA<basicSpecieMixture>(thermo_))
    {
        FatalErrorInFunction
            << "Model requires a multi-component thermo package"
            << abort(FatalError);
    }
    Foam::radiation::fsckMLP* fsck = Foam::radiation::fsckMLP::getInstance(nBands_, thermo_.T().mesh()
    );
}

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

Foam::radiation::nonGreyMeanAbsorptionEmission::~nonGreyMeanAbsorptionEmission()
{}


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

Foam::tmp<Foam::volScalarField>
Foam::radiation::nonGreyMeanAbsorptionEmission::aCont(const label bandI) const
{
    Foam::radiation::fsckMLP* fsck = Foam::radiation::fsckMLP::getInstance();
    return fsck->get_k(bandI);
}


Foam::tmp<Foam::volScalarField>
Foam::radiation::nonGreyMeanAbsorptionEmission::eCont(const label bandI) const
{
    Foam::radiation::fsckMLP* fsck = Foam::radiation::fsckMLP::getInstance();
    return fsck->get_a(bandI);
}


Foam::tmp<Foam::volScalarField>
Foam::radiation::nonGreyMeanAbsorptionEmission::ECont(const label bandI) const
{
    tmp<volScalarField> E
    (
        new volScalarField
        (
            IOobject
            (
                "ECont" + name(bandI),
                mesh_.time().timeName(),
                mesh_,
                IOobject::NO_READ,
                IOobject::NO_WRITE
            ),
            mesh_,
            dimensionedScalar(dimMass/dimLength/pow3(dimTime), Zero)
        )
    );

    const volScalarField* QdotPtr = mesh_.findObject<volScalarField>("Qdot");

    if (QdotPtr)
    {
        const volScalarField& Qdot = *QdotPtr;

        if (Qdot.dimensions() == dimEnergy/dimTime)
        {
            E.ref().primitiveFieldRef() = EhrrCoeff_*Qdot/mesh_.V();
        }
        else if (Qdot.dimensions() == dimEnergy/dimTime/dimVolume)
        {
            E.ref().primitiveFieldRef() = EhrrCoeff_*Qdot;
        }
        else
        {
            if (debug)
            {
                WarningInFunction
                    << "Incompatible dimensions for Qdot field" << endl;
            }
        }
    }
    else
    {
        // WarningInFunction
        //   << "Qdot field not found in mesh" << endl;
    }

    return E;
}

void Foam::radiation::nonGreyMeanAbsorptionEmission::correct
    (
        volScalarField& a,
        PtrList<volScalarField>& aj
    ) const
{
    Foam::radiation::fsckMLP* fsck = Foam::radiation::fsckMLP::getInstance();
    fsck->updateTref();
    fsck->updateKA();
}




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

C.2 Nongrey P1 solver

C.2.1 nonGreyP1 class

nonGreyP1.H

```
/*---------------------------------------------------------------------------*\
  =========                 |
  \\      /  F ield         | OpenFOAM: The Open Source CFD Toolbox
   \\    /   O peration     |
    \\  /    A nd           | www.openfoam.com
     \\/     M anipulation  |
-------------------------------------------------------------------------------
    Copyright (C) 2011-2017 OpenFOAM Foundation
-------------------------------------------------------------------------------
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::radiation::nonGreyP1

Group
    grpRadiationModels

Description
```

```
    Works well for combustion applications where optical thickness, tau is
    large, i.e. tau = a*L > 3 (L = distance between objects)

    Assumes
     - all surfaces are diffuse
     - tends to over predict radiative fluxes from sources/sinks
       *** SOURCES NOT CURRENTLY INCLUDED ***

SourceFiles
    nonGreyP1.C

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

#ifndef radiation_nonGreyP1_H
#define radiation_nonGreyP1_H

#include "radiationModel.H"
#include "volFields.H"

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

namespace Foam
{
namespace radiation
{

/*---------------------------------------------------------------------------*\
                           Class nonGreyP1 Declaration
\*---------------------------------------------------------------------------*/

class nonGreyP1
:
    public radiationModel
{
    // Private data

        //- Incident radiation / [W/m2]
        volScalarField G_;

        //- Total radiative heat flux [W/m2]
        volScalarField qr_;

        //- Absorption coefficient
        volScalarField a_;

        //- Emission coefficient
        volScalarField e_;

        //- Emission contribution
        volScalarField E_;

        //- internal calculation Gg_
        PtrList<volScalarField> Gg_;


    // Private Member Functions

        //- No copy construct
        nonGreyP1(const nonGreyP1&) = delete;

        //- No copy assignment
        void operator=(const nonGreyP1&) = delete;

        void initGg();

public:

    //- Runtime type information
    TypeName("nonGreyP1");


    // Constructors

        //- Construct from components
        nonGreyP1(const volScalarField& T);

        //- Construct from components
        nonGreyP1(const dictionary& dict, const volScalarField& T);


    //- Destructor
    virtual ~nonGreyP1() = default;


    // Member functions

        // Edit

            //- Solve radiation equation(s)
            void calculate();

            //- Read radiation properties dictionary
            bool read();


        // Access

            //- Source term component (for power of T^4)
            virtual tmp<volScalarField> Rp() const;

            //- Source term component (constant)
            virtual tmp<volScalarField::Internal> Ru() const;

            //- Number of bands
            label nBands() const;
};


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

} // End namespace radiation
} // End namespace Foam

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

#endif

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

nonGreyP1.C

```
/*---------------------------------------------------------------------------*\
  =========                 |
  \\      /  F ield         | OpenFOAM: The Open Source CFD Toolbox
   \\    /   O peration     |
    \\  /    A nd           | www.openfoam.com
     \\/     M anipulation  |
-------------------------------------------------------------------------------
    Copyright (C) 2011-2017 OpenFOAM Foundation
    Copyright (C) 2019-2020 OpenCFD Ltd.
-------------------------------------------------------------------------------
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 "nonGreyP1.H"
#include "fvmLaplacian.H"
#include "fvmSup.H"
#include "absorptionEmissionModel.H"
#include "extrapolatedCalculatedFvPatchFields.H"
#include "scatterModel.H"
#include "constants.H"
#include "addToRunTimeSelectionTable.H"
#include "fsckMLP.H"

using namespace Foam::constant;

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

namespace Foam
{
    namespace radiation
    {
        defineTypeNameAndDebug(nonGreyP1, 0);
        addToRadiationRunTimeSelectionTables(nonGreyP1);
    }
}


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

Foam::radiation::nonGreyP1::nonGreyP1(const volScalarField& T)
:
    radiationModel(typeName, T),
    G_
    (
        IOobject
        (
            "G",
            mesh_.time().timeName(),
            mesh_,
            IOobject::MUST_READ,
            IOobject::AUTO_WRITE
        ),
        mesh_
    ),
    qr_
    (
        IOobject
        (
            "qr",
            mesh_.time().timeName(),
            mesh_,
            IOobject::READ_IF_PRESENT,
            IOobject::AUTO_WRITE
        ),
        mesh_,
        dimensionedScalar(dimMass/pow3(dimTime), Zero)
    ),
    a_
    (
        IOobject
        (
            "a",
            mesh_.time().timeName(),
            mesh_,
            IOobject::NO_READ,
            IOobject::AUTO_WRITE
        ),
        mesh_,
        dimensionedScalar(dimless/dimLength, Zero)
    ),
    e_
    (
        IOobject
        (
            "e",
            mesh_.time().timeName(),
            mesh_,
            IOobject::NO_READ,
            IOobject::NO_WRITE
        ),
        mesh_,
        dimensionedScalar(dimless/dimLength, Zero)
    ),
    E_
    (
        IOobject
        (
            "E",
            mesh_.time().timeName(),
            mesh_,
            IOobject::NO_READ,
            IOobject::NO_WRITE
        ),
        mesh_,
        dimensionedScalar(dimMass/dimLength/pow3(dimTime), Zero)
    ),
    Gg_ (absorptionEmission_->nBands())
{
    initGg();
}


Foam::radiation::nonGreyP1::nonGreyP1(const dictionary& dict, const volScalarField& T)
:
    radiationModel(typeName, dict, T),
    G_
    (
        IOobject
        (
            "G",
            mesh_.time().timeName(),
            mesh_,
            IOobject::MUST_READ,
            IOobject::AUTO_WRITE
        ),
        mesh_
    ),
    qr_
    (
        IOobject
        (
            "qr",
            mesh_.time().timeName(),
            mesh_,
            IOobject::READ_IF_PRESENT,
            IOobject::AUTO_WRITE
        ),
        mesh_,
        dimensionedScalar(dimMass/pow3(dimTime), Zero)
    ),
    a_
    (
        IOobject
        (
            "a",
            mesh_.time().timeName(),
            mesh_,
            IOobject::NO_READ,
            IOobject::AUTO_WRITE
        ),
        mesh_,
        dimensionedScalar(dimless/dimLength, Zero)
    ),
    e_
    (
        IOobject
        (
            "e",
            mesh_.time().timeName(),
            mesh_,
            IOobject::NO_READ,
            IOobject::NO_WRITE
        ),
        mesh_,
        dimensionedScalar(dimless/dimLength, Zero)
    ),
    E_
    (
        IOobject
        (
            "E",
            mesh_.time().timeName(),
            mesh_,
            IOobject::NO_READ,
            IOobject::NO_WRITE
        ),
        mesh_,
        dimensionedScalar(dimMass/dimLength/pow3(dimTime), Zero)
    ),
    Gg_ (absorptionEmission_->nBands())
{
    initGg();
}


// * * * * * * * * * * * * * Private member Functions  * * * * * * * * * * * //

void Foam::radiation::nonGreyP1::initGg()
{
    forAll(Gg_, bandI)
    {
        string GgFileInit("Gg.");
        OStringStream Convert; // num2str
        Convert << bandI;
        string iStr(Convert.str());
        Gg_.set
        (
            bandI,
            new volScalarField
            (
                IOobject
                (
                    GgFileInit+=iStr,
                    G_.time().timeName(),
                    //fileName("nongreyRadiation"),
                    G_.mesh(),
                    IOobject::READ_IF_PRESENT,
                    IOobject::AUTO_WRITE
                ),
                G_
            )
        );
    }
}


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

bool Foam::radiation::nonGreyP1::read()
{
    if (radiationModel::read())
    {
        // nothing to read

        return true;
    }

    return false;
}


void Foam::radiation::nonGreyP1::calculate()
{
    Info << "Solving RTE on each bands" << endl;
    absorptionEmission_->correct(G_, Gg_);
    fsckMLP* fsck = fsckMLP::getInstance();
    double* wNq = fsck->getwNq();
    forAll(Gg_, bandI)
    {
        fsck->setBandI(bandI);
        a_ = absorptionEmission_->a(bandI);
        e_ = absorptionEmission_->e(bandI);
        E_ = absorptionEmission_->E(bandI);
        const volScalarField sigmaEff(scatter_->sigmaEff());

        const dimensionedScalar a0("a0", a_.dimensions(), ROOTVSMALL);

        // Construct diffusion
        const volScalarField gamma
        (
            IOobject
            (
                "gammaRad",
                G_.mesh().time().timeName(),
                G_.mesh(),
                IOobject::NO_READ,
                IOobject::NO_WRITE
            ),
            1.0/(3.0*a_ + sigmaEff + a0)
        );

        // Solve G transport equation
        solve
        (
            fvm::laplacian(gamma, Gg_[bandI])
        - fvm::Sp(a_, Gg_[bandI])
        ==
        - 4.0*(e_*a_*physicoChemical::sigma*pow4(T_)) * dimensionedScalar(dimLength, 1) - E_
        );
    }
    G_ = 0.0*G_;
    forAll(Gg_, bandI)
    {
        G_ += Gg_[bandI]*wNq[bandI];
        forAll(mesh_.boundaryMesh(), patchi)
        {
            G_.boundaryFieldRef()[patchi] == G_.boundaryFieldRef()[patchi] +
                Gg_[bandI].boundaryField()[patchi]*wNq[bandI];
        }
    }

    // !TODO: To calculate the heat flux on the wall

    fsck->updateRu(Ru());
    fsck->updateRp(Rp());
}


Foam::tmp<Foam::volScalarField> Foam::radiation::nonGreyP1::Rp() const
{
    tmp<volScalarField> tRp
    (
        new volScalarField
        (
            IOobject
            (
                "Rp",
                mesh_.time().timeName(),
                mesh_,
                IOobject::NO_READ,
                IOobject::NO_WRITE
            ),
            4.0*absorptionEmission_->eCont()*physicoChemical::sigma*0.0
        )
    );

    fsckMLP* fsck = fsckMLP::getInstance();

    forAll(Gg_, bandI)
    {
        tRp = tRp + fsck->getwNq()[bandI] * absorptionEmission_->a(bandI) *
                4*physicoChemical::sigma * absorptionEmission_->e(bandI) *
                dimensionedScalar(dimLength, 1);
    }

    return tRp;
}


Foam::tmp<Foam::DimensionedField<Foam::scalar, Foam::volMesh>>
Foam::radiation::nonGreyP1::Ru() const
{
    tmp<DimensionedField<scalar, volMesh>> tRu
    (
        new volScalarField
        (
            IOobject
            (
                "Ru",
                mesh_.time().timeName(),
                mesh_,
                IOobject::NO_READ,
                IOobject::NO_WRITE
            ),
            mesh_,
            dimensionedScalar(dimMass/dimLength/pow3(dimTime), Zero)
        )
    );

    fsckMLP* fsck = fsckMLP::getInstance();

    forAll(Gg_, bandI)
    {
        DimensionedField<scalar, volMesh> tRui
        (
            fsck->getwNq()[bandI] * absorptionEmission_->a(bandI) *
                (
                    Gg_[bandI]// -
                    // 4*physicoChemical::sigma *
                    // absorptionEmission_->e(bandI) * pow4(T_) *
                    // dimensionedScalar(dimLength, 1)
                )
        );
        tRu = tRu + tRui;
    }

    return tRu;
}


Foam::label Foam::radiation::nonGreyP1::nBands() const
{
    return absorptionEmission_->nBands();
}


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

C.2.2 `nonGreyMarshakRadiation` class

`nonGreyMarshakRadiationFvPatchScalarField.H`

```
/*---------------------------------------------------------------------------*\
  =========                 |
  \\      /  F ield         | OpenFOAM: The Open Source CFD Toolbox
   \\    /   O peration     |
    \\  /    A nd           | www.openfoam.com
     \\/     M anipulation  |
-------------------------------------------------------------------------------
    Copyright (C) 2011-2016 OpenFOAM Foundation
-------------------------------------------------------------------------------
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::nonGreyMarshakRadiationFvPatchScalarField

Group
    grpThermoBoundaryConditions

Description
    A 'mixed' boundary condition that implements a Marshak condition for the
    incident radiation field (usually written as G)

    The radiation temperature is retrieved from the mesh database, using a
    user specified temperature field name.

Usage
    \table
        Property     | Description             | Required    | Default value
        T            | temperature field name  | no          | T
    \endtable

    Example of the boundary condition specification:
    \verbatim
    <patchName>
    {
        type            MarshakRadiation;
        T               T;
        value           uniform 0;
    }
    \endverbatim

See also
    Foam::radiationCoupledBase
    Foam::mixedFvPatchField

SourceFiles
    nonGreyMarshakRadiationFvPatchScalarField.C

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

#ifndef radiation_nonGreyMarshakRadiationFvPatchScalarField_H
#define radiation_nonGreyMarshakRadiationFvPatchScalarField_H

#include "mixedFvPatchFields.H"

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

namespace Foam
{
namespace radiation
{

/*---------------------------------------------------------------------------*\
            Class nonGreyMarshakRadiationFvPatchScalarField Declaration
\*---------------------------------------------------------------------------*/

class nonGreyMarshakRadiationFvPatchScalarField
:
    public mixedFvPatchScalarField
{

    // Private data

        //- Name of temperature field
        word TName_;


public:

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


    // Constructors

        //- Construct from patch and internal field
        nonGreyMarshakRadiationFvPatchScalarField
        (
            const fvPatch&,
            const DimensionedField<scalar, volMesh>&
        );

        //- Construct from patch, internal field and dictionary
        nonGreyMarshakRadiationFvPatchScalarField
        (
            const fvPatch&,
            const DimensionedField<scalar, volMesh>&,
            const dictionary&
        );

        //- Construct by mapping given MarshakRadiationFvPatchField onto a new
        //  patch
        nonGreyMarshakRadiationFvPatchScalarField
        (
            const nonGreyMarshakRadiationFvPatchScalarField&,
            const fvPatch&,
            const DimensionedField<scalar, volMesh>&,
            const fvPatchFieldMapper&
        );

        //- Construct as copy
        nonGreyMarshakRadiationFvPatchScalarField
        (
            const nonGreyMarshakRadiationFvPatchScalarField&
        );

        //- Construct and return a clone
        virtual tmp<fvPatchScalarField> clone() const
        {
            return tmp<fvPatchScalarField>
            (
                new nonGreyMarshakRadiationFvPatchScalarField(*this)
            );
        }

        //- Construct as copy setting internal field reference
        nonGreyMarshakRadiationFvPatchScalarField
        (
            const nonGreyMarshakRadiationFvPatchScalarField&,
            const DimensionedField<scalar, volMesh>&
        );

        //- Construct and return a clone setting internal field reference
        virtual tmp<fvPatchScalarField> clone
        (
            const DimensionedField<scalar, volMesh>& iF
        ) const
        {
            return tmp<fvPatchScalarField>
            (
                new nonGreyMarshakRadiationFvPatchScalarField(*this, iF)
            );
        }


    // Member functions

        // Access

            //- Return the temperature field name
            const word& TName() const
            {
                return TName_;
            }

            //- Return reference to the temperature field name to allow
            //  adjustment
            word& TName()
            {
                return TName_;
            }


        // Evaluation functions

            //- Update the coefficients associated with the patch field
            virtual void updateCoeffs();


        // I-O

            //- Write
            virtual void write(Ostream&) const;
};


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

} // End namespace radiation
} // End namespace Foam

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

#endif

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

nonGreyMarshakRadiationFvPatchScalarField.C

```
/*---------------------------------------------------------------------------*\
  =========                 |
  \\      /  F ield         | OpenFOAM: The Open Source CFD Toolbox
   \\    /   O peration     |
    \\  /    A nd           | www.openfoam.com
     \\/     M anipulation  |
-------------------------------------------------------------------------------
    Copyright (C) 2011-2015 OpenFOAM Foundation
    Copyright (C) 2016,2020 OpenCFD Ltd.
-------------------------------------------------------------------------------
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 "nonGreyMarshakRadiationFvPatchScalarField.H"
#include "addToRunTimeSelectionTable.H"
#include "fvPatchFieldMapper.H"
#include "volFields.H"
#include "radiationModel.H"
#include "physicoChemicalConstants.H"
#include "boundaryRadiationProperties.H"
```

```
#include "fsckMLP.H"

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

Foam::radiation::nonGreyMarshakRadiationFvPatchScalarField::
nonGreyMarshakRadiationFvPatchScalarField
(
    const fvPatch& p,
    const DimensionedField<scalar, volMesh>& iF
)
:
    mixedFvPatchScalarField(p, iF),
    TName_("T")
{
    refValue() = 0.0;
    refGrad() = 0.0;
    valueFraction() = 0.0;
}


Foam::radiation::nonGreyMarshakRadiationFvPatchScalarField::
nonGreyMarshakRadiationFvPatchScalarField
(
    const nonGreyMarshakRadiationFvPatchScalarField& ptf,
    const fvPatch& p,
    const DimensionedField<scalar, volMesh>& iF,
    const fvPatchFieldMapper& mapper
)
:
    mixedFvPatchScalarField(ptf, p, iF, mapper),
    TName_(ptf.TName_)
{}


Foam::radiation::nonGreyMarshakRadiationFvPatchScalarField::
nonGreyMarshakRadiationFvPatchScalarField
(
    const fvPatch& p,
    const DimensionedField<scalar, volMesh>& iF,
    const dictionary& dict
)
:
    mixedFvPatchScalarField(p, iF),
    TName_(dict.getOrDefault<word>("T", "T"))
{
    if (dict.found("value"))
    {
        refValue() = scalarField("value", dict, p.size());
    }
    else
    {
        refValue() = 0.0;
    }

    // zero gradient
    refGrad() = 0.0;

    valueFraction() = 1.0;

    fvPatchScalarField::operator=(refValue());
}


Foam::radiation::nonGreyMarshakRadiationFvPatchScalarField::
nonGreyMarshakRadiationFvPatchScalarField
(
    const nonGreyMarshakRadiationFvPatchScalarField& ptf
)
```

```
:
    mixedFvPatchScalarField(ptf),
    TName_(ptf.TName_)
{}


Foam::radiation::nonGreyMarshakRadiationFvPatchScalarField::
nonGreyMarshakRadiationFvPatchScalarField
(
    const nonGreyMarshakRadiationFvPatchScalarField& ptf,
    const DimensionedField<scalar, volMesh>& iF
)
:
    mixedFvPatchScalarField(ptf, iF),
    TName_(ptf.TName_)
{}


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

void Foam::radiation::nonGreyMarshakRadiationFvPatchScalarField::updateCoeffs()
{
    if (this->updated())
    {
        return;
    }

    // Since we're inside initEvaluate/evaluate there might be processor
    // comms underway. Change the tag we use.
    int oldTag = UPstream::msgType();
    UPstream::msgType() = oldTag+1;

    fsckMLP* fsck = fsckMLP::getInstance();

    const word& aName_("a" + std::to_string(fsck->getBandI()));

    // Nongrey streching factor field
    const scalarField& ap =
        patch().lookupPatchField<volScalarField, scalar>(aName_);

    // Temperature field
    const scalarField& Tp =
        patch().lookupPatchField<volScalarField, scalar>(TName_);

    // Re-calc reference value
    refValue() = 4.0*constant::physicoChemical::sigma.value()*pow4(Tp)*ap;

    // Diffusion coefficient - created by radiation model's ::updateCoeffs()
    const scalarField& gamma =
        patch().lookupPatchField<volScalarField, scalar>("gammaRad");

    const boundaryRadiationProperties& boundaryRadiation =
        boundaryRadiationProperties::New(internalField().mesh());

    const tmp<scalarField> temissivity
    (
        boundaryRadiation.emissivity(patch().index(), fsck->getBandI())
    );

    const scalarField emissivity = temissivity.ref();

    const scalarField Ep(emissivity/(2.0*(scalar(2) - emissivity)));

    // Set value fraction
    valueFraction() = 1.0/(1.0 + gamma*patch().deltaCoeffs()/Ep);

    // Restore tag
    UPstream::msgType() = oldTag;
    mixedFvPatchScalarField::updateCoeffs();
}


void Foam::radiation::nonGreyMarshakRadiationFvPatchScalarField::write
(
    Ostream& os
) const
{
    mixedFvPatchScalarField::write(os);
    os.writeEntryIfDifferent<word>("T", "T", TName_);
}


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

namespace Foam
{
namespace radiation
{
    makePatchTypeField
    (
        fvPatchScalarField,
        nonGreyMarshakRadiationFvPatchScalarField
    );
}
}

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

nonGreyMarshakRadiationFixedTemperatureFvPatchScalarField.H

```
/*---------------------------------------------------------------------------*\
  =========                 |
  \\      /  F ield         | OpenFOAM: The Open Source CFD Toolbox
   \\    /   O peration     |
    \\  /    A nd           | www.openfoam.com
     \\/     M anipulation  |
-------------------------------------------------------------------------------
    Copyright (C) 2011-2016 OpenFOAM Foundation
-------------------------------------------------------------------------------
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::nonGreyMarshakRadiationFixedTemperatureFvPatchScalarField

Group
    grpThermoBoundaryConditions

Description
    A 'mixed' boundary condition that implements a Marshak condition for the
    incident radiation field (usually written as G)

    The radiation temperature field across the patch is supplied by the user
    using the \c Trad entry.

Usage
    \table
        Property     | Description             | Required    | Default value
        T            | temperature field name  | no          | T
    \endtable

    Example of the boundary condition specification:
    \verbatim
    <patchName>
    {
        type            MarshakRadiationFixedTemperature;
        Trad            uniform 1000;       // radiation temperature field
        value           uniform 0;          // place holder
    }
    \endverbatim

See also
    Foam::radiationCoupledBase
    Foam::mixedFvPatchField

SourceFiles
    nonGreyMarshakRadiationFixedTemperatureFvPatchScalarField.C

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

#ifndef nonGreyMarshakRadiationFixedTemperatureFvPatchScalarField_H
#define nonGreyMarshakRadiationFixedTemperatureFvPatchScalarField_H

#include "mixedFvPatchFields.H"

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

namespace Foam
{
namespace radiation
{
/*---------------------------------------------------------------------------*\
     Class nonGreyMarshakRadiationFixedTemperatureFvPatchScalarField Declaration
\*---------------------------------------------------------------------------*/

class nonGreyMarshakRadiationFixedTemperatureFvPatchScalarField
:
    public mixedFvPatchScalarField
    //public radiationCoupledBase
{

    // Private data

        //- Radiation temperature field
        scalarField Trad_;


public:

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


    // Constructors

        //- Construct from patch and internal field
        nonGreyMarshakRadiationFixedTemperatureFvPatchScalarField
        (
            const fvPatch&,
            const DimensionedField<scalar, volMesh>&
        );

        //- Construct from patch, internal field and dictionary
        nonGreyMarshakRadiationFixedTemperatureFvPatchScalarField
        (
            const fvPatch&,
            const DimensionedField<scalar, volMesh>&,
            const dictionary&
        );

        //- Construct by mapping given MarshakRadiationFvPatchField onto a new
        //  patch
        nonGreyMarshakRadiationFixedTemperatureFvPatchScalarField
        (
            const nonGreyMarshakRadiationFixedTemperatureFvPatchScalarField&,
            const fvPatch&,
            const DimensionedField<scalar, volMesh>&,
            const fvPatchFieldMapper&
        );

        //- Construct as copy
        nonGreyMarshakRadiationFixedTemperatureFvPatchScalarField
        (
            const nonGreyMarshakRadiationFixedTemperatureFvPatchScalarField&
        );

        //- Construct and return a clone
        virtual tmp<fvPatchScalarField> clone() const
        {
            return tmp<fvPatchScalarField>
            (
                new nonGreyMarshakRadiationFixedTemperatureFvPatchScalarField(*this)
            );
        }

        //- Construct as copy setting internal field reference
        nonGreyMarshakRadiationFixedTemperatureFvPatchScalarField
        (
            const nonGreyMarshakRadiationFixedTemperatureFvPatchScalarField&,
            const DimensionedField<scalar, volMesh>&
        );

        //- Construct and return a clone setting internal field reference
        virtual tmp<fvPatchScalarField> clone
        (
            const DimensionedField<scalar, volMesh>& iF
        ) const
        {
            return tmp<fvPatchScalarField>
            (
                new nonGreyMarshakRadiationFixedTemperatureFvPatchScalarField
                (
                    *this,
                    iF
                )
            );
        }


    // Member functions

        // Access

            //- Return the radiation temperature
            const scalarField& Trad() const
            {
                return Trad_;
            }
            //- Return reference to the radiation temperature to allow
            //  adjustment
            scalarField& Trad()
            {
                return Trad_;
            }


        // Mapping functions

            //- Map (and resize as needed) from self given a mapping object
            virtual void autoMap
            (
                const fvPatchFieldMapper&
            );

            //- Reverse map the given fvPatchField onto this fvPatchField
            virtual void rmap
            (
                const fvPatchScalarField&,
                const labelList&
            );


        // Evaluation functions

            //- Update the coefficients associated with the patch field
            virtual void updateCoeffs();


        // I-O

            //- Write
            virtual void write(Ostream&) const;
};


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

} // End namespace radiation
} // End namespace Foam

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

#endif

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

nonGreyMarshakRadiationFixedTemperatureFvPatchScalarField.C

```
/*---------------------------------------------------------------------------*\
  =========                 |
  \\      /  F ield         | OpenFOAM: The Open Source CFD Toolbox
   \\    /   O peration     |
    \\  /    A nd           | www.openfoam.com
     \\/     M anipulation  |
-------------------------------------------------------------------------------
    Copyright (C) 2011-2015 OpenFOAM Foundation
    Copyright (C) 2016 OpenCFD Ltd.
-------------------------------------------------------------------------------
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 "nonGreyMarshakRadiationFixedTemperatureFvPatchScalarField.H"
#include "addToRunTimeSelectionTable.H"
#include "fvPatchFieldMapper.H"
#include "volFields.H"
#include "radiationModel.H"
#include "physicoChemicalConstants.H"
#include "boundaryRadiationProperties.H"
#include "fsckMLP.H"

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

Foam::radiation::nonGreyMarshakRadiationFixedTemperatureFvPatchScalarField::
nonGreyMarshakRadiationFixedTemperatureFvPatchScalarField
(
    const fvPatch& p,
    const DimensionedField<scalar, volMesh>& iF
)
:
    mixedFvPatchScalarField(p, iF),
    Trad_(p.size())
{
    refValue() = 0.0;
    refGrad() = 0.0;
    valueFraction() = 0.0;
}


Foam::radiation::nonGreyMarshakRadiationFixedTemperatureFvPatchScalarField::
nonGreyMarshakRadiationFixedTemperatureFvPatchScalarField
(
    const nonGreyMarshakRadiationFixedTemperatureFvPatchScalarField& ptf,
    const fvPatch& p,
    const DimensionedField<scalar, volMesh>& iF,
    const fvPatchFieldMapper& mapper
)
:
    mixedFvPatchScalarField(ptf, p, iF, mapper),
    Trad_(ptf.Trad_, mapper)
{}


Foam::radiation::nonGreyMarshakRadiationFixedTemperatureFvPatchScalarField::
nonGreyMarshakRadiationFixedTemperatureFvPatchScalarField
(
    const fvPatch& p,
    const DimensionedField<scalar, volMesh>& iF,
    const dictionary& dict
)
:
    mixedFvPatchScalarField(p, iF),
    Trad_("Trad", dict, p.size())
{
    fsckMLP* fsck = fsckMLP::getInstance();

    const word& aName_("a" + std::to_string(fsck->getBandI()));

    // Nongrey streching factor field
    const scalarField& ap =
        patch().lookupPatchField<volScalarField, scalar>(aName_);
    // refValue updated on each call to updateCoeffs()
    refValue() = 4.0*constant::physicoChemical::sigma.value()*pow4(Trad_)*ap;

    // zero gradient
    refGrad() = 0.0;

    valueFraction() = 1.0;

    fvPatchScalarField::operator=(refValue());
}


Foam::radiation::nonGreyMarshakRadiationFixedTemperatureFvPatchScalarField::
nonGreyMarshakRadiationFixedTemperatureFvPatchScalarField
(
    const nonGreyMarshakRadiationFixedTemperatureFvPatchScalarField& ptf
)
:
    mixedFvPatchScalarField(ptf),
    Trad_(ptf.Trad_)
{}


Foam::radiation::nonGreyMarshakRadiationFixedTemperatureFvPatchScalarField::
nonGreyMarshakRadiationFixedTemperatureFvPatchScalarField
(
    const nonGreyMarshakRadiationFixedTemperatureFvPatchScalarField& ptf,
    const DimensionedField<scalar, volMesh>& iF
)
:
    mixedFvPatchScalarField(ptf, iF),
    Trad_(ptf.Trad_)
{}


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

void Foam::radiation::nonGreyMarshakRadiationFixedTemperatureFvPatchScalarField::
autoMap
(
    const fvPatchFieldMapper& m
)
{
    mixedFvPatchScalarField::autoMap(m);
    Trad_.autoMap(m);
}


void Foam::radiation::nonGreyMarshakRadiationFixedTemperatureFvPatchScalarField::rmap
(
    const fvPatchScalarField& ptf,
    const labelList& addr
)
{
    mixedFvPatchScalarField::rmap(ptf, addr);

    const nonGreyMarshakRadiationFixedTemperatureFvPatchScalarField& mrptf =
        refCast<const nonGreyMarshakRadiationFixedTemperatureFvPatchScalarField>(ptf);

    Trad_.rmap(mrptf.Trad_, addr);
}


void Foam::radiation::nonGreyMarshakRadiationFixedTemperatureFvPatchScalarField::
updateCoeffs()
{
    if (this->updated())
    {
        return;
    }

    // Since we're inside initEvaluate/evaluate there might be processor
    // comms underway. Change the tag we use.
    int oldTag = UPstream::msgType();
    UPstream::msgType() = oldTag+1;

    fsckMLP* fsck = fsckMLP::getInstance();

    const word& aName_("a" + std::to_string(fsck->getBandI()));

    // Nongrey streching factor field
    const scalarField& ap =
        patch().lookupPatchField<volScalarField, scalar>(aName_);

    // Re-calc reference value
    refValue() = 4.0*constant::physicoChemical::sigma.value()*pow4(Trad_)*ap;

    // Diffusion coefficient - created by radiation model's ::updateCoeffs()
    const scalarField& gamma =
        patch().lookupPatchField<volScalarField, scalar>("gammaRad");

    //const scalarField temissivity = emissivity();
    const boundaryRadiationProperties& boundaryRadiation =
        boundaryRadiationProperties::New(internalField().mesh());

    const tmp<scalarField> temissivity
    (
        boundaryRadiation.emissivity(patch().index(), fsck->getBandI())
    );

    const scalarField emissivity = temissivity.ref();

    const scalarField Ep(emissivity/(2.0*(scalar(2) - emissivity)));

    // Set value fraction
    valueFraction() = 1.0/(1.0 + gamma*patch().deltaCoeffs()/Ep);

    // Restore tag
    UPstream::msgType() = oldTag;

    mixedFvPatchScalarField::updateCoeffs();
}


void Foam::radiation::nonGreyMarshakRadiationFixedTemperatureFvPatchScalarField::write
(
    Ostream& os
) const
{
    mixedFvPatchScalarField::write(os);
    Trad_.writeEntry("Trad", os);
}


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

namespace Foam
{
namespace radiation
{
    makePatchTypeField
    (
        fvPatchScalarField,
        nonGreyMarshakRadiationFixedTemperatureFvPatchScalarField
    );
}
}

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