Cite as: Zhou, Y.: Implementation of growing CCM library to reduce chemistry calculation time. In Proceedings of CFD with OpenSource Software, 2022, Edited by Nilsson. H.,
http://dx.doi.org/10.17196/OS_CFD#YEAR_2022

CFD with OpenSource software

A course at Chalmers University of Technology
Taught by Håkan Nilsson

Implementation of growing CCM library to reduce chemistry calculation time

Developed for OpenFOAM-v2112

Author:
Yuchen Zhou
Lund University
yuchen.zhou@energy.lth.se

Peer reviewed by:
Xue-Song Bai
Yaquan Sun
Mohammad Hossein Arabnejad Khanouki

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

January 24, 2023

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 set a case using the CCM or the growing CCM library.
- How to set the appropriate parameters in the CCM dictionary.

The theory of it:

- The very basic idea behind most (if not all) chemistry acceleration methods.
- The theory behind the CCM method and the basic procedure of this method.

How it is implemented:

- How to implement CCM and growing CCM methods, especially about grouping/storing data and attributing those data to relevant cells.

How to modify it:

- How to modify the CCM and growing CCM library.

Prerequisites

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

- How to follow the basic official tutorials of OpenFOAM.
- Some basic CFD (Computational Fluid Dynamics) and combustion knowledge.
- How to configure a basic combustion case in OpenFOAM and the structure of the case.
- Some C++ knowledge, as well as several very basic commands of the terminal line.

Contents

Nomenclature

Acronyms

CCM	Chemistry Coordinate Mapping
CFD	Computational Fluid Dynamics
DAC	Dynamic Adaptive Chemistry
FGM	Flamelet Generated Manifold
ISAT	In-situ Adaptive Tabulation
ODE	Ordinary Differentiate Equations
TDAC	Tabulation of Dynamic Adaptive Chemistry

English symbols

$\dot{Q}$	Heat release rate rate	$\mathrm{J \cdot m^{-3} \cdot s^{-1}}$
ω_i	Reaction rate for species i	$\mathrm{kg \cdot m^{-3} \cdot s^{-1}}$
C_i	Molar concentration	$\mathrm{mol \cdot cm^{-3}}$
D_i	Molecular diffusion coefficient for species i	$\mathrm{m^2 \cdot s^{-1}}$
E_a	Activation energy	$\mathrm{J \cdot mol^{-1} \cdot K^{-1}}$
h	Enthalpy	$\mathrm{J \cdot kg^{-1}}$
K	Kinetic energy	$\mathrm{J \cdot kg^{-1}}$
$k(T)$	Rate constant	
R_u	Specific gas constant	$\mathrm{J \cdot kg^{-1} \cdot K^{-1}}$
T	Temperature	K
Y_i	Mass fraction for species i	[−]

Greek symbols

α_{eff}	Effective turbulent thermal diffusivity for enthalpy	$\mathrm{m^2 \cdot s^{-1}}$
ρ	Fluid density	$\mathrm{kg \cdot m^{-3}}$

Subscripts

i	Species i or direction i
j	Species j or direction j

Chapter 1

Introduction

This report intends to illustrate the basic theory of Chemistry Coordinate Mapping (CCM) and its implementations in OpenFOAM. Also, an improved version of CCM called growing CCM is introduced with detailed implementation tutorials.

In recent decades, we have witnessed the rapid progress of the Computational Fluid Dynamics (CFD) theory and its huge impacts in academic and industrial fields. With appropriate CFD methods used by well-trained CFD engineers, the flow field predictions with satisfactory accuracy can be obtained, which can, to a certain extent, replace some expensive and dangerous experiments [1]. This report will focus on the simulations of reactive flows, which are far more complicated than the "cold" flow without chemical reactions. Not only the basic Navier-Strokes equations (including one continuity equation, three momentum equations if this is a 3D simulation, one energy equation, and one constitutive equation) are simulated and well-resolved, N species transport equations along with a possibly very stiff reaction rate Ordinary Differentiate Equations (ODE) system should be well-addressed to guarantee the accuracy of the simulation outcome. Considering that it needs 325 reactions and 53 species to characterize the behaviors of a fuel as simple as methane [2], even more equations should be solved to describe the combustion processes of diesel fuels, biomass, and hybrid fuels. This will make the detailed simulations extremely computationally expensive and unaffordable for academic research and industry purposes. Consequently, efforts should be made to 1) simplify the chemistry mechanism and 2) accelerate the chemistry simulations. For 1), only key reactions shall be preserved to significantly reduce the number of equations during the simulation processes. However, simulations with detailed chemistry might still be a must in many cases, such as emission and efficiency predictions, where an oversimplified mechanism might fail to capture the full details [3]. For 2), we are expected to reduce the number of detailed chemistry simulations by trying to reuse the previous simulation results. This might be achieved by, for example, the Tabulation of Dynamic Adaptive Chemistry (TDAC) method [4], and the Flamelet Generated Manifold (FGM) method [5, 6]. The TDAC method available in OpenFOAM is a combination of the Dynamic Adaptive Chemistry (DAC) [7] and the In-situ Adaptive Tabulation (ISAT) [8] method. This method will both dynamically simplify the chemistry mechanism and tabulate and reuse the reaction rates using multiple linear regressions. High acceleration rates based on this method can be expected, but it might also fail in some cases to predict the species field. We will show this later in Section 4.7. As for the FGM method [5, 6], reaction rates are obtained using simple 1D or 2D simulations beforehand, and the species rate or distribution in a reaction progress variable coordinate will be stored in a data table that will be retrieved later in reactive flow simulations. The simulation is carried out with only a few additional transportation equations (e.g., mixture fraction and reaction progress variable) to save time, while the values of these transport variables are used in the retrieval processes mentioned before.

The CCM method also intends to reduce the amount of chemistry calculation by grouping the physical cells with the same "reaction states" and letting them share a common reaction rate. Those shared reaction rates can be calculated in each time step (which is the ordinary CCM method), stored in a table, and retrieved later (which is the developing target of this paper). In this report, We will

focus on explaining how to use the CCM method to accelerate a combustion simulation and the basic idea of CCM, as well as a detailed guide on even further developments of this method.

The report is organized as follows. In chapter 2 of this report, the philosophy of simplifying a reaction system will be briefly explained, followed by discussions on the procedure of CCM. Chapter 3 will focus on the implementations of the CCM library. The original CCM code is written by the colleagues of the authors using OpenFOAM-7. It is reorganized and transferred to OpenFOAM-v2112 for this project. We will discuss in which part the initialization, grouping, and reaction rate attributions are done with some explanations. Modifications to add the growing CCM feature are also shown in detail. At last, we will show how to simulate a combustion case with the CCM or growing CCM method. Some brief discussions of parameter settings are also presented there.

Chapter 2

The theory of CCM

This chapter introduces the relevant equations for calculating the reaction rates. Then we will discuss the behaviors of those equations in a reacting flow case and suggest possible ways to reduce the amount of reaction rate calculations. After that, the validity of the CCM method will be discussed.

Please note that we have emitted all the introductions of Navier-Strokes Equations. The potential readers of this report are referred to some CFD text books [1] for that knowledge.

2.1 The calculation of the reaction rates

In this section, we will show how to calculate the reaction rates during a combustion simulation and how those reaction rates are introduced into the transportation equations of the main solver.

We start with the hydrogen-air reaction described by a single-step model

$$2H_2 + O_2 \rightarrow 2H_2O \tag{2.1}$$

The equation indicates that two hydrogen molecules and one oxygen molecule might meet and react, forming two water molecules. This is generally true to describe the overall reaction process, but some modifications are still required. In fact, the hydrogen molecules cannot react with the oxygen molecules directly. What really happens is that the hydrogen molecule might collide with a third party and form two hydrogen radicals, shown as

$$H_2 + M \rightarrow H + H + M \tag{2.2}$$

Those radicals will collide with the oxygen molecule and form OH radicals, shown as

$$H + O_2 \rightarrow OH + O \tag{2.3}$$

After that, the OH radical can react with the hydrogen molecule and form the product water, shown as

$$OH + H_2 \rightarrow H_2O + H \tag{2.4}$$

The above process is just a route to reach the final products H_2O, and might not be the most important route of this reaction system during some operating conditions. However, it is enough to explain two factors.

- A single-step model is a simplified representation of many elementary reactions.
- Many elementary reactions with various reaction rates act together and determine the overall reaction rate.

We can calculate the reaction rate for each elementary reaction in the form of

$$\mathrm{A} + \mathrm{B} \rightarrow \mathrm{C} + \mathrm{D} \tag{2.5}$$

using the Arrhenius Formula given by

$$\frac{\mathrm{d}C_A}{\mathrm{d}t} = C_A C_B k(T), \quad \frac{\mathrm{d}C_A}{\mathrm{d}t} = \frac{\mathrm{d}C_B}{\mathrm{d}t} = -\frac{\mathrm{d}C_C}{\mathrm{d}t} = -\frac{\mathrm{d}C_D}{\mathrm{d}t} \tag{2.6}$$

Please note that only the elementary reaction can be done in this way.

Here $k(T)$ can be expressed as

$$k(T) = AT^b exp(-\frac{E_a}{R_u T}) \tag{2.7}$$

where A, b, E_a are three empirical parameters. Basically, we can see that the elementary reaction rate is a function of the molar concentration of reactants and temperature. The properties of the reactants and products will also determine those empirical parameters, which further influence the reaction rate.

The above information is more or less enough to understand the contents of this report. The readers are also recommended to read the combustion textbook [9] to see more detailed explanations.

2.2 Governing equations for species and energy

In this section, we will discuss how the reaction rate will be introduced into the transport equations. Generally speaking, the reaction rates will directly modify the source term of the species transport equations and sensible enthalpy equations. This represents direct changes in species mass fractions, properties and mixture temperature. These changes will immediately affect pressure which drives the flow motion, then influence all other flow field variables.

The species transport equation can be written as

$$\frac{\partial \rho Y_i}{\partial t} + \frac{\partial \rho u_j Y_i}{\partial x_j} = \frac{\partial}{\partial x_j}(\rho D_i \frac{\partial Y_i}{\partial x_j}) + \omega_i \tag{2.8}$$

where Y_i is the mass fraction for species i, and D_i is the diffusion coefficients. The two terms on the left-hand side form the material derivative of the species mass fraction, describing the changing rate of the mass fraction following a fluid particle. The first term of the right-hand side is the diffusion term, while the second term is the reaction source term. The reaction rate we calculate in 2.1, multiplied by the molar weight of a species, is inserted here to be a part of the equation system. For more detailed derivations and explanations, the readers are referred to CFD and combustion textbooks [1, 9]. Here, we only want to show that the reaction will bring this equation an additional source term representing the generation or consumption of species.

The sensible enthalpy equation is written as

$$\frac{\partial \rho h}{\partial t} + \frac{\partial \rho u_j h}{\partial x_j} + \frac{\partial \rho K}{\partial t} + \frac{\partial \rho u_j K}{\partial x_j} - \frac{\mathrm{d}p}{\mathrm{d}t} = \frac{\partial}{\partial x_j}(\alpha_{eff} \frac{\partial h}{\partial x_j}) + \dot{Q} \tag{2.9}$$

where h is the sensible enthalpy, K is the kinetic energy of the flow, and $\dot{Q}$ is the heat release rate. Please note that the viscous term is not important and emitted for simplicity. The author is not pretty sure whether OpenFOAM solves this term, but it seems that it does not (The author checked that but did not dive deeply into it). Here, we only want to emphasize that the reaction will bring an additional term, representing the conversion of chemical energy into thermal/kinetic energy.

2.3 The manifold concept

In this section, we will present some examples to show how the complex reaction rate solution system might be simplified.

As we can see from Section 2.1, the reaction rates are determined by the temperature and the molar concentration of the reactants. For general CFD software, the calculation of mass fraction might be more convenient than the molar fraction so that we can turn those molar fractions into mass fractions. Given all mass concentrations, together with the temperature, it is sufficient to obtain all the reaction rates. However, one important question is whether it is possible to obtain reaction rates with just a subset of those mass fractions. Or, can we pick up some control variables to form a control variable set, and the reaction rate system can be uniquely determined and well described by this control variable set.

For premixed flame, it is possible to configure such a control variable set because reactions in premixed flame take place in a narrow region with sharp species concentration and temperature gradients. The well-mixed fuel and air diffuse into that region and get burnt. If this thin flame region is not altered by turbulent flows, we might simulate a 1D laminar case to build up the relations between the reaction rates, and the distance from the flame front. Then, we simulate a 3D case and measure the distance of each cell from the flame front. By that means, we can look up the reaction rate of that given distance from the 1D results. It should be emphasized that the distance we discussed before can be real distance or a measurement of the reaction progress. One way of defining that distance is to use the reaction progress variable. For methane flame, it could be defined as

$$Y_c = \frac{Y_{CO_2}}{M_{CO_2}} + \frac{Y_{CO}}{M_{CO}} + \frac{Y_{HO_2}}{M_{HO_2}} \tag{2.10}$$

As for the diffusion flame, the scenario is almost the same as the premixed flame, but we should replace the progress variable with the mixture fraction if the Lewis number is assumed to be unity [10]. If we want to predict the extinction behaviors near the burner rim, the scalar dissipation rate should be introduced. If we want to further simulate the ignition process, one progress variable shall also be added in.

As we can see from the above examples, the reaction rates can be well determined in most of the cases if a well-selected set of control variables are determined. It also suggests that once those control variables are specified, other species information will be uniquely determined and explicitly included. Most (if not all) chemistry acceleration methods borrow this idea, and obtain reaction rates either in advance or simultaneously with the simulations.

2.4 The procedure of CCM

The procedure of the CCM method can be divided into four parts.

- Each cell point in physical space is assigned a unique zone index, acting as an identification for a physical cell according to its reaction state. We will call it "indexing" for short.
- The cells in physical space with the same zone index will be grouped together and assigned a common zone number representing their indexes in phase space, which is also the order of their reaction rates in the memory in the further retrieval stage. We will call it "grouping" for short. After this stage, we will obtain a much smaller set of cells in phase space.
- The cells in phase space after grouping will be dealt with by the ODE solver, and the reaction rates will be calculated. Please note that one point at this stage might represent many physical cells with similar reaction states. The physical properties of those physical cells should be averaged, and the average state is sent to calculate the reaction rate. And this reaction rate will be shared by all those physical cells. This stage is called "solving" for short.
- The reaction rates will be assigned in this stage. We will call it "assigning" for short.

2.5 CCM variables and index strategy

The indexing stage will be explained in detail here, while the grouping and assigning stages will be discussed in Chapter 3 when we introduce the code implementation. The basic knowledge of reaction

rate calculations and how it is introduced into the solution system has already been discussed in Section 2.1 and Section 2.2.

Before introducing the indexing part, it is necessary to explain the basic control variables to describe the reaction states. In CCM, our control variables (called CCM variables in this report) to describe the reaction states are (T, ϕ, χ, Y_1, Y_2, ...). Here, T is temperature, ϕ is the equivalence ratio, χ is the scalar dissipation rate, and Y_i is the mass fraction of species i. The program allows one or more species to take part in the indexing processes. If a case requires varying pressure (e.g., engine compression), the pressure should be added to the CCM variables, and the final CCM variable set will be (T, p ϕ, χ, Y_1, Y_2, ...). In this report, we only consider cases satisfying constant pressure assumptions.

The control variable set mentioned in the last paragraph is valid. We have already introduced many ideal cases in Section 2.3 and how to configure control variables in those cases. In general cases, usually the heat transfer and compression effects cannot be ignored in order to obtain a satisfactory result. Consequently, we might suggest a way to make sure the thermal states and the key species are considered [11]. If we have two thermal variables of a cell (e.g. T and p, or T alone in constant pressure case), the density, as well as other thermal variables will be uniquely determined if the composition of the cell is known. Then, if some key variables that are believed to control the reactions are introduced, we can well estimate the behaviour of the reaction system. Consequently, the CCM variable set (T, p ϕ, χ, Y_1, Y_2, ...) are valid for a general case, and more variables can be appended if more accurate simulation details are required.

The purpose of the indexing part is to assign all possible reaction states with some unique identification numbers (zone index). Similar states within the bound of accuracy tolerance will be assigned the same zone index. It does no harm to assume that the smallest possible value of the first dimension is 6, while the largest is 8, and the tolerance is 0.5. Then,

- For value within range (6, 6.5], the zone index for this dimension is 0.
- For value within range (6.5, 7], the zone index for this dimension is 1.
- ...
- For value within range (7.5, 8], the zone index for this dimension is 3.

If we have three dimensions, use three digits to represent each dimension, with the dimension values 3, 5, and 2, respectively, the final zone index, including all dimension information, will be 003 005 002, assuming that we use three digits to describe each dimension. Please note that the way of calculating the zone index is different from the method described in the original paper by Jangi and Bai. [11]. But this difference will not influence the final results. The purpose is to give each reaction state an "identification number" and make sure that cells with similar reaction states share the same "identification number", while the exact value of the zone index is not important.

Chapter 3

The implementation of CCM

In this chapter, the implementation of the CCM library will be introduced. The code shown both here and in the attached files is modified from a library developed by my colleagues, running on `OpenFOAM-7` without the growing feature. After some testing, I transferred that library to `OpenFOAM-v2112` and adjusted its structure for more convenient further development. The detailed process of the transfer and the structure adjustment is not the concern of this report, so we omit them for simplicity.

The chapter starts with a description of the general structure by discussing the main `solve` function. After that, each header file in the `solve` function, representing a solution step (already discussed in Section 2.4) during the calculation of the reaction rates with the CCM library, will be discussed in order. After that, some explanations will be made on how to add the growing feature.

3.1 General structure

A combustion solver in OpenFOAM will call `reaction->correct()` (usually in `YEqn.H`) to solve the chemistry. Assuming that no turbulent chemistry interaction is considered, the `solve` function with a `deltaTType` parameter will be called. The readers are referred to a course report in previous years written by Gadalla [12] for more detailed explanations on this topic.

The `solve` function of the CCM chemistry library is shown as follows. By observing that function, we can find that despite some information output, some header files are included here, representing different stages of the CCM solving process as we mentioned in Section 2.4. After the four main stages (i.e. indexing, grouping, solving, and assigning), the reaction rates of each cell is obtained. Please note that the "solving" process mentioned before is implemented in `avging.H` and `solving.H`. In `avging.H`, the reaction states of different cells sharing the same zone index will be averaged, then the averaged outcome will be used to calculate the reaction rates, implenented in `solving.H`. In addition, the code also includes an additional correction stage, used for correcting the inaccurate reaction rates. It is implemented based on the idea that some CCM variable sets might not be able to describe the reaction rate space fully, meaning that the reaction states are not uniquely determined with the user-specified CCM variable set. If this happens, a zone index in the CCM library might correspond to multiple reaction states with varying reaction rates, which might crash the simulation. The `correction.H` contains the code that will discover those differences and recalculate the problematic cells to avoid crashing. This correction feature is still under-development and will not be discussed in this report.

solve function in CCM

```
template<class ReactionThermo, class ThermoType>
template<class DeltaTType>
Foam::scalar Foam::CCMChemistryModel<ReactionThermo, ThermoType>::solve
(
    const DeltaTType& deltaT
)
```

```
    {
        const clockTime clockTime_= clockTime();
        clockTime_.timeIncrement();

        Info << nl << nl << "/===================== CCM Solution begins =======================/" << endl;
        #include "indexing.H"
        Info << "CCM index done "  << "("  << clockTime_.timeIncrement() << " s)" <<  "."  << endl;

        #include "grouping.H"
        Info << "CCM grouping done "  <<  "("  << clockTime_.timeIncrement() << " s)" <<  "."  << endl;

        #include "avging.H"
        Info << "CCM avging done "  <<  "("  << clockTime_.timeIncrement() << " s)" <<  "."  << endl;

        #include "correction.H"
        if (autoCorrection_) Info << "CCM correction done "  <<  "("  << clockTime_.timeIncrement() << " s)" <<  "."  << endl;

        #include "solving.H"

        Info << "CCM solving done "  <<  "("  << clockTime_.timeIncrement() << " s)" <<  "."  << endl;

        #include "attributeRR.H"
        Info << "CCM reaction attribution done "  <<  "("  << clockTime_.timeIncrement() << " s)" <<  "."  << endl;
        label  nCell_oP = this->mesh().C().size();
        reduce(nCell_oP, sumOp<label>());
        nActiveCell_out
            << scientific << this->mesh().time().timeName() << tab
            << scientific << 1 <<  tab
            << scientific << nCell_oP << tab
            << scientific << nCell_oP << tab
            << scientific << numOfNCActive << tab
            << scientific << nCell_oP/(numOfNCActive+1) << endl;
        Info
            << "meshSize " << tab
            << "oldActiveCells" << tab
            << "newActiveCell" << tab
            << "speedup factor" << endl;

        Info
            << nCell_oP << tab
            << oldTableSize_ << tab
            << newTableSize_ << tab
            <<static_cast<scalar>(nCell_oP)/static_cast<scalar>(numOfNCActive+1) << endl;


        Info << "/===================== CCM Solution completes =====================/" << endl;

        return 0;
    }
```

3.2 The implementation of CCM

In this section, the detailed implementation of the CCM library is introduced. We will start with the implementation of the reaction rate calculation, then followed by discussions on several header files we mentioned in Section 3.1.

3.2.1 The calculation of reaction rates

The function `solveCCM` is modified from the `solve` function of the `standardChemistry.C`, with the detailed explanations in a report of the previous year by Gadalla [12]. The readers of interest are

encouraged to check the original file and spot the difference. A general takeaway is that the original `solve` function in the `standardChemistry.C`

- iterates over all the cells and obtain their temperature T, pressure p, and molar concentration c
- call another solve function (usually a stiff solver), to calculate the final molar concentration after a time step `deltaT`
- calculate the mean reaction rate during that given time step

The CCM chemistry library does the same thing for only a portion of the cells. As a consequence, a list of cells should be passed to the `solveCCM` function.

If we have a look at the parameter list of the `solveCCM` function, we will find that all relevant information (T, p, c, etc.) of the whole list of cells that the reaction rates are required, and passed to the `solveCCM` function by an argument called `FieldCCM`, and the calculated reaction rates are stored in another reference argument passed into the function called `RRCCM`. Considering that all cores work in parallel, the start, end and order index are given by three parameters called `nBeg`, `nLast`, `cellIndexTmp`, respectively. The start and end index will be different for different cores, so that each cell just has to iterate over a subset of the whole list of cells and obtain their reaction rates.

Now we have already know how to calculate the reaction rates, we just need to understand how to group the points and generate the `fieldCCM` for the grouped cells including all the reaction relevant information that are required for the chemistry calculation. After that, the reaction rates will be solved by `solveCCM` function. Finally, the solved reaction rates should be assigned to the original physical cell.

solveCCM function

```
template<class ReactionThermo, class ThermoType>
Foam::scalar Foam::CCMChemistryModel<ReactionThermo, ThermoType>::solveCCM
(
    const label nBeg,
    const label nLast,
    const labelList cellIndexTmp,
    const PtrList<scalarField>& FieldCCM,
    List<List<scalar>>& RRCCM
)
{
    BasicChemistryModel<ReactionThermo>::correct();

    scalar deltaT = this->mesh().time().deltaTValue();

    scalar deltaTMin = GREAT;

    if (!this->chemistry_)
    {
        return deltaTMin;
    }

    scalarField c0(this->nSpecie_);

    for( label zonei=nBeg; zonei < nLast; zonei++)
    {
        label izone = cellIndexTmp[zonei];
        //label izone = zonei;
        scalar Ti = FieldCCM[0][izone];
        scalar pi = FieldCCM[1][izone];
        const scalar rhoi = FieldCCM[2][izone];

        for (label i=0; i<this->nSpecie_; i++)
        {
            this->c_[i] = rhoi*FieldCCM[i+5][izone]/this->specieThermo_[i].W();
            c0[i] = this->c_[i];
```

```
            }

            // Initialise time progress
            scalar timeLeft = deltaT;
            scalar deltaTChemIni = this->deltaTChemIni_;

            // Calculate the chemical source terms
            while (timeLeft > SMALL)
            {
                scalar dt = timeLeft;
                this->solve(this->c_, Ti, pi, dt, deltaTChemIni);
                timeLeft -= dt;
            }

            deltaTMin = min(deltaTChemIni, deltaTMin);

            for (label i=0; i<this->nSpecie_; i++)
            {
                RRCCM[i][izone] =
                    (this->c_[i] - c0[i])*this->specieThermo_[i].W()/deltaT/rhoi;
            }
        }


        if (deltaTMin==GREAT)
        {
            Info << "deltaTMin-ChemistryModel === " << deltaTMin << endl;
            Info << "If deltaTMin-ChemistryModel == GREAT" << endl;
            Info << "This ususlly because there is only one active cell in CCM space, and it is allocated to the last processor!" << endl;
        }

        return deltaTMin;
    }
```

3.2.2 Indexing

The indexing is achieved by the codes shown below. The program iterates over all cells and all dimensions, calculating the zone index for each dimension in every cell with the type `word` (similar to the type `string` in `C++`) and concatenating the zone index for each dimension together for each cell. It should be noted if the maximum digits for each dimension (i.e. `stringDigit`) exceeds the required digits, unused digits will be set to zero.

indexing function

```
    // attribute ZoneIndex
    for(label icell= 0; icell < fieldSize; icell++)
    {
        for(label mccm=0; mccm < nMCCM_; mccm ++) // for each dimension
        {
            scalar   pos = (MZ[mccm][icell]-MZmin_[mccm])/ZoneSpan_[mccm];
            word tempWord = std::to_string(static_cast<label>(pos));
            tempWord.insert(0,stringDigit_ - tempWord.size(),'0');
            ZoneIndex[icell] = ZoneIndex[icell] + tempWord;
        }
    }
```

3.2.3 Grouping in parallel

The grouping process is implemented with the codes shown below. The most complex part (i.e. `getZoneNumber` function) is packed to another library, which is compiled individually and linked to this library when compiling the CCM library. We will first discuss the general part in the header file `grouping.H`, and then examine the implementation of function `getZoneNumber` in detail.

The general idea of grouping is that we should generate a "minimum set" from the set of physical cells according to their reaction states. Physically, cells with similar reaction states (i.e. T, ϕ, Y, χ) share a common element in the minimum set. Technically, we can achieve that by indexing the minimum sets and tell the original cell from which index it can retrieve its reaction rates. The `getZoneNumber` function helps us to

- obtain the minimum set S_{min} and return all the zone indexes it includes with the variable name `gCCMZoneIndex`. Each element in the S_{min} will correspond to a set of reaction rates for each species.
- calculate how many elements are there in S_{min}. The number is stored with the variable name `numOfNCActive`.
- retrieve the index in the S_{min} for each physical cell, storing in a list called `NCZoneNumber`. Since `NCZoneNumber` is the order to retrieve a set of reaction rates from the S_{min}, the maximum possible value is `numOfNCActive-1`, and the minimum is `0`.

If the `growingCCM` switch is turned off (using ordinary CCM), the `NCZoneNumber` will be the final `ZoneNumber` as the `oldTableSize` is set to zero. Otherwise, we should add the previous size of table to the `NCZoneNumber`, by that means we obtain a growing table with increasing data entries as the simulation proceeds.

grouping

```
    // get ZoneNumber for each not calculated cells
    auto NCResults = getZoneNumber(notCalculatedZoneIndex, stringDigit_); //not calculated results
    scalarField NCZoneNumber = NCResults.ZoneNumber;
    List<word> gCCMZoneIndex(NCResults.ZoneIndex);
    label  numOfNCActive = NCResults.count;

    if(growingCCM_)
    {
        oldTableSize_ = curTableSize_;
        newTableSize_ = numOfNCActive;
        curTableSize_ = oldTableSize_ + newTableSize_;
    }
    else
    {
        oldTableSize_ = 0;
        newTableSize_ = numOfNCActive;
        curTableSize_ = newTableSize_;
    }


    // attribute ZoneNumber (notCalculatedCells)
    for(label icell = 0; icell < notCalculatedCells.size(); icell++)
    {
        label cellIndex = notCalculatedCells[icell];
        ZoneNumber[cellIndex] = NCZoneNumber[icell] + oldTableSize_;
    }
```

Then, we can move to the `ZoneNumber` function. As we mentioned before, `ZoneNumber` function should obtain a minimum set. An easiest way to do that is to

- build up a hash table.
- look up for each cell whether its reaction state or zone index is in the hash table.
- if not, store that in the hash table.

This strategy works pretty well for a program running on a single core. However, if we use parallel computing, each core will have its own minimum set S_{min}, and that set could be quite different for different cores (e.g. a zone index might be indexed differently in different cores). In addition, each core cannot lookup the reaction rates calculated by other cores, which could be a significant waste of

computing resources. Consequently, it is important to unify the zone indexes and gather all reaction rates globally with a better strategy.

In the CCM library, the implementation is divided into the following steps

- Execution of the look up procedure and obtain the minimum set S_{min}^{local} in each core.
- Concatenatation of S_{min}^{local} in the order of core number, and get a imperfect global minimum set $S_{min}^{global,i}$. Here the upper script i stands for "imperfection". There might be some repeating zone indexes in $S_{min}^{global,i}$ after this step.
- Iteratation over S_{min}^{local} and divide them to each core according to the remainder of the zone index divided by the number of cores. Since each zone index only has one remainder, identical zone indexes will finally be sent to the same core.
- Execution of the look up procedure again, and obtain the minimum set $S_{min}^{local,nr}$. Here the "nr" represents non-repeating.
- Gather $S_{min}^{local,nr}$ and obtain $S_{min}^{global,nr}$ (named as S_{min} for short).
- When doing the above steps, track each zone index and obtain its final order in S_{min}.

The codes below follows the above steps and works well.

grouping

```
    ZoneInfo getZoneNumber(List<word>& ZoneIndex, label stringLimit)
    {
        label nProc = Pstream::nProcs();
        label myProc = Pstream::myProcNo();

        // step 0 only oZoneIndex = ZoneIndex exists
        label oZoneSize = ZoneIndex.size();

        // step 1 group locally update
        // - 1) lZoneIndex
        // - 2) lZoneNumber (x)
        // - 3) lZoneNumberSum  (done)
        // - 4) lZoneNumberNum  (done)
        // - 5) lZoneNumberStart (done)
        // - 6) oinl (done, uncheck)

        List<word> lZoneIndex(0);
        List<label> oinl(oZoneSize);
        label      lZoneIndexCount(0);
        HashTable<scalar, word> HashMapL(2*oZoneSize);// note: the coefficient 2 might be improved later
        forAll(ZoneIndex, zi)
        {
            word target = ZoneIndex[zi];
            if (!HashMapL.found(target))
            {
                // set
                HashMapL.set(target,lZoneIndexCount);
                // append
                lZoneIndex.append(target);
                // other
                oinl[zi] = lZoneIndexCount;
                lZoneIndexCount++;
            }
            else
            {
                oinl[zi] = HashMapL.find(target)();
            }
        }
        List<label> lZoneNumberNum   = getNum  (lZoneIndex,     nProc, myProc);
```

```
        label       lZoneNumberSum   = getSum  (lZoneNumberNum, nProc, myProc);
        List<label> lZoneNumberStart = getStart(lZoneNumberNum, nProc, myProc);


        // step 2 go to global
        // - 1) gZoneIndex         (done)
        // - 2) gZoneNumber (x)
        // - 3) gZoneNumberSum      (done)
        // - 4) gZoneNumberNum      (no need)
        // - 5) gZoneNumberStart    (no need)
        // - 6) ling                (done)
        List<word> gZoneIndex = gather //List<word> ZoneIndex, label sum, List<label> start,label
    nProc, label curProc)
        (
            lZoneIndex,
            lZoneNumberSum,
            lZoneNumberStart,
            nProc,
            myProc
        );
        label gZoneNumberSum = lZoneNumberSum;

        List<label> ling(identity(lZoneNumberNum[myProc]));
        forAll(ling, li)
        {
            ling[li] += lZoneNumberStart[myProc];
        }

        // step 3 send to each core according to remainders
        // - 1) pZoneIndex         (done)
        // - 2) pZoneNumber (x)
        // - 3) pZoneNumberSum      (done)
        // - 4) pZoneNumberNum      (done)
        // - 5) pZoneNumberStart    (done)
        // - 6) ginp                (done)
        List<word> pZoneIndex(0);
        List<label> ginp(gZoneNumberSum, 0);

        label pcount = 0;

        List<label> whichCore(gZoneNumberSum,0);
        forAll(gZoneIndex, gi)
        {
            word tempIndex = gZoneIndex[gi];
            if (blockWiseRemainder(tempIndex,nProc,stringLimit) == myProc)
            {
                pZoneIndex.append(tempIndex);
                ginp[gi] = pcount;
                whichCore[gi] = myProc;
                pcount ++;
            }
        }

        List<label> pZoneNumberNum = getNum(pZoneIndex, nProc, myProc);
        label       pZoneNumberSum = gZoneNumberSum;
        List<label> pZoneNumberStart = getStart(pZoneNumberNum, nProc, myProc);

        // note: ginp is gathered for convenience
        reduce(ginp, sumOp<List<label>>());
        reduce(whichCore,sumOp<List<label>>());
        forAll(ginp, gi)
        {
            ginp[gi] += pZoneNumberStart[whichCore[gi]];
        }



        // step 4 lookup to avoid repeting numbers
```

```
        // - 1) pdZoneIndex       (done)
        // - 2) pdZoneNumber      (done)
        // - 3) pdZoneNumberSum     (done)
        // - 4) pdZoneNumberNum     (done)
        // - 5) pdZoneNumberStart    (done)
        // - 6) pinpd
        List<word> pdZoneIndex(0);
        List<label> pinpd(pZoneNumberNum[myProc]);
        HashTable<scalar, word> HashMapP(pZoneNumberSum/nProc*2);
        label pdCount = 0;
        forAll(pZoneIndex,pi)
        {
            word target = pZoneIndex[pi];
            if (!HashMapP.found(target))
            {
                pdZoneIndex.append(target);
                HashMapP.set(target,pdCount);
                pinpd[pi] = pdCount;
                pdCount++;
            }
            else
            {
                pinpd[pi] = HashMapP.find(target)();
            }
        }

        List<label> pdZoneNumberNum = getNum(pdZoneIndex, nProc, myProc);
        label pdZoneNumberSum = getSum(pdZoneNumberNum, nProc, myProc);
        List<label> pdZoneNumber = identity(pdZoneNumberSum);
        List<label> pdZoneNumberStart = getStart(pdZoneNumberNum, nProc, myProc);
        forAll(pinpd, pi)
        {
            pinpd[pi] += pdZoneNumberStart[myProc];
        }
        pinpd = gather(pinpd, pZoneNumberSum, pZoneNumberStart, nProc, myProc);



        // step 5 gather all ZoneIndex
        // - 1) gpdZoneIndex       (done)

        // step6 get ZoneNumber for each cell
        List<word> gpdZoneIndex = gather(pdZoneIndex, pdZoneNumberSum, pdZoneNumberStart, nProc,
    myProc);
        label gpdZoneNumberSum = pdZoneNumberSum;

        scalarField ZoneNumber(oZoneSize);
        forAll(ZoneNumber, zi)
        {
            label index = oinl[zi];
            index       = ling[index];
            index       = ginp[index];
            index       = pinpd[index];
            ZoneNumber[zi] = pdZoneNumber[index];
        }


        return ZoneInfo(ZoneNumber,gpdZoneIndex,gpdZoneNumberSum);

    }
```

3.2.4 The attribution of reaction rates

After obtaining the `ZoneNumber` (i.e. the order in the table of reaction rates for one cell), we can iterate over each cell and retrieve that order. Then we can ask for the table of reaction rates (`old_gRRCCM` in the program) for the value of reaction rates. This process is achieved with the code

block below in the header file `attributeRR.H`.

assigning

```
    const volScalarField& rho = this->mesh().objectRegistry::lookupObject<volScalarField>("rho");
    forAll(this->mesh().C(), icell)
    {
        const label izone = ZoneNumber[icell];
        for(label i=0; i < this->nSpecie(); i++)
        {
            this->RR_[i][icell] = old_gRRCCM_[i][izone]*rho[icell];
        }
    }
```

3.3 Adding growing features

In this section, some key pieces of code for the growing feature are introduced. Actually, the readers of interest might find far more lines in the attached code file to initialize the field and manage the index. Those lines are also important but are not listed here for simplicity.

3.3.1 Additional indexing operations

With growing CCM, we intend to save the computation time for mapping between zone indexes and their reaction rates, so that we can reuse those reaction rates in future steps. The recording of zone indexes and their reaction rates will be discussed in Subsection 3.3.2. Here, we mainly focus on how to reuse some of the reaction rates.

As we mentioned before, four main stages, including indexing, grouping, solving, and assigning, are executed one by one. It is evident that a cell should have an zone index in order to retrieve an "old" reaction rate, but it cannot enter the grouping stage, which will lead to the calculation of a new reaction rate. Consequently, an easy way to achieve that is to add a pre-lookup right after obtaining the zone index, with the codes given as follows.

pre-lookup

```
  // attribute notCalculatedCells
    if (growingCCM_)
    {
        ...
        for (label icell = 0;icell < fieldSize; icell++)
        {
            word index = ZoneIndex[icell];
            if (totalHashTable_.found(index))
            {
                ZoneNumber[icell] = totalHashTable_.find(index)();
                ...
            }
            else
            {
                notCalculatedCells.append(icell);
            }

        }
    }
```

Here, the program asks `totalHashTable` whether a zone index has been calculated before. The `totalHashTable` is a hashtable, the key of which is a zone index, while the value is the order (`ZoneNumber` in the program) of that zone index in the table storing reaction rates. If the zone index is calculated before, the order will be returned so that we can access this reaction rate by asking `old_gRRCCM[orderNumber]`, without calculating the reaction rates again. If not, the cell number will be stored to a list called `notCalculatedCells`.

Please note that if the switch `growingCCM` is turned off, all the cells should be sent to grouping and further stages to obtain reaction rates, so the `notCalculatedCells` list, in this case, includes all cells, which is a list from 0 to $N-1$ where N is the number of cells.

3.3.2 Additional grouping/assigning operations

After the grouping process, the following code piece is used to update `totalHashTable` describing the map between zone indexes and their relevant orders in the reaction rate table. In addition, the new zone index is appended to the list of zone indexes (`old_gCCMZoneIndex`). Please note that the order of zone indexes in `old_gCCMZoneIndex` is the same as that in the `old_gRRCCM`, storing the reaction rates. The first zone index corresponds to the first reaction rate, and so on and so forth.

additional operations after grouping

```
    if(growingCCM)
    {
        forAll(gCCMZoneIndex, i)
        {
            totalHashTable.set(gCCMZoneIndex[i],oldTableSize + i);
        }
        for (label i=0;i!=newTableSize;i++)
        {
            old_gCCMZoneIndex[oldTableSize+i] = gCCMZoneIndex[i];
        }
    }
```

After the solving process, the following code piece is used to append newly calculated reaction rates to the table of reaction rates.

additional assigning operations

```
    if(growingCCM_)
    {

        forAll(gRRCCM,specie)
        {
            for (label i=0; i!=newTableSize_; i++)
            {
                old_gRRCCM_[specie][oldTableSize_+i] = gRRCCM[specie][i];
            }
        }
    }
```

Chapter 4

The usage of the CCM library

The usage of CCM and the growing CCM library will be demonstrated in this chapter, with very detailed explanations of parameter settings in the CCM dictionary.

Here, we will use the SandiaD-LTS case in the OpenFOAM tutorial as an example. The author assumes that you already have a proper combustion case, and will guide you to modify that case to cater to the requirements of a case using the CCM library. For users without any cases prepared, the default case is also okay for a trial.

4.1 Prerequisite

The CCM library is designed to be applied to transient cases with fine grids. Then, we can assume that the fuel and air are well-mixed in each cell. Readers interested in applying this model in RANS simulations with coarser grids are referred to the orginal paper [11] after reading the programming guide part of this report.

In this section, we shall make some preparations. First, source the OpenFOAM bash file, and copy the `SandiaD_LTS` case to your `run` directory with commands

```
cd $FOAM_RUN
cp -r $FOAM_TUTORIALS/combustion/reactingFoam/RAS/SandiaD_LTS/ .
cd SandiaD_LTS
./Allrun # optional
```

This will copy the `SandiaD_LTS` case to your `run` folder. If you are unfamiliar with this case, you can run the `Allrun` script and have a look at the results. This script will simulate a steady flow without reactions as an initial field for the hot case, then turn on the combustion and get the final results. The `TDAC` library is used to accelerate the chemistry in this case.

For more detailed introductions and physical backgrounds of this case, the readers are referred to a report of this course written by Bertsch [13]. Only a short description is presented here. SandiaD-LTS simulates the combustion process in Sandia burner [14], presented in Fig. 4.1 at a Reynolds number of 22400. The burner injects from two concentric cylinders and inner nozzle located in the center of two cylinders. The inner nozzle called main stage (with the patch name `inletCH4`) will inject rich methane ($\phi = 3$) and air at ambient temperature, while the inner cylinder called pilot stage (with the patch name `inletPilot`) injects the hot burnt gas from methane combustion. As for the outer cylinder, it injects cold air at ambient temperature. Ambient pressure is applied to all inlets. As for the temperature, the pilot stage injects at 1880 K, while the fuel and air in the main/air stage inject at ambient temperature. The injection velocity for pilot and main stages are 11.4 m/s and 49.6 m/s, respectively.

In this report, we use the default mesh of the original case for all simulations in this report, consisting of a wedge domain with 5080 hexahedral mesh. The `k-Epsilon` model is chosen as the turbulence model. The well-stirred reaction model (`laminar`) and GRI3.0 mechanism [2] is adopted as the combustion model and the reaction mechanism, respectively.

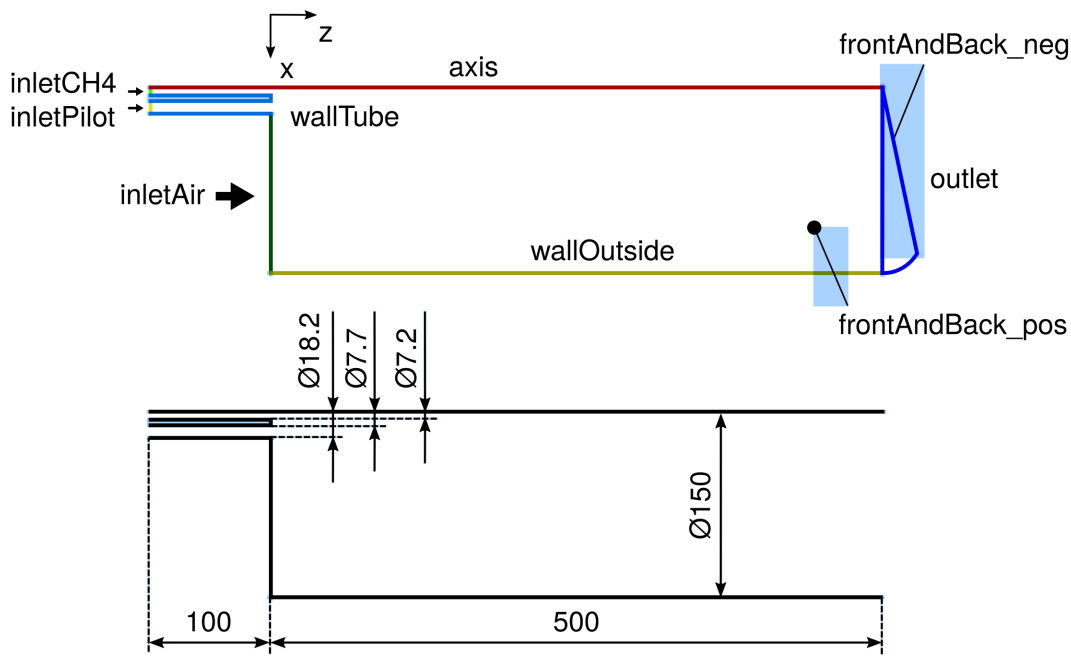

Figure 4.1: A schematic of Sandia burner

4.2 Compile the code

Go to the code directory and execute the following command.

```
./Allwmake
```

Then you are all set.

4.3 Prepare a cold case

Since we want to simulate a transient combustion case using the CCM method. We need a cold flow solution to serve as the initial condition. After that, we can turn on the chemistry to start the reacting simulation.

We might start with a steady cold case to save time. The `chemistry` entry in the `chemistryProperties` should be turned off at this stage. Then we modify the gradient scheme (`gradSchemes`) to `cellLimited Gauss linear 1` to improve convergence. After running `reactingFoam` for 10000 steps, the case will be almost ready.

Execute the commands below

```
cd $FOAM_RUN
rm -r Sandia*
cp -r $FOAM_TUTORIALS/combustion/reactingFoam/RAS/SandiaD_LTS/ .
cd SandiaD_LTS
rm -r Allrun Allclean constant/radiation* constant/boundaryRadiation*
mv 0.orig 0

foamDictionary constant/chemistryProperties -entry chemistry -set off
foamDictionary constant/combustionProperties -entry combustionModel -set laminar
foamDictionary constant/chemistryProperties -entry chemistry -set off
foamDictionary system/fvSchemes -entry gradSchemes.default
```

```
... -set "cellLimited Gauss linear 1"
foamDictionary system/fvSolution -entry relaxationFactors -remove
cp -r $FOAM_TUTORIALS/multiphase/interFoam
.../laminar/damBreak/damBreak/system/decomposeParDict system/decomposeParDict
foamDictionary system/decomposeParDict -entry method -set scotch
foamDictionary system/controlDict -entry endTime -set 15000
foamDictionary system/controlDict -entry writeFormat -set ascii
foamDictionary system/controlDict -entry timePrecision -set 18
```

to guide you through the above processes automatically.

In addition, one can paste `constant/chemistryProperties` below to replace the original `constant/chemistryProperties` file, and add the relaxationFactors as the followings to `system/fvSolutions`

relaxationFactors

```
relaxationFactors
{
    fields
    {
        p               0.4;
    }
    equations
    {
        U               0.7;
        ".*"            0.7;
    }
}
```

Then we can start the simulation for the steady cold case. Run the following commands

```
blockMesh
decomposePar -force
mpirun -n 4 reactingFoam -parallel
reconstructPar -latestTime
```

and wait until the simulation ends.

After the simulation ends, the case can be modified to a transient reacting case using commands

```
reconstructPar -latestTime
cp -r 0/Ydefault 15000/Ydefault
foamDictionary system/fvSchemes -entry ddtSchemes.default -set backward
foamDictionary system/controlDict -entry adjustTimeStep -set false
foamDictionary system/controlDict -entry endTime -set 15001 &&
foamDictionary system/controlDict -entry writeInterval -set 0.025
foamDictionary system/controlDict -entry deltaT -set 3e-5
foamDictionary system/controlDict -entry libs -set "(libCCMMars.so)"
sed -i s/"15001"/"15000.05"/g system/controlDict
foamDictionary constant/combustionProperties -entry combustionModel -set laminar
```

Then your case will be all set. We still need some modifications in the `constant/chemistryProperties` file before we can continue. These will be discussed in Section 4.4.

4.4 The usage of the CCM library

Delete all the contents in the original `constant/chemistryProperties` and add the following contents into that file.

chemistryProperties

```
/*--------------------------------*- C++ -*----------------------------------*\
  =========                 |
  \\      /  F ield         | OpenFOAM: The Open Source CFD Toolbox
   \\    /   O peration     | Website:  https://openfoam.org
    \\  /    A nd           | Version:  7
     \\/     M anipulation  |
\*---------------------------------------------------------------------------*/
FoamFile
{
    version         2;
    format          ascii;
    class           dictionary;
    location        "constant";
    object          chemistryProperties;
}


chemistryType
{
    solver          ode;
    method          CCM;
}


chemistry       on;

initialChemicalTimeStep 1e-07;

odeCoeffs
{
    solver          seulex;
    absTol          1e-08;
    relTol          1e-1;
}

CCM
{
    growingCCM           off;
    autoCorrection       off;
    correctionErrorLimit   0.10;
    correctionSpeciesCutOff  1e-6;
    stringDigit          4; //default to be 4
    maxTableSize      500000;
    hashTableSize     1500000;
    deleteRatio       0.5;



    ratioOxygenToCarbonElementInFuel 0;
    min:max:SpanZoneJe 0 5 0.01;
    min:max:SpanZoneT 300 2200 1;
    min:max:SpanZoneXi 0 1 0.025;
    chemicallyFrozenT 300;
    maxFlammabilityPhi 20;

    CCMspecies
    {
        CH4             0 0.2 0.001;
        N2              0 1 0.001;
        HO2             0 1e-3 1e-6;
    }
}


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

We will explain the settings here in the next chapter. Here we just leave as it is. After further adjusting the `endTime`, `deltaT`, switching off the `adjustTimeStep` option, and adding the library to the `controlDict` file, we can start the reacting case simulation and wait for the final results. Run the following commands

```
decomposePar -force
mpirun -n 4 reactingFoam -parallel
reconstructPar -latestTime
```

to guide you to complete those processes.

View the results with paraview by the following command

```
paraFoam
```

and you will see a flame more or less like Fig. 4.2.

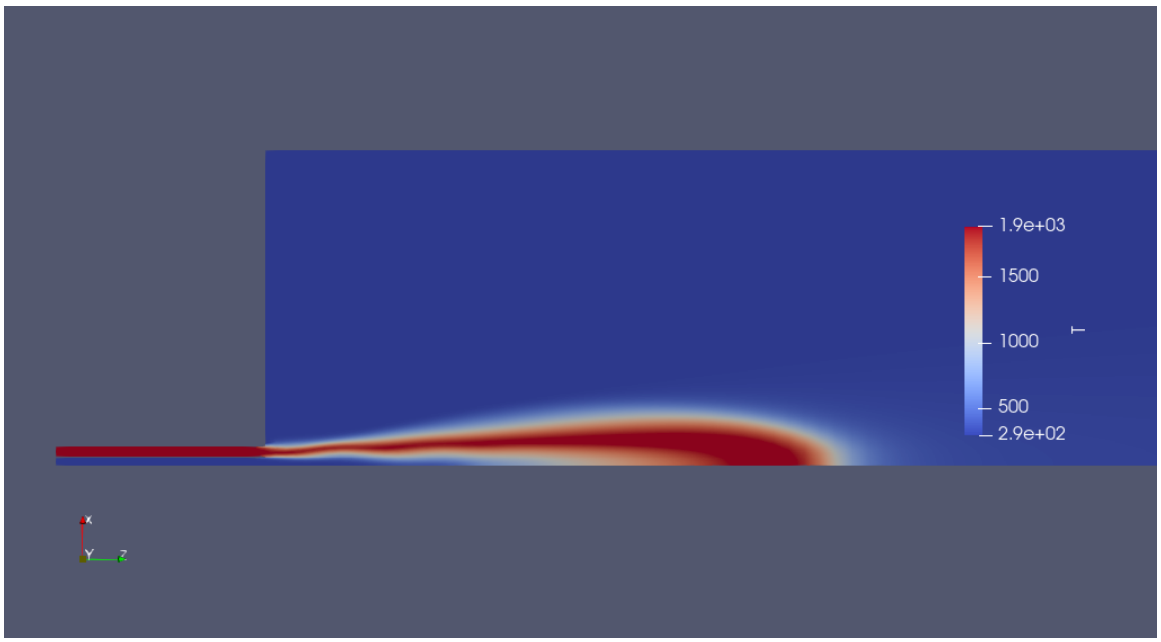

Figure 4.2: Temperature distribution in the SandiaD-LTS flame

You can redo this step, but run with the growing CCM feature. You are expected to experience a triple speedup. Just follow the commands

```
foamListTimes -time "15000.001:" -rm
rm -r pro*
foamDictionary constant/chemistryProperties -entry CCM.growingCCM -set on
mpirun -n 4 reactingFoam -parallel
reconstructPar -latestTime
```

to guide you to clean the old running files and restart a simulation with the growing CCM feature.

4.5 Setting the CCM dictionary

In this subsection, we will examine the settings of the `chemistryProperties` file in a case using the CCM library.

First, we should set the `chemistryType` dictionary. The choice of solver includes `noChemistrySolver`, `EulerImplicit`, and `ode`. It is evident that we have to choose between `EulerImplicit` and the `ode`. Note that if we are going to use a chemistry mechanism with many

species and reactions, this solution system is usually so stiff that an implicit method might not work very well. Consequently, we usually choose the stiff ODE solver `ode`. For the method option, the CCM library we have introduced in Chapter 3 is chosen.

When simulating a reacting case, the `chemistry` should be activated so that the OpenFOAM will solve the reaction system. Also, some relevant parameters for the chemistry calculation should be set. The `initialChemicalTimeStep` is the initial time step for the chemistry solution. Usually, it should be much smaller than the time step of the flow. The time step to compute reaction source term is usually much smaller than the time step of the flow. During the solution process, the chemistry solving time step might be adjusted automatically to ensure the stability of the solution, and gradually complete a full flow time step at the same time. As for the `odeCoeffs`, `solver`, absolute tolerance (`absTol`) and relative tolerance (`relTol`) should be specified. Please note that we choose a high `relTol` here, so the final results might not be very accurate. The purpose is to show the features of the library in the smallest possible time. The readers are recommended to choose a smaller value to obtain accurate results.

Then we can go to the most unique part of the `chemistryProperties` file for a case using CCM. The meaning of some important parameters are listed as follows

- `growingCCM`: the switch to turn on or off the growing CCM feature. Set `on` to turn on the feature, and vice versa for the `off` option.
- `stringDigit`: the number of digits that the program adopts to represent a zone index for each dimension.
- `maxTableSize`: the growing CCM table size, which will be preallocated in memory. This setting will only take effect with the `growingCCM` switch on.
- `hashTableSize`: the CCM utilizes the hashtable to achieve fast lookup. The size of the hashtable should be specified, and it should be larger than the `maxTableSize`. This setting will only take effect with the `growingCCM` switch on.
- `deleteRatio`: the ratio of data entries to be removed once the table is full. This setting will only take effect with the `growingCCM` switch on.
- `min:max:SpanZoneJe`: setting the minimum/ maximum/ step value for the equivalence ratio ϕ, which is one of the CCM variables
- `min:max:SpanZoneT`: setting the minimum/ maximum/ step value for the temperature.
- `min:max:SpanZoneXi`: seting the mimimum/ maximum/ step value for the scalar dissipation ratio χ, which is one of the CCM variables
- `CCMspecies`: this dictionary will set the minimum/ maximum/ step values for user specified species for grouping and lookup. Please make sure that the choice of species set should be enough to describe the reaction rate manifold of your case.
- `chemicallyFrozenT`: the threshold temperature below which a cell is considered to be non-reactive.
- `maxFlammabilityPhi`: the threshold equivalence ratio above which a cell is considered to be non-reactive.

Some settings are still under development, and might not be available before the end of this course. But the developer decides to keep it there in case someone might make use of it in the future. This will include

- `autoCorrection`: after turning on this switch, the program will check whether the current species mass fraction of a cell is close to that of an old cell stored in the reaction rate every time when a reusing operation is attempted. If the error is too large, the program will recalculate the reaction rates. This setting will only take effect when `growingCCM` switch is turned on.

Table 4.1: Base case benchmarks

Example	Time t
standard	1154.89 s
TDAC	232.11 s
CCM	1352.98 s
gCCM	351.11 s
gCCM+HO2index	454.74 s

- `correctionErrorLimit`: below which the error between the mass fractions of a new cell and an old cell is considered acceptable. This setting will only take effect when `growinCCM` and `autoCorrection` switches are turned on.
- `correctionSpeciesCutOff`: below which the mass fraction of a species is considered to be unimportant. This will only take effect when `growingCCM` and `autoCorrection` switches are turned on.

4.6 Recommended procedures

The users are suggested to follow the steps below to complete the case setup.

- Prepare a cold case with the species transportation information. Remember to turn off the chemical reaction.
- Please turn on the chemistry. And check by running with `standard` library for a few steps (Modify `constant/chemistryProperties.chemistryType.method` to `standard`).
- Then setup your CCM dictionary according to the instructions in Chapter 4.5. Modify the `chemistryType.method` to `CCM`.
- Run a few steps and stop. Check the terminal output to see the maximum/ minimum value of some key variables and species. Adjust them according to your needs.
- Restart running and check the results.

4.7 Benchmark and analysis

We run the `SandiaD_LTS` case for 0.1 physical second with the `standard`, `TDAC`, and `CCM` library. We also test `growing CCM` with and without adding HO_2 to the CCM variables. Other settings are identical. The time costs are summarized in Table 4.1. The TDAC is the fastest, followed by the growing CCM. In this case, the ordinary CCM is even slower than the standard chemistry model, this might owing to very small mesh size in the phase space and coarse grid in a 2D physical space, so that the additional lookup operations compensate the time saved by fewer chemistry calculations. In a larger 3D case, CCM will be significantly faster than the standard chemistry mode. And growing CCM will be even faster with some sacrifice to the accuracy.

Then we examine the accuracy of the acceleration methods in Table 4.1 by showing some key field distributions after running 0.05 seconds physical time.

As shown in Fig 4.3 and Fig 4.4, the temperture and heat release rate distributions are almost identical for all methods comopared to the standard chemistry results. TDAC is the worst-performing method, predicting a relatively shorter flame.

We also compared some key species field, including HCO, H_2O_2, and HO_2. Generally, both ordinary CCM and growing CCM predict slightly better HCO field than TDAC. However, growing CCM over predicts H_2O_2 field. Adding HO_2 as another CCM variable will slightly improve the results, but the discrepancy is still significant. It is worthwhile to note that the ordinary CCM

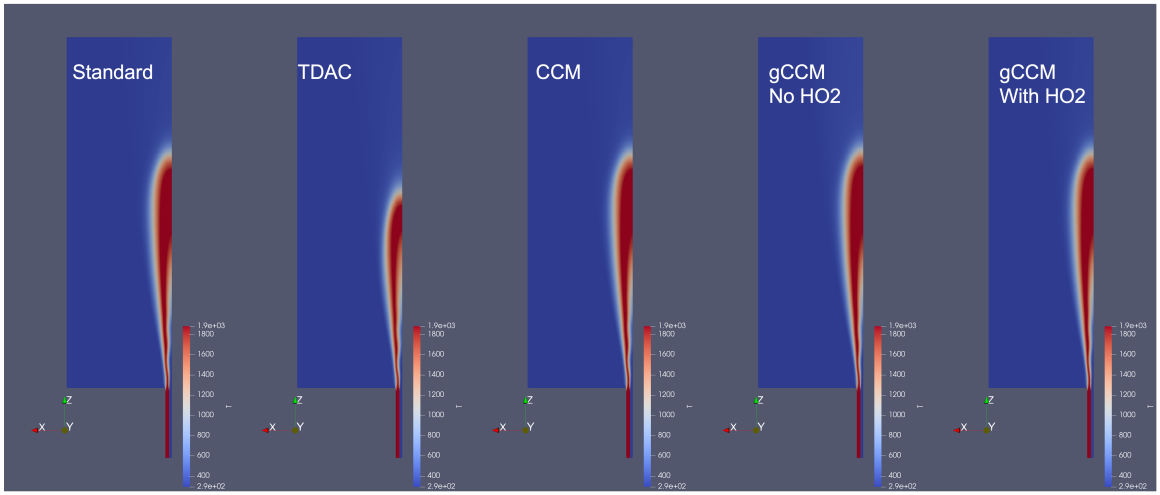

Figure 4.3: Temperature distributions in the SandiaD-LTS flame

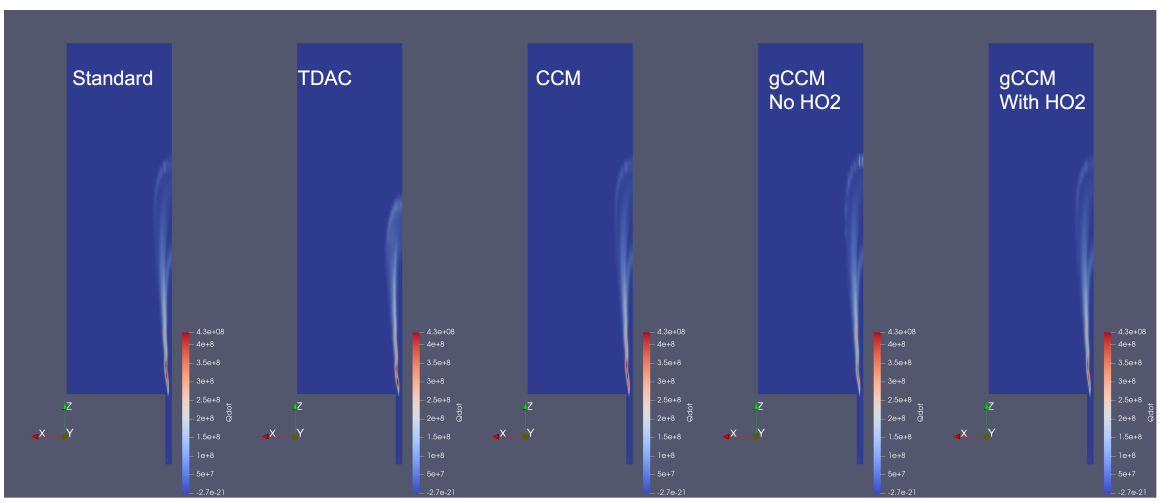

Figure 4.4: Heat release distributions in the SandiaD-LTS flame

performs better than TDAC in predicting H_2O_2. It indicates that introducing "history memory" will boost the simulation speed, but the accuracy might suffer by looking up a inaccurate history data.

Similar issues appear in HO_2 field, the ordinary CCM performs slightly better than TDAC. The growing CCM predicted fairly different HO_2 distribution downstream. By adding HO_2 as another CCM variable, this issue can be partly resolved. But it will slightly increase the simulation time.

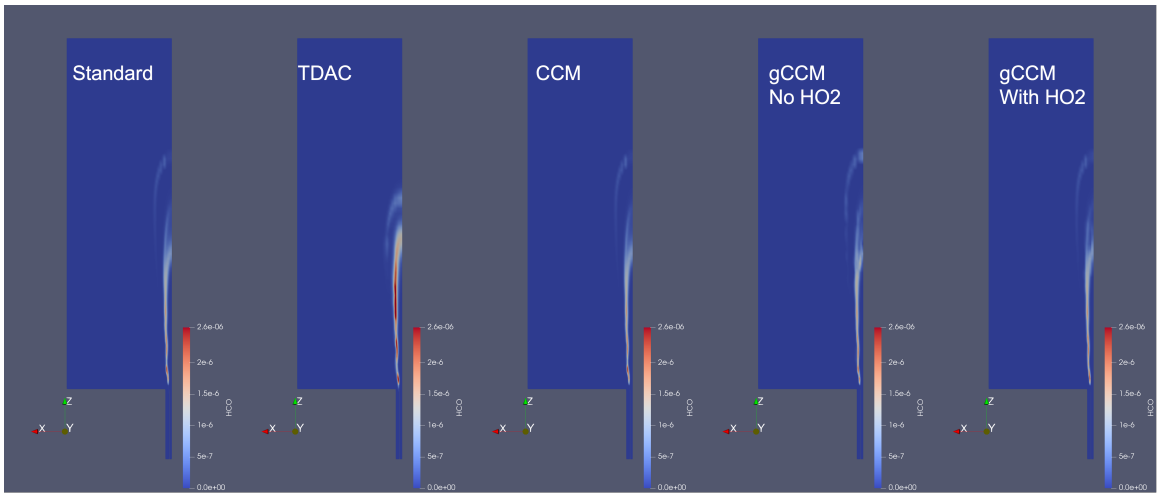

Figure 4.5: HCO distributions in the SandiaD-LTS flame

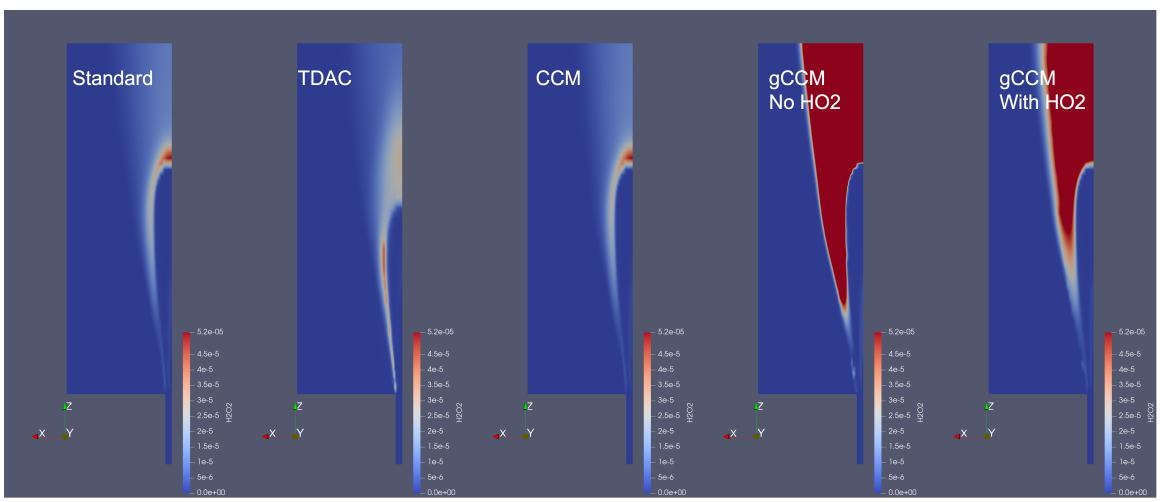

Figure 4.6: H_2O_2 distributions in the SandiaD-LTS flame

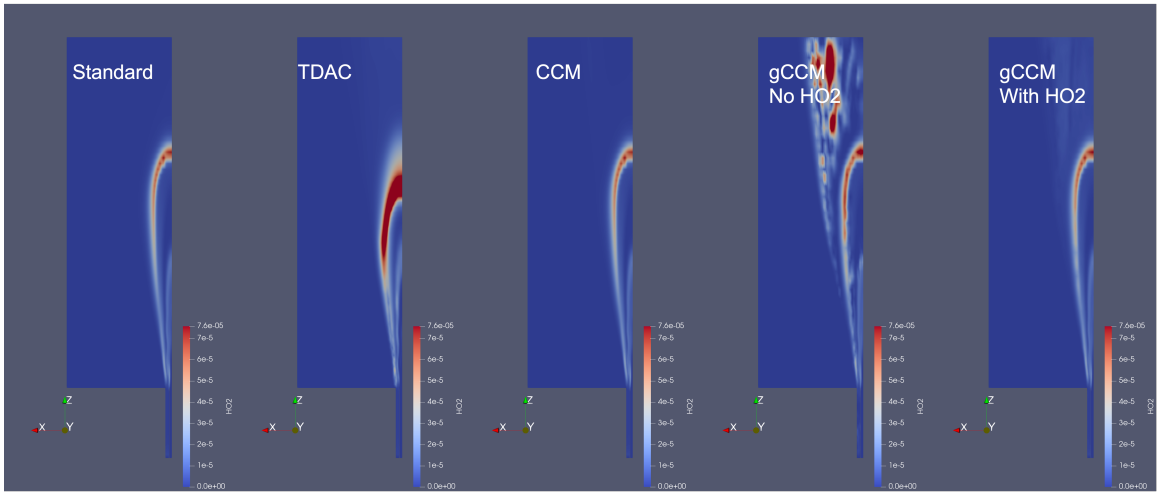

Figure 4.7: HO_2 distributions in the SandiaD-LTS flame

Bibliography

[1] H. K. Versteeg and W. Malalasekera, *An introduction to computational fluid dynamics: the finite volume method*. Pearson education, 2007.

[2] "Gri-mech 3.0." http://www.me.berkeley.edu/gri_mech/. Accessed: 2022-11-15.

[3] A. Wang, H. H. Lou, D. Chen, A. Yu, W. Dang, X. Li, C. Martin, V. Damodara, and A. Patki, "Combustion mechanism development and cfd simulation for the prediction of soot emission during flaring," *Frontiers of Chemical Science and Engineering*, vol. 10, no. 4, pp. 459–471, 2016.

[4] F. Contino, H. Jeanmart, T. Lucchini, and G. D'Errico, "Coupling of in situ adaptive tabulation and dynamic adaptive chemistry: An effective method for solving combustion in engine simulations," *Proceedings of the Combustion Institute*, vol. 33, no. 2, pp. 3057–3064, 2011.

[5] J. Van Oijen, A. Donini, R. Bastiaans, J. ten Thije Boonkkamp, and L. De Goey, "State-of-the-art in premixed combustion modeling using flamelet generated manifolds," *Progress in Energy and Combustion Science*, vol. 57, pp. 30–74, 2016.

[6] Y. Zhang, H. Wang, A. Both, L. Ma, and M. Yao, "Effects of turbulence-chemistry interactions on auto-ignition and flame structure for n-dodecane spray combustion," *Combustion Theory and Modelling*, vol. 23, no. 5, pp. 907–934, 2019.

[7] L. Liang, J. G. Stevens, S. Raman, and J. T. Farrell, "The use of dynamic adaptive chemistry in combustion simulation of gasoline surrogate fuels," *Combustion and flame*, vol. 156, no. 7, pp. 1493–1502, 2009.

[8] S. B. Pope, "Computationally efficient implementation of combustion chemistry using in situ adaptive tabulation," *Combustion Theory and Modelling*, 1997.

[9] C. K. Law, *Combustion physics*. Cambridge university press, 2010.

[10] X.-S. Bai, *Turbulent combustion*. Lund University, 2021.

[11] M. Jangi and X.-S. Bai, "Multidimensional chemistry coordinate mapping approach for combustion modelling with finite-rate chemistry," *Combustion Theory and Modelling*, vol. 16, no. 6, pp. 1109–1132, 2012.

[12] M. Gadalla, "Implementation of analytical jacobian and chemical explosive mode analysis (cema) in openfoam," in *Proceedings of CFD with OpenSource Software* (N. H, ed.), 2021.

[13] M. Bertsch, "Description of the reacting flow solver fgmfoam," in *Proceedings of CFD with OpenSource Software* (N. H, ed.), 2019.

[14] "Sandia/tud piloted ch4/air jet flames." https://tnfworkshop.org/data-archives/pilotedjet/ch4-air/. Accessed: 2022-12-13.

Study questions

1. What is Chemistry Coordinate Mapping (CCM) and the basic procedure of that?
2. What is the basic theory behind CCM, or generally the basic theory behind a chemistry acceleration method?
3. How do you choose some key variables to construct a reaction rate manifold for your simulation case?
4. How do you use CCM and growing CCM to accelerate the combustion simulation?
5. How can you modify a chemistry library?