Implementation of Analytical Jacobian and Chemical Explosive Mode Analysis (CEMA) in OpenFOAM

Developed for OpenFOAM-v2006
Requires: PyJac

Author:
Mahmoud GADALLA
Aalto University,
School of Engineering.
mahmoud.gadalla@aalto.fi
gadalla.mah@gmail.com

Peer reviewed by:
Konstantinos MISSIOS, Roskilde Univ.
Islam KABIL, Connecticut Univ.
Saeed SALEHI, Chalmers Univ.
Ville VUORINEN, Aalto Univ.

Library on Github: https://github.com/Aalto-CFD/CEMAFom

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

Disclaimer: This is a student project work, done as part of a course where OpenFOAM and some other OpenSource software are introduced to the students. Any reader should be aware that it might not be free of errors. Still, it might be useful for someone who would like 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 18, 2022
Learning outcomes

The reader will learn:

**How to use it:**
- how to use the reactingFoam solver, with complete understanding of necessary dictionaries required for the solver computations.
- how to make use of dynamic binding for loading OpenFOAM libraries in the application level.

**The theory of it:**
- the theory of finite-rate chemistry in reactive flow modeling.
- the importance of system Jacobian and derivatives to solve stiff system of ordinary differential equations (ODEs) relevant to chemical source term computations.
- the theory of Chemical Explosive Mode Analysis (CEMA) for local combustion mode characterization.

**How it is implemented:**
- how StandardChemistryModel is implemented in OpenFOAM.
- how ODE and chemistryModel libraries communicate in the lower code level for solving chemical source terms.

**How to modify it:**
- how to modify the templated StandardChemistryModel library to accommodate for analytical Jacobian and CEMA functionalities.
Prerequisites

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

- Basic knowledge of OpenFOAM usage.
- How to run standard OpenFOAM tutorials with proper knowledge of case files and dictionaries.
- Basics of “Theoretical and Numerical Combustion”, Book by T. Poinsot and D. Veynante.
- Basics of “Turbulent Combustion”, Book by N. Peters.
- Familiarization with dynamical system theory, along with the article by Lu et al. 2010, Journal of Fluid Mechanics, 652, 45-64, https://doi.org/10.1017/S002211201000039X.
- Basic understanding of object oriented programming, particularly the class inheritance and polymorphism.
A Developed codes ........................ 60
  A.1 Tree structure .............................................. 60
  A.2 cemaPyjacChemistryModel library .......................................................... 62
    A.2.1 chemistryModel .................................................. 62
    A.2.2 chemistrySolver .................................................. 79
    A.2.3 Make chemistryModel ............................................. 82
  A.3 pyJac CMake .................................................. 91
  A.4 pyJac generic header files ........................................... 92
  A.5 Utilities ...................................................... 95
Nomenclature

**Acronyms**
- CEM Chemical Explosive Mode
- CEMA Chemical Explosive Mode Analysis
- CFD Computational Fluid Dynamics
- CPU Central Processing Unit
- DI Defectiveness Index
- DRY “Don’t Repeat Yourself” concept
- EI Explosion Index
- FVM Finite Volume Method
- GPU Graphics Processing Unit
- HRR Heat Release Rate
- NSE Navier-Stokes Equations
- PI Participation Index
- PISO Pressure-Implicit with Splitting of Operators
- SIMD Single-Instruction Multiple Data
- UML Unified Modeling Language

**English symbols**
- $\bar{c}_k$ Concentration of species $k$ ........................................ [kmol/m$^3$]
- $\bar{R}$ Universal gas constant ........................................... [J/(kmol $\cdot$ K)]
- $\mathbf{u}$ Velocity vector .................................................. [m/s]
- $\mathbf{J}$ Jacobian matrix of stiff chemistry ODE system ................. [-]
- $A_r$ Pre-exponential factor for elementary reaction $r$ ...................... [varied]
- $c_{pk}(T)$ Isobaric temperature-dependent specific heat capacity for species $k$ ....... [J/(K $\cdot$ kg)]
- $D$ Mass diffusivity .......................................................... [m$^2$/s]
- $E_a$ Activation energy ......................................................... [J/kmol]
**Nomenclature**

\[ H \] Molar enthalpy ................................................. [J/kmol]
\[ h^f \] Enthalpy of formation (specific) ................................. [J/kg]
\[ h_s \] Sensible enthalpy (specific) ................................... [J/kg]
\[ h_t \] Total enthalpy (specific) ....................................... [J/kg]
\[ k_r \] Rate constant of reaction \( r \) ................................. [varied]
\[ M \] Number of elements in species composition of chemical mechanism .................. [-]
\[ p \] Pressure .......................................................... [Pa]
\[ q_r \] Progress rate of reaction \( r \) ................................ [kmol/(m\(^3\)⋅s)]
\[ S \] Molar entropy ......................................................... [J/(kmol⋅K)]
\[ T \] Temperature ......................................................... [K]
\[ t \] Time ................................................................. s
\[ u_L \] Laminar burning velocity ......................................... [m/s]
\[ W_k \] molecular weight of species \( k \) ........................... [kg/kmol]
\[ Y_k \] Mass fraction of species \( k \) .................................. [-]
\[ \text{CH}_4 \] Methane species ................................................ [-]
\[ \text{CO}_2 \] Carbon dioxide species ................................... [-]
\[ \text{H}_2\text{O} \] Water (oxidane) species ................................. [-]
\[ \text{N}_2 \] Nitrogen species ............................................... [-]
\[ \text{O}_2 \] Oxygen species ................................................ [-]

**Greek symbols**

\( \bar{\phi} \) Volumetric face flux in CFD ................................ [m\(^3\)/s]
\( \Phi \) Thermochemical state vector comprising temperature and species mass fractions ........................ [varied]
\( \Phi_c \) Thermochemical state vector comprising temperature and species concentrations .... [varied]
\( \tau \) Viscous stress tensor ................................................. [N/m\(^2\)]
\( \delta t_{\text{CFD}} \) CFD timestep ............................................ [s]
\( \delta t_{\text{ODE}} \) ODE timestep ............................................ [s]
\( \delta L \) Thermal thickness of laminar premixed flame ......................... [m]
\( \gamma \) Specific heat ratio ................................................ [-]
\( \lambda_{\text{exp}} \) Explosive eigenvalue of Jacobian matrix .................. [-]
\( \mu \) Fluid dynamic viscosity ........................................... [Pa⋅s]
\( \nabla \) Gradient operator ................................................ [1/m]
\( \nu'_{k,r} \) Molar stoichiometric coefficients of species \( k \) in reaction \( r \) products ........... [-]
\( \nu'_{k,r} \) Molar stoichiometric coefficients of species \( k \) in reaction \( r \) reactants \([-]\)

\( \nu_{k,r} \) Net molar stoichiometric coefficients of species \( k \) in reaction \( r \) \([-]\)

\( \rho \) Fluid density \([\text{kg/m}^3]\)

\( \dot{\omega}_h \) Energy source term \([\text{J/(m}^3 \cdot \text{s})]\)

\( \dot{\omega}_k \) Chemical source term of species \( k \) \([\text{kg/(m}^3 \cdot \text{s})]\)

**Superscripts**

\( N_r \) Number of reactions

\( N_{sp} \) Number of species

**Subscripts**

0 reference value

bw backward

fw forward

L Laminar flame

f formation

h enthalpy

k species index

r reaction index

s sensible

t total
Chapter 1

Background and Introduction

Reactive flow modeling plays an essential role in the field of energy and combustion research [1]. Based on the constitutive laws of continuum mechanics and particularly in computational fluid dynamics (CFD), mathematical models are derived. Conservation laws for linear momentum, represented by the Navier-Stokes equations (NSE), are solved together with the conservation of total mass (i.e. continuity) and energy governing equations. In a chemically reactive system, convection and diffusion processes of the existent chemical species in a computational domain are governed by scalar transport equation for each species. More details on the modeling of turbulent reactive flows can be revisited from the textbooks by Poinsot and Veynante [1] and by N. Peters [2]. The species production and consumption due to chemical reactions are usually represented via source terms plugged into the species and energy equations. In Figure 1.1, we show a volume-rendered visualization of spray combustion simulation using OpenFOAM, where fuel is first injected into the domain, then it starts dissociation into lighter radicals and intermediate species, indicating the onset of chemical reactions. Further downstream, and after specific induction time, chain of exothermic reactions take place where heat release is generated with a thermal runaway, hence combustion.

Figure 1.1: Volume-rendered visualization of turbulent reactive flow.

It is of particular interest to identify and study the structure of reaction fronts, defined as the leading edge of the reaction zone, by experimental [3, 4] and computational [5, 6] means. Accordingly, the chemical explosive mode analysis (CEMA) has been developed by Lu et al. [6] with motivations from the earlier concept of computational singular perturbation [7]. CEMA is considered a versatile computational diagnostics tool that enables the detection of various critical combustion features including reaction fronts, flame stabilization mechanisms, and auto-ignition and extinction zones [5]. Recently, it was further extended to account for diffusion [8] and evaporation [9] processes.
to investigate their roles toward promoting or inhibiting chemistry and auto-ignition.

In this project, we aim at implementing the basic formulations of CEMA, which are responsible for the identification of pre- and post-ignition zones and subsequently the reaction fronts, into OpenFOAM. Such an implementation is considered the crucial part in the analysis tool development since it identifies the existence of chemical explosive mode (CEM). Once a CEM is detected, all subsequent developments can be implemented including the

- explosion index (EI): which quantifies the contribution of chemical species to a CEM,
- participation index (PI): which quantifies the contribution of chemical reactions to a CEM,
- defectiveness index (DI): which measures the defectiveness of thermo-chemical Jacobian matrix that might lead into solution singularity, and
- projection of local source terms including chemistry, diffusion, and possibly spray droplet evaporation, onto CEM to quantify and compare their roles toward promoting or inhibiting chemical reactions, hence the identification of local combustion modes.

With relevance to the latest point, in combustion systems, it is critically important to understand whether the mixture is burning in a deflagrative (i.e. slow combustion or a premixed flame propagation) or auto-ignitive (i.e. spontaneous combustion with an ignition front) type. In order to characterize this, CEMA becomes useful which offers an eigenvalue analysis tool to estimate the local mode of combustion [8]. In a turbulent spray assisted combustion, both of the spontaneous and deflagration modes of combustion are known to be present, which is the subject of ongoing studies.

Another important note is that, while the derivation of CEMA is based on an analytical formulation of the thermo-chemical Jacobian matrix comprising pressure, temperature and species concentrations [6], in OpenFOAM v2006 the Jacobian matrix is only semi-analytical. Accordingly, the Python-based open-source library pyJac [10] is herein utilized. The pyJac library is a source code generator which creates C subroutines (or possibly in other programming languages) for analytical Jacobian matrix evaluation provided a particular chemical kinetic mechanism. Computationally, pyJac is optimized to minimize operations and memory usage, in addition to being thread-safe and can be easily parallelized on single-instruction multiple data (SIMD) processors such as graphics processing units (GPUs). Moreover, the performance of pyJac on central processing units (CPUs), as well as GPUs, is found to outperform other existing libraries for both finite-differencing and analytical evaluation of the Jacobian matrix. Despite the fact that other OpenFOAM releases feature implementations for analytical Jacobian evaluation, pyJac was found to be superior particularly when it comes to computing the chemical source terms via stiff ordinary differential equation (ODE) integration with tight tolerances, which we encounter in this project. As a result, pyJac will be herein employed for the generation of the analytical Jacobian matrix, and the corresponding CEMA results will be demonstrated and compared against those resulting from semi-analytical OpenFOAM routines for the Jacobian evaluation.

In the present report we first shed light on some standard OpenFOAM libraries that are responsible for the numerical modeling of combustion phenomena, with a particular focus on the chemistryModel library responsible for chemical source terms and reaction rates. Then, we discuss the implementation details to replace particular OpenFOAM functionalities responsible for numerical Jacobian matrix approximation with the analytical formulation from pyJac. After that, we utilize these implemented functionalities for analytical Jacobian formulation to develop CEMA for the identification of pre- and post-ignition zones as well as the detection of reaction fronts. Finally, a tutorial case is provided and discussed to validate and demonstrate the applicability of the developed model for analytical Jacobian and CEMA.
Chapter 2

Theory

2.1 Governing equations

The gaseous phase is herein described by the low Mach number compressible Navier-Stokes equations. Assuming laminar conditions, the corresponding formulations for the conservation of mass, momentum, species, and enthalpy are all presented as

\[
\frac{\partial \rho}{\partial t} + \nabla \cdot (\rho \mathbf{u}) = 0, \\
\frac{\partial \rho \mathbf{u}}{\partial t} + \nabla \cdot (\rho \mathbf{u} \otimes \mathbf{u}) = -\nabla p + \nabla \cdot \mathbf{\tau}, \\
\frac{\partial (\rho Y_k)}{\partial t} + \nabla \cdot (\rho \mathbf{u} Y_k) = \nabla \cdot (\rho D \nabla Y_k) + \dot{\omega}_k, \\
\frac{\partial (\rho h_t)}{\partial t} + \nabla \cdot (\rho \mathbf{u} h_t) = \frac{\partial p}{\partial t} + \nabla \cdot (\rho D \nabla h_s) + \dot{\omega}_h,
\]

where \(\rho, \mathbf{u}, p, \mathbf{\tau}, Y_k, D, h_s, h_t\) denote the density, velocity vector, pressure, viscous stress tensor, \(k^{th}\) species mass fraction, mass diffusivity, sensible and total enthalpy, respectively. The total enthalpy is herein defined as the summation of sensible enthalpy and specific kinetic energy. The \((\otimes)\) symbol refers to the outer product (dyadic), and the normal typeface symbols denote scalar quantities whereas bold ones represent higher order tensors. The viscous stress tensor for Newtonian fluids is defined as

\[
\mathbf{\tau} = \mu (\nabla \mathbf{u} + (\nabla \mathbf{u})^T - \frac{2}{3} \nabla \cdot \mathbf{u}) I,
\]

while the sensible enthalpy is defined according to the caloric state equation, i.e.

\[
h_s = \sum_{k=1}^{N_{sp}} Y_k h_{k} = \sum_{k=1}^{N_{sp}} Y_k (h_{f,k}^0 + \int_{T_0}^{T} c_{p_k}(T^*)dT^*),
\]

where \(h_{f,k}^0\) denotes \(k^{th}\) species enthalpy of formation at reference temperature \(T_0 = 298.15\) K, and \(N_{sp}\) is the total number of species in the chemical mechanism. Moreover, \(c_{p_k}(T^*)\) highlights temperature dependence of \(c_{p_k}\) which is typically retrieved using tabulated polynomial fits, as it is explained in the following section.

Equations (2.1) to (2.4) are closed by the ideal gas thermodynamic state equation, in addition to the pressure-momentum coupling using the reacting PISO (Pressure-Implicit with Splitting of Operators) algorithm [11]. The resulting set of equations is numerically discretized using the finite volume method (FVM) and it is solved using the OpenFOAM \(^\text{®}\) framework. We note that additional terms have to be considered in the previous set of equations to account for subgrid scale modeling or the phase change. However, as explained earlier, our demonstration is based on the laminar and one-phase flow assumption, without loss of generality.
Chapter 2. Theory

2.2 Finite-rate chemistry

The chemical source terms $\dot{\omega}_k$ and $\dot{\omega}_h$ denote the $k^{th}$ species net production/consumption rate, and heat release rate (HRR), respectively. The definition of HRR is given by

$$\dot{\omega}_h = \sum_{k=1}^{N_s} h_{f,k}^0 \dot{\omega}_k,$$

while the $k^{th}$ species production rate represents the mass of the species produced per unit volume and unit time (i.e. production of $\rho Y_k$ per unit time), and it is the sum of production rates $\dot{\omega}_{k,r}$ produced by all $N_r$ reactions, that is

$$\dot{\omega}_k = \sum_{r=1}^{N_r} \dot{\omega}_{k,r} = W_k \sum_{r=1}^{N_r} \nu_{k,r} q_r,$$

with $W_k$ denoting molecular weight of species $k$, and $\nu_{k,r} = \nu_{k,r}^\prime - \nu_{k,r}^\prime\prime$ denoting net stoichiometric coefficient of species $k$ in reaction $r$, while $\nu_{k,r}^\prime$ and $\nu_{k,r}^\prime\prime$ are the molar stoichiometric coefficients of species $k$ in reaction $r$ for the reactant ($\nu^\prime$) and product ($\nu^\prime\prime$) sides, respectively. The term $q_r$ indicates the progress rate of reaction $r$ and it is discussed in the following section. Further details on theoretical derivations of combustion governing equations under various conditions can be revised from Poinsot and Veynante [12] or from the book by Peters [2].

2.2 Finite-rate chemistry

Over the past fifty years, knowledge about combustion chemistry has significantly grown, especially for the gaseous phase, and nowadays it has reached a sufficient level of maturity. Considering an oxidation process of hydrocarbon fuel, this typically occurs through a chain of intermediate reactions, which becomes longer depending on the complexity of the fuel molecular structure. A reaction rate determines the speed at which chemical reaction progresses, such that it is proportional to the increase of products concentrations and decrease of reactants concentrations. In the combustion literature, information regarding the important set reactions and their rate data are gathered in a so-called chemical kinetic mechanism, hence the term finite-rate chemistry highlighting the finite set of intermediate elementary reactions of a chain reaction process.

Considering an elementary reaction $r$ with equation $a + b \rightleftharpoons c + d$, where the double arrow symbol denotes reversible reaction, the forward rate constant ($k_{fw}$) is defined according to the modified Arrhenius law as

$$k_{fw,r} = A_r T^b \exp(-E_a/\bar{R}T),$$

and the forward reaction rate is defined by

$$R_{fw,r} = [a][b]k_{fw,r},$$

with square brackets denoting molar concentrations of individual reactants $a$ and $b$.

The parameters $A_r$, $b$, $E_a$, $\bar{R}$ in (2.9) denote the pre-exponential factor, the temperature exponent, the activation energy, and the universal gas constant, respectively. The first three parameters are tunable for a given reaction and it is a research topic to fit the parameters for particular thermodynamic conditions when developing chemical kinetic mechanism for particular fuel (or fuels) oxidation. Additionally, we note that other types of elementary reactions may occur in the chain reaction and thereby be represented in the chemical mechanism, such as third-body, pressure-dependent, surface reactions, and others. However, a discussion of these reaction types is beyond the scope of this report.

The most common style, by far, to represent a chemical kinetic mechanism is through CHEMKIN-II format. Such a format provides information regarding chemical kinetics (chem file), species thermodynamic properties (thermo file), and species transport properties (transport file). The chemical kinetics file provides information for chemical elements, species, and reactions which are represented by their formulae and the tunable parameters $A$, $b$, and $E_a$ of the reaction, in addition
2.2 Finite-rate chemistry

Chapter 2. Theory

to further parameters in case of pressure-dependent reaction types. Thermodynamics file provides a set of 14 coefficients for every species (7 coefficients for each the lower and higher temperature interval) which are best fit polynomials (termed as NASA polynomials) to represent temperature dependent heat capacity $C_p(T)$, molar enthalpy $H(T)$, and molar entropy $S(T)$. The transport file provides information regarding species molecular transport properties, which are polynomial fits for the temperature dependent dynamic viscosity, thermal conductivity, and binary diffusion coefficients. Chemical mechanisms are usually publicly available. A widely common mechanism for natural gas combustion is the GRI-3.0 (which we will also use during our demonstration in the report) available at http://combustion.berkeley.edu/gri-mech and downloaded from http://combustion.berkeley.edu/gri-mech/version30/text30.html.

At this point, we may resume the derivation of the chemical source term (i.e. net production rate of $k^{th}$ species, $\dot{\omega}_k$) noted in the previous section. To compute (2.8), the progress rate of reaction $r$ in a mechanism containing $N_r$ reactions is first evaluated using the following relation

\[
q_r = R_{fw,r} - R_{bw,r} = k_{fw,r} \prod_{k=1}^{N_{sp}} (\rho \frac{Y_k}{W_k})^{\nu'_r,k} - k_{bw,r} \prod_{k=1}^{N_{sp}} (\rho \frac{Y_k}{W_k})^{\nu''_r,k},
\]

(2.11)
in which the species concentration $\bar{c}_k = (\rho \frac{Y_k}{W_k})$ is represented in terms of its mass fraction for practical reasons, as $Y_k$ is usually an input in most numerical solvers. Again, we note that the previous equation is limited to elementary reactions, and some manipulation (third-body/pressure scaling) would be needed to account for other reaction types. Now, it is obvious the relation between this equation and (2.10). Moreover, this equation can be further expanded using (2.9).

2.2.1 Evaluating source terms via direct integration

Due to the vast difference between chemistry and flow time scales, it is not feasible to limit the flow solver according to the smallest chemical time scale. Instead, it is common to utilize operator splitting approach to separate the calculation of chemistry source terms from the flow in eqs. (2.3) and (2.4). In this way, chemistry source terms will represent the change of thermo-chemical composition within a CFD time step, as we further explain in the following.

The change in local thermo-chemical composition is obtained by solving (integrating) a stiff, nonlinear initial value problem described as

\[
\begin{align*}
\frac{\partial \Phi}{\partial t} = & \begin{bmatrix} \frac{\partial T}{\partial t}, & \frac{\partial Y_1}{\partial t}, & \frac{\partial Y_2}{\partial t}, & \ldots, & \frac{\partial Y_{N_{sp}}}{\partial t} \end{bmatrix}^\top = \mathbf{f}(\Phi, t), \\
\Phi(t = t_0) = & \Phi_0,
\end{align*}
\]

(2.12)

wherein the rate of change of temperature and species mass fractions are combined into a state vector $\Phi$ of size $N_{sp} + 1$ in order to be solved while making use of eqs. (2.7) and (2.8) for the nonlinear function in the right hand side. Moreover, the change of thermo-chemical state is related to chemical source terms by the relation

\[
\frac{\partial Y_k}{\partial t} = \frac{1}{\rho} \dot{\omega}_k,
\]

(2.13)

and for HRR by the relation

\[
\frac{\partial T}{\partial t} = -\frac{1}{\rho c_p} \sum_{k=1}^{N_{sp}} h_k \dot{\omega}_k,
\]

(2.14)

which can be further expanded using (2.6). The relationship between a species mass fraction and its concentration is given by

\[
\bar{c}_k = \frac{\rho_k Y_k}{W_k},
\]

(2.15)
The aforementioned stiff ODE system in (2.12) is independent for every computational cell since the mixture of that cell is defined by the system state \( \Phi_0 \) at time \( t_0 \). After that, time integration is performed to compute source terms for chemistry and enthalpy after marching a complete CFD timestep \( \delta t_{\text{CFD}} \). In addition, due to the stiffness of such ODE systems, implicit and semi-implicit ODE solvers are preferred to ensure solution stability [13]. Accordingly, such solvers attempt to solve the stiff ODE system over \( \delta t_{\text{CFD}} \) interval by marching over subintervals denoted by \( \delta t_{\text{ODE}} \), which are smaller than \( \delta t_{\text{CFD}} \).

As in most implicit solvers for stiff ODE systems, a system Jacobian, \( J = \frac{\partial f}{\partial \Phi} \), is utilized as it is demonstrated below for a single ODE solution subinterval, i.e.

\[
\begin{align*}
\Phi_{n+1} &= \Phi_n + \int_{t_n}^{t_n + \Delta t_{\text{ODE}}} \left( f_n + J_n(\Phi_{n+1} - \Phi_n) + O(\Delta t_{\text{ODE}}^2) \right) dt \\
&= \Phi_n + f_n \Delta t_{\text{ODE}} + J_n(\Phi_{n+1} - \Phi_n) \Delta t_{\text{ODE}} + O(\Delta t_{\text{ODE}}^2),
\end{align*}
\tag{2.16}
\]

which is then linearized by neglecting the higher-order terms. As noted from the equation, the Jacobian is required for every subinterval, which can be an expensive procedure. The direct solution of the previous equation for \( \Phi_{n+1} \) requires the inverse of \( J \), which is commonly avoided through matrix factorization (i.e. LU decomposition) and back substitution techniques.

In Chapter 3 we demonstrate how the underlying derivations, herein noted, are realized in the OpenFOAM software for both the upper level and lower level implementations.

### 2.3 Chemical Explosive Mode Analysis

Chemical Explosive Mode Analysis (CEMA) is considered a powerful post-processing technique for the computational diagnostics of combustion events [6]. It is based on dynamical system theory [14, 15] and the wider concept of computational singular perturbation [16, 7]. Considering eqs. (2.3) and (2.4) while using species molar concentrations instead of mass fractions, they can be combined and rewritten in the Lagrangian form as follows

\[
\frac{D\Phi_c}{Dt} = \dot{\omega}(\Phi_c) + s(\Phi_c) \equiv g(\Phi_c),
\tag{2.17}
\]

in which \( \Phi_c \) is the thermo-chemical state vector, similarly defined in (2.12) but using species concentrations, \( \dot{\omega} \) vector is the chemistry source terms in species and energy equations, and \( s \) vector comprises the non-chemical diffusive transport terms . The Jacobian matrix of the above problem is \( J_g = \frac{\partial g}{\partial \Phi_c} \), which could be used for integration using Newton’s method. By linearity of operators, this Jacobian is decomposed into

\[
J_g = \frac{\partial g}{\partial \Phi_c} = \frac{\partial \dot{\omega}}{\partial \Phi_c} + \frac{\partial s}{\partial \Phi_c} \equiv J_\omega + J_s,
\tag{2.18}
\]

where \( J_\omega \) is comparable to the chemical Jacobian matrix in (2.16), i.e. \( J = \frac{\partial f}{\partial \Phi} \), depending on whether species concentrations or mass fractions are adopted for thermo-chemical state vector.\(^1\)

By multiplying (2.17) by \( J_\omega \) from the left side, we get the following chemical dynamical system

\[
\frac{D\dot{\omega}}{Dt} = J_\omega(\dot{\omega} + s).
\tag{2.19}
\]

CEMA is conceptually based on the stability of the aforementioned dynamical system. This is indicated by eigenvalue analysis of the Jacobian matrix \( J_\omega \). Despite time dependence of the non-linear dynamical system, it has been widely accepted in the literature to use simple eigendecomposition such that a fully decoupled set of modes (or basis vectors) is achieved [17, 18, 6]. Therefore, the diagonal matrix \( \Lambda \) denoting the Jacobian eigenvalues is obtained by

\(^1\)In case of mass fraction based state vector, Jacobian \( J_\omega \) from CEMA derivations in (2.18) is equivalent to \( J \) from (2.12).
A = B \cdot J_\omega \cdot A, \quad (2.20)

wherein the matrices $A$ and $B$ are composed of column and row basis vectors, respectively, with $A = B^{-1}$. Moreover, the time dependence of matrix $B$ is herein neglected, and $A$ is diagonal provided that $J_\omega$ is not defective such that the modes are fully decoupled with leading order accuracy.

The chemical explosive subspace (i.e. unstable subspace) of the Jacobian matrix is spanned by the basis vectors representing Chemical Explosive Modes (CEMs). CEMs are associated with eigenvalues of real part that is positive, i.e.

$$\text{Re}(\lambda_{\text{exp}}) > 0,$$

and $b_{\text{exp}}$ and $a_{\text{exp}}$ are left and right eigenvectors, respectively, of the Jacobian $J_\omega$ corresponding to $\lambda_{\text{exp}}$. Typically, when multiple CEMs are present, $\lambda_{\text{exp}}$ is chosen as the eigenvalue of the largest real part —while excluding conservation modes— hence denoting the fastest CEM along which chemical explosion occurs. On the other hand, non explosive modes are defined by eigenvalues of negative real part, i.e. stable modes of the dynamical system.

From a physical perspective, the existence of CEM indicates the propensity of a local mixture to auto-ignite if isolated, i.e. considering adiabatic and constant volume environment. Simply, a CEM is present in pre-ignition mixtures and absent in post-ignition mixtures. The transition of a CEM from explosive to non-explosive, i.e. zero-crossing eigenvalues, is strongly correlated to critical flame features such as ignition, extinction, and premixed flame front locations [6].

One important note is that, due to conservation of energy and species elements, the chemical Jacobian $J_\omega$ is typically constituted of $M+1$ zero eigenvalues, where $M$ is the number of elements in species composition. The element conservation modes are typically identified by examining the magnitude of the eigenvalues. The energy conservation mode is, however, more challenging to identify and distinguish from other slow chemical modes. The main reason is the non linear temperature-dependent thermodynamics including heat capacity, which could induce nonlinear effects from eigenvector rotation. Accordingly, for applications where dynamics of the energy conservation mode is not of particular interest, for instance when time dependence of heat capacity is not relevant, then the energy conservation mode becomes trivial and corresponds to zero eigenvalue. This is further clarified during the present implementation of CEMA in Chapter 5, where $M+1$ modes are to be skipped.
Chapter 3

Existing OpenFOAM features

3.1 High-level overview of reactingFoam solver

In this section, we highlight the high-level implementations of the reactingFoam solver as well as the mathematical equations of flow and thermo-chemistry discussed in Chapter 2.

3.1.1 Included header files

We start with the reactingFoam.C file and walk it through, first, by listing the included header files with a brief description as shown in Listings 3.1 and 3.2.

Listing 3.1: Initial header files in the reactingFoam solver

```cpp
#include "fvCFD.H"
#include "turbulentFluidThermoModel.H" // Typedef for turbulence models (compressibleTurbulenceModels)
#include "psiReactionThermo.H" // Declare class psiReactionThermo (reactionThermophysicalModels)
#include "CombustionModel.H" // Combustion models for templated thermodynamics (combustionModels)
#include "multivariateScheme.H" // Generic multi-variate discretization scheme (finiteVolume)
#include "pimpleControl.H" // Convergence information/checks for PIMPLE loop (finiteVolume)
#include "pressureControl.H" // Provide controls for the pressure reference (finiteVolume)
#include "fvOptions.H" // Finite-volume options (finiteVolume)
#include "localEulerDdtScheme.H" // First-order Euler implicit/explicit ddt scheme (finiteVolume)
#include "fvcSmooth.H" // Smooth and redistribute fields during face interpolation
```

Listing 3.2: Headers inclusion inside fvCFD.H

```cpp
#include "parRun.H" // routines for initializing the parallel run and for finalizing it
#include "Time.H" // controls the information of Time during the simulations
#include "fvMesh.H" // topological and geometric information related to mesh for FV discretization
#include "fvc.H" // 'explicit' calculus operations and geometric field discretization, return geometricField (contribute to source term b in Ax=b)
#include "fvMatrices.H"
#include "fvm.H" // 'implicit' calculus operations and geometric field discretization, return fvMatrix (contribute to coefficient matrix A in Ax=b)
#include "linear.H" // central-differencing interpolation scheme
#include "uniformDimensionedFields.H"
#include "calculatedFvPatchFields.H" // macro, add BC to run-time selection table, set debug switch
#include "extrapolatedCalculatedFvPatchFields.H" // similar to above
#include "fixedValueFvPatchFields.H" // similar to above
#include "fixedGradientFvPatchScalarField.H" // class for BC
#include "constrainHbyA.H" // relevant to PISO algorithm
#include "constrainPressure.H"
#include "adjustPhi.H"
#include "findRefCell.H" // find reference cell nearest to given index
#include "IOMRFZoneList.H" // list of MRF zones
#include "constants.H" // various constants defined within FOAM namespace
```
3.1. High-level overview of *reactingFoam* solver

We include additional header files together with brief descriptions, as shown in Listing 3.3.

### Listing 3.3: Included header files after main() in the *reactingFoam* solver

```c
#include "postProcess.H" // Execute application functionObjects to post-process existing results
#include "addCheckCaseOptions.H" // allow dry-run in supplied command-line flags
#include "setRootCaseLists.H" // setRootCase to check validity of case dir, (with additional solver-related listings)
#include "createTime.H" // constructs the runTime object of the class Time
#include "createMesh.H" // constructs mesh object (reference) of class fvMesh
#include "createTimeControls.H" // read control parameters used by setDeltaT (adjustTimeStep, maxCo, maxDeltaT)
#include "initContinuityErrs.H" // declare and initialise cumulative continuity error
#include "createFields.H"
#include "createFieldRefs.H"
```

3.1.2 Created fields with glimpse into libraries

Here, we discuss the created fields noted in *createFields.H* file of the *reactingFoam* solver directory. The creation of the GeometricField objects is fairly straightforward, and we only show one example for the velocity field $U$ in Listing 3.4. The remaining GeometricField objects that are created and stored to object registry are the density $\rho$, pressure $p$, face flux $\bar{\phi}$, transient pressure $dpdt$, kinetic energy $K$, and HRR (i.e. $\dot{\omega}_h$, c.f. (2.4)) defined by $Qdot$ in OpenFOAM.

### Listing 3.4: Example of reading geometric fields in *createFields.H*

```c
volVectorField U // variable type and name
{
    IObject
    {
        "U",
        runTime.timeName(),
        mesh,
        IObject::MUST_READ, // return error if field not found in case time dir
        IObject::AUTO_WRITE
    ),
    mesh
};
```

One important note is the selector function `New()` in the runtime selection mechanism in OpenFOAM. It is implemented in the base class with the purpose of creating an object of derived classes according to an input that we provide during runtime. In Listing 3.5, we have three selectors for thermophysical model, turbulence model, and reaction model.

### Listing 3.5: Selector functions in *createFields.H* to create object of derived classes during runTime

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

---

\(^1\)Note that `volVectorField` is just typedef of the `GeometricField<vector, fvPatchField, volMesh>` templated class, and it is defined in `volFieldsFwd.H`. 

---

16
3.1. High-level overview of `reactingFoam` solver

Chapter 3. Existing OpenFOAM features

In the thermophysical model case, the function `New()` is called from the derived class `psiReactionThermo` which overrides implementation in the base class `basicThermo` and it is implemented in the library `libreactionThermophysicalModels`.

In the turbulence model case, the pointer `turbulence` is constructed and it calls `New()` function as argument. Such a function `New()` belongs to the namespace `Foam::compressible` and the class `turbulenceModel` which is typedef `ThermalDiffusivity<CompressibleTurbulenceModel<fluidThermo>>`. Both the class `turbulenceModel` and the `New()` function are declared in `turbulentFluidThermoModel.H` with implementation in the corresponding source file that is compiled to create the shared library `libcompressibleTurbulenceModels`.

In the reaction model case, the variable `reaction` is a pointer to an object of a templated class, with the parameter being `CombustionModel` class that is also templated by the parameter `psiReactionThermo` class. The selector function `New()` belongs to the templated class `CombustionModel` declared in `CombustionModel.H` and implemented in the corresponding source code of same name, while the library `libcombustionModels` contains binaries of the compiled code. The selector function takes two non-default arguments of type `ReactionThermo` and `compressibleTurbulenceModel`. This implies that the combustion model depends on both reactive thermodynamics and turbulence properties.

Finally, we note that the implementation of `New()` functions is quite complicated and the present report does not provide detailed discussions. Therefore, we advise interested readers to further dig through the code themselves or to find other material that better covers such details.

### 3.1.3 Linear systems in the time loop

Now, we proceed with `reactingFoam.C` and take a look into the main part of time loop (Listing 3.6). There, we find inclusion of `rhoEqn.H` (located in `$($FOAM_SRC)/finiteVolume/cfdTools/compressible`) and the files `UEqn.H`, `YEqn.H`, `EEqn.H`, and `pEqn.H` (located in the solver directory) which are responsible for solving the governing equations for mass, momentum, species, energy, and pressure-momentum coupling, respectively. We will not go through all details, but we show only the relevant parts that reflect discussions from the previous chapter.

The previously mentioned files are represented in Listings 3.7 to 3.10, while the pressure-momentum coupling implementation is detailed in Listing 3.11. The finite-volume system equations for mass, momentum, species, and energy are comparable with those conservation equations in eqs. (2.1) to (2.4), assuming that we discard additional terms from e.g. MRF and `fvOptions`. Moreover, we note that `reaction->R(Yi)` (in `YEqn.H`) denotes species overall production rate $\dot{\omega}_k$ which is further discussed in Section 3.2.

Listing 3.6: Solving conservation equations inside time loop of `reactingFoam.C`

```cpp
++runTime;
```
3.1. High-level overview of reactingFoam solver

Listing 3.7: Finite volume system for mass conservation (continuity) included from rhoEqn.H

```cpp
fvScalarMatrix rhoEqn
(
    fvm::ddt(rho)
    + fvc::div(phi)
    == fvOptions(rho)
);
fvOptions.constrain(rhoEqn);
rhoEqn.solve();
fvOptions.correct(rho);
```

Listing 3.8: Finite volume system for momentum conservation included from UEqn.H

```cpp
tmp<fvVectorMatrix> tUEqn
(
    fvm::ddt(rho, U) + fvm::div(phi, U)
    + MRF.DDt(rho, U)
    + turbulence->divDevRhoReff(U)
    == fvOptions(rho, U)
);
fVectorMatrix& UEqn = tUEqn.ref();
UEqn.relax();
fvOptions.constrain(UEqn);
if (pimple.momentumPredictor())
{
    solve(UEqn == -fvc::grad(p));
    fvOptions.correct(U);
    K = 0.5*magSqr(U);
```
Listing 3.9: Finite volume system for species conservation included from YEeqn.H

```cpp
reaction->correct(); // Here chemical source term is computed
Qdot = reaction->Qdot(); // Here HRR is computed
goSFrScalarField Yt(0.0*Y[0]);

forall(Y, i)
{
  if (i != inertIndex && composition.active(i))
  {
    volScalarField& Yi = Y[i];
    fvScalarMatrix YiEqn
    {
      fvm::ddt(rho, Yi)
      + mvConvection->fvmDiv(phi, Yi)
      - fvm::laplacian(turbulence->muEff(), Yi)
      ==
      reaction->R(Yi)
      + fvOptions(rho, Yi);
    }
    YiEqn.relax();
    fvOptions.constrain(YiEqn);
    YiEqn.solve(mesh.solver("Yi"));
    fvOptions.correct(Yi);
    Yi.max(0.0);
    Yt += Yi;
  }
}
Y[inertIndex] = scalar(1) - Yt;
Y[inertIndex].max(0.0);
```

Listing 3.10: Finite volume system for energy conservation included from EEeqn.H

```cpp
volScalarField& he = thermo.he();
fvScalarMatrix EEeqn
{
  fvm::ddt(rho, he) + mvConvection->fvmDiv(phi, he)
  + fvc::ddt(rho, K) + fvc::div(phi, K)
  + {
    he.name() == "e"
    ? fvc::div
    {
      fvc::absolute(phi/fvc::interpolate(rho), U),
      p,
      "div(phiv,p)"
    }
    : -dpdt
  }
  - fvm::laplacian(turbulence->alphaEff(), he)
  ==
  Qdot
  + fvOptions(rho, he);
}
EEeqn.relax();
fvOptions.constrain(EEeqn);
```
3.1. High-level overview of reactingFoam solver

Chapter 3. Existing OpenFOAM features

Listing 3.11: Equations for pressure-momentum coupling included from pEqn.H

```cpp
rho = thermo.rho();
volScalarField rAU(1.0/UEqn.A());
surfaceScalarField rhorAUf("rhorAUf", fvc::interpolate(rho*rAU));
volVectorField HbyA(constrainHbyA(rAU*UEqn.H(), U, p));
if (pimple.nCorrPISO() <= 1)
{
    tUEqn.clear();
}
if (pimple.transonic())
{
    surfaceScalarField phid
    {
        "phid",
        fvc::interpolate(psi)
        *(fvc::flux(HbyA)
        + MRF.zeroFilter
        {
            rhorAUf*fvc::ddtCorr(rho, U, phi)/fvc::interpolate(rho)
        }
    );
    MRF.makeRelative(fvc::interpolate(psi), phid);
    while (pimple.correctNonOrthogonal())
    {
        fvScalarMatrix pEqn
        {
            fvm::ddt(psi, p)
            + fvm::div(phid, p)
            - fvm::laplacian(rhorAUf, p)
            == fvOptions(psi, p, rho.name());
        );
        pEqn.solve(mesh.solver(p.select(pimple.finalInnerIter())));
        if (pimple.finalNonOrthogonalIter())
        {
            phi == pEqn.flux();
        }
    }
}
else
{
    surfaceScalarField phiHbyA
    {
        "phiHbyA",
        fvc::flux(rho*HbyA)
        + MRF.zeroFilter(rhorAUf*fvc::ddtCorr(rho, U, phi))
    );
    MRF.makeRelative(fvc::interpolate(rho), phiHbyA);
}
```

3.2. A walk-through to compute chemistry source terms

Here, we discuss the standard implementation of finite-rate chemistry for computing the chemical source terms, \( \dot{\omega}_k \), and subsequently HRR, \( \dot{\omega}_h \). First, let us start from the solver level, again, and try to track back the function call `reaction->correct()` from `YEqn.H` (Listing 3.9). As previously noted, `reaction` is just a pointer to the employed combustion model. In the present development, we consider the implementation of finite-rate chemistry under laminar conditions, i.e. no subgrid modeling for turbulence-chemistry interactions. Therefore, we check the implementation of `correct()` function in such a combustion model from `$(FOAM_SRC)/combustionModels/laminar/laminar.C`. We see that the implementation is conditioned when active flag is enabled in the `combustionProperties`
3.2. Walk-through to compute chemistry source terms  Chapter 3. Existing OpenFOAM features

dictionary (c.f. constructor in combustionModel.C). Furthermore, the following condition is for integrating the reaction rate over CFD time step (\(\delta t_{\text{CFD}}\)). This is activated by default if the laminar model is chosen in the combustionProperties and the keyword integrateReactionRate is not added in the dictionary (c.f. constructor in laminar.C). The third condition is for the ddt scheme, whether or not it is first order Euler implicit/explicit ddt, since the reciprocal of local time step will have to be looked up from database. In combustion simulations it is common to use a higher order ddt schemes so this will leave us with the only function call

\[
\text{this->chemistryPtr_-->solve(this->mesh().time().deltaTValue());}
\]

Now, we need to track the pointer chemistryPtr_. In fact, it is a pointer to chemistry model, and it is declared in ChemistryCombustion.H with type autoPtr<BasicChemistryModel<ReactionThermo>>. Moreover, it is initialized using selector function BasicChemistryModel<ReactionThermo>::New(thermo). This gives us hint that the function solve(this->mesh().time().deltaTValue()) should be defined in some derived class of the BasicChemistryModel that is chosen during runTime.

Let us now go and check possible derived classes of BasicChemistryModel. We only see that possible models are the templated StandardChemistryModel denoting standard and direct integration of the chemistry ODE system, and the TDACChemistryModel denoting the tabulated dynamic adaptive model, which is a tabulation-based strategy for chemistry calculations. The present report is based on developments on top of the standard chemistry model, so let us proceed with this approach.

Now, after checking for the function solve() we find that it is defined in various forms inside StandardChemistryModel.C. Through a cross comparison, we realize that the function call solve(this->mesh().time().deltaTValue()) from laminar combustion model calls the function solve(const scalar deltaT) in standard chemistry model. Accordingly, we further proceed with the implementation of this function while showing the interesting part in Listing 3.12.

Listing 3.12: Part of the solve() function definition in StandardChemistryModel.C

```c
706 tmp<volScalarField> trho(this->thermo().rho());
707 const scalarField& rho = trho();
708
709 const scalarField& T = this->thermo().T();
710 const scalarField& p = this->thermo().p();
711
712 scalarField c0(nSpecie_);
713
714 forAll(rho, celli)
715 {
    scalar Ti = T[celli];
716
    if (Ti > Treact_)
717    {
        const scalar rhoi = rho[celli];
        scalar pi = p[celli];

        for (label i=0; i<nSpecie_; i++)
        {
            c_[i] = rhoi*Y_[i][celli]/specieThermo_[i].W();
            c0[i] = c_[i];
        }

        // Initialise time progress
        scalar timeLeft = deltaT[celli];

        // Calculate the chemical source terms
        while (timeLeft > SMALL)
        {
            scalar dt = timeLeft;
            this->solve(c_, Ti, pi, dt, this->deltaTChem_[celli]);
            timeLeft -= dt;
        }
    }
```
3.2. Walk-through to compute chemistry source terms  Chapter 3.  Existing OpenFOAM features

Here, we may realize few notes. First, inside the cells loop, a scalar timeLeft is initialized with, probably, $\delta t_{C_F D}$ of a computational cell. Second, as long as the value of timeLeft did not vanish, another function named solve() is also called, which is inherited from another class, and it takes the pressure, temperature, and species concentrations, hence the thermo-chemical composition of that computational cell, in addition to two more variables related to time steps which are not CFD time steps. Moreover, the comment above the while loop states that such a scope is responsible for computing chemical source terms. Based on our educated guess, this part of the code implements is relevant to the ODE solution routines which are previously noted in eqs. (2.12) to (2.14) and (2.16). More importantly, the last loop in the function definition updates the chemistry source terms for each computational cell, which is exactly (2.13) while converting mass fractions into concentrations using (2.15).

Now, let us continue our investigation to track the function call

```
this->solve(c_, Ti, pi, dt, this->deltaTChem_[celli]).
```

In fact, this function form has been declared as pure abstract in StandardChemistryModel.H. After quick check, we find that a definition of this function exists in the derived class chemistrySolver. Therefore, we understand now that the function will be executed according to the chemistry solver. In this report, we base our development on ODE approach (i.e. not implicit Euler), hence we check the solve() function definition in ode.C which is presented in the following Listing 3.13.

Listing 3.13: Definition of the solve() function in ode.C of chemistrySolver

```
template<class ChemistryModel>
void Foam::ode<ChemistryModel>::solve
(
    scalarField& c,
    scalar& T,
    scalar& p,
    scalar& deltaT,
    scalar& subDeltaT
) const
{
    // Reset the size of the ODE system to the simplified size when mechanism
    // reduction is active
    if (odeSolver_->resize())
    {
        odeSolver_->resizeField(cTp_);
    }

    const label nSpecie = this->nSpecie();

    // Copy the concentration, T and P to the total solve-vector
    for (int i=0; i<nSpecie; i++)
    {
        cTp_[i] = c[i];
    }
    cTp_[nSpecie] = T;
    cTp_[nSpecie+1] = p;

    odeSolver_->solve(0, deltaT, cTp_, subDeltaT);
```
3.2. Walk-through to compute chemistry source terms  Chapter 3. Existing OpenFOAM features

```cpp
for (int i=0; i<nSpecie; i++)
{
    c[i] = max(0.0, cTp_[i]);
}
T = cTp_[nSpecie];
p = cTp_[nSpecie+1];
```

The interesting part of this function is that it first constructs a solve vector \( c_{Tp} \) comprising thermo-chemical composition (concentrations, temperature, pressure). After that, it calls another `solve()` function using the pointer `odeSolver_` which is of mutable type `autoPtr<ODESolver>` and it is initialized using `New()` selector function from `ODESolver` class in `$(FOAM_SRC)/ODE`. At this level, the class `ODE` does not know (or need) any information about the chemistry or thermophysics of the problem. It is purely mathematical procedure at this point.

In this report, we are not interested in the implementation details of the stiff ODE solvers, nor the various algorithms and their comparative performance. Instead, we note an important point which is that `StandardChemistryModel` inherits from both `BasicChemistryModel` and the abstract class `ODESystem`. The reason of the latter inheritance is that chemistry model implements three important functions which are declared pure virtual in `ODESystem` as provided in the following Listing 3.14.

Listing 3.14: Abstract class `ODESystem` with 3 functions implemented in `chemistryModel`

```cpp
class ODESystem
{
public:

    // Constructors
    ODESystem();
    ~ODESystem() = default;

    // Member Functions
    virtual label nEqns() const = 0;
    virtual void derivatives(const scalar x, const scalarField& y, scalarField& dydx) const = 0;
    virtual void jacobian(const scalar x, const scalarField& y, scalarField& dfdx, scalarSquareMatrix& dfdy) const = 0;
};
```

As it will be explained in the following chapter, we are interested in replacing the difference-based implementation of the Jacobian function with another analytical formulation using the external source code generator `pyJac`. Accordingly, replacing the Jacobian function will result into various
functions that would need to be updated to accommodate for the changed Jacobian matrix. We provide detailed discussion in the next chapter.

### 3.3 Summary of chemistry source terms function calls

Here, we provide a unified modeling language (UML) sequence diagram, depicted in Figure 3.1, to back track the implementation and function calls relevant to computing the chemical source terms, starting from `reaction->correct()` in `YEqn.H` of solver level. Additionally, we also note that `ODESystem` abstract class includes methods for `nEqns()`, `derivatives()`, and `jacobian()`, which are all implemented in the `chemistryModel`.

![Sequence diagram of function calls to compute chemistry source terms.](image)

Figure 3.1: Sequence diagram of function calls to compute chemistry source terms.
Chapter 4

Analytical Jacobian for OpenFOAM

4.1 Motivation

The main motive behind utilizing an analytical Jacobian for the chemistry problem is that it provides i) better accuracy to the solution of stiff ODE chemistry problem compared with numerical Jacobian, ii) accurate predictions of CEMA results, since CEMA computations are sensitive to significant digits of the Jacobian matrix and thereby information might be lost due to truncation errors in numerical Jacobian [6]. Relevant details are provided in Section 6.4.

As previously discussed in Chapter 2, the Jacobian matrix $\mathbf{J}_\omega$ is required in the Newton solver when computing the chemical source terms. The Jacobian can be computed either analytically via algebraic summation of contributions from all reactions, or numerically via perturbation of thermo-chemical state vector. In numerical simulations involving implicit or semi-implicit solvers (such as the present case for Seulex algorithm), the Jacobian operations involving evaluation and factorization are computationally expensive. According to Lu and Law [19], evaluation of the numerical Jacobian scales quadratically as $(N_{sp} \times N_r)$ and consequently as $N_{sp}^2$. On the other hand, the analytical Jacobian evaluation scales linearly as $N_r$. Therefore, usage of the analytical Jacobian becomes computationally recommended whenever possible.

Besides computational performance, various computational diagnostics techniques including CEMA [6], CSP [16, 20], and Jacobian analysis [21] would require accurate evaluation of the chemical Jacobian matrix, hence the advantage of analytical formulation. However, since Jacobian factorization operation scales cubically as $N_{sp}^3$, therefore it should be performed only when needed.

The presently employed OpenFOAM version (v2006) utilizes a semi-analytical formulation of the concentration-based chemical Jacobian. We note that there has been implementations of fully analytical Jacobian formulations introduced in recent OpenFOAM versions (e.g. OpenFOAM-8). However, thus far the Open-source library pyJac seems to comprise robust implementation techniques that minimize computational and memory operations. Moreover, pyJac was reported to outperform other existing routines for analytical Jacobian evaluation either analytically or through finite-differencing [10].

As a brief note, pyJac is a software package which is intended to generate source code files of the analytical Jacobian formulation and other helper functionalities tailored for a specific chemical kinetic mechanism provided by the user. In order to generate the source files, we need to have the chemical mechanism in either Chemkin-II format (the most widely used format) or in Cantera formats.

In this chapter, we demonstrate how to link pyJac with OpenFOAM to provide fully algebraic analytical formulation of the chemical Jacobian, which will be also used for CEMA computations.

---

1It is possible to convert between Chemkin-II and Cantera formats through ck2cti and ck2yaml utilities, as described in https://cantera.org/tutorials/ck2cti-tutorial.html.
4.2 Creating cemaPyjacChemistryModel

Aiming for simplicity over a DRY (Don’t Repeat Yourself) but possibly complicated code, we create our custom chemistry model by copying the StandardChemistryModel and adding or modifying the functionalities of interest. This implies that our new chemistry model inherits from BasicChemistryModel and ODESystem and non-modified methods and class attributes will be duplicated. Indeed, a better way is to declare our model inherited from StandardChemistryModel while overriding and/or extending the desired functionalities. However, as mentioned we will proceed in the most straightforward way.

After sourcing OpenFOAM-v2006, we execute the following bash commands through a Linux terminal for creating a user-defined library cemaPyjacChemistryModel copied from StandardChemistryModel while cleaning unnecessary files.

```bash
> foam
> cp -r --parents src/thermophysicalModels/chemistryModel $WM_PROJECT_USER_DIR
> cd $WM_PROJECT_USER_DIR/src/thermophysicalModels/chemistryModel
> rm -r TDACChemistryModel BasicChemistryModel basicChemistryModel
> mv StandardChemistryModel cemaPyjacChemistryModel; cd cemaPyjacChemistryModel
> mv StandardChemistryModel.C cemaPyjacChemistryModel.C
> mv StandardChemistryModel.H cemaPyjacChemistryModel.H
> mv StandardChemistryModelI.H cemaPyjacChemistryModelI.H
> sed -i 's/Standard/cemaPyjac/g' *
```

and then we also change the TypeName for the new model in the runTime selection table

```bash
> sed -i 's/standard/cemaPyjac/g' cemaPyjacChemistryModel.H
```

Now, we need to update description in the header files (optional), and then to modify the Make/options file (Listing 4.1) to pass information to the compiler for header inclusion and library linkings. Since our chemistry model is copied from standard model which is part of the chemistryModel library (libchemistryModel.so), there will be possibly dependencies on other components of that library when we want to compile our model separately. Therefore, we need to include the path of headers (or their symbolic links) that are included by StandardChemistryModel, as well as linking to the corresponding library to enable all implementations of standard model to be also realized within our custom model.

Listing 4.1: Make/options without provided path for pyJac header files

```plaintext
EXE_INC = \n-$(LIB_SRC)/finiteVolume/lnInclude \n-$(LIB_SRC)/meshTools/lnInclude \n-$(LIB_SRC)/ODE/lnInclude \n-$(LIB_SRC)/transportModels/compressible/lnInclude \n-$(LIB_SRC)/thermophysicalModels/reactionThermo/lnInclude \n-$(LIB_SRC)/thermophysicalModels/basic/lnInclude \n-$(LIB_SRC)/thermophysicalModels/specie/lnInclude \n-$(LIB_SRC)/thermophysicalModels/functions/Polynomial \n-$(LIB_SRC)/thermophysicalModels/thermophysicalFunctions/lnInclude \n-$(LIB_SRC)/thermophysicalModels/chemistryModel/lnInclude

LIB_LIBS = \n-lfiniteVolume \n-lmeshTools \n-lODE \n-lcompressibleTransportModels \n-lfluidThermophysicalModels \n-lreactionThermophysicalModels \n-lspecie \n-lchemistryModel
```
4.3 Using pyJac functionalities

As we see, the only modifications we made in `Make/options` is adding the lines `-lchemistryModel` and `-I$(LIB_SRC)/thermophysicalModels/chemistryModel/lnInclude` for each `LIB_LIBS` and `EXE_INC` variables, respectively. This means that we allow the compiler to find all included header files needed from standard model as well as implementations of the library into our custom model. We note that we also need to add one more path related to header files from pyJac for declarations of function that we will use, as it is shown in Appendix A.1 for `pyjacInclude` subdirectory. Basically, such header files are provided for the sake of model compilation, and they can be acquired by generating source code through pyJac for an arbitrary mechanism.

Since chemistry model is a templated library by default, we also need to allow our custom chemistry model instances to be created for all possible thermodynamic and compressibility models. We achieve this thanks to the macro `makeChemistryModelType` similarly used in `BasicChemistryModels.C` from standard model. Accordingly, we copy the macros definitions and modify their usage accordingly into our own model directory as in the following.

```bash
> cd $WM_PROJECT_USER_DIR/src/thermophysicalModels/chemistryModel
> cp $FOAM_SRC/thermophysicalModels/chemistryModel/chemistryModel/makeChemistryModel.H .
> thermophysicalModels=$FOAM_SRC/thermophysicalModels
> BasicChemistryModel=chemistryModel/chemistryModel/BasicChemistryModel
> thermophysicalBasicChemistryModel=$(thermophysicalModels)/$(BasicChemistryModel)
> cp $(thermophysicalBasicChemistryModel)/BasicChemistryModels.C makeChemistryModels.C
> sed -i 's/Standard/cemaPyjac/g' makeChemistryModels.C

After that, we clean all lines of code relevant to TDAC model from `makeChemistryModels.C` as we did not include it to our development, otherwise our model will not compile. Next, we repeat the same procedure but with chemistry solver which is also templated on thermodynamics. Moreover, we modify the name of the ODE chemistry solver to `odePyjac` to reflect our intention to modify such class with functionalities from pyJac. Further discussions and demonstrations are depicted in Section 5.2 and Appendix A.3.

```bash
> cd $WM_PROJECT_USER_DIR/src/thermophysicalModels/chemistryModel
> chemistryModel=$FOAM_SRC/thermophysicalModels/chemistryModel
> cp $chemistryModel/chemistrySolver/chemistrySolver/makeChemistrySolver* .
> sed -i 's/ode,/odePyjac,/g' makeChemistrySolverTypes.H
> sed -i 's/ode.H/odePyjac.H/g' makeChemistrySolverTypes.H
```

Similar to the chemistry model, we herein remove all lines of code relevant to TDAC, including the header inclusion, from `makeChemistrySolverTypes.H`. Finally, we modify the `Make/files` (Listing 4.2) to compile the proper source files using defined macros, and we specify the executable name and location.

```
Listing 4.2: Make/files

1. chemistryModel/makeChemistryModels.C
2. chemistryModel/makeChemistrySolver.C
3. LIB = $(FOAM_USER_LIBBIN)/libcemaPyjacChemistryModel
```

Next, we dive deeper into the new chemistry model in order to incorporate functionalities from pyJac.

4.3 Using pyJac functionalities

In this section, we focus on implementing an analytical formulation of the chemical Jacobian matrix along with other exact formulations such as temporal derivatives of the thermo-chemical state vector by making use of pyJac. We also discuss the particular features of pyJac which must be considered upon implementation since they are rather different from stock OpenFOAM implementation. In brief,
the special considerations are that i) \texttt{pyJac} generates Jacobian matrix based on the mass fraction-based state vector \((\Phi)\) while standard \texttt{OpenFOAM} operates on concentration-based state vector \((\Phi_c)\), and ii) \texttt{pyJac} considers \((N_{sp} - 1)\) species in the state vector while dumping all numerical residuals into the last species, which is chosen as inert (or most abundant) species of the chemical mechanism such as Nitrogen. Hence, \(Y_{\text{last}} = 1.0 - \sum_{i=1}^{N_{sp}-1} Y_i\). In the following, we discuss how to download and compile the \texttt{pyJac} package.

### 4.3.1 Download and compile \texttt{pyJac}

The \texttt{pyJac} package is publicly available on Github through the following link \url{https://github.com/SLACKHA/pyJac}. As it is indicated from the documentation, \texttt{pyJac} package can be installed via different means. Besides building the package from the source code (while considering all dependencies, and most importantly \texttt{Cantera})

\begin{verbatim}
> python setup.py install
\end{verbatim}

it can also be installed via the \texttt{Conda} package manager

\begin{verbatim}
> conda install -c slackha pyjac
\end{verbatim}

or from the Python package index \texttt{PyPI} using the \texttt{pip} tool

\begin{verbatim}
> pip install pyjac
\end{verbatim}

The necessary header files from \texttt{pyJac} that are required for library compilation are attached in Appendix A.4. In the following, we discuss details for updating methods of Jacobian, derivatives, and heat release rate.

### 4.3.2 Updating Jacobian method

The Jacobian function of the model is declared as in the following Listing 4.3.

#### Listing 4.3: jacobian() function arguments in cemaPyjacChemistryModel.C

```cpp
template<class ReactionThermo, class ThermoType>
void Foam::cemaPyjacChemistryModel<ReactionThermo, ThermoType>::jacobian
(
    const scalar t,
    const scalarField& c,
    scalarField& dcdt,
    scalarSquareMatrix& dfdc
) const
```

The parameters \(c\) and \(dfdc\) of the previous listing correspond to the state vector and Jacobian, respectively. It is important to note that \(c\) vector originally denotes species concentrations, temperature, and pressure (i.e. \(c_{Tp}\) variable which we will also see in the chemistry solver). However, in our implementation we use pressure, temperature, and species mass fraction.

As we previously noted, the original chemistry model operates on \(\Phi_c\) which is concentration based. Since \texttt{pyJac} uses species mass fractions without last inert species, the parameter \(c\) (acting as \(\Phi\) of size of \(N_{sp} + 2\)) would be used to prepare the input variable for evaluating Jacobian from \texttt{pyJac}. Moreover, the Jacobian matrix \(dfdc\) is mass fraction based and holding size of \((N_{sp} \times N_{sp})\) for temperature and species partial derivatives while excluding last species. The Jacobian function call from \texttt{pyJac} looks like the following

```c
void eval_jacob (const double t, const double pres, const double * __restrict__ y, \
    double * __restrict__ jac)
```

in which the parameters \(t\), \(pres\), and \(y\) denote current system time, pressure, and state vector of temperature and species mass fractions of size \(N_{sp}\). Therefore, we need to prepare the proper input
data to be passed to the pyJac function call so that the parameter jac is populated with analytical Jacobian data. Additionally, we need to include proper header file (mechanism-independent) for function declaration while using dynamic binding for linking the implementation (mechanism-dependent) during runtime. The corresponding header file for derivatives declaration is dydt.h which is included as in the following Listing 4.4.

Listing 4.4: Declarations of the jacobian() and derivatives() methods in addition to helper functions in cemaPyjacChemistryModel.C

```cpp
extern "C" {
    #include "chem_utils.h"
    #include "dydt.h"
    #include "jacob.h"
};
```

In our implementation, we replace the whole definition of the semi-analytical Jacobian method with the following

Listing 4.5: Definition of the jacobian() method in cemaPyjacChemistryModel.C – without CEMA

```cpp
template<class ReactionThermo, class ThermoType>
void Foam::cemaPyjacChemistryModel<ReactionThermo, ThermoType>::jacobian
(const scalar t,
 const scalarField& c,
 scalarField& dcdt,
 scalarSquareMatrix& dfdc ) const
{
    std::vector<double> TY(nSpecie_+1, 0.0);
    std::vector<double> dfdy(nSpecie_*nSpecie_, 0.0);
    const scalar p = c[0];
    const scalar T = c[1];
    scalar csum = 0.0;
    forAll(c_, i)
    {
        c_[i] = max(c_[i+2], 0.0);
        csum += c_[i];
        TY[i+1] = c_[i];
    }
    // Then we exclude last species from csum and instead dump all
    // residuals into last species to ensure mass conservation
    csum -= c_[nSpecie_-1];
    c_[nSpecie_-1] = 1.0 - csum;
    dfdc = Zero;
    TY[0] = T;
    forAll(c_, i)
    {
        TY[i+1] = c_[i];
    }
    eval_jacob(0, p, TY.data(), dfdy.data());
    // Back substitution to update dfdc
    // Assign first row and column to zero since they correspond to const pressure
    for (label j = 0; j < nSpecie_ + 1; ++j)
    {
        dfdc(0,j) = 0.0;
        dfdc(j,0) = 0.0;
    }
    label k = 0;
    // Loop cols
```
4.3. Using \texttt{pyJac} functionalities

Here, we replace the standard implementation for temporal derivatives of state vector with that provided from \texttt{pyJac}. The \texttt{derivatives()} method has the following form

Listing 4.6: \texttt{derivatives()} function arguments in \texttt{cemaPyjacChemistryModel.C}

```cpp
312 template<class ReactionThermo, class ThermoType>
313 void Foam::cemaPyjacChemistryModel<ReactionThermo, ThermoType>::derivatives
314 (
315  const scalar time,
316  const scalarField& c,
317  scalarField& dcdt
318 ) const
319 % \end{verbatim}
```

in which the parameters \texttt{time}, \texttt{c} and \texttt{dcdt} correspond respectively to system time, state vector, and derivatives. As in the \texttt{Jacobian()} method, the state vector \texttt{c} herein denotes pressure, temperature, and species mass fraction. The derivatives function from \texttt{pyJac} is declared as follows

```cpp
void dydt (const double t, const double pres, const double * __restrict__ y, \
  double * __restrict__ dy)
```

in which the parameters \texttt{t}, \texttt{pres}, and \texttt{y} are similar as discussed in \texttt{eval_jacob} from previous section, while the array \texttt{dy} of size \((N_{sp})\) is to be populated by derivatives. The function \texttt{dydt} is defined in \texttt{dydt.c} which is mechanism dependent so we only include the header \texttt{dydt.h} (c.f. Listing 4.4) for function declaration while using dynamic binding for function definition at runtime.

In our implementation, we replace the whole definition of the \texttt{derivatives()} method with the following

Listing 4.7: Definition of the \texttt{derivatives()} method in \texttt{cemaPyjacChemistryModel.C}

```cpp
128 {
129  // Arrays to be passed into PyJac function call for derivatives
130  std::vector<double> TY(nSpecie_-1, 0.0);
131  // if TY has N+1 elements, diff(TY) has N elements
132  std::vector<double> dTYdt(nSpecie_, 0.0);
133  // state vector has now PTY instead of cTp
134  const scalar p = c[0];
135  const scalar T = c[1];
136  scalar csum = 0.0;
137  forAll(c_, i)
138  {
139    c_[i] = max(c_[i+2], 0.0);
140    csum += c_[i];
141  }
142  // Then we exclude last species from csum and dump all residuals into last
143  // species to ensure mass conservation
144  csum -= c_[nSpecie_-1];
145  c_[nSpecie_-1] = 1.0 - csum;
146  // Loop rows
147  for (label j = 1; j < nSpecie_-1; ++j)
148  {
149    // Loop rows
150    for (label i = 1; i < nSpecie_-1; ++i)
151    {
152      dfdc(i,j) = dfdy[k + i - 1];
153    }
154    k += nSpecie_;  
155  }
156  // Note that dc/dt is not needed in most ODE solvers so here we just return 0
157  dcdt = Zero;
158 }
```
4.3. Using pyJac functionalities

We hereby just replace the chemical enthalpy (i.e. enthalpy of formation) with the corresponding data from pyJac. The function responsible for chemical enthalpy from pyJac, eval_h(), is declared in chem_utils.h. Such a header declares many features that can be used (e.g. specific heat capacity) for a larger reliance on pyJac, however, we only herein demonstrate the usage of chemical enthalpy. Linking to the mechanism dependent function definition is left for runtime dynamic binding. While declarations have been included through Listing 4.4, the corresponding pyJac function for enthalpy evaluation has the following form:

```cpp
void eval_h (const double T, double * __restrict__ h)
```

Since the enthalpy of formation for a given mechanism at reference temperature is not going to change throughout the simulation, perhaps it could be better to just keep the data as member of the class. Accordingly, we declare the scalarField sp_enth_form as shown in Listing 4.8.

```cpp
// Enthalpy of formation for every species, from PyJac
scalarList sp_enth_form;
```

Then, we initialize it with size of \(N_{sp}\) as shown in Listing 4.9.

```cpp
sp_enth_form(nSpecie_)
```

After that, in the class constructor, we update it with data from pyJac as shown in Listing 4.10.

```cpp
if (this->chemistry_)
{
    Info << "\n Evaluating species enthalpy of formation using PyJac\n" << endl;
    // Enthalpy of formation for all species
    std::vector<scalar> sp_enth_form_(nSpecie_, 0.0);
    // Enthalpy of formation is taken from pyJac at T-standard (chem_utils.h)
    eval_h(298.15, sp_enth_form_.data());
    for (label i = 0; i < nSpecie_; ++i)
    {
        sp_enth_form[i] = sp_enth_form_[i];
    }
}
```

Finally, the implementation for \(\dot{\omega}_h\) is updated to incorporate the pyJac data as shown in Listing 4.11.
4.3. Using pyJac functionalities

Listing 4.11: Modify heat release rate implementation to incorporate formation enthalpy from pyJac

```cpp
forAll(Y_, i)
{
    forAll(Qdot, celli)
    {
        scalar hi = sp_enth_form[i];
        Qdot[celli] -= hi*RR_[i][celli];
    }
}
```

4.3.5 Updating solve method from chemistry model

The most important implications due to pyJac usage is the mass fraction based Jacobian matrix. This implies that, according to (2.16), the state vector needs to be also based on mass fractions instead of concentrations. This is realized in the solve() method of the chemistryModel, wherein the data passed to the function call of chemistrySolver are pressure, temperature, and mass fractions, c.f. Section 3.3. In the code, replacement of concentrations into mass fractions is adopted for the data to be passed to solve() function call of chemistrySolver as in the following Listing 4.12.

Listing 4.12: Use species mass fractions instead of concentrations in cemaPyjacChemistryModel.C

```cpp
for (label i=0; i<nSpecie_; i++)
{
    // c_[i] = rhoi*Y_[i][celli]/specieThermo_[i].W();
    c_[i] = Y_[i][celli];
    c0[i] = c_[i];
}
```

Then, as shown in Listing 4.13, the concentration-mass conversion also needs to be achieved when computing the source terms, noting that $\dot{\omega}_k$ units must be consistent in (2.3), i.e. $(c_iW_i = \rho Y_i)$.

Listing 4.13: Use species mass fractions instead of concentrations in chemistry source terms

```cpp
for (label i=0; i<nSpecie_; i++)
{
    // CHEMICAL SOURCE TERM PER SPECIES
    // RR_[i] = c_[i] - c0[i])*specieThermo_[i].W()/deltaT[celli];
    this->RR_[i][celli] = rhoi*(this->c_[i] - c0[i])/deltaT[celli];
}
```

At this point, we discussed all modifications necessary to incorporate chemical Jacobian matrix from pyJac in the chemistry model. In the following section we discuss modifications required in the ODE approach of the chemistry solver.

4.3.6 Updating solve method from chemistry solver

Prior to discussion, we note that the following modifications concern the ODE approach of the chemistrySolver class, i.e. within the chemistryModel library and not the ODE library of OpenFOAM.

It is demonstrated in the previous section that species mass fractions are used instead of concentrations. The mass fraction based state vector, defined by $c_-$, $T_i$, and $p_i$ variables is used for calling the odePyjac::solve() method in chemistrySolver class as shown in Listing 4.14.

Listing 4.14: Using species mass fractions instead of concentrations while calling odePyjac::solve() in chemistrySolver

```cpp
while (timeLeft > SMALL)
{
    scalar dt = timeLeft;
    // Calls ode::solve() from chemistrySolver
    this->solve(c_, Ti, pi, dt, this->deltaTChem_[celli]);
    timeLeft -= dt;
}
```
Here, we provide insight toward the required modifications in the `odePyjac::solve()` method as a response to the adopted changes in the state vector. First, as the Jacobian matrix is evaluated from `pyJac` for species mass fractions excluding last species, we have a total of $N_{sp} + 1$ equations to be solved when also considering pressure and temperature. Accordingly, we modify the total solve vector $cTp_-$ to have a size of $N_{sp} + 1$ before which the ODE solver is called, c.f. Section 3.3. This is achieved by modifying the inline method to exclude last species from computations of the solve vector as shown in Listing 4.15.

Listing 4.15: Modify number of equations to exclude solving for last species

```cpp
template<class ReactionThermo, class ThermoType>
inline Foam::label Foam::cemaPyjacChemistryModel<ReactionThermo, ThermoType>::nEqns() const
{
  // nEqns = number of species (N-1) + temperature + pressure
  return nSpecie_ + 1;
}
```

The updated size ($N_{sp} + 1$) is used to initialize the total solve vector in the class constructor as follows

Listing 4.16: Initialize total solve vector with size of $N_{sp} + 1$ during construction of class `odePyjac`

```cpp
cTp_(this->nEqns())
```

Finally, the full implementation of the `odePyjac::solve()` method is presented in Listing 4.17.

Listing 4.17: Definition of `odePyjac::solve()` method in chemistrySolver

```cpp
template<class ChemistryModel>
void Foam::odePyjac<ChemistryModel>::solve
(
  scalarField& c,
  scalar& T,
  scalar& p,
  scalar& deltaT,
  scalar& subDeltaT
) const
{
  // Reset the size of the ODE system to the simplified size when mechanism
  // reduction is active
  if (odeSolver_-&->resize())
  {
    odeSolver_-&->resizeField(cTp_);
  }
  const label nSpecie = this->nSpecie();
  // Copy the concentration, T and P to the total solve-vector (N+1)
  cTp_[0] = p;
  cTp_[1] = T;
  // Update for N-1 species
  for (label i=0; i<nSpecie-1; i++)
  {
    cTp_[i+2] = c[i];
  }
  // Here, we call ODE solver...
  odeSolver_-&->solve(0, deltaT, cTp_, subDeltaT);
  // Back substitute results
  p = cTp_[0];
  T = cTp_[1];
  scalar csum = 0;
  for (label i=0; i<nSpecie-1; i++)
  {
```

34
c[i] = max(0.0, cTp_[i+2]);
csum += c[i];
}
// Last species

// Last species

// Last species

c[nSpecie-1] = 1.0 - csum;
Chapter 5

CEMA for OpenFOAM

5.1 Implementing CEMA

In order to implement CEMA, as presented in Section 2.3, first we need to store the chemical Jacobian matrix as member of the class so that we can operate on the data using separate methods. After that, we need to create a new field to be populated using CEMA. Accordingly, we declare the square matrix chemJacobian_ for the Jacobian matrix, and the field cem_ as it follows in Listing 5.1.

Listing 5.1: Declarations of the Jacobian matrix and CEMA field as members of the class cemaPyjacChemistryModel

```c
label nElements_; 
// Jacobian from chemistry problem, from pyJac 
mutable scalarSquareMatrix chemJacobian_; 
```

After that, we initialize both chemJacobian_ and cem_ as shown in Listing 5.2.

Listing 5.2: Initialization of chemJacobian_ and cem_ variables in cemaPyjacChemistryModel.C

```c
sp_enthalpy_(nSpecie_), 
nElements_ (BasicChemistryModel<ReactionThermo>::template get<label>("nElements")), 
chemJacobian_ (nSpecie_), 
cem_ 
( 
  IDobject 
  ( 
    "cem", 
    this->mesh_.time().timeName(), 
    this->mesh_, 
    IDobject::NO_READ, 
    IDobject::AUTO_WRITE 
  ), 
  this->mesh_, 
dimensionedScalar("cem", dimless, 0),
```

Here, we note that the chemical Jacobian has size of $N_{sp}$ in order to hold information for temperature and species mass fractions excluding last species. After that, we read the number of elements of the chemical mechanism as specified by the user in chemistryProperties dictionary using the keyword nElements. This information is necessary in order to exclude the $M + 1$ insignificant eigenvalues due to conservation modes, as discussed in Section 2.3. This is achieved through the following declaration and definition of the variable nElements_ in Listing 5.4 and Listing 5.4.

Listing 5.3: Declaration of nElements_ variable in cemaPyjacChemistryModel.H
5.2 Building cemaPyjacChemistryModel with Make

In order to build the library cemaPyjacChemistryModel, we use the OpenFOAM wmake compilation script. The corresponding Make/files and Make/options are depicted in Listing 5.7 and Listing 5.8, respectively.

Listing 5.7: Make (files) for the cemaPyjacChemistryModel library compilation

```bash
makeChemistryModels.C
makeChemistrySolvers.C
LIB = $(FOAM_USER_LIBBIN)/libcemaPyjacChemistryModel
```

Listing 5.8: Make (options) for the cemaPyjacChemistryModel library compilation

```bash
EXE_INC = \
-$(LIB_SRC)/finiteVolume/include \n-$(LIB_SRC)/meshTools/include \n-$(LIB_SRC)/ODE/include \
```

Now, in order to perform eigendecomposition, we can use implementations from EigenMatrix class. All we need is to include declaration headers, as shown in Listing 5.5, and the implementations are defined in standard OpenFOAM library which is already linked.

Listing 5.5: Include declaration header file for EigenMatrix class

```c
#include "simpleMatrix.H"
```

Finally, the full implementation is presented in the following Listing 5.6.

Listing 5.6: Definition of the cema() method to compute $\lambda_{\exp}$

```c
template<class ReactionThermo, class ThermoType>
void Foam::cemaPyjacChemistryModel<ReactionThermo, ThermoType>::cema
(scalar& cem)
{
    const Foam::EigenMatrix<scalar> EM(chemJacobian_);
    DiagonalMatrix<scalar> EValsRe(EM.EValsRe());
    DiagonalMatrix<scalar> EValsIm(EM.EValsIm());
    DiagonalMatrix<scalar> EValsMag(EValsRe.size(), 0.0);
    forAll(EValsRe, i)
    {
        EValsMag[i] = (EValsRe[i]*EValsRe[i] + EValsIm[i]*EValsIm[i]);
    }
    const auto ascend = [&](scalar a, scalar b){ return a < b; };  
    const List<label> permut(EValsMag.sortPermutation(ascend));
    // Skip conservation modes for elements and temperature
    for (label i=0; i<nElements_+1; ++i)
    {
        label idx = permut[i];
        EValsRe[idx] = -1E30;
    }
}
```
As it is observed, the Make files are actually templates to create instances of the chemistry models and chemistry solvers templated on the type of thermodynamics. The macros required to instantiate chemistry models and chemistry solvers according to compressibility and transport types, as well as adding them to runTime selection table, are depicted respectively in Listings 5.9 and 5.10.

Listing 5.9: Macros for chemistry models based on compressibility and transport types

```cpp
#define makeChemistryModel(Comp) 
  typedef BasicChemistryModel<Comp> BasicChemistryModel##Comp; 
  defineTemplateTypeNameAndDebugWithName 
  ( 
    BasicChemistryModel##Comp, 
    "BasicChemistryModel<"##Comp"">", 
    0 
  ); 
  defineTemplateRunTimeSelectionTable 
  ( 
    BasicChemistryModel##Comp, 
    thermo 
  );

#define makeChemistryModelType(SS, Comp, Thermo) 
  typedef SS<Comp, Thermo> SS##Comp##Thermo; 
  defineTemplateTypeNameAndDebugWithName 
  ( 
    SS##Comp##Thermo, 
    (SS"<"##Comp""," + Thermo::typeName() + ">")\c_str(), 
    0 
  );
```

Listing 5.10: Macros for chemistry solvers based on compressibility and transport types

```cpp
#define makeChemistrySolverType(SS, Comp, Thermo) 
  typedef SS<Comp, Thermo> SS##Comp##Thermo; 
  defineTemplateTypeNameAndDebugWithName 
  ( 
    SS##Comp##Thermo, 
    (SS"<"##Comp""," + Thermo::typeName() + ">")\c_str(), 
    0 
  );
```
Finally, the templated instances of the chemistry models and chemistry solvers, which are specified in `Make/files`, c.f. Listing 5.7, create the instances for all combinations of OpenFOAM thermodynamic and transport models. Due to their lengthy code, definitions of the corresponding files are shown in Appendix A.2.3.
Chapter 6

Tutorial on 1D planar premixed flame

In this tutorial, we demonstrate the applicability of the developed model on a laminar one-dimensional unstrained planar premixed flame comprising methane and air at equivalence ratio of $\phi = 0.5$ and thermodynamic conditions of $T = 900$ K and $p = 1$ atm. The core file structure for this demonstration is based on the default OpenFOAM tutorial counterFlowFlame2D_GRI which employs the GRI-3.0 mechanism [22] that we are going to use as well. Accordingly, The tutorial case is copied to a convenient workspace together with some initial cleaning as follows.

```bash
> cp -r $FOAM_TUTORIALS/combustion/reactingFoam/laminar/counterFlowFlame2D_GRI \\
$FOAM_RUN/cemaPyjac_tutorial
> rm $FOAM_RUN/cemaPyjac_tutorial/0/alphat
> rm $FOAM_RUN/cemaPyjac_tutorial/constant/reactions
> rm $FOAM_RUN/cemaPyjac_tutorial/constant/thermo.compressibleGas
```

In the following, we first discuss the generation and compilation of the relevant pyJac subroutines for the GRI-3.0 mechanism. After that, we proceed by discussing the case setup in terms of initial and boundary conditions, mesh generation, finite-volume settings, and IO and control options. After executing all necessary commands to run the simulation using the newly developed model, the results are presented with validations and demonstrations of the pyJac analytical Jacobian and subsequently CEMA fields.

6.1 Preparation of the chemical mechanism

First, it is expected from the reader to have successfully installed pyJac on their computer. The package is hosted by GitHub and can be accessed through https://github.com/SLACKHA/pyJac. Moreover, brief instructions on the package installation are found in Section 4.3.1, while detailed instructions can be found through the online documentation http://slackha.github.io/pyJac.

After navigating to the tutorial workspace, we find that the mechanism reaction file (reactionsGRI) and thermodynamic file(thermo.compressibleGasGRI), which are both located in constant directory, are in OpenFOAM format. In order to generate corresponding pyJac subroutines, the mechanism files in either Chemkin or Cantera format should be provided. Accordingly, we download the original files of the GRI-3.0 mechanism from the website http://combustion.berkeley.edu/gri-mech/version30/text30.html and we store the 3 files, namely grimech30.dat, thermo30.dat, and transport.dat in directory named mechanism ¹ in the case folder.

Now, we are ready to use pyJac to generate the corresponding subroutines. This can be executed as follows.

¹Remember to save the files in *.dat format and not *.dat.txt.
6.1. Preparation of the chemical mechanism

Chapter 6. Tutorial on 1D planar premixed flame

```bash
> cd $FOAM_RUN/cemaPyjac_tutorial/mechanism
> python3 -m pyjac --lang c --input grimech30.dat --thermo thermo30.dat --last_species N2
```

The previous Python command with `-m` switch executes the `pyJac` module through `__main__.py`. The remaining arguments of the command are `pyJac` related, and they are responsible for the programming language of the output source files (i.e. `--lang`), the input mechanism filename (i.e. `--input`), the thermodynamic database (i.e. `--thermo`), and the species name to be set as the last species for dumping all numerical residuals (i.e. `--last_species`), cf. Section 4.3.

The result of the previous command is a new subdirectory named `out`, located inside `mechanism` directory. We can see that definition of analytical Jacobian evaluation of the GRI-3.0 mechanism is realized in `eval_jacob()` function located in `out/jacob.c` and `out/jacobs/jacob_*.c`. Also, functionalities for derivatives and enthalpy of formation are realized in `dydt()` and `eval_h()`, located in `out/dydt.c` and `out/chem_utils.c`, respectively.

Therefore, we need compile all these source code files into a shared object to be dynamically linked to OpenFOAM, providing all function definitions previously declared when compiling the model 2. The compilation process can be achieved through CMake, and the corresponding set of directives and instructions for the source code files and targets are described in the following `CMakeLists.txt`.

Listing 6.1: `CMakeLists.txt` to compile mechanism source code files generated from `pyJac`

```cmake
# cmake_minimum_required(VERSION 2.6)
project(pyJac)

set(CMAKE_BUILD_TYPE Release)

enable_language(C)

set(CMAKE_C_FLAGS "-std=c99 -O3 -mtune=native -fPIC")

include_directories(out)
include_directories(out/jacobs)

file(GLOB_RECURSE SOURCES "out/*.c")

add_library(c_pyjac SHARED $(SOURCES))

install(TARGETS c_pyjac DESTINATION .)
```

After creating `CMakeLists.txt` inside the `mechanism` directory, the build process can be achieved by executing the following commands.

```bash
> cd $FOAM_RUN/cemaPyjac_tutorial/mechanism
> mkdir build
> cd build && cmake .. -DCMAKE_C_COMPILER=cc
> make
> cp libc_pyjac.so ../../constant/
```

Here, the target name of the resulting shared object is `libc_pyjac.so` and it is located inside `build` directory, so we conveniently copy it to `constant` so that all mechanism files are there. The shared library object can be dynamically linked, together with the developed model, through the `controlDict` by specifying them as follows.

```bash
libs ("libcemaPyjacChemistryModel.so"
      "$FOAM_CASE/constant/libc_pyjac.so")
```

2Recall that all function declarations of `pyJac` are mechanism independent.

---

41
It is worth noting that the decision of locating the shared object `libc_pyjac.so` inside the case directory (i.e. to be case specific) is to avoid interference of mechanism function definitions when running multiple OpenFOAM cases using different chemical mechanisms. In simple words, locating the `pyJac` mechanism object into the conventional OpenFOAM user libraries path (i.e. `$FOAM_USER_LIBBIN`) might result into erroneous results when other simulations, using the developed model, run while using different mechanisms than the one compiled in `libc_pyjac.so`.

At this point, the `constant` directory contains the original OpenFOAM reaction and thermodynamic mechanism files for GRI-3.0, along with the `pyJac` analytical based functionalities. There are two important notes to herein consider. First, reaction type and Arrhenius coefficients are no longer required when `pyJac` is used. The reason is that `pyJac` already contains all the information regarding reaction rates, progress rate of reactions, thermo-chemical Jacobian, among others, which are necessary for computing the chemistry source terms. Second, the ordering of chemical species in the mechanism reaction file must follow the same ordering to that in `pyJac`. This is particularly important since we specified $N_2$ to be the last species (i.e. using `--last_species` switch) to handle all residuals, c.f. Section 4.3. The correct species ordering that must be followed can be retrieved from `mechanism/out/mechanism.h`. Therefore, we create a mechanism reaction file `reactionsGRIPyjac`, modified from `constant/reactionsGRI`, and it is depicted in Listing 6.2.

Listing 6.2: Reaction mechanism file `reactionsGRIPyjac` located in `constant`
A final note of this section is to remind the user to manually specify —again— the number of elements of the chemical mechanism (defined in reactionsGRIPyjac) into the chemistryProperties dictionary, so as to be used by CEMA subroutine in the chemistry model. The keyword nElements needs to be used as follows for this particular mechanism.

Listing 6.3: Number of elements of the reaction mechanism file reactionsGRIPyjac to be specified in chemistryProperties for CEMA

```plaintext
nElements 5;
```

After reaching this point, we are ready to proceed with the case setup.

### 6.2 Case setup

In this section, we discuss the choices for initial and boundary conditions, domain discretization, finite-volume numerical schemes and solver settings, and the IO control options.

#### 6.2.1 Internal and boundary conditions

The present tutorial aims at simulating a one-dimensional (1D) unstrained planer premixed flame of methane and air mixture at equivalence ratio of $\phi = 0.5$ and at temperature and pressure of $T = 900$ [K] and $p = 1$ [atm], respectively. In order to ensure a quasi-stabilized flame, we initialize the domain with non-uniform internal fields for the temperature and key species, namely CH$_4$, O$_2$, N$_2$, CO$_2$, and H$_2$O, such that an ignition event is initialized. The spatial distribution of the aforementioned profiles is set to attain temperature rise after nearly halfway of the domain length from inlet boundary. Stabilization of the reaction front is achieved by setting the inlet velocity equivalent to the laminar burning velocity $u_L \approx 1.39$ [m/s]. A data interpolation is performed on the initialized profiles of temperature and the aforementioned key species to match the desired grid points, as it is further elaborated in Section 6.2.2. The boundary conditions are presented in Table 6.1.
Table 6.1: Internal and boundary conditions of the tutorial case. Symbol $\gamma$ denotes specific heat ratio.

| Field variable | Inlet                                | Internal                        | Outlet                               |
|----------------|--------------------------------------|----------------------------------|--------------------------------------|
| Pressure [Pa]  | waveTransmissive ($\gamma = 1.32$) | 101325                           | waveTransmissive ($\gamma = 1.27$) |
| Temperature [K]| 900                                  | non-uniform $^a$                 | zero gradient                        |
| CH$_4$ molar % | 4.98812 (2.83654)                    | non-uniform                      | zero gradient                        |
| O$_2$ molar %  | 19.9525 (22.631)                     | non-uniform                      | zero gradient                        |
| N$_2$ molar %  | 75.05938 (74.53246)                  | non-uniform                      | zero gradient                        |
| CO$_2$ molar % | 0.0 (0.0)                            | non-uniform                      | zero gradient                        |
| H$_2$O molar % | 0.0 (0.0)                            | non-uniform                      | zero gradient                        |
| Equivalence ratio $^b$ ($\phi$) | 0.5                                  | 0.5                              | 0.5                                  |
| Velocity [m/s] | (1.456, 0, 0)                        | 0.0                              | zero gradient                        |

$^a$ Spatial distribution profiles are interpolated. Corresponding scripts are provided in Appendix A.5.

$^b$ For unity Lewis number of all species, which is the case in the present simulation, local equivalence ratio is expected to be rather constant and equals to that in the reactants. For simulations with non-unity Lewis number, variation is expected to lower down across the reaction front, before approaching again the local value of the reactants after crossing the front toward product side [23]. This is beyond the scope of this project.

Regarding the corresponding files for initial and boundary conditions, they are presented in Listings 6.4 and 6.5 for temperature and methane (CH$_4$), respectively. The remaining key species are specified in an analogous manner to that for CH$_4$ while modifying the uniform value of the inlet boundary by using Table 6.1, and updating the species names in the two occurrences of the dictionary. The interpolated data that is included using the command `#include"CH4.dat"` are provided in the tutorial case folder under 0 directory. Moreover, it is possible to re-generate the data using the attached helper scripts in utilities directory. The utilities scripts are written in Python and they require Cantera package to be installed. After executing `compute_initial_fields.py` and then `interpolate_initial_fields.py`, the generated *.dat files will be located inside out_states subdirectory which are then required to be copied into 0 directory of the tutorial case.

Listing 6.4: Initial and boundary conditions for temperature field

```plaintext
FoamFile
{
  version 2.0;
  format ascii;
  class volScalarField;
  location "0";
  object T;
}

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

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

internalField  nonuniform #include"T.dat" ;

boundaryField
{
  inlet
  {
    type fixedValue;
    value uniform 900;
  }

  outlet
  {
    type zeroGradient;
  }
```

44
6.2. Case setup

Chapter 6. Tutorial on 1D planar premixed flame

---

It is worth mentioning that the chosen approach of initializing the ignition event by interpolating spatial profiles of temperature and key species is made for simulation feasibility. In particular, for the initial conditions of pressure and temperature presented in Table 6.1, ignition delay time (IDT) of methane/air mixture at $\phi = 0.5$ is about 5.39 [s] which is considered too long, considering the relatively small temporal and spatial length scales of the simulation as it will be further elaborated in Sections 6.2.2 and 6.2.5.

6.2.2 Domain discretization

Aiming for resolving the flame thermal thickness ($\delta_L \approx 3.72 \times 10^{-4}$ [m] for the conditions specified in Table 6.1), the grid spacing is chosen to be $1.5 \times 10^{-5}$ [m]. Therefore, thermal thickness is resolved with more than 20 grid points. The domain length is set to $2 \times 10^{-2}$ [m] hence a total amount of 1350 grid points, in which the number is inserted to the interpolator to generate arrays $0/*.dat$ of 1350 internal grid points in the discretized domain. The `blockMesh` dictionary is depicted in Listing 6.6.

---

**Listing 6.5: Initial and boundary conditions for methane field**

```plaintext
FoamFile
{
  version 2.0;
  format ascii;
  class volScalarField;
  location "0";
  object CH4;
}
// ******************************************************************************
//dimensions [0 0 0 0 0 0 0];
internalField nonuniform
#include"CH4.dat"
;
boundaryField
{
  inlet
  {
    type fixedValue;
    value uniform 0.0283654;
  }
  outlet
  {
    type zeroGradient;
  }
  frontAndBack
  {
    type empty;
  }
}
```

**Listing 6.6: blockMesh dictionary**

```plaintext
FoamFile
{
  version 2.0;
  format ascii;
  class dictionary;
}
```

---


6.2. Case setup

Chapter 6. Tutorial on 1D planar premixed flame

```plaintext
object blockMeshDict;

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

scale 0.01;

vertices
(
    (0 0 0)
    (2.0 0 0)
    (2.0 0.25 0)
    (0 0.25 0)
    (0 0 0.25)
    (2.0 0 0.25)
    (2.0 0.25 0.25)
    (0 0.25 0.25)
);

blocks
(
    hex (0 1 2 3 4 5 6 7) (1350 1 1) simpleGrading (1 1 1)
);

edges
(
);

boundary
(
    inlet
    {
        type patch;
        faces
        (4 7 3 0)
    );

    outlet
    {
        type patch;
        faces
        (1 2 6 5)
    );

    frontAndBack
    {
        type empty;
        faces
        (0 3 2 1)
        (4 5 6 7)
        (7 6 2 3)
        (1 0 4 5)
    );

    mergePatchPairs
    ( );

);
6.2.3 Finite volume settings

Modifications in the finite-volume numerical schemes and solver settings, with respect to default ones from `counterFlowFlame2D_GRI` tutorial, should not have major impact on the results. Nevertheless, our preferences for the numerical schemes and solver options are herein presented. Modifications on numerical schemes are adopted through the following commands

```bash
> cd $FOAM_RUN/cemaPyjac_tutorial
> sed -i 's/Euler/backward/g' system/fvSchemes
> sed -i 's/limitedLinearV/limitedLinear/g' system/fvSchemes #not necessary
> sed -i 's/Gauss linear orthogonal/Gauss linear corrected/g' system/fvSchemes
```

For solver settings, the updated `fvSolution` dictionary is depicted in Listing 6.7.

```
FoamFile
{
    version 2.0;
    format ascii;
    class dictionary;
    location "system";
    object fvSolution;
}

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

solvers
{
  rho
  {
    solver PCG;
    preconditioner DIC;
    tolerance 1e-8;
    relTol 0.005;
  }
  rhoFinal
  {
    $rho;
    tolerance 1e-8;
    relTol 0.0;
  }
  p
  {
    solver PCG;
    preconditioner DIC;
    tolerance 1e-8;
    relTol 0.005;
  }
  pFinal
  {
    $p;
    tolerance 1e-8;
    relTol 0.0;
  }
  "(U|h|k|epsilon)"
  {
    solver PBiCGStab;
    preconditioner DILU;
    tolerance 1e-7;
    relTol 0.005;
  }
  "(U|h|k|epsilon)Final"
  {
```

Listing 6.7: fvSolution dictionary
### 6.2.4 Chemistry and thermophysical properties

At this point, the tutorial case files are updated for the mechanism files (inside `constant` directory, copied from `mechanism` directory), the initial and boundary conditions (inside `0` directory), the mesh data (inside `constant` directory), and the finite-volume settings (inside `system` directory).

Now, we update the chemistry and thermophysical properties in the `constant` directory. The `chemistryProperties` dictionary needs to be properly updated to allow using `cemaPyjac` chemistry model, while modifying stiff ODE solver tolerances for faster computations. The modifications can be adopted through the following commands.

```bash
> sed -i 's/ode;/odePyjac;/g' constant/chemistryProperties
> sed -i 's/initialChemicalTimeStep 1e-7/initialChemicalTimeStep 1e-8/g' constant/chemistryProperties
> sed -i 's/1e-12/1e-08/g' constant/chemistryProperties # absTol
> sed -i 's/1e-1/1e-05/g' constant/chemistryProperties # relTol
```

Another note is to remember specifying the number of elements (using `nElements` keyword) of the chemical mechanism, which can be looked up from `reactionsGRIPyjac`. This is particularly important for CEMA computations to skip the conservative modes and thereby accurately identify the CEM. The final form of the `chemistryProperties` dictionary is depicted in Listing 6.8.

**Listing 6.8: chemistryProperties dictionary**

```plaintext
FoamFile
{
    version 2.0;
    format ascii;
    class dictionary;
    location "constant";
    object chemistryProperties;

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

    chemistryType
    {
        solver odePyjac; // ode; odePyjac;
        method cemaPyjac; // standard; cemaPyjac;
    }
}
```
6.2. Case setup

Chapter 6. Tutorial on 1D planar premixed flame

```plaintext
chemistry on;
initialChemicalTimeStep 1e-8;
odeCoeffs
{
    solver seuler;
    absTol 1e-08;
    relTol 1e-05;
}
nElements 5;
```

The `thermophysicalProperties` dictionary needs to be updated only to specify the path of the modified reaction mechanism file (i.e. with species correct ordering and without reaction rate data, c.f. Section 6.1). This is achieved through the following command.

```plaintext
> sed -i 's/reactionsGRI/reactionsGRIPyjac/g' constant/thermophysicalProperties
```

The updated `thermophysicalProperties` dictionary should look like the following `Listing 6.9`. The remaining dictionaries in `constant` directory for turbulence and combustion properties are kept as laminar, similar to the original `counterFlowFlame2D_GRI` tutorial.

```
FoamFile
{
    version 2.0;
    format ascii;
    class dictionary;
    location "constant";
    object thermophysicalProperties;
}
// * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * //
thermoType
{
    type hePsiThermo;
    mixture reactingMixture;
    transport sutherland;
    thermo janaf;
    energy sensibleEnthalpy;
    equationOfState perfectGas;
    specie specie;
}
inertSpecie N2;
chemistryReader foamChemistryReader;
foamChemistryFile "<constant>/reactionsGRIPyjac";
foamChemistryThermoFile "<constant>/thermo.compressibleGasGRI";
```

6.2.5 IO control and dynamic linking

When it comes to IO control through `controlDict` file, it is always up to user’s personal preferences to decide on the total simulation time as well as the data writing format and frequency. However, the essential parts to be noted are that the numerical time step and total simulation time both need to be adequate with regard to domain characteristic length and time scales. Moreover, the simulation time should be specified sufficiently long to allow the reactive flow to fully develop in terms of species concentrations and reaction front establishment. Compared with the default `controlDict` from `counterFlowFlame2D_GRI`, the modifications can be adopted through the following commands.

```plaintext
> sed -i 's/0.5/0.002/g' system/controlDict #endTime
```
6.2. Case setup

Chapter 6. Tutorial on 1D planar premixed flame

> sed -i 's/1e-6/1e-8/g' system/controlDict #deltaT
> sed -i 's/adjustable/adjustableRunTime/g' system/controlDict #writeControl
> sed -i 's/0.05/1e-04/g' system/controlDict #writeInterval
> sed -i 's/0.4/0.3/g' system/controlDict #maxCo

The shared object for the chemistry model and for pyJac are both dynamically linked as previously discussed in Section 6.1. The controlDict file after all modifications is depicted in Listing 6.10

Listing 6.10: controlDict

```plaintext
FoamFile
{
    version 2.0;
    format ascii;
    class dictionary;
    location "system";
    object controlDict;
}
// * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * //
application reactingFoam;
startFrom startTime;
startTime 0;
stopAt endTime;
endTime 0.002;
deltaT 1e-8;
writeControl adjustableRunTime;
writeInterval 1e-04;
purgeWrite 0;
writeFormat ascii;
writePrecision 6;
writeCompression off;
timeFormat general;
timePrecision 6;
runTimeModifiable true;
adjustTimeStep yes;
maxCo 0.3;
libs {
    "libcemaPyjacChemistryModel.so"
    "$FOAM_CASE/constant/libc_pyjac.so"
};
```
At this point, the tutorial case is ready to be run. To summarize what we have done, we list the procedure in brief as in the following.

1. Copy the original tutorial `counterFlowFlame2D_GRI` of `reactingFoam` solver to a desired workspace.

2. Download chemical mechanism (GRI-3.0) in Chemkin format from the original website. While having `pyJac` installed, generate source code for the GRI-3.0 mechanism involving analytical Jacobian using `pyJac`. After that, build the files using `CMake` and copy the shared object `libc_pyjac.so` to somewhere inside the case directory such as `constant`.

3. Modify original chemical reaction file such that reaction rate information are omitted, and species index ordering is consistent with `pyJac`, which can be retrieved from `out/mechanism.h` in the generated mechanism source code.

4. Adjust case setup in terms of initial and boundary conditions, domain discretization, and finite-volume settings as desired, or according to the prescribed settings in the report.

5. Modify `chemistryProperties` dictionary to use the developed chemistry model `cemaPyjac` while specifying `odePyjac` for the analytical Jacobian based stiff ODE solver. Moreover, specify number of elements of the chemical mechanism, retrieved from mechanism reaction file, using the keyword `nElements`.

6. Modify `thermophysicalProperties` dictionary to specify location of the updated mechanism reaction file, using keyword `foamChemistryFile`, with the corrected species index ordering and omitted reaction rate data.

7. Perform dynamic binding of the developed model shared object `libcemaPyjacChemistryModel.so` and the mechanism’s analytical Jacobian functionality definitions in `libcemaPyjacChemistryModel.so`. Both objects are dynamically linked at runtime through `controlDict`.

Finally, the simulation can be executed through the following commands, while the results are shown in the following section.

```bash
# Remember to source OpenFOAM-v2006 and to compile CEMAFoam
> blockMesh
> reactingFoam
```
6.3 Results-I: Analytical Jacobian validation

The validation results of the simulation after 0.002 [s] are depicted in Figure 6.1. Here, profiles of temperature, heat release rate (HRR), in addition to mass fractions of O₂, CH₄, CH₃, and OH are plotted and compared with standard chemistry model (without pyJac). The presented results verify the implementation accuracy of the developed cemaPyjac chemistry model with the analytical Jacobian formulations from pyJac. We note a small shift in the reaction zone which indicates a slower burning rate in the case of using an analytical Jacobian. A possible reasons behind such a shift could be the mass-based versus concentration-based Jacobian matrix which, in turn, might possibly lead into variations in the iterative procedure and solution convergence rate for both cases. This justification could seem reasonable especially that the relative velocity between inlet flow and laminar burning rate is not exactly zero (i.e. there is slight slip in the front location), therefore the front location is not fixed with respect to its initial location. Next, we present validations of the developed chemistry model for CEMA.

Figure 6.1: Validation of the developed cemaPyjacChemistryModel to computing chemical source terms using analytical Jacobian formulations. Data are sampled after running the simulation for 0.002 [s]. Small shift is noted for the reaction zone indicating smaller burning rate for the case adopting cemaPyjac model with analytical Jacobian.
6.4 Results-II: CEMA demonstration

In this section, we demonstrate results from CEMA functionality of the developed model. The variable $\text{cem}^3$ is the leading non-conservative eigenvalue of the thermo-chemical analytic Jacobian matrix. In Figure 6.2, we present field plots of temperature, OH and CH$_3$ mass fractions, in addition to the cem field. The positive values of cem indicate pre-ignition zones (unstable mode of the dynamical system) and the negative values indicate post-ignition zones (stable mode of the dynamical system), whereas zero-crossing interface can be regarded as the reaction front. As previously mentioned in Chapter 1, the identification of pre- and post-ignition zones as well as zero-crossing is a crucial analysis for various subsequent developments as it is briefly demonstrated in Section 6.5.

Figure 6.2: Validation of the developed cemaPyjacChemistryModel for CEMA using either analytical Jacobian. Zero-crossing CEMA field clearly indicates the reaction front.

\[^3\text{Note that the variable cem denotes leading eigenvalue (i.e. } \lambda_{\text{exp}}\text{) and it is not the chemical explosive mode (CEM) denoting the corresponding eigenvector.}\]
The final results of this section is a comparison between the analytical and numerical Jacobian based implementation for CEMA computations. In Figure 6.3, field plots of temperature, HRR, and \( \text{cem} \) are presented for the numerical based Jacobian (left panel) and analytical based Jacobian (right panel). While both results were almost identical using GRI-3.0 mechanism, the presented results using Yao54 mechanism \([24]\) (another long-chain hydrocarbon skeletal mechanism involving 54 species and 269 reactions) show discrepancies in the preheat zone of \( \text{cem} \) which is evaluated using the numerical Jacobian. This is possibly due to numerical inaccuracies and insufficient significant digits of the Jacobian matrix resulting by finite-differencing. Such notes are further supported by discussions of the original CEMA developments by Lu et al. \([6]\), hence the necessity of incorporating \texttt{pyJac} for analytical Jacobian evaluation in the present developments. Next, we briefly demonstrate the validation of projected CEMA for combustion mode characterization.

Figure 6.3: Comparison of semi-analytical (left panel) and analytical (right panel) based development of \texttt{cemaPyjacChemistryModel} for CEMA. Note the discrepancies in the preheat zone for the case of semi-analytical Jacobian due to numerical inaccuracies and insufficient significant digits of the Jacobian matrix as consistently explained in the original CEMA paper \([6]\).
6.5 Results-III: Projected CEMA demonstration

In this section, we demonstrate one application of CEMA which is the projection of diffusion and reaction terms onto CEM to characterize the local combustion mode [8]. In brief, by recalling derivations from Section 2.3, the projected reaction and diffusion terms onto CEM are described as by the following equations.

\[ \phi_\omega = b_{\exp} \cdot \dot{\omega}, \]  
\[ \phi_s = b_{\exp} \cdot s \]  

The local combustion mode indicator, \( \alpha = \phi_s / \phi_\omega \), compares relative alignment of diffusion and chemistry contributions with relevance to fastest CEM. The validation of the developed model for projected CEMA (against reference data set computed using \textsc{premix} code) along with definitions for the local combustion modes are depicted in Figure 6.4. Implementation details of the projected CEMA approach are not shown in the present report.

- burnt (\( \lambda_{\exp} \leq 0 \))
- fresh (\( \lambda_{\exp} > 0 \cap (\lambda_{\exp} \leq 1 \cap T \leq 1000) \))
- auto-ign. (\( \lambda_{\exp} > 1 \cap T > 1000 \cap |\alpha| < 1 \))
- diff. assisted (\( \lambda_{\exp} > 1 \cap T > 1000 \cap \alpha > 1 \))
- extinction (\( \lambda_{\exp} > 1 \cap T > 1000 \cap \alpha < -1 \))

Figure 6.4: Validation of the developed \textsc{cemaPyjacChemistryModel} with implementations of reaction and diffusion terms projections onto CEM for local combustion mode identification. Results are compared against reference case from \textsc{premix} code. Implementation details are not demonstrated in the present report.
Acknowledgment

The incorporation of the PyJac package into OpenFOAM chemistryModel has been originated in the thesis works of Dr. Heikki Kahila, Wärtsilä Finland Oy, using OpenFOAM-v6. Further information on particular implementations and framework can be found on his doctoral dissertation at Aalto University.
Bibliography

[1] T. Poinsot and D. Veynante, *Theoretical and numerical combustion*. 2011.

[2] N. Peters, “Turbulent combustion,” 2001.

[3] V. Sick, “High speed imaging in fundamental and applied combustion research,” *Proceedings of the Combustion Institute*, vol. 34, pp. 3509–3530, Jan. 2013.

[4] J. G. Lee and D. A. Santavicca, “Experimental diagnostics for the study of combustion instabilities in lean premixed combustors,” *Journal of Propulsion and Power*, vol. 19, pp. 735–750, Sept. 2003.

[5] R. Shan, C. S. Yoo, J. H. Chen, and T. Lu, “Computational diagnostics for n-heptane flames with chemical explosive mode analysis,” *Combustion and Flame*, vol. 159, pp. 3119–3127, Oct. 2012.

[6] T. F. Lu, C. S. Yoo, J. H. Chen, and C. K. Law, “Three-dimensional direct numerical simulation of a turbulent lifted hydrogen jet flame in heated coflow: a chemical explosive mode analysis,” *Journal of Fluid Mechanics*, vol. 652, pp. 45–64, May 2010.

[7] S. Lam, “Singular perturbation for stiff equations using numerical methods,” in *Recent advances in the aerospace sciences*, pp. 3–19, Springer, 1985.

[8] C. Xu, J.-W. Park, C. S. Yoo, J. H. Chen, and T. Lu, “Identification of premixed flame propagation modes using chemical explosive mode analysis,” *Proceedings of the Combustion Institute*, vol. 37, no. 2, pp. 2407–2415, 2019.

[9] D. Mohaddes, W. Xie, and M. Ihme, “Analysis of low-temperature chemistry in a turbulent swirling spray flame near lean blow-out,” *Proceedings of the Combustion Institute*, vol. 38, no. 2, pp. 3435–3443, 2021.

[10] K. E. Niemeyer, N. J. Curtis, and C.-J. Sung, “pyJac: Analytical jacobian generator for chemical kinetics,” *Computer Physics Communications*, vol. 215, pp. 188–203, June 2017.

[11] R. Issa, B. Ahmadi-Befrui, K. Beshay, and A. Gosman, “Solution of the implicitly discretised reacting flow equations by operator-splitting,” *Journal of Computational Physics*, vol. 93, pp. 388–410, 1991.

[12] T. Poinsot and D. Veynante, *Theoretical and numerical combustion*. RT Edwards, Inc., 2005.

[13] E. Hairer and G. Wanner, *Solving Ordinary Differential Equations II*. Berlin, Heidelberg: Springer Berlin Heidelberg, 1996.

[14] P. Lawrence, “Differential equations and dynamical systems,” 1991.

[15] J. P. La Salle, *The stability of dynamical systems*. SIAM, 1976.

[16] S. Lam and D. Goussis, “Understanding complex chemical kinetics with computational singular perturbation,” *Symposium (International) on Combustion*, vol. 22, pp. 931–941, Jan 1989.
[17] U. Maas and S. Pope, “Simplifying chemical kinetics: Intrinsic low-dimensional manifolds in composition space,” *Combustion and Flame*, vol. 88, pp. 239–264, Mar. 1992.

[18] T. Lu, Y. Ju, and C. K. Law, “Complex CSP for chemistry reduction and analysis,” *Combustion and Flame*, vol. 126, pp. 1445–1455, July 2001.

[19] T. Lu and C. K. Law, “Toward accommodating realistic fuel chemistry in large-scale computations,” *Progress in Energy and Combustion Science*, vol. 35, pp. 192–215, Apr. 2009.

[20] S. H. Lam, “Using CSP to understand complex chemical kinetics,” *Combustion Science and Technology*, vol. 89, pp. 375–404, Mar. 1993.

[21] T. Turanyi, “Reduction of large reaction mechanisms,” *New journal of chemistry (1987)*, vol. 14, no. 11, pp. 795–803, 1990.

[22] G. P. Smith, D. M. Golden, M. Frenklach, N. W. Moriarty, B. Eiteneer, M. Goldenberg, C. T. Bowman, R. K. Hanson, S. Song, W. C. G. Jr., V. V. Lissianski, and Z. Qin, “GRI-Mech 3.0,” [http://combustion.berkeley.edu/gri-mech/](http://combustion.berkeley.edu/gri-mech/), 1999.

[23] H. Lee, P. Dai, M. Wan, and A. Lipatnikov, “Influence of molecular transport on burning rate and conditioned species concentrations in highly turbulent premixed flames,” *Journal of Fluid Mechanics*, vol. 928, Oct. 2021.

[24] T. Yao, Y. Pei, B.-J. Zhong, S. Som, T. Lu, and K. H. Luo, “A compact skeletal mechanism for n-dodecane with optimized semi-global low-temperature chemistry for diesel engine simulations,” *Fuel*, vol. 191, pp. 339–349, Mar. 2017.

[25] M. Gadalla, M. Tezzele, A. Mola, and G. Rozza, “BladeX: Python blade morphing,” *Journal of Open Source Software*, vol. 4, p. 1203, Feb. 2019.
Study questions

1. What is the purpose of direct integration of finite-rate chemistry for chemical source terms?
2. What is the importance of CEMA in the field of computational combustion?
3. Why is it important to incorporate analytical Jacobian for CEMA computations?
4. How to generate analytical Jacobian from given mechanism in Chemkin-II format?
5. What is the purpose of object orientation in software design?
6. I am not very confident of my programming skills and I would like to use the existing Jacobian formulation from OpenFOAM standard chemistry model without linking any external third-party libraries. Is that fine?
Appendix A

Developed codes

A.1 Tree structure

CEMAFoam
  |__ src
  |    |__ thermophysicalModels
  |    |    |__ chemistryModel
  |    |    |    |__ chemistryModel
  |    |    |    |    __ cemaPyjacChemistryModel
  |    |    |    |    |__ cemaPyjacChemistryModelI.H
  |    |    |    |    |__ cemaPyjacChemistryModel.H
  |    |    |    |    |__ cemaPyjacChemistryModel.C
  |    |__ chemistrySolver
  |    |    |__ odePyjac
  |    |    |    |__ odePyjac.H
  |    |    |    |__ odePyjac.C
  |__ Make
  |    |__ files
  |    |__ options
  |    |__ makeChemistryModel.H
  |    |__ makeChemistryModels.C
  |    |__ makeChemistrySolverTypes.H
  |    |__ makeChemistrySolvers.C
  |__ pyjacInclude
  |    |__ chem_utils.h
  |    |__ dydt.h
  |    |__ header.h
  |    |__ jacob.h
  |    |__ mass_mole.h
  |    |__ mechanism.h
  |    |__ rates.h
  |    |__ sparse_multiplier.h
CEMAFoam
  | tutorials
  |   | premixedFlame1D
  |   |   | Allrun
  |   |   |   | U
  |   |   |   | P
  |   |   |   | T
  |   |   |   | T.dat
  |   |   |   | CH4
  |   |   |   | CH4.dat
  |   |   |   | O2
  |   |   |   | O2.dat
  |   |   |   | N2
  |   |   |   | N2.dat
  |   |   |   | CO2
  |   |   |   | CO2.dat
  |   |   |   | H2O
  |   |   |   | H2O.dat
  |   |   |   | Ydefault
  |   | constant
  |   |   | chemistryProperties
  |   |   | reactionsGRIPyjac
  |   |   | thermo.compressibleGasGRI
  |   |   | thermophysicalProperties
  |   |   | turbulenceProperties
  |   |   | combustionProperties
  |   | system
  |   |   | blockMeshDict
  |   |   | decomposeParDict
  |   |   | controlDict
  |   |   | fvSchemes
  |   |   | fvSolution
  |   | mechanism
  |   |   | grimech30.dat
  |   |   | thermo30.dat
  |   |   | transport.dat
  |   |   | out.tgz
  |   |   | runCmake.sh
  |   |   | CMakeLists.txt
  | utilities
  |   | compute_initial_fields.py
  |   | interpolate_initial_fields.py
  |   | ndInterpolator.py
A.2 cemaPyjacChemistryModel library

A.2.1 chemistryModel

cemaPyjacChemistryModel.H

/*-------------------------------------------------------------
------- F ield OpenFOAM: The Open Source CFD Toolbox
------- O peration | www.openfoam.com
------- A nd
------- M anipulation
-------------------------------------------------------------
Copyright (C) 2011-2017 OpenFOAM Foundation -- Author: Mahmoud Gadalla
-------------------------------------------------------------
License
This file is part of CEMAFoam, derived from OpenFOAM.

https://github.com/Aalto-CFD/CEMAFoam

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::cemaPyjacChemistryModel

Description
Extends base chemistry model by adding a thermo package, and ODE functions. Introduces chemistry equation system and evaluation of chemical source terms. Replaces approximated jacobian implementation with analytical one. Adds new functionality for extended Chemical Explosive Mode Analysis (CEMA)

SourceFiles
cemaPyjacChemistryModelI.H
cemaPyjacChemistryModel.C

/*-------------------------------------------------------------
#ifndef cemaPyjacChemistryModel_H
#define cemaPyjacChemistryModel_H
#include "BasicChemistryModel.H"
#include "Reaction.H"
#include "ODESystem.H"
#include "volFields.H"
#include "simpleMatrix.H"

#include "EigenMatrix.H" // FOR EIGEN DECOMPOSITION

extern "C" {
    #include "chem_utils.h"
    #include "dydt.h"
    #include "jacob.h"
};

/*-------------------------------------------------------------
namespace Foam
{
// Forward declaration of classes
class fvMesh;

/*---------------------------------------------------------------------------*
Class cemaPyjacChemistryModel Declaration
\---------------------------------------------------------------------------*/

template<class ReactionThermo, class ThermoType>
class cemaPyjacChemistryModel
:
    public BasicChemistryModel<ReactionThermo>,
    public ODESystem
{
    // Private Member Functions

    // Solve the reaction system for the given time step
    // of given type and return the characteristic time
    template<class DeltaTType>
    scalar solve(const DeltaTType& deltaT);

    // No copy construct
    cemaPyjacChemistryModel
    (const cemaPyjacChemistryModel<ReactionThermo, ThermoType>&)
    = delete;

    // No copy assignment
    void operator=
    (const cemaPyjacChemistryModel<ReactionThermo, ThermoType>&)
    = delete;

protected:

typedef ThermoType thermoType;

    // Protected data

    // Reference to the field of specie mass fractions
    PtrList<volScalarField>& Y_; 

    // Reactions
    const PtrList<Reaction<ThermoType>>& reactions_; 

    // Thermodynamic data of the species
    const PtrList<ThermoType>& specieThermo_; 

    // Number of species
    label nSpecie_; 

    // Number of reactions
    label nReaction_; 

    // Temperature below which the reaction rates are assumed 0
    scalar Treact_; 

    // List of reaction rate per specie [kg/m3/s]
    PtrList<volScalarField::Internal> RR_; 

    // Temporary concentration field
    mutable scalarField c_; 

    // Temporary rate-of-change of concentration field
    mutable scalarField dcdt_; 
}
// Enthalpy of formation for every species, from PyJac ###
scalarList sp_enthalpy_; 

// Number of elements specified in the reaction file
label nElements_;  

// Jacobian from chemistry problem, from pyJac
mutable scalarSquareMatrix chemJacobian_;  

// CEMA geometric fields
volScalarField cem_; 

// Protected Member Functions

//- Write access to chemical source terms
// (e.g. for multi-chemistry model)
inline PtrList<volScalarField::Internal>& RR();

public:

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

// Constructors

//- Construct from thermo
cemaPyjacChemistryModel(ReactionThermo& thermo);

//- Destructor
virtual ~cemaPyjacChemistryModel();

// Member Functions

//- The reactions
inline const PtrList<Reaction<ThermoType>>& reactions() const;

//- Thermodynamic data of the species
inline const PtrList<ThermoType>& specieThermo() const;

//- The number of species
virtual inline label nSpecie() const;

//- The number of reactions
virtual inline label nReaction() const;

//- Temperature below which the reaction rates are assumed 0
inline scalar Treact() const;

//- Temperature below which the reaction rates are assumed 0
inline scalar& Treact();

//- dc/dt = omega, rate of change in concentration, for each species
virtual void omega
(const scalarField& c,
 const scalar T,
 const scalar p,
 scalarField& dcdt ) const;

//- Return the reaction rate for reaction r and the reference
// species and characteristic times
virtual scalar omega
A.2. cemaPyjacChemistryModel library

Appendix A. Developed codes

(` const Reaction<ThermoType>& r,
 const scalarField& c,
 const scalar T,
 const scalar p,
 scalar& pf,
 scalar& cf,
 label& lRef,
 scalar& pr,
 scalar& cr,
 label& rRef
 ) const;

//- Return the reaction rate for iReaction and the reference
// species and characteristic times
virtual scalar omegaI
(
 label iReaction,
 const scalarField& c,
 const scalar T,
 const scalar p,
 scalar& pf,
 scalar& cf,
 label& lRef,
 scalar& pr,
 scalar& cr,
 label& rRef
 ) const;

//- Calculates the reaction rates
virtual void calculate();

// Chemistry model functions (overriding abstract functions in
// basicChemistryModel.H)

//- Return const access to the chemical source terms for specie, i
inline const volScalarField::Internal& RR
(
 const label i
 ) const;

//- Return non const access to chemical source terms [kg/m3/s]
virtual volScalarField::Internal& RR
(
 const label i
 );

//- Return reaction rate of the speciei in reactionI
virtual tmp<volScalarField::Internal> calculateRR
(
 const label reactionI,
 const label speciei
 ) const;

//- Solve the reaction system for the given time step
// and return the characteristic time
virtual scalar solve(const scalar deltaT);

//- Solve the reaction system for the given time step
// and return the characteristic time
virtual scalar solve(const scalarField& deltaT);

//- Return the chemical time scale
virtual tmp<volScalarField> tc() const;

//- Return the heat release rate [kg/m/s3]
virtual tmp<volScalarField> Qdot() const;

// ODE functions (overriding abstract functions in ODE.H)

//- Number of ODE’s to solve
inline virtual label nEqns() const;

virtual void derivatives
{
    const scalar t,
    const scalarField& c,
    scalarField& dcdt
} const;

virtual void jacobian
{
    const scalar t,
    const scalarField& c,
    scalarField& dcdt,
    scalarSquareMatrix& dfdc
} const;

virtual void solve
{
    scalarField &c,
    scalar& T,
    scalar& p,
    scalar& deltaT,
    scalar& subDeltaT
} const = 0;

// Perhaps to propagate for other models
void cema
{
    scalar& cem
} const;
};

// ************************************************************************* //
cemaPyjacChemistryModelI.H
/*---------------------------------------------------------------------------*\
========= |
\ / F ield | OpenFOAM: The Open Source CFD Toolbox
\ / O peration |
\ / A nd | www.openfoam.com
66
A.2. cemaPyjacChemistryModel library

---

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/>.

---

```cpp
#include "volFields.H"
#include "zeroGradientFvPatchFields.H"

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

template<class ReactionThermo, class ThermoType>
inline Foam::label
Foam::cemaPyjacChemistryModel<ReactionThermo, ThermoType>::nEqns() const
{
    // nEqns = number of species (N-1) + temperature + pressure
    return nSpecie_ + 1;
}

// !!! REMAINING ARE SAME AS STANDARD

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

---

```
#include "volFields.H"
#include "zeroGradientFvPatchFields.H"

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

template<class ReactionThermo, class ThermoType>
inline Foam::label
Foam::cemaPyjacChemistryModel<ReactionThermo, ThermoType>::nEqns() const
{
    // nEqns = number of species (N-1) + temperature + pressure
    return nSpecie_ + 1;
}

// !!! REMAINING ARE SAME AS STANDARD

```

---

```
#include "volFields.H"
#include "zeroGradientFvPatchFields.H"

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

---

#include "cemaPyjacChemistryModel.H"
#include "reactingMixture.H"
#include "UniformField.H"
#include "extrapolatedCalculatedFvPatchFields.H"

// * * * * * * * * * * * * * * * * Constructors * * * * * * * * * * * * * * //
template<class ReactionThermo, class ThermoType>
Foam::cemaPyjacChemistryModel<ReactionThermo, ThermoType>::cemaPyjacChemistryModel
(ReactionThermo& thermo)
:
    BasicChemistryModel<ReactionThermo>(thermo),
    ODESystem(),
    Y_(this->thermo().composition().Y()),
    reactions_
    (dynamic_cast<const reactingMixture<ThermoType>&>(this->thermo())
    ),
    specieThermo_
    (dynamic_cast<const reactingMixture<ThermoType>&>(this->thermo()).speciesData()),
    nSpecie_(Y_.size()),
    nReaction_(reactions_.size()),
    Treact_
    (BasicChemistryModel<ReactionThermo>::template getOrDefault<scalar>(
        "Treact",
        0.0
    )
    ),
    RR_(nSpecie_),
    c_(nSpecie_),
    dcdt_(nSpecie_),
    sp_enthalpy_(nSpecie_),
    nElements_(BasicChemistryModel<ReactionThermo>::template get<label>("nElements")),
    chemJacobian_(nSpecie_),
    chem_
    (IOobject
        ("cem",
         this->mesh_.time().timeName(),
         this->mesh_,
         IOobject::NO_READ,
         IOobject::AUTO_WRITE
    ),
    this->mesh_,
    dimensionedScalar("cem", dimless, 0),
    extrapolatedCalculatedFvPatchScalarField::typeName
    )
{
    // Create the fields for the chemistry sources
    forAll(RR_, fieldi)
    {
        RR_.set
        (fieldi,
         new volScalarField::Internal
         {
            IOobject
        })
    }
}
A.2. cemaPyjacChemistryModel library

Appendix A. Developed codes

( "RR." + Y_[fieldi].name(),
  this->mesh().time().timeName(),
  this->mesh(),
  IOobject::NO_READ,
  IOobject::NO_WRITE
),
this->mesh(),
dimensionedScalar(dimMass/dimVolume/dimTime, Zero)
);});

Info<< "cemaPyjacChemistryModel: Number of species = " << nSpecie_
<< "\n and reactions (from reaction file, expected 0 with PyJac) = " << nReaction_ << endl;
// Note that nReaction_ should be updated with PyJAC
// PERHAPS TO OVERWRITE IN THE SRC DURING DYNAMIC BINDING
Info<< "cemaPyjacChemistryModel: Number of elements = " << nElements_ << endl;

if (this->chemistry_) {
  Info << "\n Evaluating species enthalpy of formation using PyJac\n" << endl;
  // Enthalpy of formation for all species
  std::vector<scalar> sp_enth_form(nSpecie_, 0.0);
  // Enthalpy of formation is taken from pyJac at T-standard (chem_utils.h)
  eval_h(298.15, sp_enth_form.data());
  for (label i = 0; i < nSpecie_; ++i)
    { sp_enthalpy_[i] = sp_enth_form[i]; }
  chemJacobian_ = Zero;
}

} // * * * * * * * * * * * * * * * * Destructor * * * * * * * * * * * * * * * //

template<class ReactionThermo, class ThermoType>
Foam::cemaPyjacChemistryModel<ReactionThermo, ThermoType>::
cemaPyjacChemistryModel()
{
}

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

template<class ReactionThermo, class ThermoType>
void Foam::cemaPyjacChemistryModel<ReactionThermo, ThermoType>::omega
( const scalarField& c, const scalar T, const scalar p, scalarField& dcdt
) const
{
  scalar pf, cf, pr, cr;
  label lRef, rRef;
  dcdt = Zero;
  forAll(reactions_, i)
    { const ReactionThermo& R = reactions_[i];
      scalar omegai = omega
      ( R, c, T, p, pf, cf, lRef, pr, cr, rRef
      );
      forAll(R.lhs(), s)


```cpp
{  
    const label si = R.lhs()[s].index;
    const scalar sl = R.lhs()[s].stoichCoeff;
    dcdt[si] -= sl*omegai;
}

forAll(R.rhs(), s)
{
    const label si = R.rhs()[s].index;
    const scalar sr = R.rhs()[s].stoichCoeff;
    dcdt[si] += sr*omegai;
}
}

template<class ReactionThermo, class ThermoType>
Foam::scalar Foam::cemaPyjacChemistryModel<ReactionThermo, ThermoType>::omegaI
(
    const label index,
    const scalarField& c,
    const scalar T,
    const scalar p,
    scalar& pf,
    scalar& cf,
    label& lRef,
    scalar& pr,
    scalar& cr,
    label& rRef
) const
{
    const Reaction<ThermoType>& R = reactions_[index];
    scalar w = omega(R, c, T, p, pf, cf, lRef, pr, cr, rRef);
    return(w);
}

template<class ReactionThermo, class ThermoType>
Foam::scalar Foam::cemaPyjacChemistryModel<ReactionThermo, ThermoType>::omega
(
    const Reaction<ThermoType>& R,
    const scalarField& c,
    const scalar T,
    const scalar p,
    scalar& pf,
    scalar& cf,
    label& lRef,
    scalar& pr,
    scalar& cr,
    label& rRef
) const
{
    const scalar kf = R.kf(p, T, c);
    const scalar kr = R.kr(kf, p, T, c);

    pf = 1.0;
    pr = 1.0;

    const label Nl = R.lhs().size();
    const label Nr = R.rhs().size();

    label slRef = 0;
    lRef = R.lhs()[slRef].index;

    pf = kf;
    for (label s = 1; s < Nl; s++)
    {
        const label si = R.lhs()[s].index;
```
if (c[si] < c[lRef])
{
    const scalar exp = R.lhs()[slRef].exponent;
    pf *= pow(max(c[lRef], 0.0), exp);
    lRef = si;
    slRef = s;
}
else
{
    const scalar exp = R.lhs()[s].exponent;
    pf *= pow(max(c[si], 0.0), exp);
}
}
cf = max(c[lRef], 0.0);

{  
    const scalar exp = R.lhs()[slRef].exponent;
    if (exp < 1.0)
    {
        if (cf > SMALL)
        {
            pf *= pow(cf, exp - 1.0);
        }
        else
        {
            pf = 0.0;
        }
    }
    else
    {
        pf *= pow(cf, exp - 1.0);
    }
}

label srRef = 0;
rRef = R.rhs()[srRef].index;

// Find the matrix element and element position for the rhs
pr = kr;
for (label s = 1; s < Nr; s++)
{
    const label si = R.rhs()[s].index;
    if (c[si] < c[rRef])
    {
        const scalar exp = R.rhs()[srRef].exponent;
        pr *= pow(max(c[rRef], 0.0), exp);
        rRef = si;
        srRef = s;
    }
    else
    {
        const scalar exp = R.rhs()[s].exponent;
        pr *= pow(max(c[si], 0.0), exp);
    }
}
cr = max(c[rRef], 0.0);

{  
    const scalar exp = R.rhs()[srRef].exponent;
    if (exp < 1.0)
    {
        if (cr > SMALL)
        {
            pr *= pow(cr, exp - 1.0);
        }
        else
        {
            pr = 0.0;
        }
    }
}
A.2. cemaPyjacChemistryModel library

```cpp
pr = 0.0;
}
else
{
    pr *= pow(cr, exp - 1.0);
}
}
return pf*cf - pr*cr;
}

template<class ReactionThermo, class ThermoType>
void Foam::cemaPyjacChemistryModel<ReactionThermo, ThermoType>::derivatives
(
    const scalar time,
    const scalarField& c,
    scalarField& dcdt
) const
{
    std::vector<double> TY(nSpecie_+1, 0.0);
    // if TY has N+1 elements, diff(TY) has N elements
    std::vector<double> dTYdt(nSpecie_, 0.0);

    const scalar p = c[0];
    const scalar T = c[1];

    scalar csum = 0.0;
    forAll(c_, i)
    {
        c_[i] = max(c[i+2], 0.0);
        csum += c_[i];
    }
    // Then we exclude last species from csum and dump all residuals
    // into last species to ensure mass conservation
    csum -= c_[nSpecie_-1];
    c_[nSpecie_-1] = 1.0 - csum;

    TY[0] = T;
    forAll(c_, i)
    {
        TY[i+1] = c_[i];
    }

dydt(0, p, TY.data(), dTYdt.data());
// dp/dt = 0
dcdt[0] = 0.0;

// Back substitute into dcdt (dcdt has nSpecie+1 elements for diff(PTY))
for (label i = 0; i < nSpecie_; ++i)
{
    dcdt[i+1] = dTYdt[i];
}
}

template<class ReactionThermo, class ThermoType>
void Foam::cemaPyjacChemistryModel<ReactionThermo, ThermoType>::jacobian
(
    const scalar t,
    const scalarField& c,
    scalarField& dcdt,
    scalarSquareMatrix& dfdc
) const
{
    std::vector<double> TY(nSpecie_+1, 0.0); //###
```
```cpp
std::vector<double> dfdy(nSpecie_*nSpecie_, 0.0); //###

const scalar p = c[0];
const scalar T = c[1];

scalar csum = 0.0;
forAll(c_, i)
{
    c_[i] = max(c[i+2], 0.0);
    csum += c_[i];
}
// Then we exclude last species from csum and instead dump all
// residuals into last species to ensure mass conservation
// csum = c_[nSpecie_-1];
// c_[nSpecie_-1] = 1.0 - csum;
dfdc = Zero;

TY[0] = T;
// Assign nSpecies-1 species mass fractions to the TY vector
forAll(c_, i)
{
    TY[i+1] = c_[i];
}
eval_jacob(0, p, TY.data(), dfdy.data());

// Back substitution to update dfdc
// Assign first row and column to zero as they correspond to const pressure
for (label j = 0; j < nSpecie_ + 1; ++j)
{
    dfdc(0,j) = 0.0;
    dfdc(j,0) = 0.0;
}

label k = 0;
// Loop cols
for (label j = 1; j < nSpecie_+1; ++j)
{
    // Loop rows
    for (label i = 1; i < nSpecie_+1; ++i)
    {
        dfdc(i,j) = dfdy[k + i - 1];
        chemJacobian_(i-1,j-1) = dfdy[k + i - 1];
    }
    k += nSpecie_;
}

// Note that dcdt is not needed in most ODE solvers so here we just return 0
dcdt = Zero;
```

```cpp
template<class ReactionThermo, class ThermoType>
Foam::tmp<Foam::volScalarField>
Foam::cemaPyjacChemistryModel<ReactionThermo, ThermoType>::tc() const
{
    tmp<volScalarField> ttc
    (
        new volScalarField
        (
            IOobject
            (
                "tc",
                this->time().timeName(),
                this->mesh(),
                IOobject::NO_READ
            ),
            0
        )
    );
```
scalarField& tc = ttc.ref();

tmp<volScalarField> trho(this->thermo().rho());
const scalarField& rho = trho();
const scalarField& T = this->thermo().T();
const scalarField& p = this->thermo().p();
const label nReaction = reactions_.size();

scalar pf, cf, pr, cr;
label lRef, rRef;

if (this->chemistry_)
{
    forAll(rho, celli)
    {
        const scalar rhoi = rho[celli];
        const scalar Ti = T[celli];
        const scalar pi = p[celli];
        scalar cSum = 0.0;

        for (label i=0; i<nSpecie_; i++)
        {
            c_[i] = rhoi*Y_[i][celli]/specieThermo_[i].W();
            cSum += c_[i];
        }

        forAll(reactions_, i)
        {
            const Reaction<ThermoType>& R = reactions_[i];
            omega(R, c_, Ti, pi, pf, cf, lRef, pr, cr, rRef);

            forAll(R.rhs(), s)
            {
                tc[celli] += R.rhs()[s].stoichCoeff*pf*cf;
            }

            tc[celli] = nReaction*cSum/tc[celli];
        }
    }
}

ttc.ref().correctBoundaryConditions();

return ttc;


}
A.2. cemaPyjacChemistryModel library

Appendix A. Developed codes

```cpp
IOobject
(
    "Qdot",
    this->mesh_.time().timeName(),
    this->mesh_,
    IOobject::NO_READ,
    IOobject::NO_WRITE,
    false
),
this->mesh_,
dimensionedScalar("zero", dimEnergy / dimVolume / dimTime, 0)
);

if (this->chemistry_)
{
    scalarField& Qdot = tQdot.ref();
    forAll(Y_, i)
    {
        forAll(Qdot, celli)
        {
            // const scalar hi = specieThermo_[i].Hc();
            scalar hi = sp_enthalpy_[i];
            Qdot[celli] -= hi*RR_[i][celli];
        }
    }
    return tQdot;
}

template<class ReactionThermo, class ThermoType>
Foam::tmp<Foam::DimensionedField<Foam::scalar, Foam::volMesh>>
Foam::cemaPyjacChemistryModel<ReactionThermo, ThermoType>::calculateRR
(
    const label ri,
    const label si
) const
{
    scalar pf, cf, pr, cr;
    label lRef, rRef;
    tmp<volScalarField::Internal> tRR
    (
        new volScalarField::Internal
        (
            IOobject
            (
                "RR",
                this->mesh().time().timeName(),
                this->mesh(),
                IOobject::NO_READ,
                IOobject::NO_WRITE
            ),
            this->mesh(),
            dimensionedScalar(dimMass/dimVolume/dimTime, Zero)
        )
    );
    volScalarField::Internal& RR = tRR.ref();
    tmp<volScalarField> trho(this->thermo().rho());
    const scalarField& rho = trho();
    const scalarField& T = this->thermo().T();
    const scalarField& p = this->thermo().p();
```
A.2. cemaPyjacChemistryModel library

Appendix A. Developed codes

```c++
forAll(rho, celli)
{
    const scalar rhoi = rho[celli];
    const scalar Ti = T[celli];
    const scalar pi = p[celli];

    for (label i=0; i<nSpecie_; i++)
    {
        const scalar Yi = Y_[i][celli];
        c_[i] = rhoi*Yi/specieThermo_[i].W();
    }

    const scalar w = omegaI
    (
        ri,
        c_,
        Ti,
        pi,
        pf,
        cf,
        lRef,
        pr,
        cr,
        rRef
    );

    RR[celli] = w*specieThermo_[si].W();
}
return tRR;
}

template<class ReactionThermo, class ThermoType>
void Foam::cemaPyjacChemistryModel<ReactionThermo, ThermoType>::calculate()
{
    if (!this->chemistry_)
    {
        return;
    }

tmp<volScalarField> trho(this->thermo().rho());
const scalarField& rho = trho();
const scalarField& T = this->thermo().T();
const scalarField& p = this->thermo().p();

forAll(rho, celli)
{
    const scalar rhoi = rho[celli];
    const scalar Ti = T[celli];
    const scalar pi = p[celli];

    for (label i=0; i<nSpecie_; i++)
    {
        const scalar Yi = Y_[i][celli];
        c_[i] = rhoi*Yi/specieThermo_[i].W();
    }

    omega(c_, Ti, pi, dcdt_);

    for (label i=0; i<nSpecie_; i++)
    {
        RR_[i][celli] = dcdt_[i]*specieThermo_[i].W();
    }
}
}
```
template<class ReactionThermo, class ThermoType>
template<class DeltaTType>
Foam::scalar Foam::cemaPyjacChemistryModel<ReactionThermo, ThermoType>::solve
(const DeltaTType& deltaT
)
{
  BasicChemistryModel<ReactionThermo>::correct();

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

tmp<volScalarField> trho(this->thermo().rho());
const scalarField& rho = trho();
const scalarField& T = this->thermo().T();
const scalarField& p = this->thermo().p();
scalarField c0(nSpecie_);

forall(rho, celli)
{
  scalar Ti = T[celli];
  if (Ti > Treact_)
  {
    const scalar rhoi = rho[celli];
    scalar pi = p[celli];
    
    for (label i=0; i<nSpecie_; i++)
    {
      // c_[i] = rhoi*Y_[i][celli]/specieThermo_[i].W();
      c_[i] = Y_[i][celli];
      c0[i] = c_[i];
    }

    // Initialise time progress
    scalar timeLeft = deltaT[celli];

    // Calculate the chemical source terms
    while (timeLeft > SMALL)
    {
      scalar dt = timeLeft;
      // Calls ode::solve() from chemistrySolver
      this->solve(c_, Ti, pi, dt, this->deltaTChem_[celli]);
      timeLeft -= dt;
    }
  
  deltaTMin = min(this->deltaTChem_[celli], deltaTMin);
  this->deltaTChem_[celli] = min(this->deltaTChem_[celli], this->deltaTChemMax_);

  for (label i=0; i<nSpecie_; i++)
  {
    // CHEMICAL SOURCE TERM PER SPECIES
    // (c_[i] - c0[i])*specieThermo_[i].W()/deltaT[celli]; // ###
    this->RR_[i][celli] = rhoi*(this->c_[i] - c0[i])/deltaT[celli];
  }

  if (this->time().write()) {
    // Info << "\nCELL: " << celli << "\t Temperature = " << Ti << endl;
  }
}
A.2. cemaPyjacChemistryModel library

```
scalar cem_cell;
cema(cem_cell);
cem_[celli] = cem_cell;
}
else
{
  for (label i=0; i<nSpecie_; i++)
  {
    RR_[i][celli] = 0;
  }
}

cem_.correctBoundaryConditions();
return deltaTMin;
}

template<class ReactionThermo, class ThermoType>
Foam::scalar Foam::cemaPyjacChemistryModel<ReactionThermo, ThermoType>::solve
(const scalar deltaT)
{
  // Don't allow the time-step to change more than a factor of 2
  return min
  (this->solve<UniformField<scalar>>(UniformField<scalar>(deltaT)),
   2*deltaT);
}

template<class ReactionThermo, class ThermoType>
Foam::scalar Foam::cemaPyjacChemistryModel<ReactionThermo, ThermoType>::solve
(const scalarField& deltaT)
{
  return this->solve<scalarField>(deltaT);
}

template<class ReactionThermo, class ThermoType>
void Foam::cemaPyjacChemistryModel<ReactionThermo, ThermoType>::cema
(scalar& cem) const
{
  const Foam::EigenMatrix<scalar> EM(chemJacobian_);
  DiagonalMatrix<scalar> EValsRe(EM.EValsRe());
  DiagonalMatrix<scalar> EValsIm(EM.EValsIm());
  DiagonalMatrix<scalar> EValsMag(EValsRe.size(), 0.0);
  forAll(EValsRe, i)
  {
    EValsMag[i] = (EValsRe[i]*EValsRe[i] + EValsIm[i]*EValsIm[i]);
  }
  // Sort eigenvalues in ascending order, and track indices
  const auto ascend = [&](scalar a, scalar b){ return a < b;};
  const List<label> permut(EValsMag.sortPermutation(ascend));
  // Skip conservation modes for elements and temperature
  for (label i=0; i<nElements_+1; ++i)
```
A.2. cemaPyjacChemistryModel library

Appendix A. Developed codes

```cpp
{ 
    label idx = permut[i];
    EValsRe[idx] = -1E30;
}

cem = gMax(EValsRe);
}

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

A.2.2 chemistrySolver

odePyjac.H

苑 /*-----------------------------------------------------------*/
苑 |                              |
苑 | Field | OpenFOAM: The Open Source CFD Toolbox |
苑 | Oeration | |
苑 | And | www.openfoam.com |
苑 | Manipulation | |
苑 /*** * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * */
苑 Copyright (C) 2011-2017 OpenFOAM Foundation
苑 /*** * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * */
苑 License
苑 This file is part of OpenFOAM.
苑
苑 OpenFOAM is free software: you can redistribute it and/or modify it
苑 under the terms of the GNU General Public License as published by
苑 the Free Software Foundation, either version 3 of the License, or
苑 (at your option) any later version.
苑
苑 OpenFOAM is distributed in the hope that it will be useful, but WITHOUT
苑 ANY WARRANTY; without even the implied warranty of MERCHANTABILITY or
苑 FITNESS FOR A PARTICULAR PURPOSE. See the GNU General Public License
苑 for more details.
苑
苑 You should have received a copy of the GNU General Public License
苑 along with OpenFOAM. If not, see <http://www.gnu.org/licenses/>.
苑
苑 Class
苑 Foam::odePyjac
苑
苑 Description
苑 An ODE solver for chemistry
苑
苑 SourceFiles
苑 odePyjac.C
苑
苑 /*-----------------------------------------------------------*/
苑 #ifndef odePyjac_H
苑 #define odePyjac_H
苑
苑 #include "chemistrySolver.H"
苑 #include "ODESolver.H"
苑
苑 // ************************************************************************* //
苑 namespace Foam
苑 {
苑
苑 /*-----------------------------------------------------------*/
苑 Class odePyjac Declaration
苑 /*-----------------------------------------------------------*/
苑
template<class ChemistryModel>
```
class odePyjac
:
public chemistrySolver<ChemistryModel>
{
    // Private data
    dictionary coeffsDict_;  
    mutable autoPtr<ODESolver> odeSolver_;  
    // Solver data
    mutable scalarField cTp_;  

    public:
        // Runtime type information
        TypeName("odePyjac");  

        // Constructors
        //- Construct from thermo
        odePyjac(typename ChemistryModel::reactionThermo& thermo);  

        //- Destructor
        virtual ~odePyjac();  

        // Member Functions
        //- Update the concentrations and return the chemical time
        virtual void solve
        {
            scalarField& c,  
            scalar& T,  
            scalar& p,  
            scalar& deltaT,  
            scalar& subDeltaT
            ) const;
        };

    // ************************************************************************* //
} // End namespace Foam

// ************************************************************************* //
#endif  
// ************************************************************************* //

#ifndef NoRepository
    #include "odePyjac.C"
#endif  

// ************************************************************************* //
# Copyright (C) 2011-2017 OpenFOAM Foundation

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/>.

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

template<class ChemistryModel>
Foam::odePyjac<ChemistryModel>::odePyjac(typename ChemistryModel::reactionThermo& thermo):
    chemistrySolver<ChemistryModel>(thermo),
    coeffsDict_(this->subDict("odeCoeffs")),
    odeSolver_(ODESolver::New(*this, coeffsDict_),
    cTp_(*this->nEqns))
{}

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

template<class ChemistryModel>
Foam::odePyjac<ChemistryModel>::~odePyjac()
{}

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

template<class ChemistryModel>
void Foam::odePyjac<ChemistryModel>::solve
(
    scalarField& c,
    scalar& T,
    scalar& p,
    scalar& deltaT,
    scalar& subDeltaT
) const
{
    // Reset the size of the ODE system to the simplified size when mechanism
    // reduction is active
    if (odeSolver_->resize())
    {
        odeSolver_->resizeField(cTp_);
    }

    const label nSpecie = this->nSpecie();

    // Copy the concentration, T and P to the total solve-vector (N+1)
    cTp_[0] = p;
    cTp_[1] = T;
    // Update for N-1 species
    for (label i=0; i<nSpecie-1; i++)
    {  

{
  cTp_[i+2] = c[i];
}

// Here, we call ODE solver...
odeSolver_->solve(0, deltaT, cTp_, subDeltaT);

// Back substitute results
p = cTp_[0];
T = cTp_[1];
scalar csum = 0;

for (label i=0; i<nSpecie-1; i++)
{
  c[i] = max(0.0, cTp_[i+2]);
  csum += c[i];
}

// Last species
c[nSpecie-1] = 1.0 - csum;
}

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

A.2.3 Make chemistryModel

Listing A.1: Make (files) for the cemaPyjacChemistryModel library compilation

makeChemistryModels.C
makeChemistrySolvers.C

LIB = $(FOAM_USER_LIBBIN)/libcemaPyjacChemistryModel

Listing A.2: Make (options) for the cemaPyjacChemistryModel library compilation

EXE_INC = \n  -I$(LIB_SRC)/finiteVolume/lnInclude \n  -I$(LIB_SRC)/meshTools/lnInclude \n  -I$(LIB_SRC)/ODE/lnInclude \n  -I$(LIB_SRC)/transportModels/compressible/lnInclude \n  -I$(LIB_SRC)/thermophysicalModels/reactionThermo/lnInclude \n  -I$(LIB_SRC)/thermophysicalModels/basic/lnInclude \n  -I$(LIB_SRC)/thermophysicalModels/specie/lnInclude \n  -I$(LIB_SRC)/thermophysicalModels/functions/Polynomial \n  -I$(LIB_SRC)/thermophysicalModels/thermophysicalFunctions/lnInclude \n  -I$(LIB_SRC)/thermophysicalModels/chemistryModel/lnInclude \n  -IpyjacInclude

LIB_LIBS = \n  -lfiniteVolume \n  -lmeshTools \n  -lODE \n  -lcompressibleTransportModels \n  -lfluidThermophysicalModels \n  -lreactionThermophysicalModels \n  -lspecie \n  -lchemistryModel

82
Listing A.3: Macros for chemistry models based on compressibility and transport types

```plaintext
#ifndef makeChemistryModel_H
#define makeChemistryModel_H

#include "addToRunTimeSelectionTable.H"

namespace Foam {

    typedef BasicChemistryModel<Comp> BasicChemistryModel##Comp;

defineTemplateTypeNameAndDebugWithName
    (B}

    defineTemplateRunTimeSelectionTable
    (B)

define makeChemistryModelType(SS, Comp, Thermo)

    typedef SS<Comp, Thermo> SS##Comp##Thermo;

}
```

Listing A.4: Macros for chemistry solvers based on compressibility and transport types

```c++
#ifndef makeChemistrySolverTypes_H
#define makeChemistrySolverTypes_H

#include "chemistrySolver.H"
#include "cemaPyjacChemistryModel.H"
#include "noChemistrySolver.H"
#include "EulerImplicit.H"

#define makeChemistrySolverType(SS, Comp, Thermo) \
    typedef SS<cemaPyjacChemistryModel<Comp, Thermo>> SS##Comp##Thermo; \
    defineTemplateTypeNameAndDebugWithName \
    ( \
        SS##Comp##Thermo, \
        (#SS"<" + word(cemaPyjacChemistryModel<Comp, Thermo>::typeName_()) + ">").c_str(), \
        0 \
    );

// *******************************************************
#endif

// ************************************************************************* //
```
Listing A.5: Templates to instantiate chemistry models based on compressibility and transport types

```cpp
#define makeChemistrySolverTypes(Comp, Thermo)
{
    makeChemistrySolverType
    (   noChemistrySolver,
        Comp,
        Thermo
    );

    makeChemistrySolverType
    (   EulerImplicit,
        Comp,
        Thermo
    );

    makeChemistrySolverType
    (   odePyjac,
        Comp,
        Thermo
    );

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

InClass
Foam::psiChemistryModel

Description
Creates chemistry model instances templated on the type of thermodynamics

\*---------------------------------------------------------------------------*/
#include "makeChemistryModel.H"
#include "psiReactionThermo.H"
#include "rhoReactionThermo.H"
#include "cemaPyjacChemistryModel.H"
#include "thermoPhysicsTypes.H"

// * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * //
namespace Foam
{
    // Make base types
    makeChemistryModel(psiReactionThermo);
    makeChemistryModel(rhoReactionThermo);

    // Chemistry models based on sensibleEnthalpy
    makeChemistryModelType
    (cemaPyjacChemistryModel,
     psiReactionThermo,
     constGasHThermoPhysics)
    ;

    makeChemistryModelType
    (cemaPyjacChemistryModel,
     psiReactionThermo,
     gasHThermoPhysics)
    ;

    makeChemistryModelType
    (cemaPyjacChemistryModel,
     psiReactionThermo,
     constIncompressibleGasHThermoPhysics)
    ;

    makeChemistryModelType
    (cemaPyjacChemistryModel,
     psiReactionThermo,
     incompressibleGasHThermoPhysics)
    ;

    makeChemistryModelType
    (cemaPyjacChemistryModel,
     psiReactionThermo,
     icoPoly8HThermoPhysics)
    ;

    makeChemistryModelType
    (cemaPyjacChemistryModel,
     psiReactionThermo,
     constFluidHThermoPhysics)
    ;

    makeChemistryModelType
    (cemaPyjacChemistryModel,
     psiReactionThermo,
     icoPoly8HThermoPhysics)
    ;
A.2. cemaPyjacChemistryModel library

Developed codes (cemaPyjacChemistryModel, psiReactionThermo, constAdiabaticFluidHThermoPhysics);
makeChemistryModelType
(cemaPyjacChemistryModel, psiReactionThermo, constHThermoPhysics);
makeChemistryModelType
(cemaPyjacChemistryModel, rhoReactionThermo, constGasHThermoPhysics);
makeChemistryModelType
(cemaPyjacChemistryModel, rhoReactionThermo, gasHThermoPhysics);
makeChemistryModelType
(cemaPyjacChemistryModel, rhoReactionThermo, constIncompressibleGasHThermoPhysics);
makeChemistryModelType
(cemaPyjacChemistryModel, rhoReactionThermo, incompressibleGasHThermoPhysics);
makeChemistryModelType
(cemaPyjacChemistryModel, rhoReactionThermo, icoPoly8HThermoPhysics);
makeChemistryModelType
(cemaPyjacChemistryModel, rhoReactionThermo, constFluidHThermoPhysics);
makeChemistryModelType
(cemaPyjacChemistryModel, rhoReactionThermo, constAdiabaticFluidHThermoPhysics);
makeChemistryModelType
(cemaPyjacChemistryModel, rhoReactionThermo, constHThermoPhysics)
makeChemistryModelType
    (cemaPyjacChemistryModel, psiReactionThermo, constGasEThermoPhysics);
makeChemistryModelType
    (cemaPyjacChemistryModel, psiReactionThermo, gasEThermoPhysics);
makeChemistryModelType
    (cemaPyjacChemistryModel, psiReactionThermo, constIncompressibleGasEThermoPhysics);
makeChemistryModelType
    (cemaPyjacChemistryModel, psiReactionThermo, incompressibleGasEThermoPhysics);
makeChemistryModelType
    (cemaPyjacChemistryModel, psiReactionThermo, icoPoly8EThermoPhysics);
makeChemistryModelType
    (cemaPyjacChemistryModel, psiReactionThermo, constFluidEThermoPhysics);
makeChemistryModelType
    (cemaPyjacChemistryModel, psiReactionThermo, constAdiabaticFluidEThermoPhysics);
makeChemistryModelType
    (cemaPyjacChemistryModel, psiReactionThermo, constEThermoPhysics);
makeChemistryModelType
    (cemaPyjacChemistryModel, rhoReactionThermo, constGasEThermoPhysics);
makeChemistryModelType
(
    cemaPyjacChemistryModel,
    rhoReactionThermo,
    gasEThermoPhysics
);

makeChemistryModelType
(
    cemaPyjacChemistryModel,
    rhoReactionThermo,
    constIncompressibleGasEThermoPhysics
);

makeChemistryModelType
(
    cemaPyjacChemistryModel,
    rhoReactionThermo,
    incompressibleGasEThermoPhysics
);

makeChemistryModelType
(
    cemaPyjacChemistryModel,
    rhoReactionThermo,
    icoPoly8EThermoPhysics
);

makeChemistryModelType
(
    cemaPyjacChemistryModel,
    rhoReactionThermo,
    constFluidEThermoPhysics
);

makeChemistryModelType
(
    cemaPyjacChemistryModel,
    rhoReactionThermo,
    constAdiabaticFluidEThermoPhysics
);

makeChemistryModelType
(
    cemaPyjacChemistryModel,
    rhoReactionThermo,
    constEThermoPhysics
);

Listing A.6: Templates to instantiate chemistry solvers based on compressibility and transport types

// ************************************************************************* //
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 "makeChemistrySolverTypes.H"
#include "thermoPhysicsTypes.H"
#include "psiReactionThermo.H"
#include "rhoReactionThermo.H"

namespace Foam {

    // Chemistry solvers based on sensibleEnthalpy
    makeChemistrySolverTypes(psiReactionThermo, constGasHThermoPhysics);
    makeChemistrySolverTypes(psiReactionThermo, gasHThermoPhysics);
    makeChemistrySolverTypes(psiReactionThermo, constIncompressibleGasHThermoPhysics);
    makeChemistrySolverTypes(psiReactionThermo, incompressibleGasHThermoPhysics);
    makeChemistrySolverTypes(psiReactionThermo, icoPoly8HThermoPhysics);
    makeChemistrySolverTypes(psiReactionThermo, constFluidHThermoPhysics);
    makeChemistrySolverTypes(psiReactionThermo, constAdiabaticFluidHThermoPhysics);

    makeChemistrySolverTypes(rhoReactionThermo, constGasHThermoPhysics);
    makeChemistrySolverTypes(rhoReactionThermo, gasHThermoPhysics);
    makeChemistrySolverTypes(rhoReactionThermo, constIncompressibleGasHThermoPhysics);
    makeChemistrySolverTypes(rhoReactionThermo, incompressibleGasHThermoPhysics);
    makeChemistrySolverTypes(rhoReactionThermo, icoPoly8HThermoPhysics);
    makeChemistrySolverTypes(rhoReactionThermo, constFluidHThermoPhysics);
    makeChemistrySolverTypes(rhoReactionThermo, constAdiabaticFluidHThermoPhysics);

};
A.3 pyJac CMake

We assume that pyJac was already used to generate a mechanism source code, and subsequently that out directory is created.

Listing A.7: Shell script to build mechanism source code into the shared object libc_pyjac.so

```
rm -rf build
mkdir build
cd build && cmake .. -DCMAKE_C_COMPILER=cc
make
```
Listing A.8: **CMakeLists.txt** containing set of directives and instructions to compile using **CMake**

```cmake
# Listing A.8: CMakeLists.txt

cmake_minimum_required(VERSION 2.6)
project(pyJac)
set(CMAKE_BUILD_TYPE Release)
enable_language(C)

# GNU on DESKTOP
set(CMAKE_C_FLAGS "-std=c99 -O3 -mtune=native -fPIC")

include_directories(out)
include_directories(out/jacobs)

file(GLOB_RECURSE SOURCES "out/*.c")
add_library(c_pyjac SHARED ${SOURCES})
install(TARGETS c_pyjac DESTINATION .)
```

### A.4 pyJac generic header files

These header files are required for the compilation of the developed library, while definitions if the functions declarations are compiled for specific chemical mechanism and the generated binary is linked to solver during runtime.

Listing A.9: **header.h** initial header file

```c
// Listing A.9: header.h

#ifndef HEAD
#define HEAD
#include <stdlib.h>
#include <math.h>
/** Constant pressure or volume. */
#define CONP
#ifndef CONV
#define CONV

/** Include mechanism header to get NSP and NN **/
#include "mechanism.h"

#ifndef _OPENMP
#define _OPENMP
#else
#define omp_get_max_threads() 1
#define omp_get_num_threads() 1
#endif
#endif
```

Listing A.10: **mechanism.h** for GRI–3.0. Note that file needs to be updated for new mechanism in case enclosed directives are to be used. Otherwise all good

```c
// Listing A.10: mechanism.h

#ifndef MECHANISM_h
#define MECHANISM_h
#include <string.h>
/* Species Indexes */
0 H2 1 H 2 O 3 O2 4 OH 5 H2O 6 HO2 7 H2O2 8 C
```

92
/ *//Number of species
#define NSP 53
//Number of variables. NN = NSP + 1 (temperature)
#define NN 54
//Number of forward reactions
#define FWD_RATES 325
//Number of reversible reactions
#define REV_RATES 309
//Number of reactions with pressure modified rates
#define PRES_MOD_RATES 41

//Must be implemented by user on a per mechanism basis in mechanism.c
void set_same_initial_conditions(int, double**, double**);

#if defined (RATES_TEST) || defined (PROFILER)
    void write_jacobian_and_rates_output(int NUM);
#else
#endif

//apply masking of ICs for cache optimized mechanisms
void apply_mask(double*);
void apply_reverse_mask(double*);
A.4. pyJac generic header files

Listing A.11: jacob.h to evaluate Jacobian

```c
#ifndef JACOB_HEAD
#define JACOB_HEAD

#include "header.h"
// #include "jacobs/jac_include.h"
#include "chem_utils.h"
#include "rates.h"
void eval_jacob (const double, const double, const double * __restrict__, double * __restrict__);
#endif
```

Listing A.12: dydt.h to evaluate derivatives

```c
#ifndef DYDT_HEAD
#define DYDT_HEAD

#include "header.h"
void dydt (const double, const double, const double * __restrict__, double * __restrict__);
#endif
```

Listing A.13: rates.h to evaluate reaction rates

```c
#ifndef RATES_HEAD
#define RATES_HEAD

#include "header.h"

void eval_rxn_rates (const double, const double, const double * __restrict__, double * __restrict__,
                     double * __restrict__);
void eval_spec_rates (const double * __restrict__, const double * __restrict__,
                     const double * __restrict__, double * __restrict__,
                     double * __restrict__);
void get_rxn_pres_mod (const double, const double, const double * __restrict__, double * __restrict__);
#endif
```

Listing A.14: chem_utils.h to evaluate thermodynamic quantities

```c
#ifndef CHEM_UTILS_HEAD
#define CHEM_UTILS_HEAD

#include "header.h"

void eval_conc (const double, const double, const double * __restrict__, double * __restrict__,
                double * __restrict__, double * __restrict__, double * __restrict__);
void eval_conc_rho (const double, const double, const double * __restrict__, double * __restrict__,
                    double * __restrict__, double * __restrict__);
void eval_h (const double, double * __restrict__);
void eval_u (const double, double * __restrict__);
void eval_cv (const double, double * __restrict__);
void eval_cp (const double, double * __restrict__);
#endif
```

Listing A.15: mass_mole.h to convert between molar and mass basis

```c
#ifndef MASS_MOLE_HEAD
#define MASS_MOLE_HEAD

#include "header.h"

void mole2mass (const double*, double*);
```
A.5 Utilities

We note that the interpolator module is taken from Bladex Python package [25], available at https://github.com/mathLab/Bladex.

### computeInitialFields.py

```python
import os
import numpy as np
import cantera as ct

if __name__ == '__main__':
    print(ct.__version__)

basename = os.path.basename(__file__)
# User-defined input - fuel is CH4
pgas = 1*ct.one_atm
Tin = 900.0
phi = 0.5
mixture = 'O2:0.21, N2:0.79'
dir_out = 'out_states'

if not os.path.exists(dir_out):
    os.makedirs(dir_out)

#Import gas phases with mixture transport model
gas = ct.Solution('gri30.xml')
#Set gas state to that of the unburned gas

#Solver settings
initial_grid = 2*np.array([0.0, 0.001, 0.01, 0.02, 0.029, 0.03], 'd')/3
```

---

Listing A.16: sparse_multiplier.h

```c
#ifndef SPARSE_HEAD
#define SPARSE_HEAD

#define N_A 2809
#include "header.h"

void sparse_multiplier (const double *, const double *, double*);

#ifdef COMPILE_TESTING_METHODS
int test_sparse_multiplier();
#endif
#endif
```

---

**void mass2mole (const double*, double**); double getDensity (const double, const double, const double**);**
tol_ss = [1.0e-5, 1.0e-8] # [rtol atol] for steady-state problem
tol_ts = [1.0e-5, 1.0e-8] # [rtol atol] for time stepping
loglevel = 1
refine_grid = True

f = ct.FreeFlame(gas, initial_grid)
f.flame.set_steady_tolerances(default=tol_ss)
f.flame.set_transient_tolerances(default=tol_ts)

f.inlet.X = gas.X
f.inlet.T = Tin

# First solve - energy off
f.energy_enabled = False
f.set_max_jac_age(50, 50)

# Max number of times the Jacobian will be used before re-evaluation
f.set_time_step(5.e-06, [10, 20, 80]) # s
f.set_transport_model = 'UnityLewis' # Mix, Multi, UnityLewis

f.solve(loglevel, refine_grid)

# Second solve - energy on
f.energy_enabled = True
f.set_transport_model = 'Multi'

f.solve(loglevel, refine_grid)

# Third solve:
#f.set_transport_model = 'Multi'
f.solve(loglevel, refine_grid)

#Fourth solve:
#f.set_transport_model = 'Multi'
f.solve(loglevel, refine_grid)

grid = f.flame.grid
n_points = f.flame.n_points
T = np.zeros(n_points)

for n in np.arange(n_points):
    T[n] = f.T[n]

np.savetxt(dir_out + '/T', T)
np.savetxt(dir_out + '/grid', grid)

n_species = gas.n_species
for i in np.arange(n_species):
    species_name = gas.species_name(i)
    if species_name in ['CH4', 'O2', 'N2', 'CO2', 'H2O']:
        print('species: ' + str(species_name))
        Yi = np.zeros(n_points)

        for n in np.arange(n_points):
            f.set_gas_state(n)
            Yi[n] = gas.Y[i]
        np.savetxt(dir_out + '/' + str(species_name), Yi)
interpolateInitalFields.py

```python
""
Usage: Interpolate initial fields according to grid resolution

ndinterpolator module is part of the Bladex package
https://github.com/mathLab/BladeX

M. Gadalla, M. Tezzele, A. Mola, G. Rozza, (2019).
BladeX: Python Blade Morphing.
Journal of Open Source Software, 4(34), 1203,
https://doi.org/10.21105/joss.01203
""
import numpy as np
from ndinterpolator import reconstruct_f

def openfoam_write_internalField_scalar(filename, rbf_points, array):
    outfile = ''
    outfile += str(rbf_points)
    outfile += '\n(\n    for val in array:
        outfile += "{:.7f}".format(val) + '"
        outfile += ')'
    f = open(filename, "w")
    f.write(outfile)
    f.close()

rbf_points = 1350
out_states = 'out_states'

grid = np.loadtxt(out_states + '/grid')

for f in ['T', 'CH4', 'O2', 'N2', 'CO2', 'H2O']:
    field = np.loadtxt(out_states + '/' + f)
    xx = np.linspace(grid[0], grid[-1], num=rbf_points)
    yy = np.zeros(rbf_points)
    reconstruct_f(original_input=grid, original_output=field,
                  rbf_input=xx, rbf_output=yy,
                  basis='beckert_wendland_c2_basis', radius=2.0)
    openfoam_write_internalField_scalar(filename=out_states + '/' + f + '.dat',
                                         rbf_points=rbf_points, array=yy)
```