051176 - Computational Techniques for Thermochemical Propulsion

Master of Science in Aeronautical Engineering

Theoretical and Numerical Combustion

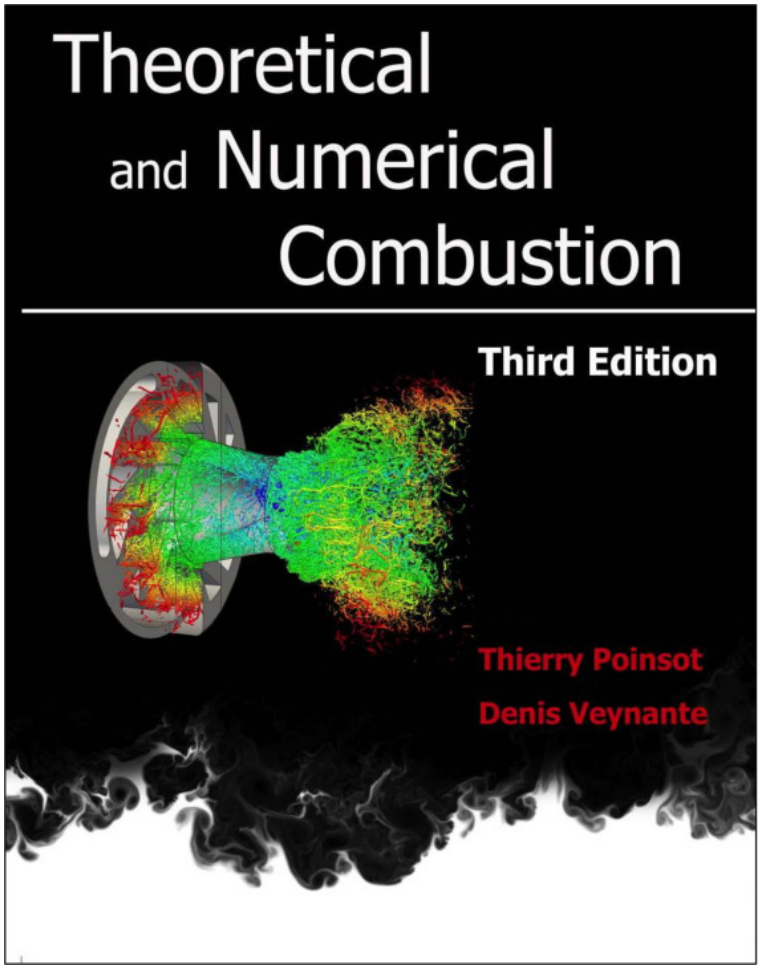
Prof. **Federico Piscaglia**

Dept. of Aerospace Science and Technology (DAER)

POLITECNICO DI MILANO, Italy

federico.piscaglia@polimi.it

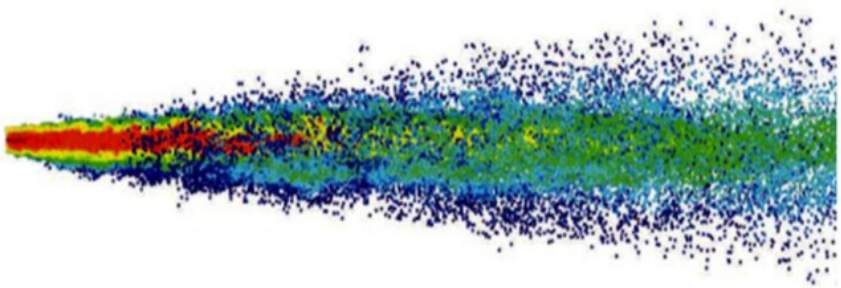
Bibliography

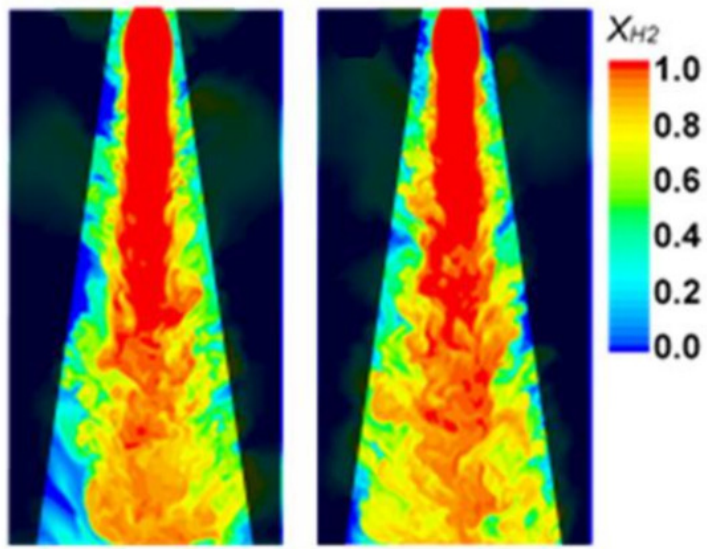


T. Poinsot, D. Veynante. "**Theoretical and Numerical Combustion**" (Third Edition), Cerfacs, 2015.

The Importance of Chemistry

- Nowadays most of the energy generated and commonly used in everyday life is provided by combustion. In a combustion chamber, different chemical species are introduced. The heat release from the development of exothermic chemical reactions in a combustion chamber can be stored or transformed in different forms of energy, to supply to common demand.
- Combustion products typically include pollutants: the main goal of experimental and computational studies is to improve our understanding about the combustion process, to improve quality and efficiency fo the combustion, to increase the amount of energy extracted, to reduce the level of pollutants, and much more.





The Importance of Chemistry

Chemistry can be very expensive.

Example:

- simple H_2 (simplified!) reaction mechanism can be of 10 species and 27 reactions
- a detailed nC_7H_{16} reaction mechanism implies 544 species and 2446 reactions!!!

In a CFD code, when reaction mechanism is triggered:

- the mechanism must be provided in a specific input format (CHEMKIN);
- the complexity of the solver increases, as well as the number of parameters to control.

Lab experiments vs. Computational analysis

Analysis can be performed in chemical laboratories requiring great time and cost, while posing at risk the staff due to particular complexity of the processes and safety issues. This type of analysis is necessary though, as the experimental process must follow up the computational study and work toghether for the best possible comprehensive result.

Computational analysis allows to work in a safe environment, and obtain visual and accurate quantitative results saving cost and time. However, in order to save time, for complex fluid-dynamic and chemical (complete) problems very high computational power is requested.

The Importance of Chemistry

Combustion involves multiple species reacting through multiple chemical reactions. Navier-Stokes equations apply for such a multi-species multi-reaction gas but they require additional terms.

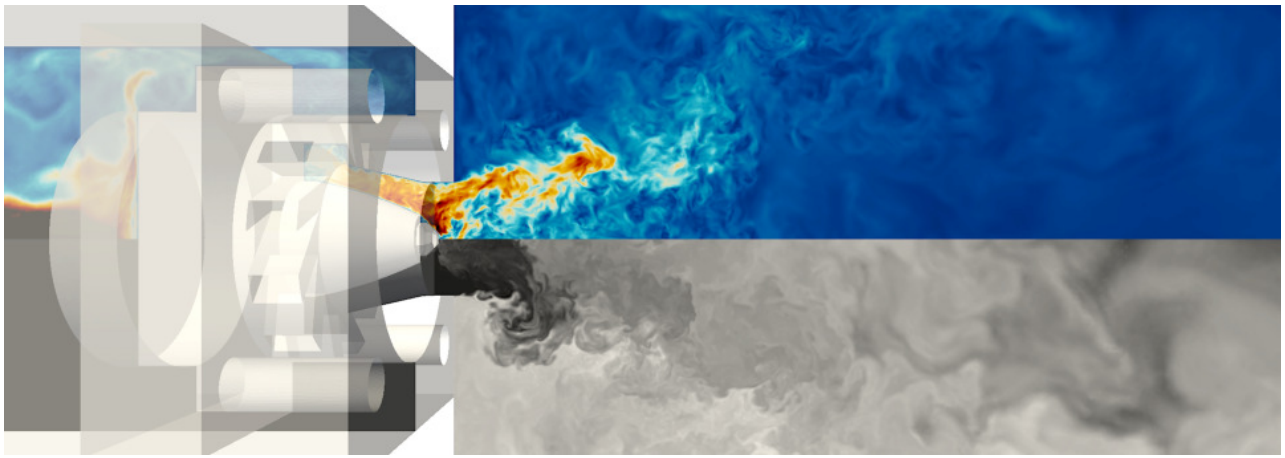


Figure 1: Jet-A (source: http://www.coria-cfd.fr)

Governing equations for reactive flows:

- reacting gas is non-isothermal mixture of multiple species which must be tracked individually;
- heat capacities vary with temperature and composition;
- species react chemically and the reaction rates require specific modeling;
- since the gas is a mixture of gases, transport coefficients (heat diffusivity, species diffusion, viscosity, etc) require specific attention;

Chemical Reaction Chain

"*Activation of chemistry*" means combine chemical reacting substances, allowing the developement of their chemical reactions towards the arise of chemical combustion products, once an energetic level (ie activation energy) is overcome.

- typically the flame is a very thin discontinuity region due to the finite dimension of flame front, which produces a separation between reactants and combustion products
- usually the chemical reactions are considered oxidation phenomena strongly exothermic and extremely fast, dependently on the involved species
- a reacting gas is a non-isothermal mixture of multiple species which must be tracked individually;
- since mixture of gases are considered, transport coefficients (heat diffusivity, species diffusion, viscosity, etc) require specific attention.

The typical chemical reaction construction is described by means of Penner's formalism

$$A \rightleftharpoons B$$

$$A + B \rightleftharpoons X + Z$$

$$\sum_{i=0}^{n} \nu_i' M_i \rightleftharpoons \sum_{i=0}^{n} \nu_i'' M_i$$

Chemical Reaction Chain

$$\sum_{i=0}^{n} \nu_i^{'} M_i \rightleftharpoons \sum_{i=0}^{n} \nu_i^{''} M_i$$

- $\nu_i^{'}$ is i^{th} stoich coeff relative to reactants
- $\nu_i^{''}$ is i^{th} stoich coeff relative to products
- M_i is i^{th} chemical specie of mixture
- n total n° of chemical species in reactions

Moreover:

Being the reactions strongly exothermic, by introducing the reaction hentalpy (thermal energy released or absorbed during chemical reaction at specified conditions Δh_T) the balance must state

$$\sum_{i=0}^{n} \nu_i^{'} M_i \rightarrow \sum_{i=0}^{n} \nu_i^{''} M_i + \Delta h_{react}$$

Also:

When chemical kinetics is taken into account, the balanced (or global) reaction is considered

$$A + B \rightleftharpoons X + Z$$

But if reaction comes out to be very complex, the mechanism can't occur in just one step, so the single elementary reaction step, one after the other, creates a global reaction chain.

Chemical Reaction Chain

Elementary R.	$A + B \rightarrow C$			
	$\Downarrow$	$\Longleftrightarrow$	$A + B \rightarrow X + Z$	Global R.
Elementary R.	$C \rightarrow X + Z$			

Elementary reactions, which form chemical reaction chains, occur at a certain rate that varies as function of reactants and their concentrations, specified conditions such as pressure and temperature, or some other external factors.

Chemical Reaction Rate ($\dot{\omega}$):

Chemical Reaction Rates indicate the rate of change of reactant (consumption) or products (increase) concentrations.
If one takes **elementary reactions**, concentrations of species change in the interval Δt so that

$$A + B \rightarrow C$$

$$\frac{d[A]}{dt} = \frac{d[B]}{dt} < 0 \text{ while } \frac{d[C]}{dt} > 0$$

While defining as Ψ the reaction development, the Reaction Rate ($\dot{\omega}$) indicates the change in time of the advancement of the reaction, $\frac{d\Psi}{dt}$.

Transport of chemical (reacting) species

Species are characterized through their mass fractions Y_k ($k = [1 : N]$), where N is the number of species in the reacting mixture.

$$Y_k = \frac{m_k}{m}$$

So one has to consider as primitive variables for the reacting flows as:

- density $\rho = \frac{m}{V}$;
- velocity field $\boldsymbol{U}$;
- Energy (e), OR enthalpy (h), OR temperature (T), as they are connected;
- mass fraction Y_k of the N species;

Going from non-reactive to reactive flows
requires to solve for $N - 1 + 5$ variables instead of 5

Also:

For a mixture of N gasses, **total pressure is the sum of partial pressures**:

$$p = \sum_{k=1}^{N} p_k \quad \text{where} \quad p_k = \rho_k \frac{RT}{W_k} \quad \text{being} \quad R = 8.314 \frac{J}{molK}$$

Mathematics recap: thermodynamics

The last one thanks to the Equation of State:

$$p = \rho \frac{RT}{W} \quad \text{where W is the mean molecular weight:} \quad \frac{1}{W} = \sum_{k=1}^{N} \frac{Y_k}{W_k}$$

$$\text{While } \rho = \sum_{k=1}^{N} \rho_k$$

Mind the different definitions:

Quantity	Definition	Useful relations
Mass fraction* (Y_k)	Mass of species k^{th} / Total mass	Y_k
Mole fraction (X_k)	Mass of species k^{th} / Total moles	$X_k = \frac{W}{W_k} Y_k$
Molar concentration ($[X_k]$)	Mass of species k / Unit volume	$[X_k] = \rho \frac{Y_k}{W_k} = \rho \frac{X_k}{W}$

* Mass fraction is the one commonly used in combustion codes

Mathematics recap: thermodynamics

For a reacting flow, **there are multiple possible variables to represent energy or enthalpy**: definitions for energy (e_k), enthalpy (h_k), sensible energy (e_{sk}) and sensible enthalpy (h_{sk}) for one species are given in this table.

Form	Energy	Enthalpy
Sensible	$e_{sk} = \int_{T_0}^{T} c_{vk} dT - RT_0/W_k$	$h_{sk} = \int_{T_0}^{T} c_{pk} dT$
Sensible + chemical	$e_k = e_{sk} + \Delta h^0_{f,k}$	$h_k = h_{sk} + \Delta h^0_{f,k}$

- $\Delta h^0_{f,k}$ = mass enthalpy of formation of species k at T_0 (standard reference state is $T_0 = 298.15$ K: it is diffult to do measurements at $T_0 = 0$ K!).
- In addition to the reference temperature, a *reference enthalpy* (or energy value) is chosen:

$$h_k = \underbrace{\int_{T_0}^{T} c_{pk} dT}_{\text{sensible}} + \underbrace{\Delta h^0_{f,k}}_{\text{chemical}}$$

The sensible enthalpy h_{sk} is *zero* at T=T_0 for all substances. **NOTE**: at T=T_0, $h_k = \Delta h^0_{f,k}$. The mass enthalpy of formation h_k of species k at temperature T_0 is NOT zero!

Mathematics recap: Sensible Energy

Sensible energies are defined to satisfy:

$$h_k = e_{sk} + p_k/\rho_k$$

This choice requires the introduction of the RT_0/W_k term in e_{sk}: the sensible enthalpy h_{sk} is zero at $T = T_0$ but the sensible energy e_{sk} is not:

$$e_{sk}(T_0) = -RT_0/W_k$$

In all these forms, energies and enthalpies are mass quantities: for example, the formation enthalpies $\Delta h^0_{f,k}$, are the enthalpies needed to form 1 kg of species k at the reference temperature T_0 = 298.15 K. These values are linked to molar values $\Delta h^{o,m}_{f,k}$ by:

$$\Delta h^0_{f,k} = \Delta h^{o,m}_{f,k}/W_k$$

Recap: Mass Formation Enthalpy $\Delta h^0_{f,k}$

Some reference values of $\Delta h^0_{f,k}$ for some typical fuels and products:

Substance	Molecular Weight W_k $(mole/kg)$	Mass formation enthalpy (kJ/kg)	Molar formation enthalpy $(kJ/mole)$
CH_4	0.016	-4675	-74.8
C_3H_8	0.044	-2360	-103.8
C_8H_{18}	0.114	-1829	-208.5
CO_2	0.044	-8943	-393.5
H_2O	0.018	-13435	-241.8
O_2	0.032	0	0
H_2	0.002	0	0
N_2	0.028	0	0

Recap: Heat Capacity $C_{p,k}$

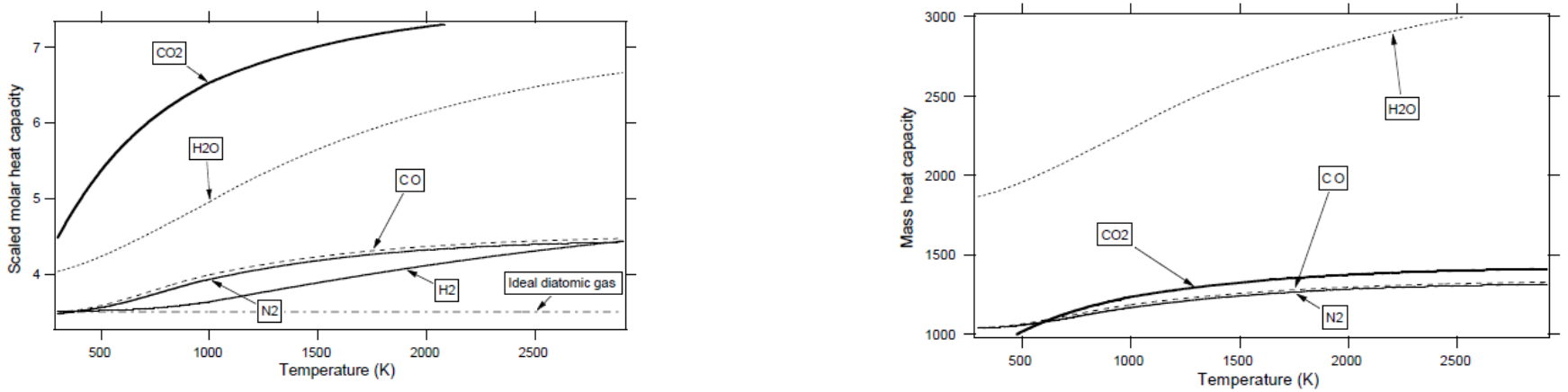


Figure 2: heat capacities C_{pk} (divided by the perfect gas constant R) as a function of temperature. While N_2 and H_2 heat capacities are of the order of 3.5 R at low temperatures, they deviate rapidly from this value at high temperature.

The heat capacities at constant pressure of species k (C_{pk}) are mass heat capacities related to molar capacities $^{m}_{pk}$ by:

$$C_{pk} = C^m_{pk}/W_k$$

For a perfect diatomic gas: $C^m_{pk} = 3.5\,R$ and $C_{pk} = 3.5\,R/W_k$

In practice, the changes of C_{pk} with temperature are large in combusting flows and C^m_{pk} values are usually tabulated as temperature functions using **polynomials** (Stull and Prophet, Heywood).

Recap: Heat capacity $C_{p,k}$

The heat capacity at constant pressure of the mixture, C_p, is:

$$C_p = \sum_{k=1}^{N} C_{pk} Y_k = \sum_{k=1}^{N} C_{pk}^m \frac{Y_k}{W_k}$$

From the equation above, it is apparent that the mixture heat capacity C_p is a function both of temperature T and composition Y_k and that it may change significantly from one point to another.

In most hydrocarbon/air flames, the properties of nitrogen dominate and the mass heat capacity of the mixture is very close to that of Nitrogen. Furthermore, this value changes only from 1000 to 1300 J/(kg K) when temperature goes from 300 K to 3000 K so that C_p is often assumed to be constant in many theoretical approaches and some combustion codes.

Recap: Equations

Mathematical point of view:

The fluid-dynamic problem is solvable thanks to the common equations of

- **mass conservation**

$$\frac{\partial \rho}{\partial t} + \nabla \cdot (\rho \boldsymbol{U}) = 0$$

- **momentum balance**

$$\frac{\partial (\rho \boldsymbol{U})}{\partial t} + \nabla \cdot (\rho \boldsymbol{U}\boldsymbol{U}) = -\nabla p + \frac{2}{3}\mu \nabla \boldsymbol{U} I + \nabla \cdot [\mu(\nabla \boldsymbol{U} + (\nabla \boldsymbol{U})^T] + \rho \boldsymbol{g} + S$$

- **energy eqn.**

$$\frac{\partial (\rho e)}{\partial t} + \nabla \cdot (\rho e \boldsymbol{U}) + \frac{\partial (\rho K)}{\partial t} + \nabla \cdot (\rho K \boldsymbol{U}) = -\nabla \cdot (p \boldsymbol{U}) + \nabla \cdot (\alpha_{eff} \nabla e) + \rho r$$

The energy equation could be written in its enthalpy form thanks to the controller "*he*" (please referr to previous chapters), but with respect to its complete form:

- **mechanical sources** $\nabla \cdot (\tau \boldsymbol{U})$ and $\rho g \boldsymbol{U}$ are neglected
- **heat flux** $q = -\alpha_{eff} \nabla e$ is assumed, where thermal diffusivity $\alpha_{eff} = \alpha_{lam} + \alpha_{turb}$
- **thermal sources terms** ρr are specific to each solver

Chemical ODEs

Chemical reactions impose to solve an additional number of **Ordinary Differential Equations (ODEs)**, function of number of species, and dependant on number of reactions.

Ex: *consider a simple generic chain, and consider specie concentration as "[specie]"*

$$A + B \overset{k_1}{\rightarrow} C$$

$$C \overset{k_2}{\rightarrow} X + Z$$

then

$$\frac{d[A]}{dt} = \frac{d[B]}{dt} = -k_1[A][B]$$

$$\frac{d[C]}{dt} = k_1[A][B] - k_2[C]$$

$$\frac{d[X]}{dt} = \frac{d[Z]}{dt} = k_2[C]$$

Let Y_k be the mass fraction of species k and that N species participate in the chemical mechanism describing our problem, than the **conservation equation for the specie** Y_k is:

$$\frac{\partial \rho Y_k}{\partial t} + \nabla \cdot (\rho \boldsymbol{U} Y_k) = \nabla \cdot (\rho \Gamma_k \nabla Y_k) + \dot{\omega}$$

The **energy equation** is modified with inclusion of chemistry source terms:

$$\frac{\partial (\rho e)}{\partial t} + \nabla \cdot (\rho e \boldsymbol{U}) + \frac{\partial (\rho K)}{\partial t} + \nabla \cdot (\rho K \boldsymbol{U}) = -\nabla \cdot (p \boldsymbol{U}) + \nabla \cdot (\alpha_{eff} \nabla e) + \dot{q} + \rho r$$

Chemical source terms

Therefore, the source terms to be accounted and studied are:

- the reaction rate ($\dot{\omega}$);
- the reaction heat (realeased or absorbed ρr);

They must be computed by a combustion/chemistry model!

Also: The reaction heat (released or absorbed) ρr is given by

$$\rho r = -\sum_{k=1}^{N} h_k \omega_k + \rho \mathcal{D} \frac{\partial T}{\partial x_k} \sum_{k=1}^{N} \left(c_{p,k} \frac{\partial Y_k}{\partial x_k} \right)$$

In which the main quantities appearing are:

- mass fraction of species Y_k;
- diffusion coefficient $\mathcal{D}$;
- chemical source term ω_k;
- absolute enthalpy;
- mixture specific heat capacity $c_{p,m} = \sum_{k=1}^{N} (c_{p,k} Y_k)$;
- mixture enthalpy $h = \sum_{k=1}^{N} (h_k Y_k)$;

Mathematics recap

$A + B \xrightarrow{k_1} C$

"k" is given by the **Arrhenius Law**

Arrhenius Law:

So as to determine the **rate constant** or **specific rate** (k) that depends ONLY on temperature

$$k = BT_a e^{\left(-\frac{E_a}{R_u T}\right)}$$

- BT_a: collision frequency
- E_a: activation energy
- $e^{\left(-\frac{E_a}{R_u T}\right)}$ Boltzmann factor

The eponential term inclides the greatest non-linearity of the system and it is the main reason why turbulent reactive flows are quite difficult to be modelled

Extremely important as this is linked to the reaction rate!

$$\dot{\omega} = k \prod_{i=1}^{N} [M_i]^{\nu_i'}$$

Mind also that the reaction rate for a *"forward"* reaction (reactants to products) is not the same as the one in the *"opposite direction"*

Mathematics recap

Computationally, ODEs must be solved for a number of chemical subtimesteps for every fluid dynamic timestep. This is necessary in order to obtain **source terms**, while **preserving mass conservation**.

Solve ODEs for all chemical subtimestep until next fluid dynamic timestep

Calculate HEAT SOURCE terms

Insert terms into fluid dynamic equations

$$\frac{\partial(\rho \boldsymbol{U})}{\partial t} + \nabla \cdot (\rho \boldsymbol{U}\boldsymbol{U}) = -\nabla p + \frac{2}{3}\mu \nabla \boldsymbol{U} I + \nabla \cdot [\mu(\nabla \boldsymbol{U} + (\nabla \boldsymbol{U})^T] + \rho \boldsymbol{g} + S$$

and

$$\frac{\partial(\rho e)}{\partial t} + \nabla \cdot (\rho e \boldsymbol{U}) + \frac{\partial(\rho K)}{\partial t} + \nabla \cdot (\rho K \boldsymbol{U}) =$$
$$-\nabla \cdot (p\boldsymbol{U}) + \frac{2}{3}\mu(\nabla \cdot \boldsymbol{U})\boldsymbol{U} + \nabla \cdot \left[\mu(\nabla \boldsymbol{U} + (\nabla \boldsymbol{U})^T)\boldsymbol{U}\right] + \nabla \cdot (\lambda T) + \rho \dot{Q} + \rho \boldsymbol{g}\boldsymbol{U}$$

while preserving $\frac{\partial \rho}{\partial t} + \nabla \cdot (\rho \boldsymbol{U}) = 0$

Repeat

Mass conservation

By linking mass conservation with Penner's rule, it must be enforced that

$$\sum_{i=1}^{N} \nu'_{i,j} W_k = \sum_{i=1}^{N} \nu''_{i,j} W_k \quad \text{or} \quad \sum_{k=1}^{N} \nu'_{k,j} W_k = 0 \quad \text{for } j = 1 : M$$

$$\text{in which} \quad \nu_{k,j} = \nu''_{k,j} - \nu'_{k,j}$$

Total mass conservation equation is unchanged compared to non-reacting flows (**as combustion DOES NOT generate mass**). Thus, mass conservation for the k^{th} specie yelds:

$$\frac{\partial(\rho Y_k)}{\partial t} + \nabla \cdot \rho(\boldsymbol{U} + \boldsymbol{V_k}) Y_k = \dot{\omega}, \text{ for } k = 1 : N$$

where $\boldsymbol{V_k}$ is the **diffusion velocity** of the k^{th} specie and $\dot{\omega}$ is the reaction rate.

Summing all species equations and combining it with $\sum_{k=1}^{N} \dot{\omega} = 0$ gives:

$$\frac{\partial \rho}{\partial t} + \frac{\partial(\rho U_i)}{\partial x_i} = -\frac{\partial \left(\rho \sum_{k=1}^{N} (Y_k V_{k,i}) \right)}{\partial x_i} + \sum_{k=1}^{N} \dot{\omega_k} = -\frac{\partial}{\partial x_i} \left(\rho \sum_{k=1}^{N} (Y_k V_{k,i}) \right)$$

Mass conservation

The last one must lead to the total mass conservation that states $\frac{\partial \rho}{\partial t} + \frac{\partial \rho U_i}{\partial x_i} = 0$
So that a **necessary condition** is:

$$\boxed{\sum_{k=1}^{N} Y_k \boldsymbol{V_{k,i}} = 0}$$

The full equations to calculate the diffusion velocities $\boldsymbol{V_k}$ are obtained by solving the system:

$$\nabla X_p = \sum_{k=1}^{N} \frac{X_p X_k}{\mathcal{D}_{p,k}} (\boldsymbol{V_k} - \boldsymbol{V_p}) + (Y_p - X_p) \frac{\nabla P}{P} + \frac{\rho}{p} \sum_{k=1}^{N} Y_p Y_k (f_p - f_k)$$

Where

- $\mathcal{D}_{p,k} = \mathcal{D}_{k,p}$ is the binary diffusion coefficient of species p into species k;
- X_k is the mole fraction of specie k;

The last equation is a linear system of size N^2 which must be solved in each direction at each point and at each instant for unsteady flows. Mathematically, this task is difficult and expensive, so two simplifications are commonly used:

- the **Fick's law** for theoretical and analytical flame studies;
- the **Hirschfelder and Curtiss approxiametion** in most numerical tools;

Fick's Law

If pressure gradients are small and volume forces are neglected, the full system of equations can be solved only in two cases:

1) If the mixture contains only two species, the system reduces to a scalar equation where the unknowns are the two diffusion velocities V_1 and V_2.

$$\nabla X_1 = \frac{X_1 X_2}{\mathcal{D}_{12}}(V_2 - V_1)$$

Starting from $Y_1 = W_1 X_1 / W$, it is simple to show that $\nabla X_1 = W^2/(W_1 W_2)\nabla Y_1)$. Knowing that $Y_1 V_1 + Y_2 V_2 = 0$ it follows:

$$V_1 X_1 = -\mathcal{D}_{12}\frac{Y_2}{X_2}\nabla X_1 \quad \rightarrow \quad \boxed{V_1 = -\mathcal{D}_{12}\nabla ln(Y_1)}$$

2) A second case where Fick's law is exact is multispecies diffusion (N>2) when all binary diffusion coefficients are equal $\mathcal{D}_{ij} = \mathcal{D}$. In this case, V_p can be calculated as:

$$V_p = -\mathcal{D}\nabla ln(Y_p)$$

As soon as a more detailed description of transport is required (typically to describe more complex kinetics), Fick's law should not be used and in most codes it should be replaced by the Hirschfelder and Curtiss approximation.

Hirschfelder and Curtiss approximation

When Fick's law can not be used, the rigorous inversion of the system in a multispecies gas is often replaced by the Hirschfelder and Curtiss approximation which is the best first-order approximation to the exact resolution of the full equations to calculate the diffusion velocities $\boldsymbol{V}_k$:

$$V_k X_k = -\mathcal{D}_k \nabla X_k \quad \text{with} \quad \mathcal{D}_k = \frac{1 - Y_k}{\sum_{j \neq k} X_j / \mathcal{D}_{jk}}$$

The coefficient $\mathcal{D}_k$ is not a binary diffusion but an equivalent diffusion coefficient of species k into the rest of the mixture. The species equation takes the form:

$$\frac{\partial (\rho Y_k)}{\partial t} + \frac{\partial (\rho u_i Y_k)}{\partial x_i} = \frac{\partial}{\partial x_i} \left(\rho \mathcal{D}_k \frac{W_k}{W} \frac{\partial X_k}{\partial x_i} \right) + \dot{\omega}_k$$

The Hirschfelder and Curtiss approximation is a convenient approximation because the diffusion coefficients $\mathcal{D}_k$ can be simply linked to the heat diffusivity $\mathcal{D}_{th}$ in many flames: the Lewis numbers of individual species $Le_k = \frac{\mathcal{D}_{th}}{\mathcal{D}_k}$ are usually varying by small amounts in flame fronts.

Lewis numbers of individual species

The Hirschfelder and Curtiss approximation is a convenient approximation because the diffusion coefficients $\mathcal{D}_k$ can be simply linked to the heat diffusivity $\mathcal{D}_{th}$ in many flames: the Lewis numbers of individual species $Le_k = \frac{\mathcal{D}_{th}}{\mathcal{D}_k}$ are usually varying by small amounts in flame fronts.

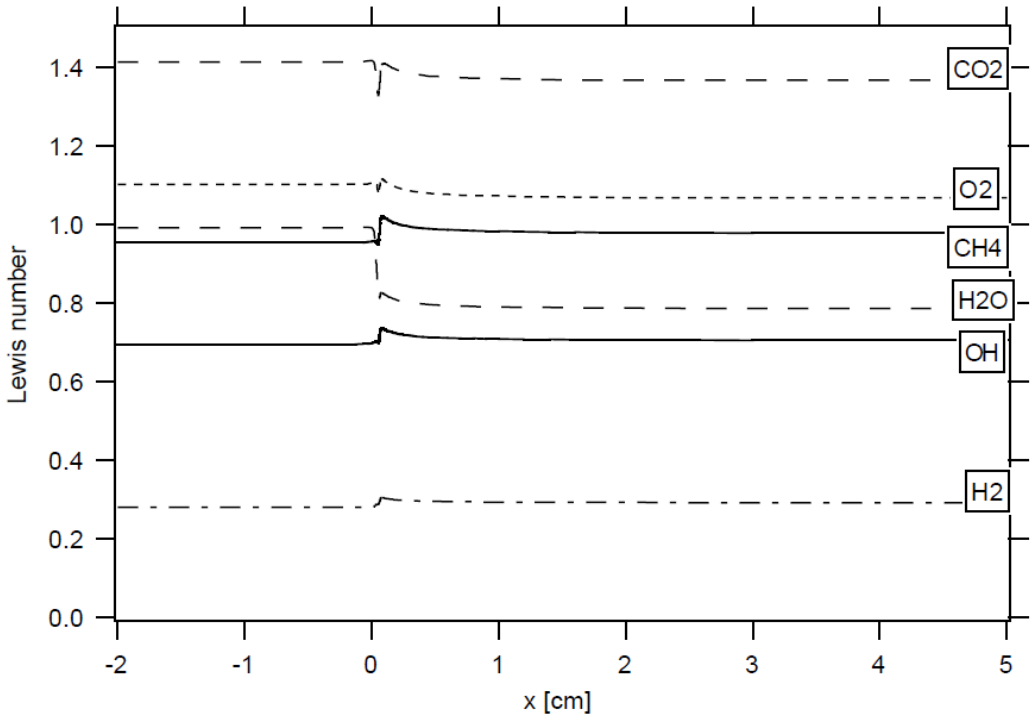


Figure 3: Computation of Lewis numbers of main species for a premixed stoichiometric methane/air flame plotted versus spatial coordinate through the flame front. Lewis numbers change through the flame front (located here at x=0) but these changes are small.

Energy Equation (from previous lectures..)

The energy equation is generally implemented in the form of

- Total Energy

$$\frac{\partial(\rho e)}{\partial t} + \nabla \cdot (\rho \boldsymbol{U} e) + \frac{\partial(\rho K)}{\partial t} + \nabla \cdot (\rho \boldsymbol{U} K) + \nabla \cdot (\boldsymbol{U}\, p) = \alpha_{\text{eff}} \nabla e + \rho\, r$$

- Total Enthalpy

$$\frac{\partial(\rho h)}{\partial t} + \nabla \cdot (\rho \boldsymbol{U} h) + \frac{\partial(\rho K)}{\partial t} + \nabla \cdot (\rho \boldsymbol{U} K) = \alpha_{\text{eff}} \nabla e + \rho\, r$$

With respect to its complete form, in the implementation of the energy equation in OpenFOAM:

- **mechanical sources** $\nabla \cdot (\tau \cdot \boldsymbol{U})$ and $\rho \boldsymbol{g} \cdot \boldsymbol{U}$ are neglected;
- a **heat flux** $q = -\alpha_{\text{eff}} \nabla e$ is assumed, where the effective thermal diffusivity α_{eff} is the sum of laminar and turbulent thermal diffusivities;
- **thermal source terms** $\rho\, r$ are specific to the particular solver.

Energy Equation

For a mixture of N species, the different forms used for energy and enthalpy (eight because of the possible combinations between sensible, kinetic and chemical parts) are:

Form	Energy	Enthalpy
Sensible	$e_s = \int_{t_0}^{T} C_v dT - RT_0/W$	$h_s = \int_{t_0}^{T} C_p dT$
Sensible+Chemical	$e = e_s + \sum_{k=1}^{N} \Delta h^0_{f,k} Y_k$	$h = h_s + \sum_{k=1}^{N} \Delta h^0_{f,k} Y_k$
Total Chemical	$e_t = e + \frac{1}{2} u_i u_i$	$h_t = h + \frac{1}{2} u_i u_i$
Total non Chemical	$E = e_s + \frac{1}{2} u_i u_i$	$H = h_s + \frac{1}{2} u_i u_i$

The enthalpy h is defined by:

$$h = \sum_{k=1}^{N} h_k Y_k = \sum_{k=1}^{N} \left(\int_{T_0}^{T} C_{pk} dT + \Delta h^0_{f,k} \right) Y_k = \int_{T_0}^{T} C_p dT + \sum_{k=1}^{N} \Delta h^0_{f,k} Y_k$$

while the energy $e = h - p/\rho$ is:

$$e = \sum_{k=1}^{N} \left(\int_{T_0}^{T} C_{pk} dT - RT/W_k + \Delta h^0_{f,k} \right) Y_k = \sum_{k=1}^{N} \left(\int_{T_0}^{T} C_{vk} dT - RT_0/W_k + \Delta h^0_{f,k} \right) Y_k$$

$$= \int_{T_0}^{T} C_v dT - RT_0/W + \sum_{k=1}^{N} \Delta h^0_{f,k} Y_k = \sum_{k=1}^{N} e_k Y_k$$

Molecular transport of species and heat 1/2

The heat diffusion coefficient is called λ.

- Diffusion processes involve binary diffusion coefficients (D_{kj}) and require the resolution of a system giving diffusion velocities. This is not done in most combustion codes, because solving the diffusion problem in a multi-species gas is a problem in itself.

- The diffusion coefficient of species k in the rest of the mixture (used in **Fick's law**, see below) is called D_k. Simplified diffusion laws (usually Fick's law) are used in a majority of combustion codes and this text is restricted to this case. The coefficients D_k are often characterized in terms of Lewis number defined by:

$$Le_k = \frac{\lambda}{\rho C_p D_k} = \frac{D_{th}}{D_k}$$

  where $D_{th} = \frac{\lambda}{\rho C_p}$ is the heat diffusivity coefficient. The Lewis number Le_k compares the diffusion speeds of heat and species k.

Molecular transport of species and heat 2/2

Le_k is a local quantity but, in most gases, it changes very little from one point to another. The kinetic theory of gases shows that:

$$\lambda \sim T^{0.7} \qquad \rho \sim 1/T \qquad D_k \sim T^{1.7}$$

$\rightarrow Le_k$ is changing only by a few percents in a flame.

The **Prandtl Number** compares momentum and heat transport

$$Pr = \frac{\nu}{\lambda/(\rho C_p)} = \frac{\rho \nu C_p}{\lambda} = \frac{\mu C_p}{\lambda}$$

The **Schmidt Number**, S_{C_k} compares momentum and species k molecular diffusion:

$$S_{C_k} = \frac{\nu}{D_k} = Pr\, Le_k$$

Global mass conservation with species

Mass conservation is a specific issue when dealing with reacting flows. Obviously, the sum of mass fractions must be unity: $\sum_{k=1}^{N} Y_k = 1$. This equation adds N equations while there are only N unknowns (the Y_k's): the system is over-determined and there are only N independent equations. Any one of the N species equations or the mass conservation equation may be eliminated: this apparent problem is not a difficulty when exact expressions for diffusion velocities are used.

However, this is no longer the case when Hirschfelder's law is used. In this case, the RHS of equation

$$\frac{\partial \rho}{\partial t} + \frac{\partial \rho u_i}{\partial x_i} = -\frac{\partial \left(\rho \sum_{k=1}^{N} Y_k V_{k,i} \right)}{\partial x_i} + \sum_{k=1}^{N} \dot{\omega}_k = -\frac{\partial}{\partial x_i} \left(\rho \sum_{k=1}^{N} Y_k V_{k,i} \right)$$

becomes:

$$\frac{\partial}{\partial x_i} \left(\rho \sum_{k=1}^{N} D_k \frac{W_k}{W} \frac{\partial X_k}{\partial x_i} \right)$$

which is not zero: global mass is not conserved.

Global mass conservation with species

Despite this drawback, most codes in the combustion community use Hirschfelder's law (or even Fick's law) because solving for the diffusion velocities is too complex. **Two methods can be used to implement such laws and still maintain global mass conservation**.

- the **first and the simplest method** is to solve the global mass conservation equation and only N-1 species equations using directly $V_k X_k = -\mathcal{D}_k \nabla X_k$ to express diffusion velocities. The last species mass fraction (usually a diluent such as N_2) is obtained by writing $Y_N = 1 - \sum_{k=1}^{N-1} Y_k$ and absorbs all inconsistencies introduced by the calculation in the transport of the other species.

  $\rightarrow$ this simplification is dangerous and should be used only when flames are strongly diluted (for example in air so that Y_{N_2} is large).

NOTE: in the simulation of **multiphase flows based on Volume of Fluids (VoF) methods**, a method based on a correction velocity V_c is applied to ensure global conservation of the volume fraction. This is quite similar to what is done with reactive flows!

Global mass conservation with species

- In the **second method**, a correction velocity V_c is added to the convection velocity in the species equations:

$$\frac{\partial \left(\rho Y_k\right)}{\partial t} + \frac{\partial}{\partial x_i}\rho\left(u_i - V_i^c\right) Y_k = \frac{\partial}{\partial x_i}\left(\rho \mathcal{D}_k \frac{W_k}{W}\frac{\partial X_k}{\partial x_i}\right) + \dot{\omega}_k$$

The correction velocity V_c is evaluated to ensure global mass conservation. If all species equations are summed, the mass conservation equation must be recovered:

$$\frac{\partial \rho}{\partial t} + \frac{\partial \rho u_i}{\partial x_i} = \frac{\partial}{\partial x_i}\left(\rho \sum_{k=1}^{N} \mathcal{D}_k \frac{W_k}{W}\frac{\partial X_k}{\partial x_i} - \rho V_i^c\right) = 0 \quad \rightarrow \quad \boxed{V_i^c = \sum_{k=1}^{N} \mathcal{D}_k \frac{W_k}{W}\frac{\partial X_k}{\partial x_i}}$$

At each time step, the correction velocity is computed and added to the convecting field $\vec{u}$ to ensure the compatibility of species and mass conservation equations.

In this case, it is still possible to solve for (N-1) species and for the total mass but, unlike for the first method, the result for Y_N is correct.

NOTE: in the simulation of **multiphase flows based on Volume of Fluids (VoF) methods**, a method based on a correction velocity V_c is applied to ensure global conservation of the volume fraction. This is quite similar to what is done with reactive flows!

Linking chemistry and OpenFOAM

There are 3 important values that must be considered for every specie, in every elementary chemical reaction composing a chemical chain: sign, stoichiometric coefficient and exponent.

Ex: H_2 reaction (5 elements, 10 species, 27 elementary reactions)

Elements: H, O, C, N, Ar *Species:* $H_2, H, O_2, O, OH, HO_2, H_2O_2, H_2O, Ar, N_2$

$$OH + H_2 \rightarrow H + H_2O$$

This table is referred to reaction 1 on 27.

Therefore, ideally one should construct 27 tables considering these parameters

Species	Index	Stoich coeff	Exponent
H_2	-1 (left)	1.0	1
H	1 (right)	1.0	1
0_2	0	-	-
0	0	-	-
OH	-1 (left)	1.0	1
$H0_2$	0	-	-
...	...	...	...

These tables are automatically constructed within OpenFOAM.

The CHEMKIN format

But how can `OpenFOAM` access all these informations?

This is possible thanks to the new files introduced into a new folder named **Chemkin Folder**

Chemkin II format: Fortran chemical kinetics package

[See CHEMKIN Collection Release 3.6, September 2000]

The folder contains different files such as:

chem.inp

- Elements are listed;
- Species are listed;
- Reactions are listed;

```
ELEMENTS
H   O   N
END

SPECIES
H2  H  O2  O  OH  HO2  H2O2  H2O  N2
END

REACTIONS
H+O2+M=HO2+M                          3.61E17  -0.72
  H2O/18.6/  H2/2.86/
H+H+M=H2+M                            1.0E18   -1.0
H+H+H2=H2+H2                          9.2E16   -0.6
                                      6.0E19   -1.25
                                      1.6E22   -2.0

                                      6.2E16   -0.6

                                      1.89E13   0.0    -1
                                      1.3E17    0.0    45
                                      1.7E13    0.0    47
                                      1.17E9    1.3     3
O+OH=O2+H                             3.61E14  -0.5
O+H2=OH+H                             5.06E4    2.67    6
OH+HO2=H2O+O2                         7.5E12    0.
H+HO2=2OH                             1.4E14    0.0     1
```

For complete description of the CHEMKIN interpreter input files see:

Chemkin-II: A Fortran Chemical Kinetics Package for the Analysis of Gas-Phase Chemical Kinetics, R.J.Kee, F.M. Rupley, and J.A.Miller, Sandia Report (1995).

The CHEMKIN format

First, elements and species which have been retained for the scheme are listed. For each reaction, the table then gives A_{fj} in cgs units, β_j and E_j in cal/mole. The backwards rates K_{rj} are computed from the forward rates through the equilibrium constants:

$$K_{rj} = \frac{K_{fj}}{\left(\frac{p_a}{RT}\right)^{\sum_{k=1}^{N}} e^{\left(\frac{\Delta S_J^0}{R} - \frac{\Delta H_j^0}{RT}\right)}}$$

where p_a= 1 bar.

- The Δ symbols refer to changes occurring when passing from reactants to products in the j_{th} reaction;
- ΔH_{j0} is the enthalpy changes for reaction $j \rightarrow$ from tabulations
- ΔS_{j0} is the entropy changes for reaction $j \rightarrow$ from tabulations

The CHEMKIN format

therm.dat

- Initial values of the problem:

  T_{low};
  T_{common};
  T_{high};
  T_{max};

- Relevant quantities for every specie:

  number of moles;
  molar weight;
  c_p coefficients at different states;

Moreover:
transportProperties

And some other files
which are contained in the **constant folder**.

```
THERMO ALL
   200.000  1000.000  5000.000
AR                120186AR  1               G  0200.00   5000.
 0.02500000E+02 0.00000000E+00 0.00000000E+00 0.00000000E+00 0
-0.07453750E+04 0.04366001E+02 0.02500000E+02 0.00000000E+00 0
 0.00000000E+00 0.00000000E+00-0.07453750E+04 0.04366001E+02
H                 120186H   1               G  0200.00   5000.
 0.02500000E+02 0.00000000E+00 0.00000000E+00 0.00000000E+00 0
 0.02547163E+06-0.04601176E+01 0.02500000E+02 0.00000000E+00 0
 0.00000000E+00 0.00000000E+00 0.02547163E+06-0.04601176E+01
H2                121286H   2               G  0200.00   5000.
 0.02991423E+02 0.07000644E-02-0.05633829E-06-0.09231578E-10 0
-0.08350340E+04-0.01355110E+02 0.03298124E+02 0.08249442E-02-0
-0.09475434E-09 0.04134872E-11-0.01012521E+05-0.03294094E+02
H2O                20387H   2O   1          G  0200.00   5000.
 0.02672146E+02 0.03056293E-01-0.08730260E-05 0.01200996E-08-0
-0.02989921E+06 0.06862817E+02 0.03386842E+02 0.03474982E-01-0
 0.06968581E-07-0.02506588E-10-0.03020811E+06 0.02590233E+02
H2O2               120186H   2O   2          G  0200.00   5000.
 0.04573167E+02 0.04336136E-01-0.01474689E-04 0.02348904E-08-0
-0.01800696E+06 0.05011370E+01 0.03388754E+02 0.06569226E-01-0
-0.04625806E-07 0.02471515E-10-0.01766315E+06 0.06785363E+02
HO2                20387H   1O   2          G  0200.00   5000.
 0.04072191E+02 0.02131296E-01-0.05308145E-05 0.06112269E-09-0
-0.01579727E+04 0.03476029E+02 0.02979963E+02 0.04996697E-01-0
 0.02354192E-07-0.08089024E-11 0.01762274E+04 0.09222724E+02
O                 120186O   1               G  0200.00   5000.
 0.02923080E+06 0.04920308E+02 0.02946429E+02-0.01638166E-01 0
 0.02542060E+02-0.02755062E-03-0.03102803E-07 0.04551067E-10-0
-0.01602843E-07 0.03890696E-11 0.02914764E+06 0.02963995E+02
O2                121386O   2               G  0200.00   5000.
 0.03697578E+02 0.06135197E-02-0.01258842E-05 0.01775281E-09-0
-0.01233930E+05 0.03189166E+02 0.03212936E+02 0.01127486E-01-0
 0.01313877E-07-0.08768554E-11-0.01005249E+05 0.06034738E+02
OH                1212860   1H   1          G  0200.00   5000.
 0.02882730E+02 0.01013974E-01-0.02276877E-05 0.02174684E-09-0
 0.03886888E+05 0.05595712E+02 0.03637266E+02 0.01850910E-02-0
 0.02387203E-07-0.08431442E-11 0.03606782E+05 0.01358860E+02
```

Chemical Kinetics and Combustion Modeling

At this point, for numerical combustion users, it is important to mention that data on $\mathcal{Q}_j$ correspond to models: except for certain reactions, these data are obtained experimentally and values of kinetic parameters are often disputed in the kinetics community.

- For many flames, very different schemes are found in the literature with very comparable accuracy and the choice of a scheme is a difficult and controversial task. Unfortunately, the values of the parameters used to compute $\mathcal{Q}_j$ and the stiffness associated with the determination of $\mathcal{Q}_j$ create a central difficulty for numerical combustion: the space and time scales corresponding to the $\mathcal{Q}_j$ terms are usually very small and require meshes and time steps which can be orders of magnitude smaller than in non reacting flows.
- Particular attention must be paid to the activation energy E_j (usually measured in kcal/mole in the combustion community): the exponential dependence of rates to E_j leads to considerable difficulties when E_j takes large values (typically more than 60 kcal/mole).
- A given combustion code for laminar flames may work perfectly with a given mesh when E_j is small but will require many more points to resolve the flame structure when E_j is increased even by a small amount.

Chemistry Activation in OpenFOAM

Activation of Chemistry complicates an already intricate problem. Whenever Chemistry is accounted in OpenFOAM, computational time and cost MUST increase:

- the subset of equations becomes more complex;
- ODEs must be solved in addition to fluid dynamic equations;
- chemical subtimestep is smaller than flui dynamic one by several order of magnitude;
- within OpenFOAM a **serial process** imposes the calculation of solution **one cell at time**;

The latter is one of the biggest sources for the increase in computational cost and time. This construction is anyway forced by the several "*if statements control*" applied on chemical reactions:

- if $T > T_{react}$
- if $nStep < nMax$
- if $err < tol$

There are two models of chemistry treatment: **Standard** and **TDAC**
In this course only the **Standard Chemistry Model** will be taken into consideration.

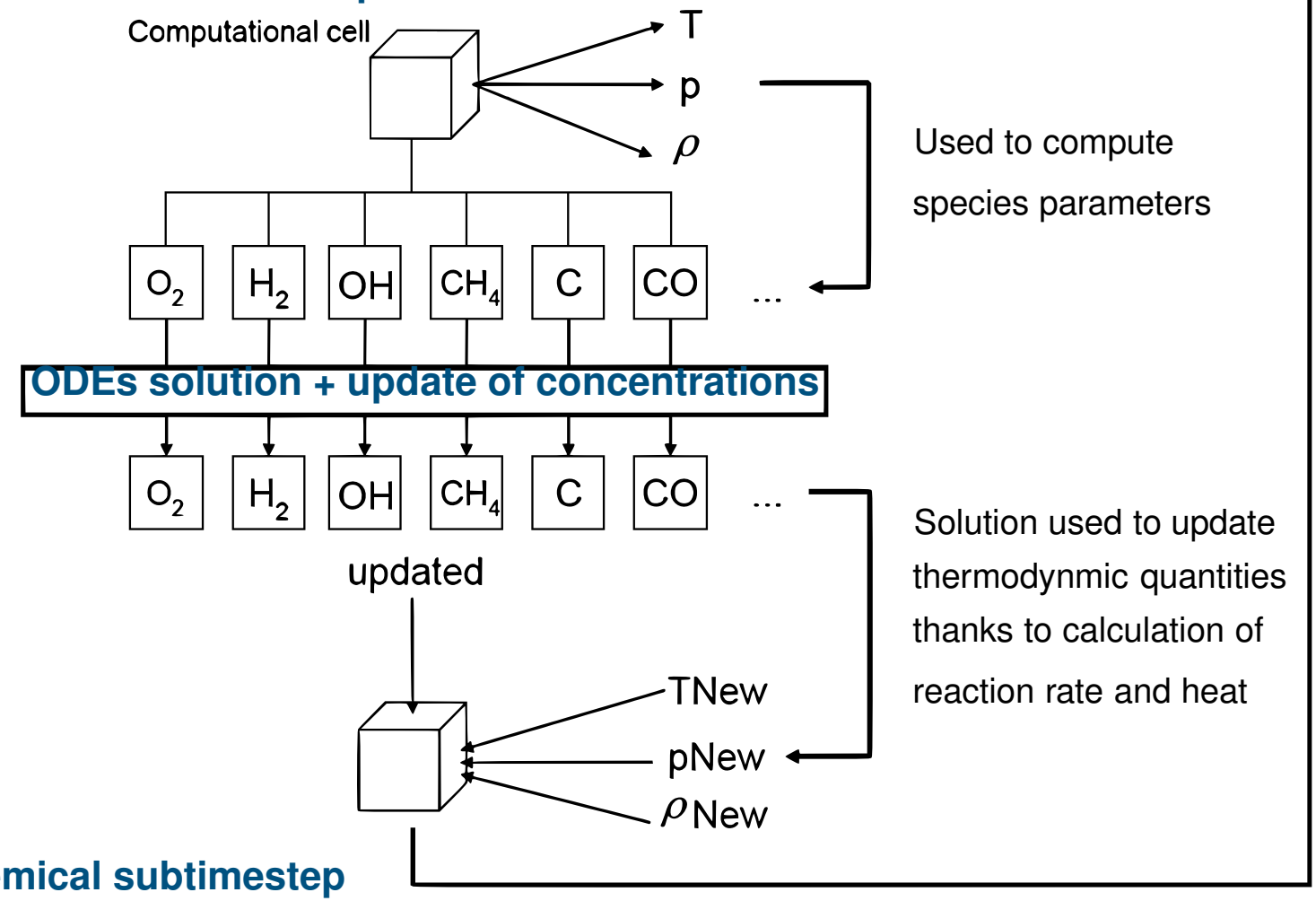
Single Reactor Simulations:

chemFoam

Single reactor case: chemFoam

Begin simulation

Begin chemical subtimestep



End chemical subtimestep

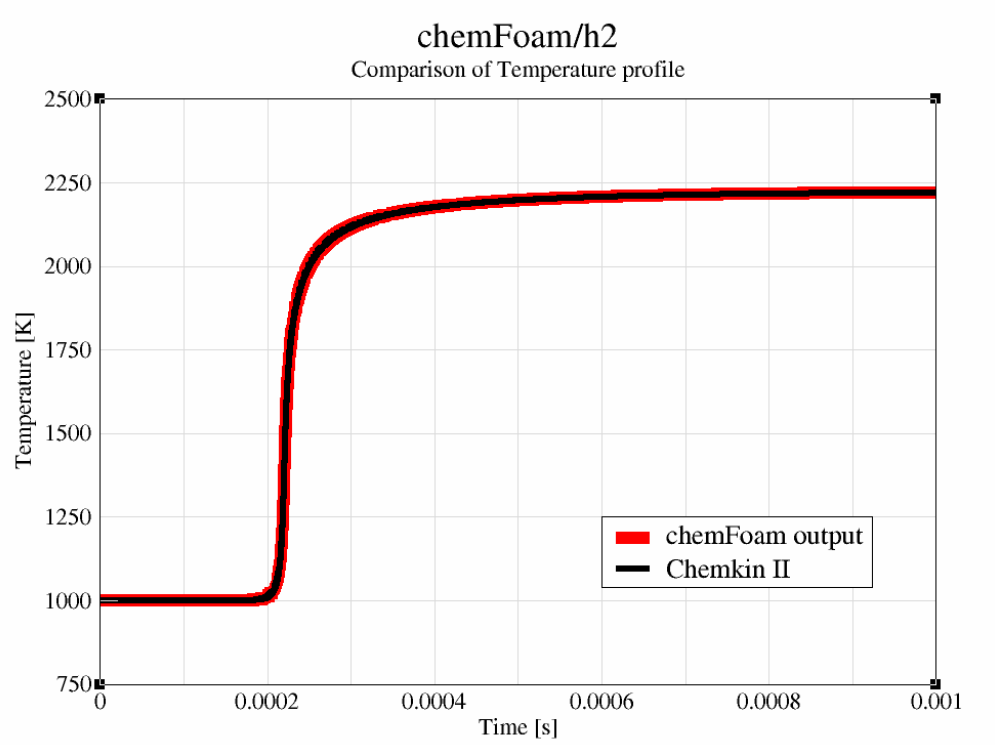
End simulation

Let's consider an example

It was introduced at the very beginning the very easy H_2 reaction.
Let's consider a **single cell (single reactor cell) example**, thanks to the use of `chemFoam`

Chemical reactions must be considered, BUT fluid dynamic conditions are "fixed", as

- temperature is allowed to change (see graph);
- pressure is fixed (isobaric process);
- velocity is absent ($U = 0$);



Let's look at the Solver Code in detail!

chemFoam - 1/3

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

    Info<< "\nStarting time loop\n" << endl;

    while (runTime.run())
    {
        #include "readControls.H"

        #include "setDeltaT.H"

        runTime++;
        Info<< "Time = " << runTime.timeName() << nl << endl;

        #include "solveChemistry.H"
        #include "YEqn.H"
        #include "hEqn.H"
        #include "pEqn.H"

        #include "output.H"

        Info<< "ExecutionTime = " << r
            << "  ClockTime = " << ru
            << nl << endl;
    }

    Info << "Number of steps = " << runTime.timeIndex() << endl;
    Info << "End" << nl << endl;

    return 0;
}
```

Chemistry is solved in this position, before the `Yeqn` in which the conservation of mass species is produced

```
dtChem = chemistry.solve(runTime.deltaT().value());
scalar Qdot = chemistry.Qdot()()[0]/rho[0];
integratedHeat += Qdot*runTime.deltaT().value();
```

MIND THE LACK OF `Ueqn` IN THE PROCESS, BUT THE PRESENCE OF THE ENTHALPY ONE as heat is present!

chemFoam - 2/3

```
dtChem = chemistry.solve(runTime.deltaT().value());
scalar Qdot = chemistry.Qdot()()[0]/rho[0];
integratedHeat += Qdot*runTime.deltaT().value();
```

The other two lines calculate the `Qdot` and finally the Heat which is the source term for the Enthalpy equation!

The entire calculation of Reaction Rate occurs in the very first line of this code contained in `solveChemistry.H`

HOW CAN THE REACTION RATE BE OBTAINED?

Let's look at `chemistry.solve(time)`

```
if (constProp != "temperature")
{
    volScalarField& h = thermo.he();

    if (constProp == "volume")
    {
        h[0] = u0 + p[0]/rho[0] + integratedHeat;
    }
    else
    {
        h[0] = h0 + integratedHeat;
    }

    thermo.correct();
}
```

chemistry.solve() shows by means of the file `createFieldRefs.H` the link between `chemistry` and `pChemistry`, which is associated with the `BasicChemistryModel` template, and in particular by means of `rhoReactionThermo` is generated as `New(ReactionThermo & thermo)`

[for this very last passage look at line 45 of `BasicChemistryModel.C` *- OpenFOAM 7]*

This file in turn invokes `basicChemistryModel.H` which, thanks to its `basicChemistryModel.C` file, is able to read all the impositions of the thermodynamic models and chemistry that are selected within the files contained in the constant folder.

[Mind the difference between "Uppercase B" of `BasicChemistryModel` *, which is a template, and "lower-case b"* `basicChemistryModel`

WHAT ABOUT THESE FILES?

constant/chemistryProperties

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

```
chemistryType
{
    solver              ode;
    method              TDAC;
}
```

The two main keywords are:

- Solver
- Method

```
chemistry       on;
```

Command to activate the chemistry processes in `OpenFOAM`

```
initialChemicalTimeStep 1e-7;
```

Initial chemical subtimestep must be provided.
Usually smaller than fluid dynamic one by several order of magnitude

```
odeCoeffs
{
    solver          seulex;
    absTol          1e-12;
    relTol          1e-1;
}
```

```
reduction
{
    tolerance   1e-4;

    // Search initiating set
    initialSet
    {
        CO;
        IC8H18;
        HO2;
    }
}
```

Typically **ode** is chosen as solver method, and a dictionary containing the **type of ODE solver, as much as tolerances for convergence**, must be provided (*2nd red box*). If no choice is made on tolerances, default ones are applied.

Talking about **method**, the default one is **standard**. So, if no other choice is produced and the keyword is not set, the standard method will be applied. Otherwise, chosen **TDAC**, an interpolation starting from tables will be applied.

constant/chemistryProperties

```
initialChemicalTimeStep 1e-7;

odeCoeffs
{
    solver          seulex;
    absTol          1e-12;
    relTol          1e-1;
}
```

TDAC methodology consists in linear interpolation starting from tabulated quantities, but since this may cause an increase in the error on the calculation, and moreover it lays beyond our interests, this option is neglected and only the standard mode (then StandardChemistryModel) is the one accounted.

```
reduction
{
    tolerance   1e-4;

    // Search initiating set (SIS) of species, needed for most methods
    initialSet
    {
        CO;
        IC8H18;
        HO2;
    }
}

// Tabulation is not effective for single-cell ignition calculations
tabulation
{
    method    none;
}

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

The different ODEs treatment methods still have to be analyzed!

OpenFOAM Chemistry ODEs treatment

In order to analyze in details how OpenFOAM treats chemical problems one must not forget that:

- space and time are computationally discretized;
- chemistry adds equations (ODEs) to fluid dynamic ones;
- chemistry adds SOURCE TERMS to fluid dynamic equations;
- coupling thermo/fluid dynamic is present;

But ODEs can be solved in two different ways!

Explicit Method	**Implicit Method**
Runge-Kutta 45;	seulex;
Runge-Kutta Cash-Karp;	trapezoid;
...	...

When problems become STIFF, the explicit integration algorithms are still potentially feasible, but to maintain stability they must undergo to a decrease in step dimension that leads the latter to be so small that the number of steps becomes unacceptably large, and sometimes **the risk of underflow is present**.

EXPLICIT chemistry resolution method

Explicit Method

- Runge-Kutta 45;
- Runge-Kutta Cash-Karp;
- ...

Discretize timestep must be extremely small in order to fulfill convergence and accuracy of results for every discretized cell. In this way it is guaranteed the LOW COST PER TIMESTEP, requiring anyway HIGH NUMBER OF STEPS.
Computational cost for a complex problem can become thus very big.

- Low cost per timestep;
- Easy mathematical implementation;
- No discard of solution required;
- No Jacobian computation required;
- Optimal if small fluid dynamic timestep;
- Optimal if few cells are considered;

- High number of chemical timestep for every fluid-dynamic one
- Very bad if high number of cells
- Progressively bad increasing species and/or reactions

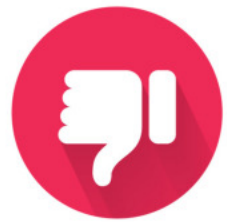
IMPLICIT chemistry resolution method

Implicit Method

- seulex;
- trapezoid;
- ...

Discretize timestep can increase its size ad much as convergence under tolerance is guaranteed, providing less chemical subtimesteps for every fluid-dynamic one. The cost per timestep increases, and the mathematical implementation becomes much more complex.
Global cost for complex or particular problems can become thus very big.

- Small number of chemical timesteps per fluid dynamic one;
- Solution gets ricalculated until convergence is guaranteed;
- Optimal if big fluid dynamic timestep;
- Optimal when high number of cells;



- Jacobian calculation increases cost;
- Discarding solution means ricalculation;
- with smaller timestep;
- High cost per timestep;

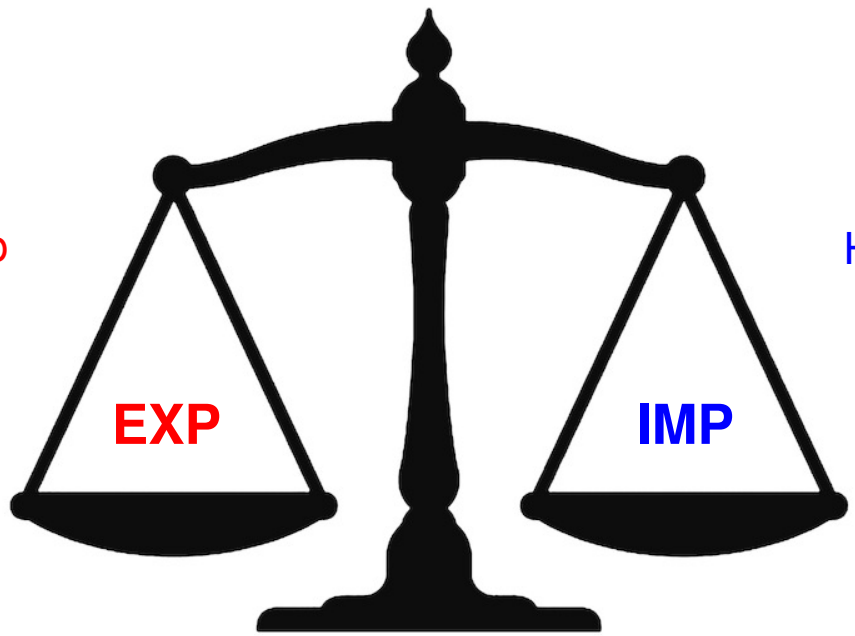
Which method is the best?

BEST METHOD IS PROBLEM DEPENDANT

Whenever a "*single reactor*" (ie a single cell) is considered and/or fluid dynamic timestep is small, explicit method is usually the optimal choice.
If the number o cells, species, and/or reaction increases, then implicit method becomes the best option!

The MAIN GOAL is to always REDUCE as much as possible COMPUTATIONAL TIME AND COST WHILE PRESERVING ACCURACY ON RESULTS

Low cost per timestep
BUT more timesteps



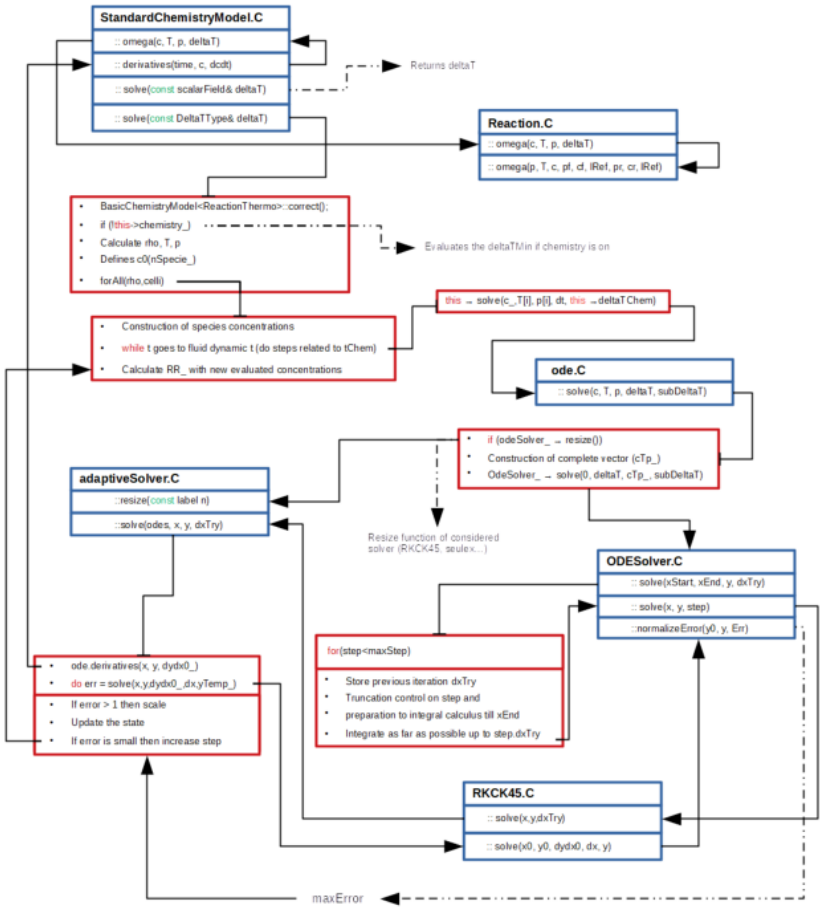
High cost per timestep
BUT less timesteps

Parallelization plays an important role too!

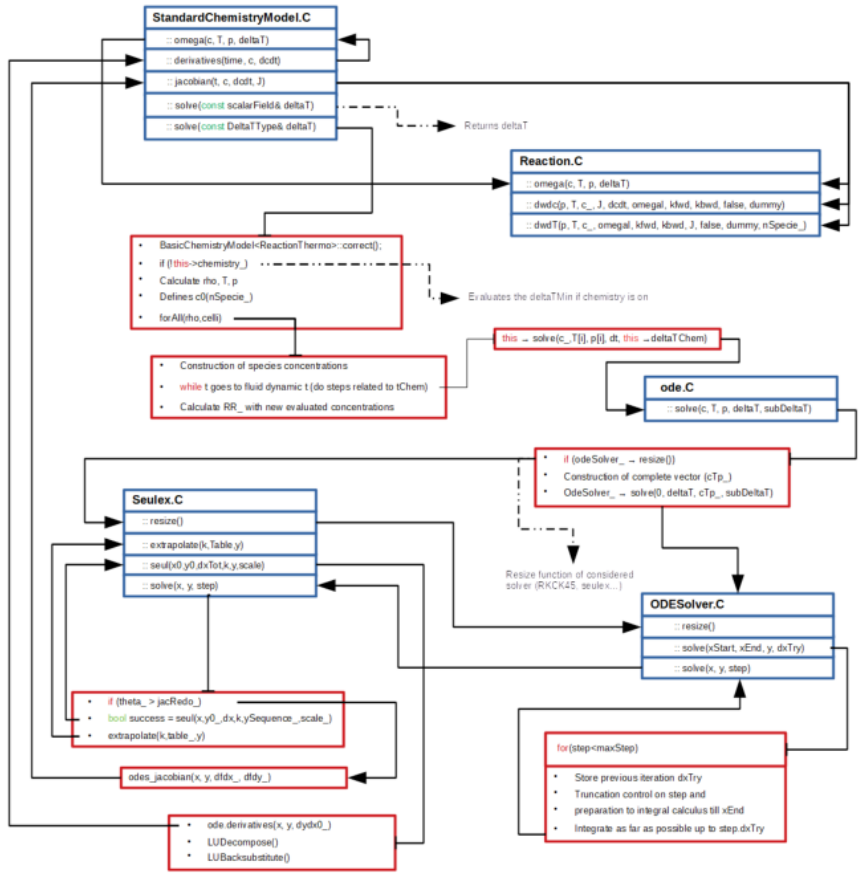
Which method is the best?

DON'T ASSUME that **Explicit "*simpler mathematics*"** means **"*simpler implementation*"**

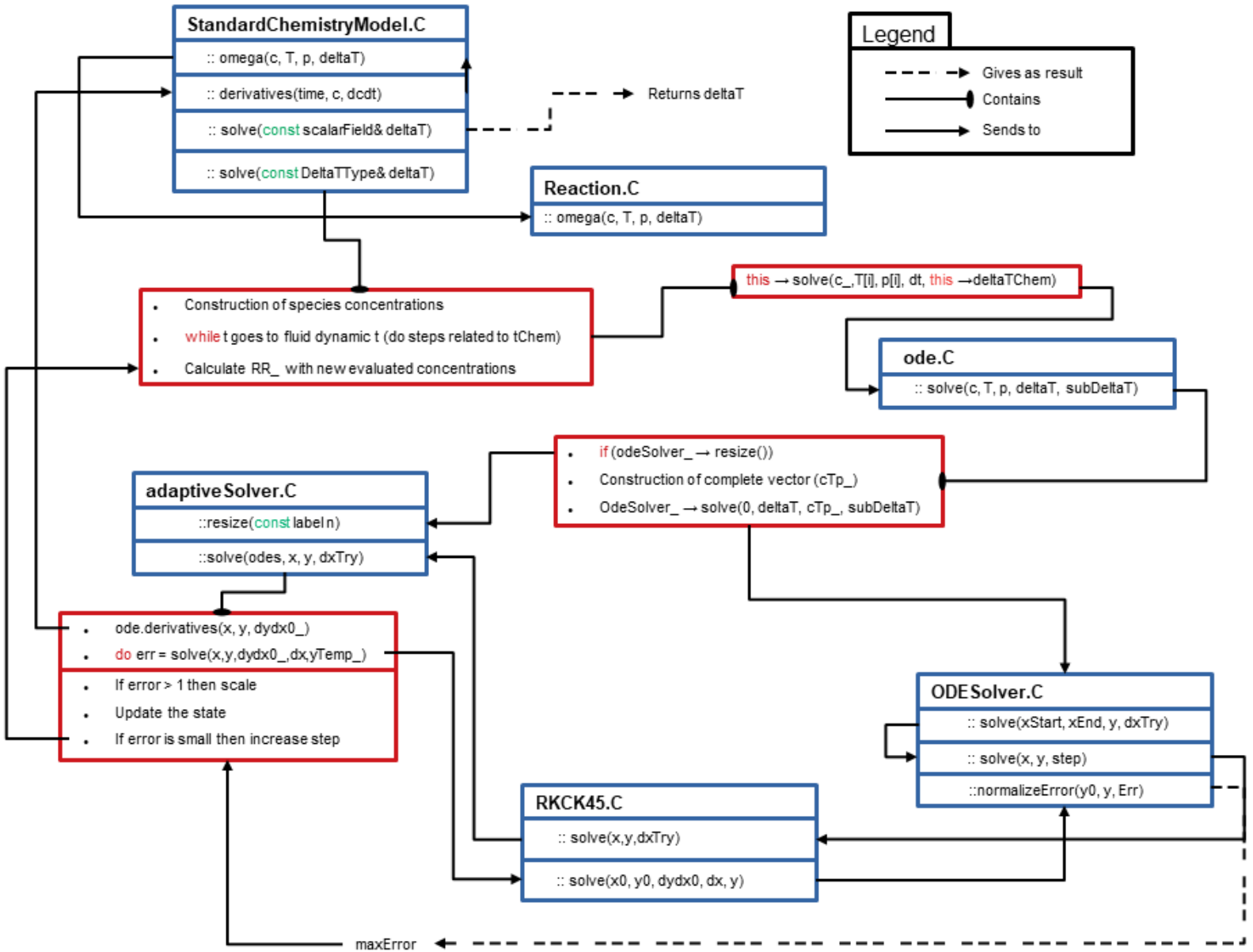
EXPLICIT CHEMISTRY TREATMENT in OpenFOAM



IMPLICIT CHEMISTRY TREATMENT in OpenFOAM



Practical solution in detail: EXPLICIT



Chemistry ODEs treatment - 1

Expected output type **Name of container** **Name of function**

```
template<class ReactionThermo, class ThermoType>
Foam::scalar Foam::StandardChemistryModel<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
    );
}
```

input

This function is asking for the return of a chemical sub-timestep.
That must result from a MINIMUM operation between two values:

- the **SECOND** one is the double of the previous chemical sub-timestep (imposing in this way a limitation on the increase of chemical sub-timestep so as to avoid divergence);
- the **FIRST** one comes out from a **SOLVE** function;

Chemistry ODEs treatment - 2

The previous function called for a **SOLVE** that is contained in the same file:
`StandardChemistryModel.C`

```
template<class ReactionThermo, class ThermoType>
template<class DeltaTType>
Foam::scalar Foam::StandardChemistryModel<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_);
```

It retuns a scalar (*timeStep*)!

`deltaTMin` is initialized and posed equal to a very big value

Then, if chemistry is active, the chemical sub-timestep is acquired

Thermodynamic values are extracted from the cells: `this->thermo().x()` copies the values "`x`". Also, a vector (`scalarField`) of INITIAL concentrations `c0` is created, with dimension equal to `nSpecie`

Chemistry ODEs treatment - 3

Followingly, a **`forAll`** cycle starts, which imposes the repetition of the operations **FOR ONE CELL AFTER THE OTHER**, this means a **SERIAL PROCESS**, which is one of the two principal causes of huge increase in computational time

```
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();
            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;
        }
```

Serial process is hardly avoidable, as the "`if controls`" are implemented in a specifical serial manner

Concentrations are calculated and a copy is stored for future operations of differences!

While the chemical sub-timestep is greater than a value which avoids underflow, perform the sub-iteration.
Perform it as long as the subsequent FLUID DYNAMIC TIMESTEP is reached

Chemistry ODEs treatment - 4

The previous **SOLVE** function sends to **`ode.C`**

```
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);

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

A new compact vector is constructed, so that the first elemets are the concentrations of nSpecies, followed by Temperature and pressure

Another SOLVE func is called

At the end of the solve the concentrations will be updated as much as Temperature and pressure

Chemistry ODEs treatment - 5

WE ARE IN `ODESolver` NOW

```
void Foam::ODESolver::solve
(
    const scalar xStart,
    const scalar xEnd,
    scalarField& y,
    scalar& dxTry
) const
{
    stepState step(dxTry);
    scalar x = xStart;

    for (label nStep=0; nStep<maxSteps_; nStep++)
    {
        // Store previous iteration dxTry
        scalar dxTry0 = step.dxTry;

        step.reject = false;

        // Check if this is a truncated step and set dxTry to integrate to xEnd
        if ((x + step.dxTry - xEnd)*(x + step.dxTry - xStart) > 0)
        {
            step.last = true;
            step.dxTry = xEnd - x;
        }

        // Integrate as far as possible up to step.dxTry
        solve(x, y, step);

        // Check if reached xEnd
        if ((x - xEnd)*(xEnd - xStart) >= 0)
        {
            if (nStep > 0 && step.last)
            {
                step.dxTry = dxTry0;
            }

            dxTry = step.dxTry;

            return;
        }
```

"*Number of steps cycle*"
Iteration must continue as long as convergence is not obtained OR maximum number of steps is produced

Another `SOLVE` function!
This one sends to a function in this very same file.

Chemistry ODEs treatment - 5bis

If the convergence is NOT obtained, than solution must be discarded, sub-timestep must be reduced and calculation performed AGAIN

```
    step.first = false;

    // If the step.dxTry was reject set step.prevReject
    if (step.reject)
    {
        step.prevReject = true;
    }
}

FatalErrorInFunction
    << "Integration steps greater than maximum " << maxSteps_ << nl
    << "    xStart = " << xStart << ", xEnd = " << xEnd
    << ", x = " << x << ", dxDid = " << step.dxDid << nl
    << "    y = " << y
    << exit(FatalError);
```

Chemistry ODEs treatment: explicit way

```
void Foam::ODESolver::solve
(
    scalar& x,
    scalarField& y,
    stepState& step
) const
{
    scalar x0 = x;
    solve(x, y, step.dxTry);
    step.dxDid = x - x0;
}
```

The previous `SOLVE` sent to another function contained in `ODESolver.C` **WHICH AGAIN HAS A DIFFERENT SOLVE FUNCTION**

Up to now, no solution has yet been produced for the chemical subtimestep, but the controls on activation and convergence have been posed for the SERIAL PROCESS. At this point if an EXPLICIT SOLVER is selected, the solution must be calculated.

```
void Foam::RKCK45::solve
(
    scalar& x,
    scalarField& y,
    scalar& dxTry
) const
{
    adaptiveSolver::solve(odes, x, y, dxTry);
}
```

Upper `solve` function sends to the **RUNGE-KUTTA CASH KARP (RKCK45)** file

This one instead sends to a different file called `adaptiveSolver.C`

RKCK45: explicit way to solution - 1

```
void Foam::adaptiveSolver::solve
(
    const ODESystem& odes,
    scalar& x,
    scalarField& y,
    scalar& dxTry
) const
{
    scalar dx = dxTry;
    scalar err = 0.0;

    odes.derivatives(x, y, dydx0_);

    // Loop over solver and adjust step-size as necessary
    // to achieve desired error
    do
    {
        // Solve step and provide error estimate
        err = solve(x, y, dydx0_, dx, yTemp_);

        // If error is large reduce dx
        if (err > 1)
        {
            scalar scale = max(safeScale_*pow(err, -alphaDec_), minScale_);
            dx *= scale;

            if (dx < vSmall)
            {
                FatalErrorInFunction
                            << "stepsize underflow"
                            << exit(FatalError);
            }
        }
    } while (err > 1);
```

!!!

This particular construction allows to perform the operations at least once in a *"forced manner"* and then to repeat the process as long as the convergence is not obtained

This is the **FINAL SOLVE step** which **sends back to RKCK45**

If the error is small, the subtimestep has to de-crease! Solution must be discarded and calculation is performed once again

RKCK45: explicit way to solution - 1bis

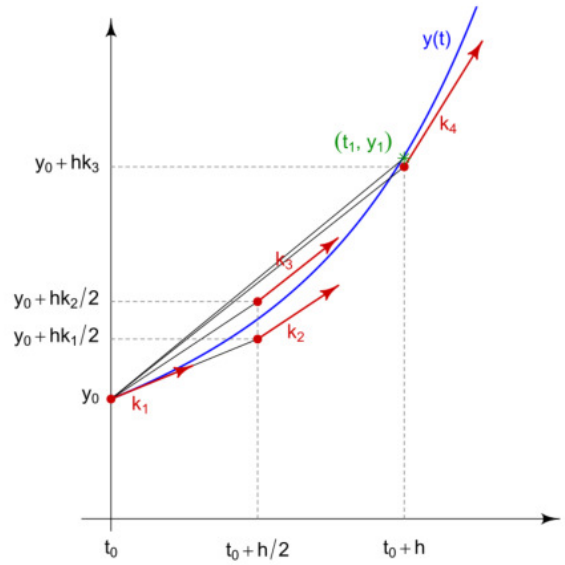
```
    // Update the state
    x += dx;
    y = yTemp_;

    // If the error is small increase the step-size
    if (err > pow(maxScale_/safeScale_, -1.0/alphaInc_))
    {
        dxTry =
            min(max(safeScale_*pow(err, -alphaInc_), minScale_), maxScale_)*dx;
    }
    else
    {
        dxTry = safeScale_*maxScale_*dx;
    }
}
```

Also, mind that the timestep can't become too small!
Underflow must be avoided!

RKCK45: explicit way to solution - 2

```
Foam::scalar Foam::RKCK45::solve
(
    const scalar x0,
    const scalarField& y0,
    const scalarField& dydx0,
    const scalar dx,
    scalarField& y
) const
{
    forAll(yTemp_, i)
    {
        yTemp_[i] = y0[i] + a21*dx*dydx0[i];
    }

    odes_.derivatives(x0 + c2*dx, yTemp_, k2_

    forAll(yTemp_, i)
    {
        yTemp_[i] = y0[i] + dx*(a31*dydx0[i]
    }

    odes_.derivatives(x0 + c3*dx, yTemp_, k3_

    forAll(yTemp_, i)
    {
        yTemp_[i] = y0[i] + dx*(a41*dydx0[i]
    }

    odes_.derivatives(x0 + c4*dx, yTemp_, k4_);

    forAll(yTemp_, i)
    {
        yTemp_[i] = y0[i]
```

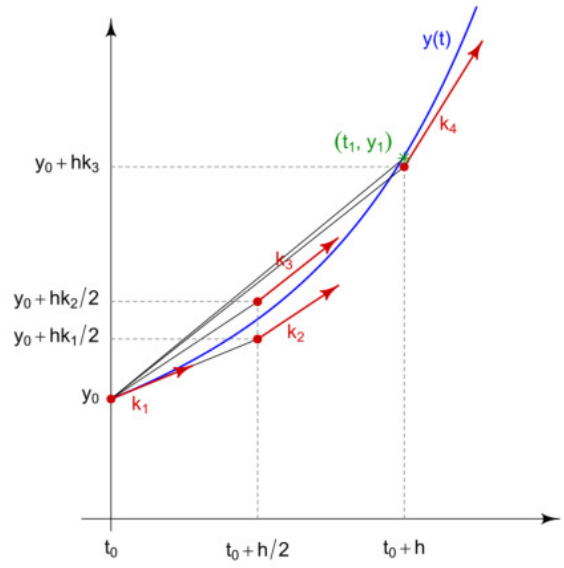
This solve function produces the common Runge Kutta 45 (Cash-Karp) operations, by means of a temporary `yTemp_` that gets updated between two contiguous chemical subtimesteps



RKCK45: explicit way to solution - 2bis

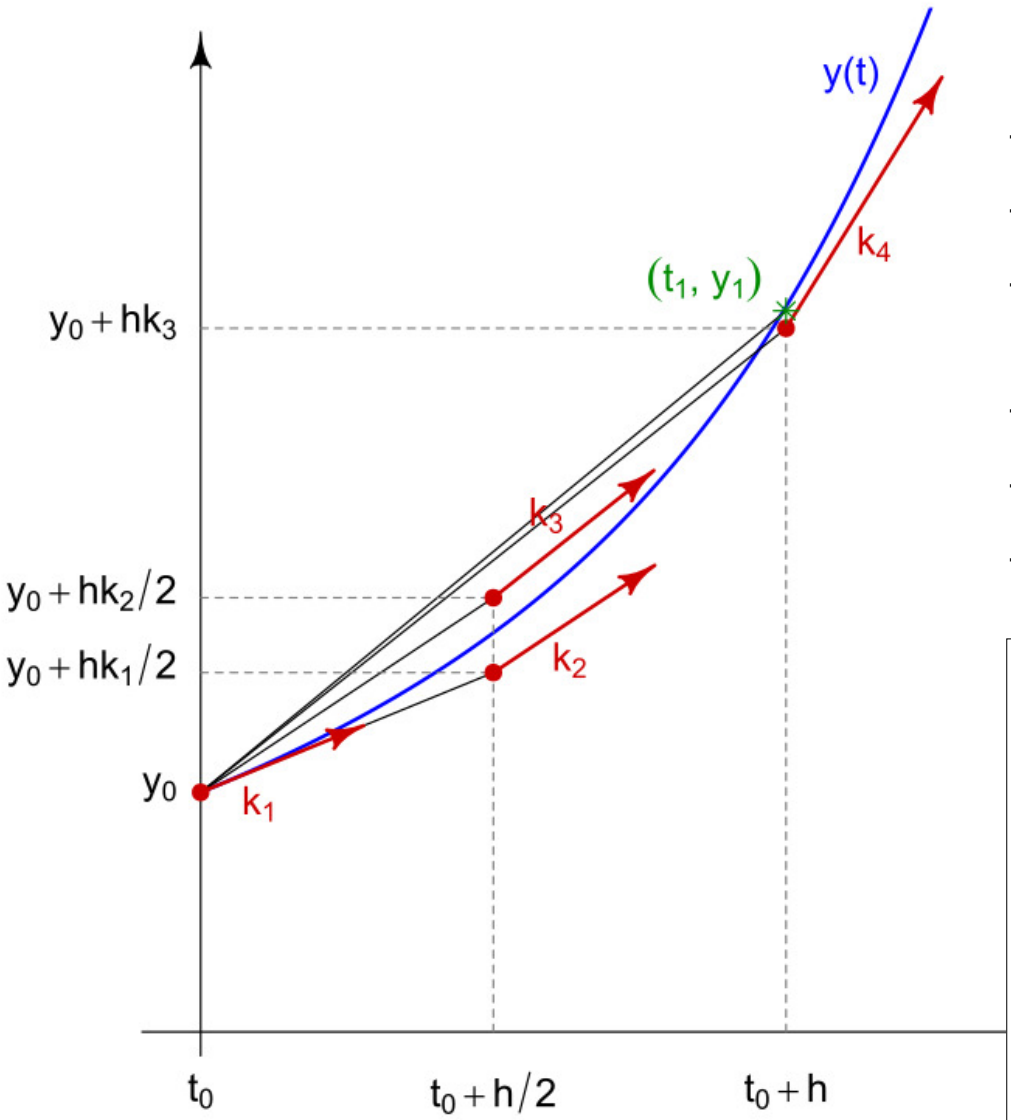
```
    odes_.derivatives(x0 + c5*dx, yTemp_, k5_

    forAll(yTemp_, i)
    {
        yTemp_[i] = y0[i]
          + dx
           *(a61*dydx0[i] + a62*k2_[i] + a63*
    }

    odes_.derivatives(x0 + c6*dx, yTemp_, k6_

    forAll(y, i)
    {
        y[i] = y0[i]
          + dx*(b1*dydx0[i] + b3*k3_[i] + b4*
    }

    forAll(err_, i)
    {
        err_[i] =
            dx
           *(e1*dydx0[i] + e3*k3_[i] + e4*k4_
    }

    return normalizeError(y0, y, err_);
}
```

This solve function produces the common Runge Kutta 45 (Cash-Karp) operations, by means of a temporary `yTemp_` that gets updated between two contiguous chemical subtimesteps



A final normalization of the error must be produced on the list of error calculated for all the species plus T and p! This operation is contained back in `ODESolver.C`

RKCK45: integration over time



Regarding integration, inside the code one has to consider

- $x = t_{init}$ (starting point of chemical sub-timestep);
- $xEnd = t_{end}$ (end point of chemical sub-timestep);
- $y = [c]$ (concentration of species OR Temp OR pressure);
- $h = \Delta t$ (of chemical sub-timestep);
- $dcdt$: final result of integration;
- k : slopes;

With respect to this figure:

- $x = t0$;
- $xEnd = t0 + h$;
- $y = y0$;
- $h = h$;
- $dcdt = y1$;
- $k = k_i$;

RKCK45: odes.derivatives()

```
template<class ReactionThermo, class ThermoType>
void Foam::StandardChemistryModel<ReactionThermo, ThermoType>::derivatives
(
    const scalar time,
    const scalarField& c,
    scalarField& dcdt
) const
{
    const scalar T = c[nSpecie_];
    const scalar p = c[nSpecie_ + 1];

    forAll(c_, i)
    {
        c_[i] = max(c[i], 0);
    }

    omega(c_, T, p, dcdt);

    // Constant pressure
    // dT/dt = ...
    scalar rho = 0;
    scalar cSum = 0;
    for (label i = 0; i < nSpecie ; i++)
    {
        const scalar W = specieThermo_[i].W();
        cSum += c_[i];
        rho += W*c_[i];
    }
    ...

    dcdt[nSpecie_] = -dT;

    // dp/dt = ...
    dcdt[nSpecie_ + 1] = 0;
}
```

Here are calculated the slopes which are necessary for `RKCK45`, for ALL CHEMICAL SPECIES, TEMPERATURE and PRESSURE

Remember that concentrations must be always greater than or equal to 0!

This command provides the calculation of ω by means of a complex function contained in this same file

All the informations requested for molecular weight, c_p, and other parameters **are STORED at the BEGINNING of the simulation, as `OpenFOAM` READS and PROCESSES** what is contained in the aforementioned **CHEMKIN FILES**

Mind the **ISOBARIC process**: dc/dt of the last element (ie pressure dp/dt) is equal to 0!

RKCK45: normalizeError()

```
Foam::scalar Foam::RKCK45::solve
(
    const scalar x0,
    const scalarField& y0,
    const scalarField& dydx0,
    const scalar dx,
    scalarField& y
) const
{
    ...

    return normalizeError(y0, y, err_);
}
```

↓

```
Foam::scalar Foam::ODESolver::normalizeError
(
    const scalarField& y0,
    const scalarField& y,
    const scalarField& err
) const
{
    // Calculate the maximum error
    scalar maxErr = 0.0;
    forAll(err, i)
    {
        scalar tol = absTol_[i] + relTol_[i]*max(mag(y0[i]), mag(y[i]));
        maxErr = max(maxErr, mag(err[i])/tol);
    }

    return maxErr;
}
```

Final step in `RKCK45` solve function is the calculatio of the normalized error To do so, a function in `ODESolver.C` is invoked

"`maxErr`" is computed and returned in `RKCK45`

Chemistry ODEs treatment - end

```
            while (timeLeft > small)
            {
                scalar dt = timeLeft;
                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++)
            {
                RR_[i][celli] =
                    (c_[i] - c0[i])*specieThermo_[i].W()/deltaT[celli];
            }
        }
        else
        {
            for (label i=0; i<nSpecie_; i++)
            {
                RR_[i][celli] = 0;
            }
        }
    }

    return deltaTMin;
}
```

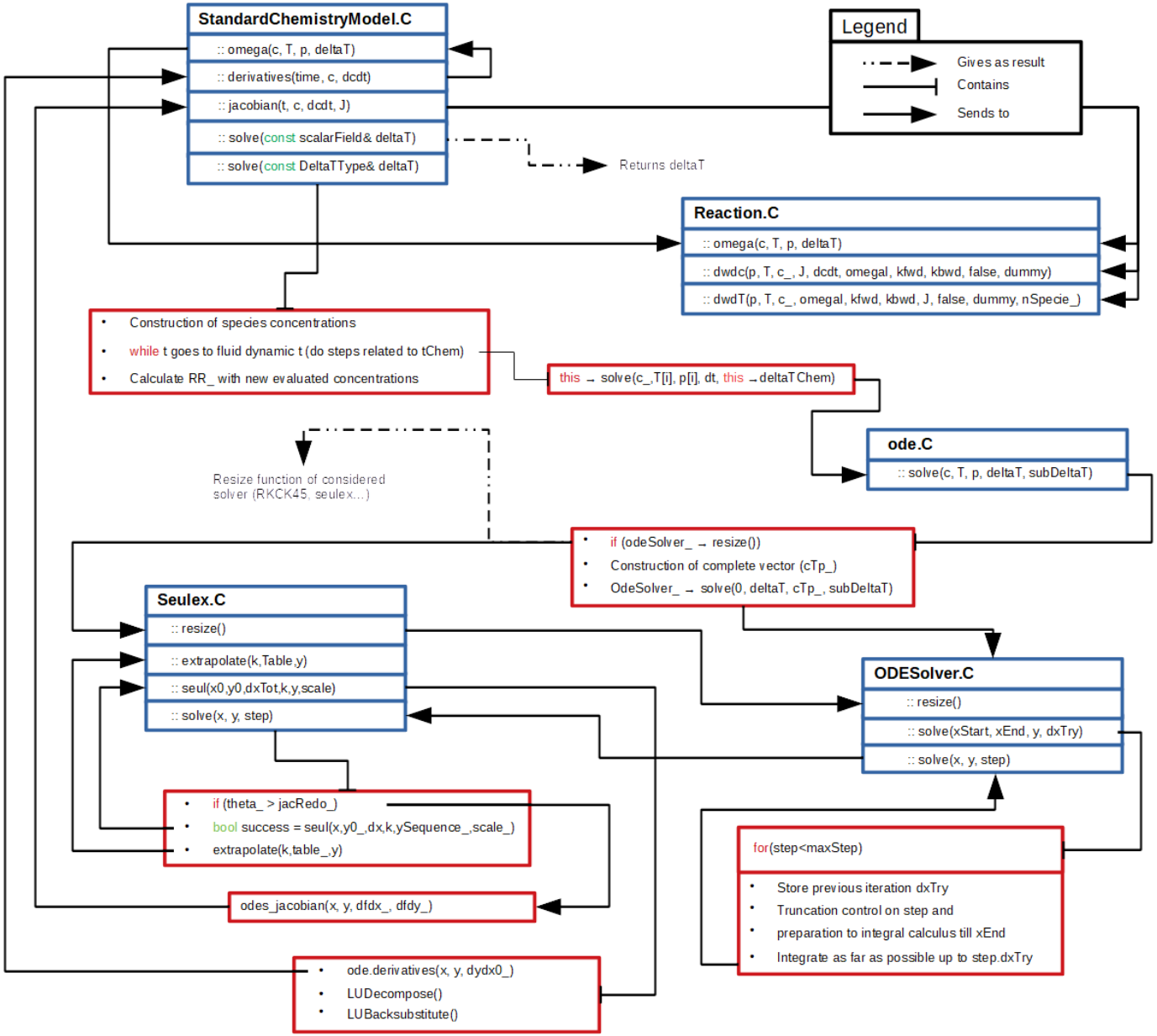
Now a solution is obtained, for ALL CHEMICAL SUBTIMESTEPS so as to reach the next fluid dynamic timestep, and FOR THE SPECIFIC SINGLE CELL

deltaTMin is updated in order to obtain the requested scalar at the very beginning of the process

The Reaction Rate is updated for EVERY SINGLE REACTING CELL...

... because if the the cell is NON-REACTIVE, all the process is irrelevant, and the Reaction Rate is equal to 0!

Practical solution in detail: IMPLICIT



Chemistry ODEs treatment: implicit way

```
void Foam::ODESolver::solve
(
    scalar& x,
    scalarField& y,
    stepState& step
) const
{
    scalar x0 = x;
    solve(x, y, step.dxTry);
    step.dxDid = x - x0;
}
```

The previous SOLVE sent to another function contained in `ODESolver.C` **WHICH AGAIN HAS A DIFFERENT SOLVE FUNCTION**

Up to here, what was presented above remains valid.
Process remains SERIAL.
At this point if an IMPLICIT SOLVER is selected, the solution must be calculated.

The upper `solve` function sends to this one from the `seulex` solver

```
void Foam::seulex::solve
(
    scalar& x,
    scalarField& y,
    stepState& step
) const
{
    temp_[0] = great;
    scalar dx = step.dxTry;
    y0_ = y;
    dxOpt_[0] = mag(0.1*dx);

    if (step.first || step.prevReject)
    {
        theta_ = 2*jacRedo_;
    }

    if (step.first)
    {
```

This function is much longer and more complex than the explicit counterparts (ex. `RKCK45`)

SEULEX: extrapolation along time

Seulex is an extrapolation-algorithm based on linear implicit Euler schemes.

Extrapolation methods for ODEs solutions usually exploit **Richardson extrapolation**, that here can be seen in its general formulation.

Let's consider for the numerical solution $z(t,h)$ as the numerical approximation of $y(t)$ at time time t, obtained using the step size h. If $t_n = nh$, then $y_n = z(t_n, h)$.
Suppose the global error of numerical method has an expansion in powers of h so that

$$z(t,h) = y(t) + \sum_{i=1}^{q} c_i(t) h^i + o(h^{q+1})$$

The idea of extrapolation allows to combine the solution of the form $z(t,h_j)$ considering different timesteps, so that successive terms of the error expansion are eliminated, resulting, thus, in a higher-order approximation.

Assume for a simpler notation that Y^* must be approximated, and that the relationship $Y(h)$, ie $z(t,h)$, is valid. In this way

$$Y(h) = Y^* + c_1 h^p + o(h^p)$$

SEULEX: extrapolation over time

The equation can be modified with a parameter k of amplification

$$Y(kh) = Y^* + c_1 k^p h^p + o(h^p)$$

Whenever $k < 1$ the initial method is improved and order of convergence is increased

Indeed, provided the knowledge of the order of $Y(h)$ defined as "p", then **the h^p term can be cancelled out by considering a linear combination of $Y(h)$ and $Y(kh)$**

$$Y(kh) - k^p Y(h) = (1 - k^p)Y^* + o(h^p)$$

This results to be the simplest way, but unfortunately the formulation **DOES NOT** converge to Y^*, so it must be provided a division for the right side parentheses

$$\boxed{\frac{Y(kh) - k^p Y(h)}{1 - k^p} = Y^* + o(h^p)}$$

This represents a better numerical method than the initial one, able to compute Y^*, which means that **this method has higher order of convergence with respect to the initial one**.

SEULEX: implicit way to solution - 1

```
void Foam::seulex::solve
(
    scalar& x,
    scalarField& y,
    stepState& step
) const
{
    temp_[0] = great;
    scalar dx = step.dxTry;
    y0_ = y;
    dxOpt_[0] = mag(0.1*dx);

    if (step.first || step.prevReject)
    {
        theta_ = 2*jacRedo_;
    }

    if (step.first)
    {
        // NOTE: the first element of relTol_ and absTol_ are used here.
        scalar logTol = -log10(relTol_[0] + absTol_[0])*0.6 + 0.5;
        kTarg_ = max(1, min(kMaxx_ - 1, int(logTol)));
    }

    forAll(scale_, i)
    {
        scale_[i] = absTol_[i] + relTol_[i]*mag(y[i]);
    }

    bool jacUpdated = false;

    if (theta_ > jacRedo_)
    {
        odes_.jacobian(x, y, dfdx_, dfdy_);
        jacUpdated = true;
    }
```

In this particular function there is a direct connection with the JACOBIAN construction! This is computationally VERY EXPENSIVE, progressively by increasing the number of species/reactions!

BUT timestep dimension increases and global computational cost might be LOWER than explicit counterparts, specially when stiff problems are accounted!

Computational advantage between IMPLICIT and EXPLICIT is PROBLEM DEPENDANT

→

```
if (theta_ > jacRedo_)
{
    odes_.jacobian(x, y, dfdx_, dfdy_);
    jacUpdated = true;
}
```

SEULEX: implicit way to solution - 2

```
temp_[0] = great;
scalar dx = step.dxTry;
y0_ = y;
dxOpt_[0] = mag(0.1*dx);

if (step.first || step.prevReject)
{
    theta_ = 2*jacRedo_;
}

if (step.first)
{
    // NOTE: the first element of relTol_ and
    scalar logTol = -log10(relTol_[0] + absTol_[0])*0.6 + 0.5;
    kTarg_ = max(1, min(kMaxx_ - 1, int(logTol)));
}

forAll(scale_, i)
{
    scale_[i] = absTol_[i] + relTol_[i]*mag(y[i]);
}

bool jacUpdated = false;

if (theta_ > jacRedo_)
{
    odes_.jacobian(x, y, dfdx_, dfdy_);
    jacUpdated = true;
}
```

Some important informations:

- The use of temporary storage is useful to avoid dynamic memory allocation between calls;
- `table` is a scalar rectangular matrix and its construction is dependant by the number of species;
- `a_` is a scalar square matrix;
- Initially dx, y, and optimal dx must be set;

`theta` is imposed by default as equal to twice the `jacRedo` which is chosen as minimum between 10^{-4} and `min(relTol)`. The code must solve this at least once at the beginning by producing "jacobian" (ie `dfdy_`)

SEULEX: implicit way to solution - 3

```
int k;
scalar dxNew = mag(dx);
bool firstk = true;

while (firstk || step.reject)
{
    dx = step.forward ? dxNew : -dxNew;
    firstk = false;
    step.reject = false;

    if (mag(dx) <= mag(x)*sqr(small))
    {
        WarningInFunction
            << "step size underflow :"  << dx << endl;
    }

    scalar errOld = 0;
```

This must be performed at least at the beginning of each fluid dynamic iteration, and whenever the solution is discarded.
Moreover, an underflow control is present!

SEULEX: implicit way to solution - 4

```
for (k=0; k<=kTarg_+1; k++)
{
    bool success = seul(x, y0_, dx, k, ySequence_, scale_);

    if (!success)
    {
        step.reject = true;
        dxNew = mag(dx)*stepFactor5_;
        break;
    }
```

`seul` is a different function, contained in this very same file `seulex.C`

If the convergence is NOT obtained, than solution must be discarded, sub-timestep must be reduced and calculation performed AGAIN

`table` is updated at least at the first iteration

```
if (k != 0)
{
    extrapolate(k, table_, y);
    scalar err = 0;
    forAll(scale_, i)
    {
        scale_[i] = absTol_[i] + relTol_[i]*mag(y0_[i]);
        err += sqr((y[i] - table_(0, i))/scale_[i]);
    }
    err = sqrt(err/n_);
```

Again, `extrapolate` is a different function, contained in this very same file `seulex.C`

Followingly, the `x` must be updated, thanks to an increment of `dx`, ie, the sub-iteration continues bewteen two subsequent fluid dynamic timesteps

SEULEX: implicit way to solution - 5

```
bool Foam::seulex::seul
(
    const scalar x0,
    const scalarField& y0,
    const scalar dxTot,
    const label k,
    scalarField& y,
    const scalarField& scale
) const
{
    label nSteps = nSeq_[k];
    scalar dx = dxTot/nSteps;

    for (label i=0; i<n_; i++)
    {
        for (label j=0; j<n_; j++)
        {
            a_(i, j) = -dfdy_(i, j);
        }

        a_(i, i) += 1/dx;
    }

    LUDecompose(a_, pivotIndices_);

    scalar xnew = x0 + dx;
    odes_.derivatives(xnew, y0, dy_);
    LUBacksubstitute(a_, pivotIndices_, dy_);

    yTemp_ = y0;
```

Construction of `dx` starts from `dxTot` which means that the `dx` passed from `seulex::solve`, here is defined as `dxTot`
BE REALLY AWARE OF THE CHANGE IN NOMENCLATURE AMONG THE FUNCTIONS

Matrix `a_` is filled with the "table" values (ie `dfdy_`)

Mind the presence of **LU Decomposition** that must be applied on the jacobian, that requires moreover backward substitution. Also derivatives must be computed using the same function seen above `LUDecompose` and `LUBacksubstitute` come from `LUSolve` inside `scalarMatrices.C` and they produce the common LU decomposition.

These operations increase the total computational cost PER TIMESTEP!
Also, parallel treatment of multiple cells at the same time becomes prohibitive!

SEULEX: implicit way to solution - 6

```
void Foam::seulex::extrapolate
(
    const label k,
    scalarRectangularMatrix& table,
    scalarField& y
) const
{
    for (int j=k-1; j>0; j--)
    {
        for (label i=0; i<n_; i++)
        {
            table[j-1][i] =
                table(j, i) + coeff_(k, j)*(table(j, i) - table[j-1][i]);
        }
    }

    for (int i=0; i<n_; i++)
    {
        y[i] = table(0, i) + coeff_(k, 0)*(table(0, i) - y[i]);
    }
}
```

Mind the presence of a backward extrapolation in the cycle!

In which «table» is directly referred to the chemical species concentrations jacobian construction, with time and chemical species concentrations

Moreover:

- Jacobian construction is produced inside the `StandardChemistryModel.C` file, as it happened for the derivatives;
- it is requested the identification of `omega`, `dwdc` and `dwdT` which are again computed thanks to the link with the `Reaction.C` file, and the construction produced at the beginning of the simulation, thanks to the CHEMKIN files;

SEULEX: odes.jacobian() - 1

```
template<class ReactionThermo, class ThermoType>
void Foam::StandardChemistryModel<ReactionThermo, ThermoType>::jacobian
(
    const scalar t,
    const scalarField& c,
    scalarField& dcdt,
    scalarSquareMatrix& J
) const
{
    const scalar T = c[nSpecie_];
    const scalar p = c[nSpecie_ + 1];

    forAll(c_, i)
    {
        c_[i] = max(c[i], 0);
    }

    J = Zero;
    dcdt = Zero;

    // To compute the species derivatives of the temperature term,
    // the enthalpies of the individual species is needed
    scalarField hi(nSpecie_);
    scalarField cpi(nSpecie_);
    for (label i = 0; i < nSpecie_; i++)
    {
        hi[i] = specieThermo_[i].ha(p, T);
        cpi[i] = specieThermo_[i].cp(p, T);
    }
    scalar omegaI = 0;
    List<label> dummy;
    forAll(reactions_, ri)
    {
        const Reaction<ThermoType>& R = reactions
        scalar kfwd, kbwd;
        R.dwdc(p, T, c_, J, dcdt, omegaI, kfwd, k
        R.dwdT(p, T, c_, omegaI, kfwd, kbwd, J, false, dummy, nSpecie_);
    }
```

Same initial construction seen in `odes.derivatives()` within `StandardChemistryModel.C`

A square `jacobian J` matrix is generated as function of nSpecie and filled with zeros, as much as the vector referred to change in concentrations (dc/dt)

c_p values as much as Hi ones (which were seen in `odes.derivatives()` as well) must be evaluated thanks to a connection with `Reaction.C` file. Also, not only `omega`, as for `odes.derivatives()`, but also `dwdc` and `dwdt` are necessary! Again, linking them to the `Reaction.C` file in which they're accounted at the BEGINNING OF THE SIMULATION

Mind that also in this case a cycle on the entire set of REACTIONS must be produced!

SEULEX: odes.jacobian() - 2

```
// The species derivatives of the temperature term are partially computed
// while computing dwdc, they are completed hereunder.
scalar cpMean = 0;
scalar dcpdTMean = 0;
for (label i=0; i<nSpecie_; i++)
{
    cpMean += c_[i]*cpi[i]; // J/(m^3 K)
    dcpdTMean += c_[i]*specieThermo_[i].dcpdT(p, T);
}
scalar dTdt = 0.0;
for (label i=0; i<nSpecie_; i++)
{
    dTdt += hi[i]*dcdt[i]; // J/(m^3 s)
}
dTdt /= -cpMean; // K/s

for (label i = 0; i < nSpecie_; i++)
{
    J(nSpecie_, i) = 0;
    for (label j = 0; j < nSpecie_; j++)
    {
        J(nSpecie_, i) += hi[j]*J(j, i);
    }
    J(nSpecie_, i) += cpi[i]*dTdt; // J/(mol s)
    J(nSpecie_, i) /= -cpMean;     // K/s/(mol/m3)
}
```

Then, the vector `dcdt` must be calculated, which means to request the computation of the mean c_p variation in the mixture

Once the mean dc_p/dt has been computed it is also possible to update `dt`, or Δt, ie the variation of temperature

In order to perform the calculation above, at least one guess of `dcdt` must be accounted! And this operation is PARTIALLY DONE while computing `dwdc` (see previous slide)

Within `dwdc` it must be provided the jacobian so that it can be updated thanks to `omega` computed as requested for `odes.derivatives()`

→ The operation takes advantage of a splitting between left-hand and right-hand sides of the chemical reaction equations

SEULEX: odes.jacobian() - 3

```
    for (label i = 0; i < nSpecie_; i++)
    {
        J(nSpecie_, i) = 0;
        for (label j = 0; j < nSpecie_; j++)
        {
            J(nSpecie_, i) += hi[j]*J(j, i);
        }
        J(nSpecie_, i) += cpi[i]*dTdt; // J/(mol s)
        J(nSpecie_, i) /= -cpMean;     // K/s/(mol/m3)
    }

    // ddT of dTdt
    J(nSpecie_, nSpecie_) = 0;
    for (label i = 0; i < nSpecie_; i++)
    {
        J(nSpecie_, nSpecie_) += cpi[i]*dcdt[i] + hi[i]*J(i, nSpecie_);
    }
    J(nSpecie_, nSpecie_) += dTdt*dcpdTMean;
    J(nSpecie_, nSpecie_) /= -cpMean;
    J(nSpecie_, nSpecie_) += dTdt/T;
}
```

The last Jacobian row, Temperature related, gets upated

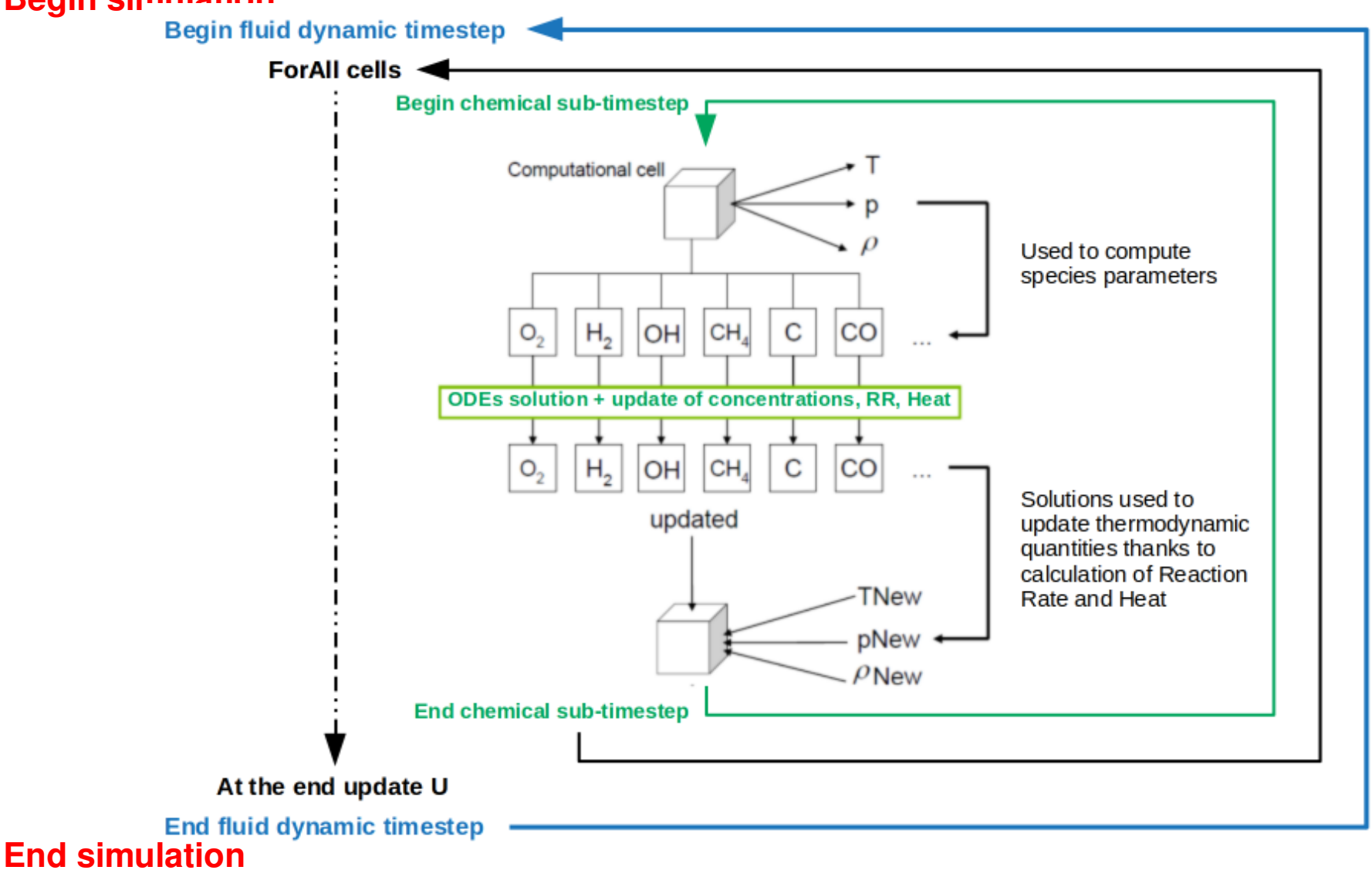
And finally the last term, ie the rate of change in temperature variation $d(\Delta T)/dt$ is updated as a function of dT/dt

Within `dwdc` it must be provided the jacobian so that it can be updated thanks to `omega` computed as requested for `odes.derivatives()`

→ The operation takes advantage of a splitting between left-hand and right-hand sides of the chemical reaction equations

More complex Simulations: multiple cells treatment

More complex cases



More complex cases

In complex cases one should not just consider high number of species and/or reactions, but multiple cells and therefore DIFFERENT SOLVERS

There are thus a lot of different possibilities, as it might be taken into account `sprayFoam`, `reactingParcelFoam`, `rhoReactingFoam`, `fireFoam`, `XiFoam`, ecc.

These different solvers have important differences and they are suitable (or much more convenient) for different situlations and simulations.

Mind that the chemistry treatment previously seen does not change

QUESTION: how does OpenFOAM treat multi-cell cases?

The `StandardChemistryModel` previously consideredtakes into account a very expensive `forAll` cells cycle, which imposes the calculation of solution for every single cell one after the other. **It is clear how the computational cost and time increase due to this loop.**

```
forAll(rho, celli)
{
    scalar Ti = T[celli];

    if (Ti > Treact_)
    {
        const scalar rhoi = rho[celli];
        scalar pi = p[celli];
```

⟵ `StandardChemistryModel.C`

reactingFoam & rhoReactingFoam

Whenever combustion with gaseous diffusive or premixed chemical reactions is considered, the two solvers that must be analyzed are `reactingFoam` and `rhoReactingFoam`.

One must be really aware that the differences are not related to treatment of "*compressible*" or "*uncompressible*" fluids, but the separation bewteen these two is more in dept and thermo-related.
Indeed, by looking at the codes, it will be seen how the thermal treatment of the two is different, and because of this, the use of these two is commonly referred to different specific reactive/heat exchange problems.

Moreover, due to the consideration of an absolute value of pressure "p" (and not a "$p_{rgh} = p - \rho g h$"), it is well known how **both solvers can treat up to transonic cases**.

No mention will be done related to the SIMPLEC treatment of the pressure equation, which requires to include a so-called "`pcEqn`", for which a consistent pressure equation gets solved:

If consistency is activated the p-U coupling requires only a little under-relaxation for velocity and other transport quantities, referred to the equations involved, while it is not requested to apply any under-relaxation for pressure. Typically, results have more robust solution and faster convergence.

psiThermo & rhoThermo: definitions

As it was pointed out, the differences are thermo-related. Indeed, reactingFoam implies a psiThermo construction/treatment of the reactions and heat exchange, while rhoReactingFoam implies rhoThermo, from which comes its name.

The two ways are different, and in order to analyze them some nomenclature is due:

- *Type*: the type of a model is a function of the way is used to compute thermal variables. For us it will be related here to psiThermo or rhoThermo

  PsiThermo: thermophysical model for a mixture with fixed chemical composition, which accounts for compressibility PSI ($\psi = 1/RT$). Equations are compressibility related, as well as on pressure changes. Density can be calculated from closure equation (ie, for example the equation of state: $\rho = p/RT = \psi p$) . Usually this is proven to be more suitable for combustion complex problems;

  RhoThermo: thermophysical model for mixture with fixed chemical composition, which accounts for density RHO (ρ) to calculate the basic thermodynamic quantities. Usually this results to be more suitable for heat transfer un/compressible problems, such as heat exchangers;

Transport & Thermodynamic Models

And the nomenclature is complete by considering different *transport models*

const: it considers a constant Prandtl and constant dynamic viscosity;

Sutherland: dynamic viscosity is varying depending on temperature thanks to the relation $\mu = \frac{(A_s\sqrt{T})}{1+\frac{T_s}{T}}$ in which A_s is Sutherland coefficient and T_s is the Sutherland temperature;

As well as *thermodynamic models*

hConst: it considers a constant c_p value and a fusion heat H_f;

eConst: it does not consider a constant c_p, but a constant c_v instead, and again a fusion heat H_f;

janaf: it provides a custom relation for the c_p calculation, as function of temperature by means of coefficient sets, starting from a Janaf thermodynamic table. Typically, the function is evaluated thanks to two sets, one from T_{high} to T_{com}, and one from T_{com} to T_{low}

$$c_p = R(a_0 + a_1 T + a_2 T^2 + a_3 T^3 + a_4 T^4)$$

polynomial: option available in order to compute c_p value as dependant on temperature in polynomial way

$$c_p = \sum_{i=0}^{N} a_i T^i$$

reactingFoam: 1/3

```
turbulence->validate();

if (!LTS)
{
    #include "compressibleCourantNo.H"
    #include "setInitialDeltaT.H"
}

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

Info<< "\nStarting time loop\n" << endl;

while (runTime.run())
{
    #include "readTimeControls.H"

    if (LTS)
    {
        #include "setRDeltaT.H"
    }
    else
    {
        #include "compressibleCourantNo.H"
        #include "setDeltaT.H"
    }

    runTime++;
```

`LTS` stands for Local Time Stepping: LTS can be used to accelerate the convergence towards a steady solution, and the steady state is obtained when the difference in values of two successive solutions is small. One way to exploit "*local*" time stepping for transient cases is to use inner sub-iterations. For each global timestep (identical for every cell in the mesh) sub-iterations are performed to eliminate factorization errors. The inner iterations are commonly performed in pseudo-time, which accelerates convergence to the pseudo-time steady state for the current global timestep.

SIMULATION BEGINS, and after the inclusion of timestep evaluation and reaction time, the iteration occurs

Within this file there is a maximum change in terms of Temperature per iteration (by default $0.05K$) and a maximum change in cell concentration per iteration (default 1,0)

reactingFoam: 2/3

```
Info<< "Time = " << runTime.timeName() << nl << endl;

#include "rhoEqn.H"

while (pimple.loop())
{
    #include "UEqn.H"
    #include "YEqn.H"
    #include "EEqn.H"

    // --- Pressure corrector loop
    while (pimple.correct())
    {
        if (pimple.consistent())
        {
            #include "pcEqn.H"
        }
        else
        {
            #include "pEqn.H"
        }
    }

    if (pimple.turbCorr())
    {
        turbulence->correct();
    }
}

rho = thermo.rho();
```

This first include is an initial set for density transport, it behaves as a mass conservation, and sets an initial density

$$\frac{d\rho}{dt} + \nabla \cdot \phi = fvOptions(\rho)$$

A DETAIL ON `YEqn`:
For every substance, IF reactive (ie not inert or absent), the calculation occurs

$$\frac{d\rho Y_i}{dt} + mvConvective() - \nabla^2(\mu_{eff} Y_i) = RR(Y_i) + fvOpts(\rho, Y_i)$$

And for the final INERT substance the calculation

$$Y_{inert} = 1 - Y_{tot}$$

is performed, which considers $Y_{tot} = \sum Y_{react,i}$

LET'S LOOK AT THE `pEqn` IN DETAIL

reactingFoam: pEqn - 1

This is the 1st time this function is called in the solver: it pulls the ρ value from thermo, and makes a copy of it. ρ-field is computed from P and T field terms

```
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())
{
```

`pEqn` can treat transonic conditions

...

```
    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();

        if (pimple.finalNonOrthogonalIter())
        {
            phi == pEqn.flux();
        }
    }
}
```

BE REALLY CAREFUL:
here PSI is INSIDE the derivations!
Thus, PSI is NOT considered as constant

$$\frac{d(\psi p)}{dt} + \nabla \cdot [interp(\psi * HbyA)p] - \nabla^2 \left(\rho \frac{1}{UEqn.A()} p\right) = fvOptions(\psi, p, \rho)$$

reactingFoam: pEqn - 2

If transonic is NOT selected

```
while (pimple.correctNonOrthogonal())
{
    fvScalarMatrix pEqn
    (
        fvm::ddt(psi, p)
      + fvc::div(phiHbyA)
      - fvm::laplacian(rhorAUf, p)
     ==
        fvOptions(psi, p, rho.name())
    );

    pEqn.solve();

    if (pimple.finalNonOrthogonalIter())
    {
        phi = phiHbyA + pEqn.flux();
    }
}
}

#include "rhoEqn.H"
```

In this case the pEqn is simpler, but AGAIN note how the PSI value is not constant, and must be included in the derivation and its variation must be accounted!

$$\frac{d(\psi p)}{dt} + \nabla \cdot (\rho * HbyA) - \nabla^2 \left(\rho \frac{1}{UEqn.A()} p \right) = fvOptions(\psi, p, \rho)$$

Once pEqn is solved, the pressure value changes, PHI is updated, thus, the density value ρ must have changed! The rhoEqn guarantees that the new ρ is obtained correctly.

```
U = HbyA - rAU*fvc::grad(p);
U.correctBoundaryConditions();
fvOptions.correct(U);
K = 0.5*magSqr(U);

if (pressureControl.limit(p))
{
    p.correctBoundaryConditions();
    rho = thermo.rho();
}

if (thermo.dpdt())
{
    dpdt = fvc::ddt(p);
}
```

Finally, observe that is pressure is "*limited*" (by fvOptions) then RHO is called once again.
BUT WHAT IF P IS NOT LIMITED?

reactingFoam: 3/3

```
        while (pimple.correct())
        {
            if (pimple.consistent())
            {
                #include "pcEqn.H"
            }
            else
            {
                #include "pEqn.H"
            }
        }

        if (pimple.turbCorr())
        {
            turbulence->correct();
        }
    }

    rho = thermo.rho();

    runTime.write();

    Info<< "ExecutionTime = " << runTime.elapsedCpuTime() << " s"
        << "  ClockTime = " << runTime.elapsedClockTime() << " s"
        << nl << endl;
}

Info<< "End\n" << endl;

return 0;
}
```

Finally, observe that pressure is "*limited*" (by `fvOptions`) then RHO is called once again.

After the correction of the turbulence parameters, the same usual RHO function is called!

The question is
WHAT ARE THE DIFFERENCES WITH `rhoReactingFoam`?

rhoReactingFoam: what differs

In rhoReactingFoam is lacking an initial include of the rhoEqn before the PIMPLE loop.

While both (rhoReactingFoam and reactingFoam) call for a thermo.rho() in order to extract the thermo-value of the density, **the pEqn in which this happens are different for the two solvers!** Indeed, rhoReactingFoam uses the pEqn from the rhoPimpleFoam solver, while reactingFoam uses the one previously seen.

Furthermore, in rhoReactingFoam the evolution of "phiHbyA" is computed, which depends on the interpolation of the density by the flux "HbyA", so as to subsequently update the boundary conditions as a function of flux consistency enforcement.

While both have transonic treatment, and take into consideration a pressure "p" evaluation and not a $p-\rho gh$, the way to calculate transonic effects is different as the two pEqn files are different. The same is valid for the nonOrthogonality treatment.

BUT, the most relevant difference is related to the presence of a function that must be analized in detail: thermo.correctRho(), which provides the update of the thermo-dynamic quantities by a density based approach.

This latter function is present in rhoReactingFoam, but is absent in reactingFoam, as in a psiThermo-based solver this function has NO effect.

rhoReactingFoam: what differs

While in a psiThermo approach the function thermo.correctRho() is a completely empty function, ie it does NOTHING, in a rhoThermo approach this function provides for a change in density computation. The new density is defined as increased by a variation, which is $\Delta\rho$, that comes from the values contained in the parenthesis.

So, while for psiThermo, a function correctRho() is empty, for rhoThermo, the same function states

$$\rho_{new} = \rho_{old} + \Delta\rho$$

For which $\Delta\rho$ is defined from what is contained within the parenthesis, that for the pEqn under consideration is thermo.correctRho(psip-psip0), ie $(\psi p - \psi p_{old})$, in which ψ is considered constant before correction, p is computed, while p_{old} is stored at the beginning of the pEqn.

Indeed, $\psi p - \psi p_{old} = \frac{p}{RT} - \frac{p_{old}}{RT} = \rho - \rho_{old} = \Delta\rho$

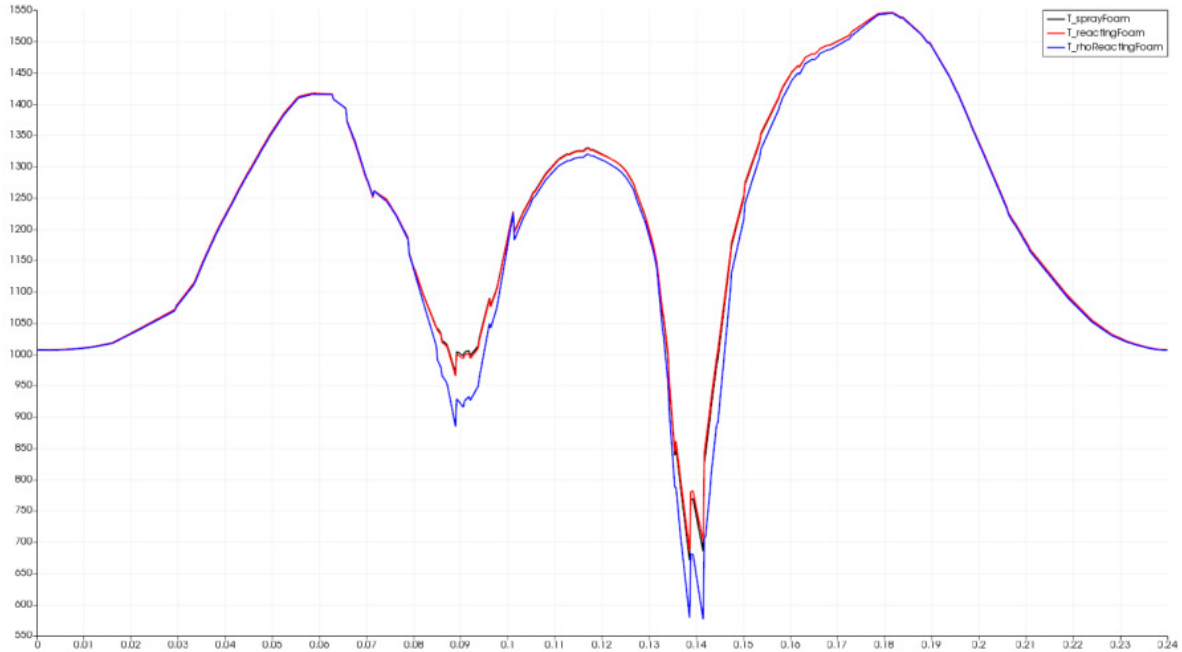
THIS CLEARLY SHOWS THE PRINCIPAL DIFFERENCES BETWEEN THE TWO SOLVERS AND THERMO MODELS

One can look up for the function correctRho() in the
src/thermophysicalModels/basic/psiThermo.H
src/thermophysicalModels/basic/psiThermo.H

rhoReactingFoam: thermo.correctRho()

By looking at the codes, and in particular at the `rhoReactingFoam pEqn`, which comes from the `rhoPimpleFoam` solver, it will be clear how the compressibility ψ is out of the derivation within the equations, thus preserving it as constant, during the iteration, while changing the pressure, in order to subsequently use the updated pressure to calculate the change in density, and in this way modify ψ for the next iteration.

While this procedure DOES NOT provide ANY difference with respect to the `psiThermo` one WHENEVER COMPRESSIBLE TRANSONIC or NON TRANSONIC - **NON REACTIVE** - flow are considered, a change is PROVIDED to occur when REACTIVE FLOWS are considered.



rhoReactingFoam: 1/3

```
if (!LTS)
{
    #include "compressibleCourantNo.H"
    #include "setInitialDeltaT.H"
}
```

LTS considerations as done for `reactingFoam`. Mind that the presence of dynamic variable files are due to intrinsic implementation of dynamic mesh treatment, but the same modifications can be done to the `reactingFoam` solver files and are out of the scope

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

Info<< "\nStarting time loop\n" << endl;

while (runTime.run())
{
    #include "readDyMControls.H"

    // Store divrhoU from the previous mesh so that it can be mapped
    // and used in correctPhi to ensure the corrected phi has the
    // same divergence
    autoPtr<volScalarField> divrhoU;
    if (correctPhi)
    {
        divrhoU = new volScalarField
        (
            "divrhoU",
            fvc::div(fvc::absolute(phi, rho, U))
        );
    }

    if (LTS)
    {
        #include "setRDeltaT.H"
    }
    else
    {
        #include "compressibleCourantNo.H"
        #include "setDeltaT.H"
    }
```

This correction of fluxes and storage of informations is again due to the dynamic mesh default treatment and are not important for the study of the solver

Up to now, no substantial differences are present between `reactingFoam` and `rhoReactingFoam`

rhoReactingFoam: 2/3

```
runTime++;

Info<< "Time = " << runTime.timeName() << nl << endl;

    ...

if (pimple.firstPimpleIter() && !pimple.simpleRho())
{
    #include "rhoEqn.H"
}

#include "UEqn.H"
#include "YEqn.H"
#include "EEqn.H"

// --- Pressure corrector loop
while (pimple.correct())
{
    if (pimple.consistent())
    {
        #include "../../../compressible/rhoPimpleFoam/pcEqn.H"
    }
    else
    {
        #include "../../../compressible/rhoPimpleFoam/pEqn.H"
    }
}

if (pimple.turbCorr())
{
    turbulence->correct();
}
```

← **# include "rhoEqn" is missing!**

The effect is anyway very limited as the update of the density is done by `correctRho()` as a function of pressure change and compressibility. And the initialization of the density is anyway done in the 2nd red box.

No substantial changes are present within these files. Small differences related to mesh motion treatment, which is out of the scope

This is much more relevant! The pressure equations (consistent for `SIMPLEC` or not, ie for `SIMPLE`) come from the `rhoPimpleFoam` solver! And the `pEqn` contains the most important differences with `reactingFoam`, so they must be looked carefully!

rhoReactingFoam: pEqn - 1

```
if (!pimple.simpleRho())
{
    rho = thermo.rho();
}

// Thermodynamic density needs to be updated by psi*d(p) after the
// pressure solution
const volScalarField psip0(psi*p);

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();
}

surfaceScalarField phiHbyA
(
    "phiHbyA",
    fvc::interpolate(rho)*fvc::flux(HbyA)
  + MRF.zeroFilter(rhorAUf*fvc::ddtCorr(rho, U, phi, rhoUf))
);

fvc::makeRelative(phiHbyA, rho, U);
MRF.makeRelative(fvc::interpolate(rho), phiHbyA);

// Update the pressure BCs to ensure flux consistency
constrainPressure(p, rho, U, phiHbyA, rhorAUf, MRF);
```

As explained, the compressibility ψ is considered as constant in the ieration, while pressure is undergoing its update

The `rhoThermo` model imposes the calculation of the density update as dependant on $\Delta\rho$ change $\Delta\rho = \psi\Delta p$, thus, the initial ψp_{old} must be stored

Scalar, surface, and volumetric fields functions must be created which are used in the pressure update.

$$phiHbyA = interpolate(\rho)\frac{H(\boldsymbol{U})}{a_p}$$

rhoReactingFoam: pEqn - 2 (transonic)

IF transonic is active: as seen rhoReactingFoam supports transonic study, as its pEqn comes from rhoPimpleFoam. The SUBSONIC case is presented in the next slides.

```
if (pimple.transonic())
{
    surfaceScalarField phid
    (
        "phid",
        (fvc::interpolate(psi)/fvc::interpolate(rho))*phiHbyA
    );

    phiHbyA -= fvc::interpolate(psi*p)*phiHbyA/fvc::interpolate(rho);

    fvScalarMatrix pDDtEqn
    (
        fvc::ddt(rho) + psi*correction(fvm::ddt(p))
      + fvc::div(phiHbyA) + fvm::div(phid, p)
     ==
        fvOptions(psi, p, rho.name())
    );

    while (pimple.correctNonOrthogonal())
    {
        fvScalarMatrix pEqn(pDDtEqn - fvm::laplacian(rhorAUf, p));

        // Relax the pressure equation to ensure diagonal-dominance
        pEqn.relax();

        pEqn.solve();

        if (pimple.finalNonOrthogonalIter())
        {
            phi = phiHbyA + pEqn.flux();
        }
    }
}
```

Surface scalar field "phid" is computed and "phiHbyA" is updated as it is used in the pressure update equation!

This is part of the "pEqn" and takes into account the updated "phiHbyA"

$$\frac{d\rho}{dt} + \psi * corr\left(\frac{dp}{dt}\right) + \nabla \cdot (phiHbyA) + \nabla \cdot \left(\frac{interp(\psi)}{interp(\rho)} p\right) = fvOptions(\psi, p, \rho)$$

rhoReactingFoam: pEqn - 3 (transonic)

```
while (pimple.correctNonOrthogonal())
{
    fvScalarMatrix pEqn(pDDtEqn - fvm::laplacian(rhorAUf, p));

    // Relax the pressure equation to ensure diagonal-dominance
    pEqn.relax();

    pEqn.solve();

    if (pimple.finalNonOrthogonalIter())
    {
        phi = phiHbyA + pEqn.flux();
    }
}
```

One of the things that was presented is that ψ is treated as constant in the `pEqn`. Indeed, by looking at the construction one can see how the value is out of the derivation of density. To be more clear, whenever the TOTAL DERIVATIVE of a quantity is considered, it is well known how that it equals the summation of two values: a partial time derivative, plus the transport term

$$\frac{D\rho}{Dt} = \frac{\partial \rho}{\partial t} + \boldsymbol{U} \cdot \nabla \rho$$

Now, consider $\rho = \frac{p}{RT} = \psi p$
If the ψ value is defined as constant (as in `rhoThermo` model), then it is possible to peroduce a simplification

$$\frac{D\rho}{Dt} = \frac{\partial \rho}{\partial t} + \boldsymbol{U} \cdot \psi \nabla \mathbf{p}$$

rhoReactingFoam: pEqn - 4 (low mach)

```
else
{
    fvScalarMatrix pDDtEqn
    (
        fvc::ddt(rho) + psi*correction(fvm::ddt(p))
      + fvc::div(phiHbyA)
     ==
        fvOptions(psi, p, rho.name())
    );

    while (pimple.correctNonOrthogonal())
    {
        fvScalarMatrix pEqn(pDDtEqn - fvm::laplacian(rhorAUf, p));

        pEqn.solve();

        if (pimple.finalNonOrthogonalIter())
        {
            phi = phiHbyA + pEqn.flux();
        }
    }
}
```

This situation has a simpler formulation, as the transonic effect is not taken into account, but again, this is just part of the `pEqn` written below, which again takes into account the SAME updated "`phiHbyA`"

$$\frac{\partial \rho}{\partial t} + \psi \cdot corr\left(\frac{dp}{dt}\right) + \nabla \cdot (phiHbyA) = fvOptions(\psi, p, \rho)$$

The same operation is followingly done on the pressure equation for the non-orthogonal corrections and the update of ϕ is done.

rhoReactingFoam: pEqn - 5

```
bool limitedp = pressureControl.limit(p);

// Thermodynamic density update
thermo.correctRho(psi*p - psip0);

if (limitedp)
{
    rho = thermo.rho();
}

#include "rhoEqn.H"
#include "compressibleContinuityErrs.H"

// Explicitly relax pressure for momentum corrector
p.relax();
```

Then the density is updated by means of `thermo.correctRho()`
see next slide to clarify that within the parentheses a $\Delta\rho$ is considered, and what this function does.

And if the pressure is limited (by means of a pressure control), then the updated density is copied before performing once again the `rhoEqn` and check for correct solution.

After the review of what `thermo.correctRho()` function does, it surely is interesting to see what happens if the pressure is not limited.

Is the density updated anyway?
Where?
How?

rhoReactingFoam: rhoThermo - correctRho()

```
Foam::volScalarField& Foam::rhoThermo::rho()
{
    return rho_;
}
```

rho = thermo.rho() sends to this location if rhoThermo is the model used and extracts the PRIVATE variable rho_

```
void Foam::rhoThermo::correctRho(const Foam::volScalarField& deltaRho)
{
    rho_ += deltaRho;
}
```

thermo.correctRho($\Delta\rho$) sends to this location if rhoThermo is the model used and modifies the PRIVATE variable rho_ by adding the variation $\Delta\rho$ to it. This $\Delta\rho$ comes from pEqn which considers $\psi p - \psi p_{old}$

```
const Foam::volScalarField& Foam::rhoThermo::psi() const
{
    return psi_;
}
```

psi = thermo.psi() sends to this location if rhoThermo is the model used and extracts the PRIVATE variable psi_. This option is presented but never used in this scope.

MIND WELL: if you look at the same function in the psiThermo.C file, that is an EMPTY function, ie IT DOES NOTHING
(src/thermophysicalModels/basic/psiThermo)

rhoReactingFoam: 3/3

The big question was related to the acquisition of the modified density!
It was seen indeed how if the pressure is limited in the `fvSolution` file, then the density ρ is extracted from the thermo to be equal to the updated private variable "`rho_`" thanks to the command "`rho = thermo.rho()`"

But, ìf the pressure is not limited, the operation is anyway done out of the pressure equation. Then, by calling the same function just written and highlighted below, the density changed value is again extracted, and it's stored in order to be used in the next iteration.

```
        rho = thermo.rho();

        runTime.write();

        Info<< "ExecutionTime = " << runTime.elapsedCpuTime() << " s"
            << "  ClockTime = " << runTime.elapsedClockTime() << " s"
            << nl << endl;
    }

    Info<< "End\n" << endl;

    return 0;
}
```

Combustion

Combustion can be defined in a major class of chemical reaction commonly referred to as burning.
Combustion is an exothermic high-temperature redox chemical reaction which sees the interaction of fuel, ie the reactant, and oxidant, that produces oxidizer.

It was analyzed as the reaction process proceeds following a multiple step chain, and what appears to be a single global reaction is instead composed by a massive sequence of intermediate steps. Moreover it was noted how the mass fraction transport equations of every species have to be computed, to relate them with the energy (enthalpy) equation.

Hence even the simplest combustion reaction involves very tedious and rigorous calculation if all the intermediate steps of the combustion process, all transport equations and all flow equations have to be satisfied simultaneously. All these factors have a significant effect on the computational speed and time of the simulation. But with proper simplifying assumptions the simulation can be done without substantial compromise on the accuracy and convergence of the solution.

Damköler number

One of the most important parameters is the Damköler number. The Damköler number is an adimensional parameter used to compute a relation between chemical reaction timescales and transport phenomena occurring within the system.
Taking into consideration the two relevant quantities, ie the reaction rate $\dot{\omega}$ and the released/absorbed heat of combustion $\dot{Q}$, which are dependant on the turbulent mixing and chemical kinetics, the Damköler number is computed as:

$$Da = \frac{\text{reaction rate}}{\text{convective mass transport}} = \frac{\text{diffusion time}}{\text{reaction time}}$$

Thus, 2 cases are relevant:

- **if** $Da >> 1$ the chemical reactions are supposed to be much faster than turbulent phenomena, so that the reaction rate can be controlled only by mixing, and therefore leading a infinitely fast chemistry combustion model set;
- **if** $Da << 1$ then mixing is much faster than chemistry, and turbulence does not influence predominantly the reaction rate, so that the listed above terms are determined only by chemistry;

One step simple chemically-reacting systems

`singleStepCombustion` models considers a chemistry which occurs in an infinitely fast way, thus the chemical reaction is present in the form of the global one, therefore avoiding the presence on intermediate species.
The reaction sees global initial reactants and global final combustion products, provided the input files (and a `Ydefault` one). Each species production/consumption rate is dependant on the stoichiometric conditions mass ratio and the fuel consumption ratio.

$$R = \dot{\omega}_f \alpha_{st,i}$$

Released heat instead is computed by considering the LHV_f, ie the *fuel lower heating*, and Y_f which was defined ad the fuel mass concentration

$$\dot{Q} = LHV_f \underbrace{\dot{\omega}_f}_{\text{which has to be investigated}} Y_f$$

Infinitely fast & diffusion chemistry model

Infinitely fast chemistry model:

In *infinitely fast chemistry model* the basic assumption is that "*mixed is burnt*".
If this simple chemically-reacting system model is applied the reaction is taken as irreverisble, ie the rate of the reverse reaction is presumed to be very low.
The model assumes that the reactions are completed at the moment of mixing, so that the reaction rate is complitely controlled by turbulent mixing. The reaction rate is given by the consumption of the limiting reactant in the cell

$$\dot{\omega}_f = \rho \frac{1}{C\Delta t} \min \left[Y_f, \frac{Y_{ox}}{\alpha_{st}} \right]$$

Diffusion chemistry model:

Diffusion of reactants (fuel and oxidiser) is considered by taking the inner product of the concentration gradients

$$\dot{\omega}_f = C\mu_{eff} |\nabla Y_f \cdot \nabla Y_{O2}| \text{pos}_0(Y_f) \text{pos}_0(Y_{O2})$$

where μ_{eff} is the effective viscosity computed by the turbulence model. $\text{pos}_0(x)$ is a function that is equal to 1 if $x \geq 0$ and 0 otherwise.

Finite-rate chemistry model

Models are generally made to represent a reactor, which is the space in which the combustion reaction occurs.

The laminar finite-rate model computes the chemical source terms using Arrhenius expressions, and ignores the effects of turbulent fluctuations. The model is exact for laminar flames, but is generally inaccurate for turbulent flames due to highly non-linear Arrhenius chemical kinetics. The laminar model may, however, be acceptable for combustion with relatively slow chemistry and small turbulent fluctuations, such as supersonic flames.

Partially Stirred Reactor (PaSR) models, instead, are commonly used to deal with pollutant formation. A perfectly stirred reactor (PSR) has an inflow rate equal to its outflow rate and a high rate of mixing. The high rate of mixing causes a uniform dispersion of fuel and oxidiser throughout the entire reactor. Therefore if a sample was taken from the outflow of the reactor it should match the concentrations within the reactor. Similar to a PSR, a partially stirred reactor has an equal inflow and outflow, however the rate of mixing or residence time is lower meaning that a uniform dispersion is not achieved. Therefore within the reactor there will be parts where the concentration is different to other parts of the reactor. For combustion this means that some areas could possibly have a concentration that does not allow complete combustion

Laminar & PaSR chemistry model

Laminar:

It does not consider turbulent mixing. Reaction rate and heat of reaction are computed by the chemistry solver:

$$R_i = \dot{\omega}_{i,chem}$$
$$\dot{Q} = \dot{Q}_{chem}$$

Partially Stirred Reactor (PaSR):

The mixing time scale is computed as:

$$\tau_k = C_{mix}\sqrt{\frac{\mu_{eff}}{\rho\varepsilon}}$$

and it is used to compute the reaction limiter κ:

$$\kappa = \begin{cases} \dfrac{\tau_{chem}}{\tau_{chem}+\tau_k} & \text{if} \quad \tau_k > 0 \\ 1 & \text{otherwise} \end{cases}$$

Then:

$$R_i = \kappa\,\dot{\omega}_{i,chem}$$
$$\dot{Q} = \kappa\,\dot{Q}_{chem}$$

combustionModels: Psi & Rho

Regarding combustion, two models have to be considered again: **Rho Combustion Model** and **Psi Combustion Model**.

It is possible to compare the two different models side by side to see that the `autoPtr` (auto pointer) is linked to two different models which are rispectively referred one to `rhoReaction` and the other to `psiReaction`.

Indeed within the `createFields.H` files of the solver

- `reactingFoam` is linked to `combustionModel<psiReactionThermo>` reaction;
- `rhoReactingFoam` is linked to `combustionModel<rhoReactionThermo>` reaction;

```
Info<< "Creating reaction model\n" << endl;
autoPtr<CombustionModel<psiReactionThermo>> reaction
(
    CombustionModel<psiReactionThermo>::New(thermo, turbulence())
);
```

reactingFoam

```
Info<< "Creating reaction model\n" << endl;
autoPtr<CombustionModel<rhoReactionThermo>> reaction
(
    CombustionModel<rhoReactionThermo>::New(thermo, turbulence())
);
```

rhoReactingFoam

In order to give a look at the details it is sufficient to look at `CombustionModels.C` file

CombustionModels.C

```
#include "makeCombustionTypes.H"

#include "CombustionModel.H"

#include "rhoReactionThermo.H"
#include "psiReactionThermo.H"

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

namespace Foam
{

makeCombustion(psiReactionThermo);
makeCombustion(rhoReactionThermo);

}

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

There are just two different combustion models which are linked to the `reactionThermo` models. This file is an abstract, for which `combustionModel<ReactionThermo>` is the class template reference

Remember that `psiReaction` calculates the enthalpy for combustion mixture as a function of compressibility PSI ψ, while `rhoReaction` does this by means of the density RHO ρ

Thank you for your attention!

contact: federico.piscaglia@polimi.it