Coal Combustion – Devolatilization Model

Empirical model

- **Single step model**

$$\text{coal} \xrightarrow{k_1} X\ \text{volatiles} + (1-X)\text{char} \quad \textit{where } X \textit{ is volatile fraction}$$

$$\frac{dX(t)}{dt} = k_1(X^{\infty} - X(t)) \qquad k_1 = A_1 \exp(-E_1/RT)$$

- **Two competing rates model (at low and high temperatures)**
  - Heating rate affects both devolatilization rate and yield

$$\xrightarrow{k_1} X_1 volatile1 + (1-X_1)\,\text{Residue1}$$

$$\text{coal} \xrightarrow{k_2} X_2 volatile2 + (1-X_2)\,\text{Residue2}$$

$$\frac{dX(t)}{dt} = \frac{dX_1(t)}{dt} + \frac{dX_2(t)}{dt} = (\alpha_1 k_1 + \alpha_2 k_2)\,m_{daf} \qquad k_1 = A_1 \exp(-E_1/RT) \qquad k_2 = A_2 \exp(-E_2/RT)$$

- **Distributed Activation Energy Model (DAEM)**
  - Volatiles can be released from coal of different activation energies in parallel

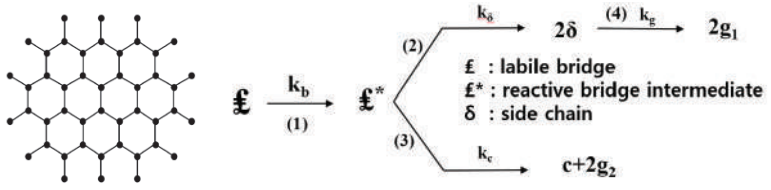
$$\text{coal} \xrightarrow{k_1} X\ \text{volatiles} + (1-X)\text{char} \quad \textit{where } X \textit{ is volatile fraction}$$

$$1 - \frac{X(t)}{X^{\infty}} = \int_0^{\infty} \exp\left[-k_0 \int_0^t e^{-E/RT} dt\right] f(E)dE \qquad \int_0^{\infty} f(E)dE = 1$$

Coal Combustion – Devolatilization Model

Network model

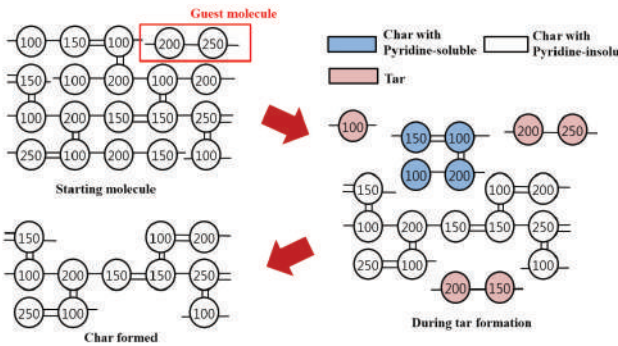
- **CPD (Chemical Percolation Devolatilization)**



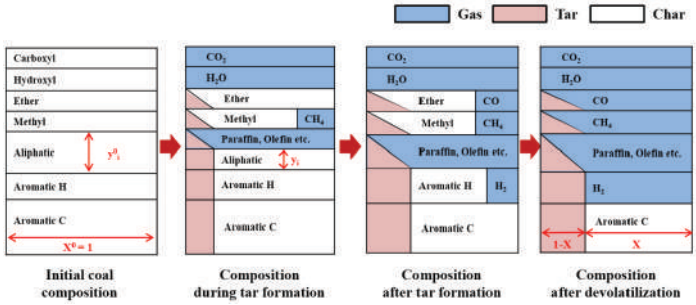
Honeycomb Lattice

(1) formation of a reactive bride intermediate
(2) formation of a side chain
(3) formation of char bridge and gas
(4) conversion of side chain into light gases

- **FG-DVC (Function Group–Depolymerization Vaporization Crosslinking)**



DVC model

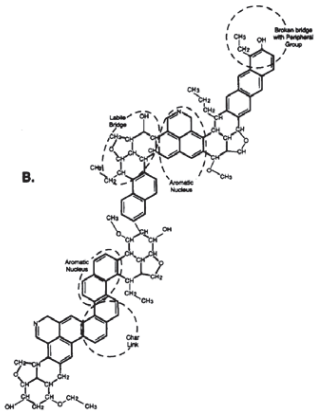


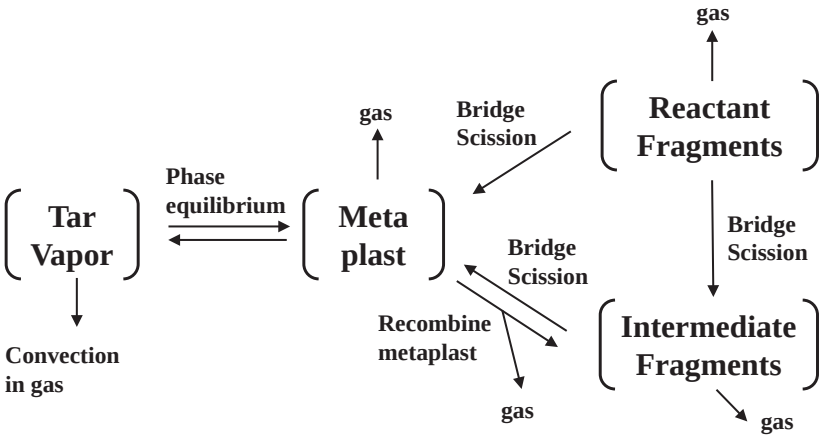
FG model

Coal Combustion – Devolatilization Model

Network model

- **FLASHCHAIN**
  - Depolymerization with broad size distribution by bridge scission
  - A phase equilibrium is established and tar is convected out of the particle
  - Conversion of labile bridges into char link to generate non-condensable gases
  - Fragments crosslink to form char





Coal Combustion – Char Combustion Model

Global reaction rate model (based on external surface area)

- **Diffusion limited model**
  - Reaction rate is decided by diffusion rate and appropriate to use in Zone Ⅲ

$$q_{diff} = k_d\left(P_\infty - P_s\right) \quad where \;\; k_d = \left(\frac{2}{1+\psi}\right)\frac{D_{AB}MW_C Sh}{d_p RT_g}$$

$$\frac{dm_p}{dt} = -\pi d_p^2 q_{diff} = -\pi d_p^2 k_d (P_\infty - P_s) = -\pi d_p^2 k_d P_\infty \qquad \therefore \frac{dm_p}{dt} = -\pi d_p^2 k_d P_\infty$$

D_{AB} : *diffusion coefficient*,
Sh : *Sherwood number*,
MW_c : *carbon molecular weight*,
P_∞ : *bulk pressure*,
ψ : *char burnout ratio*

- **Apparent rate / Diffusion rate model**
  - Most often used for char combustion model in CFD

$$q_{reaction} = k_c P_s \qquad where \; k_c = A\exp(-E/RT)$$

$$\frac{1}{k} = \frac{1}{k_d} + \frac{1}{k_c}$$

$$\underbrace{k_d = \left(\frac{2}{1+\psi}\right)\frac{D_{AB}MW_C Sh}{d_p RT_g}}_{Diffusion} \longleftrightarrow \underbrace{k_c = A\exp(-E/RT)}_{chemical\ kinetics} \qquad \therefore \frac{dm_p}{dt} = -\pi d_p^2 kP_\infty = -\pi d_p{}^2\left(\frac{k_d k_c}{k_d + k_c}\right)P_\infty$$

- **Global n-th order model**
  - It's appropriate to use the model in specific condition

$$q = k_s P_s^n \quad where \;\; k_s = A\exp(-E/RT_p) \qquad \therefore \frac{dm_p}{dt} = -\pi d_p^2 q = -\pi d_p^2 A\exp(-E/RT_p)P_s^n$$

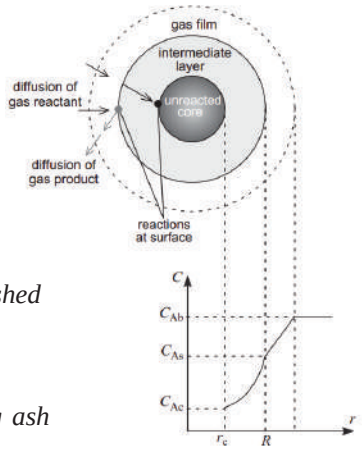
P_s : *satured pressure of oxgen at the particle surface*

- **Hurts and Mitchell model**
  - One of global n-th order model
  - Pre-exponential factor and activation energy can be gained by coal analysis
  - Correlations for K_{Tm} and E (T_g > 1500K, 75 μm < d_p < 200 μm , P_{O2} > 0.03 atm)

Coal Combustion – Char Combustion Model

Intrinsic reactivity model (based on BET)

- **Unreacted core shrinking model**
  - Char is divide into Un-reacted core and ash layer with porous structure



$$q_{un\_core} = \left\{ \frac{1}{k_{diff}} + \frac{1}{k_{surface} Y^2} + \frac{1}{k_{ash}} \left(\frac{1}{Y} - 1 \right) \right\} \times (P - P^*)$$

$$\text{where } Y = \frac{r_c}{R} = \left(\frac{1-x}{1-f} \right)^{\frac{1}{3}}$$

$$k_{ash} = k_{surface} \cdot \varepsilon^{2.5}$$

f : *conversion when pyrolysis is finished*
ε : *voidage* of ash layer
r_c : *radius of unreacted core*
R : *radius of whole particle including ash*
x : *conversion ratio of char*

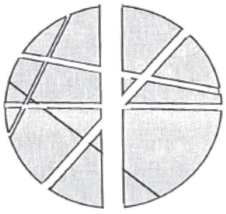
- **Random pore model**
  - Char particle is porous structure with internal surface such as pores

$$\frac{dx}{dt} = k_p (1-x) \sqrt{1 - \psi \ln(1-x)}$$

$$\frac{S}{S_0} = (1-x) \sqrt{1 - \psi \ln(1-x)}$$

$$\psi = 4\pi L_0 (1-\varepsilon_0) / S_0^{\,2}$$

ψ : *initial pore structure,*
L_0 : *initial pore length,*
ε_0 : *initial porosity,*
S_0 : *initial specific surface area*
x : *conversion ratio of char*



GAVALAS
(random pore model)

Coal Combustion – Char Combustion Model

Intrinsic reactivity model (based on BET)

- **CBK/E (Char Burnout kinetic / extended) model**
  - Assume oxidation reaction to 3 step intrinsic kinetics

$$\begin{aligned} R_1 &: C + O_2 \rightarrow 2C(O) \\ R_2 &: C(O) + O_2 \rightarrow CO_2 + C(O) \\ R_3 &: C(O) \rightarrow CO \end{aligned}$$

θ : *the fraction of sites occupied by the absorbed oxygen complex*
M_T : *Thiele* modulus

$$q_1 = k_1 P_{O_2} (1-\theta)$$
$$q_2 = k_2 P_{O_2} \theta$$
$$q_3 = k_3 \theta$$

Where k_1, k_2, k_3 *is Arrhenius form*

$$A_3 = 10^{14.38 - 0.0764 C_{daf}} \qquad E_1 = 25\ KJ/mole$$
$$A_2 / A_3 = 5.0 \times 10^4 \quad \text{and} \quad E_2 = 117\ KJ/mole$$
$$A_3 / A_1 = 1.0 \times 10^{-6} \qquad E_3 = 133.8\ KJ/mole$$

Ratio of CO and CO_2 *generation*

$$\frac{CO}{CO_2} = \frac{k_3}{k_2 P_{O_2}}$$

Overall oxidation rate

$$r_{gas} = \frac{k_1 k_2 P_{O_2}^2 + k_1 k_3 P_{O_2}}{k_1 P_{O_2} + k_3 / 2} \Rightarrow q_{\text{intrinsic}} = \frac{d_p}{6} \rho r_{gas} \Rightarrow \underline{q_{CBK/E} = \eta q_{\text{intrinsic}}}$$

$$\text{where} \quad \eta(\text{effective factor}) = \frac{1}{M_T} \left(\frac{1}{\tanh(3M_T)} - \frac{1}{3M_T} \right)$$

Coal Combustion – OpenFOAM Development

OpenFOAM solver development

- **SimpleCoalCombustionFoam**
  - Steady state blended or single coal combustion solver
  - Developed based on OpenFOAM ver. 2.3.x
  - Includes various improved devolatilization, char surface reaction and gas combustion models

simpleCoalCombustionFoam (based on ver. 2.3.x)

- **Injection of parcels**
- **Parcel tracking**
- **Initial inert heating**
- **Devolatilization**
- **Char surface reaction**

Accumulate source terms due to the coal particle

- **Momentum**
- **Species**
- **Sensible enthalpy**
- **Pressure**

libCOALPOSTECH (based on ver. 2.3.x)

User libraries : coal combustion models

- **coalCombustion_POSTECH**
  - Blended coal capability
  - Char reaction submodels
- **Intermediate_POSTECH**
  - Injection, tracking, impaction on wall
  - Phase change, devolatilization, particle radiation

User libraries : gas combustion models

- **chemistryModel_POSTECH**
  - Multiple reaction step capability
- **combustionModels_POSTECH**
  - Gas-phase combustion (finite-rate/EDM/EDC, etc)
- **radiationModels_POSTECH**
  - Radiation models

The schematic diagram of the simpleCoalCombustionFoam

Thank you

Turbulence and Thermodynamics Modelling Libraries

Hrvoje Jasak

hrvoje.jasak@fsb.hr, h.jasak@wikki.co.uk

University of Zagreb, Croatia and

Wikki Ltd, United Kingdom

Background

Objective

- Describe the structure and interface to turbulence modelling libraries
- Describe implementation and capability of thermodynamics model library

Topics

1. Turbulence Model Library Interface
   - Early implementation interface: turbulence viscosity models
   - `fvVectorMatrix` interface
   - RANS and LES model library; compressible and incompressible flow
   - Wall function implementation
2. Thermodynamics Library: Interface and Implementation
   - Overview of overall capability: `thermo` package interface
   - Defining a thermo type
   - Efficiency concerns: run-time and compile-time polymorphism
3. Summary

Turbulence Library

Turbulence Modelling Library Overview

- A turbulence model accounts for the “unresolved” part of flow fluctuations on the mean or resolved flow field
- Within this framework, Reynolds-Averaged Navier-Stokes (RANS) models, as well as Large Eddy Simulation (LES) can be covered
- Turbulence model interfaces with the momentum equation, heat and species transfer models and various other components in a less predictable way

Turbulence Model Library

- All RANS/LES turbulence models serve the same purpose: virtual base class `turbulenceModel` with run-time selection
- For reasons of top-level solver structure, LES and RANS models can be grouped together under the common interface
- . . . but keep in mind that their **physical meaning is different**
- Note: run-time selection table for `RASModel` used directly for steady-state solvers

```
class turbulenceModel {};

class RASModel: public turbulenceModel {};
class LESModel: public turbulenceModel {};
```


Turbulence Library

Interface to the Momentum Equation: Historical Form

- Assuming **eddy viscosity models**, interface to the momentum equation can be trivial: turbulent viscosity `nut()`

$$\frac{\partial \mathbf{u}}{\partial t} + \nabla\bullet(\mathbf{uu}) - \nabla\bullet\left[((\nu + \nu_t)\nabla\mathbf{u}\right] = -\nabla p$$

```
class turbulenceModel
{
    virtual const volScalarField& nut() const = 0;
    virtual void correct() = 0;
};
    fvVectorMatrix UEqn
    (
        fvm::ddt(rho, U)
      + fvm::div(phi, U)
      - fvm::laplacian(nu + turbulence->nut(), U)
     ==
      - fvc::grad(p)
    );
    turbulence->correct();
```

Turbulence Library

Interface to the Momentum Equation: Historical Form

- This is the original implementation (1994), but it is not sufficiently flexible. In fact, it is wrong: $\nabla \mu_t \bullet \nabla \mathbf{u}$ term is missing
- Furthermore, Reynolds stress transport models cannot be implemented: explicit correction to the momentum source
- Seeking a more general interface: turbulence model provides the “laplacian” contribution to the momentum equation, comprising a matrix and a source
- In the new hierarchy, a `turbulenceModel` interacts with a laminar viscosity model, which in turn contains its own virtual base class with run-time selection
- `turbulenceModel` class may provide a further interface functions, (*e.g.* $k()$). Please keep in mind th generality of the interface: are all turbulence models capable of providing the required quantity?
- Example: Lagrangian spray model requires a turbulence dispersion term based on $\mathbf{u}'$ and length-scale. How is $\mathbf{u}'$ provided from the Reynolds stress transport model?
- Coupling to the energy equation currently done via “eddy diffusivity” field `alphaEff()`: see above, with same limitations (scalar fluxes)

Turbulence Library

Interface to the Momentum Equation: Historical Form

```
class turbulenceModel
{
    virtual volTensorField R() const = 0;
    virtual tmp<fvVectorMatrix> divR
    (
        volVectorField& U
    ) const = 0;
    virtual void correct() = 0;
};

    fvVectorMatrix UEqn
    (
        fvm::ddt(rho, U)
      + fvm::div(phi, U)
      + turbulence->divR(U)
     ==
      - fvc::grad(p)
    );
    turbulence->correct();
```

Turbulence Library

Interface to the Momentum Equation: Current Implementation

- Momentum coupling term in the $k - \epsilon$ model

```
tmp<fvVectorMatrix> kEpsilon::divDevReff(volVectorField& U) const
{
    return
    (
      - fvm::laplacian(nuEff(), U)
      - fvc::div(nuEff()*dev(fvc::grad(U)().T()))
    );
}
```

- `turbulence->correct()` call typically involves a solution of further transport equations

Near-Wall Distance

- Near-wall distance functionality provided in 2 modes
  - `nearWallDist`, calculated only for cells touching walls
  - `wallDist`, calculated for all cells in the mesh
- Note: mark patch as `wall` type to be used for reference distance!

Turbulence Library

Note on Implementation

- Individual turbulence models are not layered into own hierarchies but are implemented as self-contained units. Example: $k - \epsilon$ and realisable $k - \epsilon$ models do no share code, even though they are closely related to each other
- All turbulence models carry the `fvm::ddt` terms and relaxation calls: it is not known if they are used from a steady or transient solver
- New turbulence model implementation derives form existing infrastructure and implements the virtual function interface. Example: Spalart-Allmaras

```
class SpalartAllmaras : public turbulenceModel{};
```

Some Limitations

- Compressible models interface with the form `fvm::ddt(rho, U)` and require dynamic viscosity, while incompressible models interface via `fvm::ddt(U)` and kinematic viscosity. **Separate hierarchies required**
- Some incompressible models require modification of interface, eg VOF free surface or multiphase flow models: Changed in FOAM-2.3.1
- It is very difficult to make the model porous- or phase fraction-aware: new model implementation is required

Turbulence Library

Near-Wall Modelling

- Wall functions are a quasi-general model of turbulence in the near-wall region: can be used with multiple RANS turbulence models
- Note: near-wall treatment in LES can be similar and answers to similar interface, but with substantially different physical meaning
- Two main categories of implementation ϵ and ω wall functions
- Effect of wall functions on momentum equation and turbulence model
  - Account for wall drag in the momentum equation: accounted for implicitly via a modification in `nuEff()` at the wall
  - Modify turbulence generation term G in the near-wall cell
  - Impose turbulence length-scale (ϵ) in the near-wall cell

Turbulence Library

Implementation of Wall Functions

- Initial implementation via include files in turbulence models
- Current implementation
  - Length-scale equation (ϵ or ω) evaluates y^+ from near-wall distance and k and calculates the wall function properties
  - G and ν_t enforced via a `objectRegistry` lookup of `volScalarField` objects
  - `kqRWallFunction` passively accepts the field and (internally) evaluates as a zero gradient condition

Thermodynamics Library

`thermo` Package Interface

- In functionality and layout, `thermo` package library is similar to the `turbulenceModel` package, but the implementation is much more complex
- Objective:
  - Material properties of the medium (fluid) are a function of state variables, in easiest case, p and T. In reality, this is much more complex
  - For each property, multiple choices for evaluation exist. Example: laminar dynamic viscosity
  - Energy equation (controlling temperature T) can be formulated in many different ways: internal energy, enthalpy, rothalpy, thermo-chemical energy etc. The user interface is always define in terms of T and conversion of boundary conditions (eg. fixed value, fixed flux) should happen automatically
  - Interface to incompressible and compressible flow solvers is substantially different: formulation of the pressure equation and compressibility. This needs to be handled by the interface

Thermodynamics Library

`thermo` Package Interface

- Efficiency concerns
  - In general, evaluation of material properties may be costly or complex (JANAF polynomials, table lookup) and should be optimised
  - Derivative properties can be required only on a subset of field and should be calculated only where needed. Example: speed of sound at a non-reflective boundary condition
  - **Efficiency dictates the structure of implementation**!
  - Object-orientation paradigm used to achieve efficiency is **compile-time polymorphism**, with a single level of run-time selection at the top (`basicThermo`) level

Thermodynamics Library

Defining a `thermo` Type

1. `specie`: encoding basic information of the material: name, mol. weight
2. `equationOfState`: basic (equilibrium) behaviour of the material. Example: incompressible with constant or variable density, perfect gas, real gas etc.
3. `mixture` type: for multi-component systems, the definition of mixture operations, modifying basic properties based on mixture fraction. In eg. combustion simulations, multiple mixtures may co-exist in unburnt and burnt state or eg. depending on fuel fraction
4. `mixtureThermo` type: defining specific heat capacity C_p as a function of state, including conversion from temperature to energy/enthalpy
5. `transport` type: defining dynamic viscosity and thermal conductivity as a function of state variables
6. `basicThermo` type: choice of the standard form of the energy equation to be solved to represent heat transfer. Example: sensible enthalpy, internal energy

`basicThermo` Type

- Up to level 6, all properties can be seen as “single point” properties, depending on state variables and composition
- `basicThermo` holds **field variables** and update function: `thermo->correct()`

Thermodynamics Library

Notes

- The system needs to be capable of dealing with simple mixtures, but also with eg. combustion mixtures. Therefore, basic properties (molar mass, composition, equation of state constants, thermo and transport properties need to be “mixed” in a consistent manner
- Formulation of the compressible flow solvers requires different types of `basicThermo`, depending on the choice of energy equation(s) to be solved at top level
- Formulation of the pressure equation for compressible flow solvers requires the definition of fluid density in terms of compressibility ψ

$$\rho = \psi p$$

  where, for ideal gas $\psi = \frac{1}{RT}$
- Wealth of model and interface does not allow all possibilities to be handled. Example: real gas model and interaction with the energy equation

Thermodynamics Library

Run-Time Selection?

- `thermo` hierarchy provides 7 distinct choices to be made in selection of thermodynamic model behaviour of a fluid
- Most combinations of choices are legal and sensible: lots of possibilities!
- Standard object-oriented implementation of such structures is via virtual base classes and virtual function calls, **but for our purpose this is unacceptable!**. The cost of virtual function call **for every cell and for every iteration** is prohibitive
- Top-level `basicThermo` demands virtual functions for interfacing with flow solvers: allow run-time selection of a thermodynamics model
- Choice of a `thermo` model is in any case done for the complete domain
- Efficiency concern is solved by **performing an operation over an array within a single virtual function call**, as opposed to performing a virtual function call cell-per-cell

Thermodynamics Library

Design Considerations

- `basicThermo` holds **field variables** and update function under virtual functions
- All other virtual function calls should be eliminated: **compile-time polymorphism**

Compile-Time Polymorphism

- Cost of run-time polymorphism is in **run-time execution decisions**
  - RTTI analysis of the object on which the function is called
  - Hash table lookup of the function pointer
  - Long jump function call: no optimisation across the call
- Compile-time polymorphism relocates the decision to compile-time
  - Calling code is templated on type (equivalent to run-time selection)
  - At template instantiation (compile-time) the called function is explicitly known and can be inlined and optimised: no further run-time cost

```
hPsiThermo
<
    pureMixture
    <
        constTransport<specieThermo<hConstThermo<perfectGas>>>
    >
>;
```

Thermodynamics Library

Integration to Top-Level Code

- Create `basicThermo` form run-time selection
- Refer to state variables inside the thermo
- Make local copy of `rho` and keep updated: `ddt(rho,U)` terms

```
Info<< "Reading thermophysical properties" << endl;
autoPtr<basicPsiThermo> pThermo
(
    basicPsiThermo::New(mesh)
);
basicPsiThermo& thermo = pThermo();

volScalarField& T = const_cast<volScalarField&>(thermo.T());
volScalarField& p = thermo.p();
volScalarField& e = thermo.e();
const volScalarField& psi = thermo.psi();
```

- Update `rho` and `thermo.correct()` in the time loop, when state variables are at new time level

Summary

Turbulence Modelling

- `turbulenceModel` library encapsulates incompressible and compressible turbulence models under a single interface, with run-time selection
- Steady solvers typically access `RASModel` to avoid “steady-state LES”
- Implementation is a simplest form of run-time selection

Thermodynamics Library

- Thermodynamics modelling library provides a comprehensive interface to thermophysical properties of liquids and gasses, under an interface for easy integration into top-level solvers
- Each level of interface chooses evaluation method for material properties; this can be combined at will
- Top-level `basicThermo` is grouped depending on choice of energy equation and compressibility model
- For efficiency, lower level functionality is implemented using compile-time polymorphism

Application of OpenFOAM for Various Industrial Combustion Devices

Workshop for OpenFOAM and its Application in Industrial Combustion Devices

26-27th Feb. 2015
POSCO International Center, Pohang, Korea

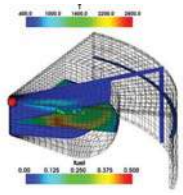
Kang Y. Huh
Pohang University of Science and Technology

Contents

Contents

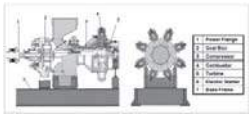
Introduction

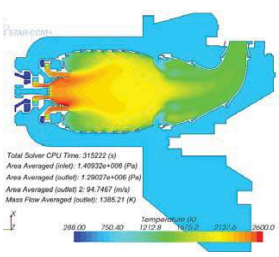
Most industrial combustion devices operate in the regime of turbulent combustion

Diesel engine


Pulverized coal power plant





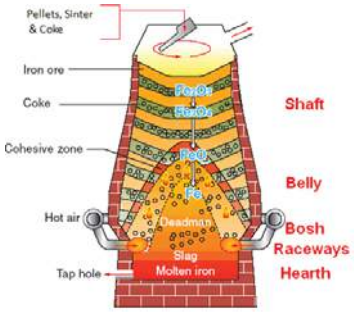
Gas turbine

CFD analysis of turbulent combustion is a crucial design process to improve performance of practical combustion devices

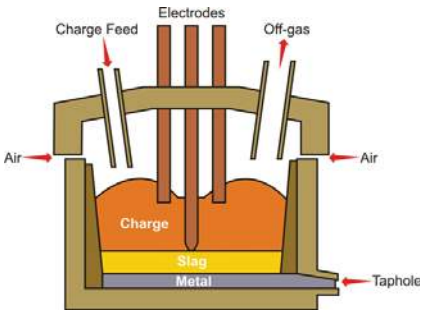
Introduction

Most industrial combustion devices operate in the regime of turbulent combustion

	Transient	Statistically steady / Steady	
Gas	Spark ignition engine	Gas Turbine / HRSG	
Liquid	Compression ignition engine	Heavy-oil furnace	
Solid	Blast furnace / Electric arc furnace	FINEX	Pulverized coal furnace



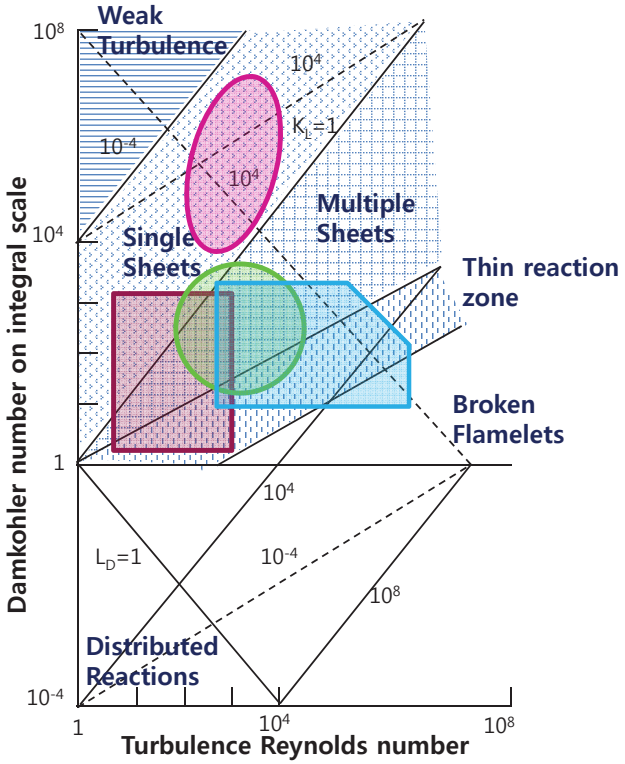
Blast furnace

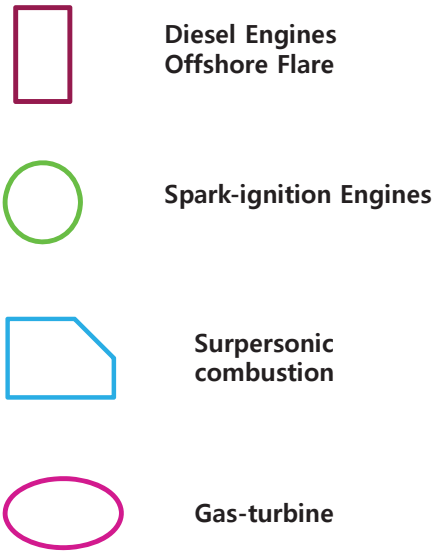


Electric arc furnace

Introduction

Classification of combustion devices related to combustion regime





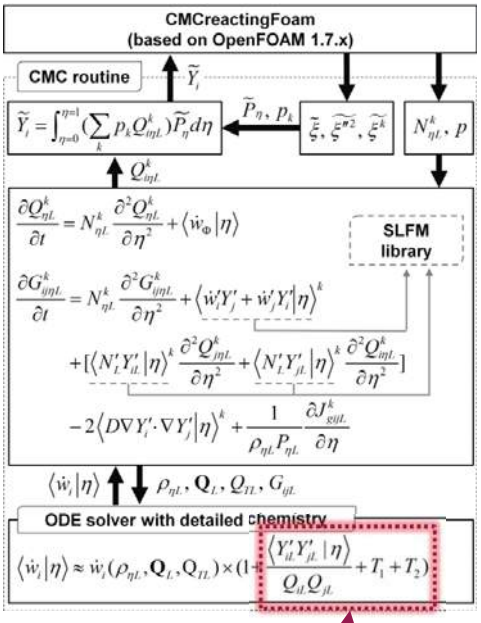
Turbulent Nonpremixed Flames

Conditional Moment Closure Model

Implementation strategies

- Open source CFD toolbox, OpenFOAM, is coupled with Lagrangian CMC routine
- OpenFOAM solves flow and mixing field in the physical space,
  - Favre mean mass, momentum, energy, turbulence
  - Favre mean mixture fraction and its variance
- Lagrangian CMC routine solves conditionally averaged equations in the mixture fraction space
  - Conditional mean mass fractions and enthalpy
  - Conditional variances and covariances (2nd order CMC, $G_{ij\eta L} \equiv \langle Y''_i Y''_j | \eta \rangle$)
- Source terms of chemical reaction are integrated by stiff ordinary differential equation solver, SIBS, with GRI 3.0 mechanism.
- Correction is made up to the second order terms in Taylor expansion of the Arrhenius reaction rate for the following four rate limiting steps

$$H + O_2 \rightarrow OH + O$$
$$CO + OH \rightarrow CO_2 + H$$
$$CH_4 + H \rightarrow CH_3 + H_2$$
$$H + O_2 + M \rightarrow HO_2 + M$$

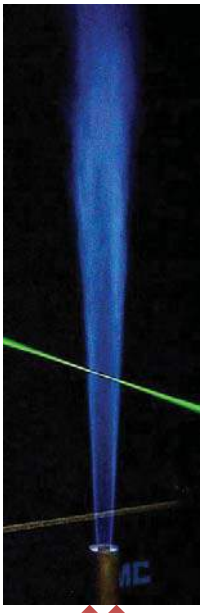


Schematic diagram of interaction between OpenFOAM and CMC routines

Second order correction terms

Basic Flame - Sandia Flame D & E

Case description



Sandia/TUD Piloted CH4/Air Jet Flames

Fuel : 25% CH4, 75% Air (% vol.)

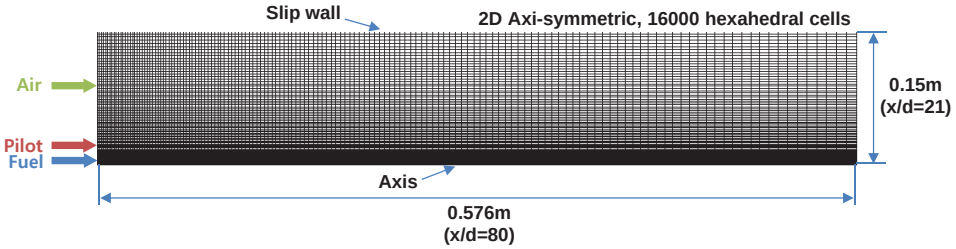
Stoichiometric mixture fraction = 0.351

Nozzle diameter = 7.2mm, Pilot diameter = 18.2mm

Fuel Temp = 294K, Pilot Temp = 1880K, Coflow Temp = 291K

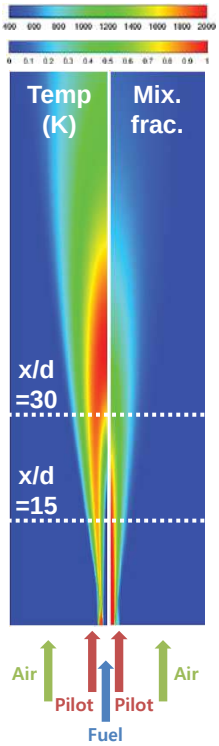
Flame D	Flame E
Fuel velocity = 49.6m/s	Fuel velocity = 74.4m/s
Pilot velocity = 11.4m/s	Pilot velocity = 17.1m/s
Reynolds number = 22,400	Reynolds number = 33,600

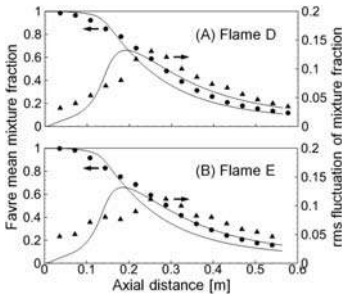
R. S. Barlow and J. H. Frank, Proc. Combust. Inst. 27:1087-1095 (1998)
R. S. Barlow, J. H. Frank, A. N. Karpetis and J. Y. Chen, Combust. Flame 143:433-449 (2005)
Ch. Schneider, A. Dreizler, J. Janicka, Combust. Flame 135:185-190 (2003)



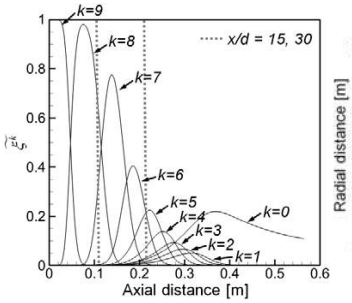
Basic Flame - Sandia Flame D & E

Results

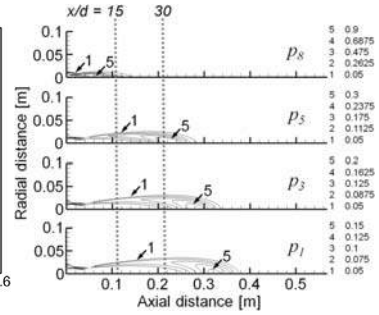




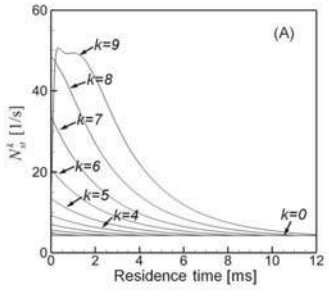
Favre mean mixture fraction and rms fluctuations of mixture fraction along the axis



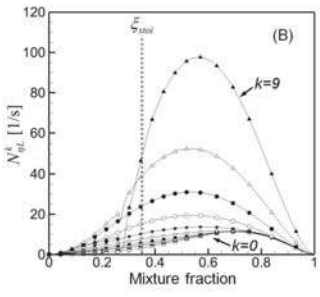
Distribution of $\widetilde{\xi^k}$ for ten flame groups along the axis



Axial distribution of p_k for the 1st, 3rd, 5th and 8th flame groups in Flame D



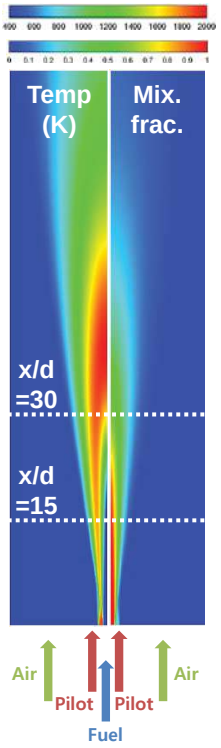
Conditional SDR's at stoichiometry w.r.t. the residence time

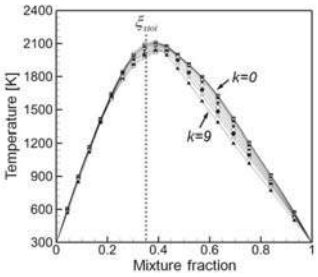


Conditional SDR's for ten flame groups in Flame D

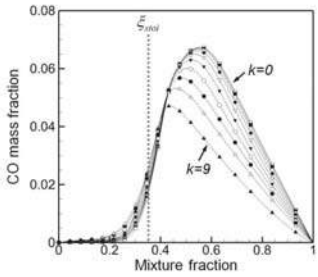
Basic Flame - Sandia Flame D & E

Results

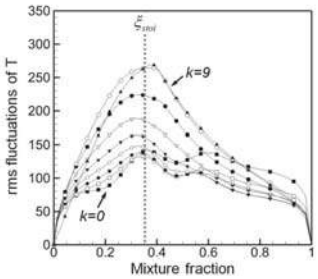




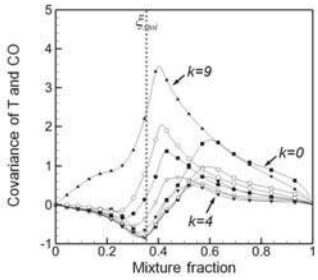
Conditional temperature for ten flame groups in Flame D



Conditional CO mass fraction for ten flame groups in Flame D



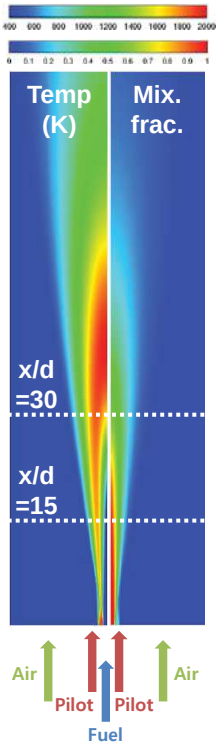
Conditional rms fluctuations of T for ten flame groups in Flame D



Conditional covariance of T and CO for ten flame groups in Flame D

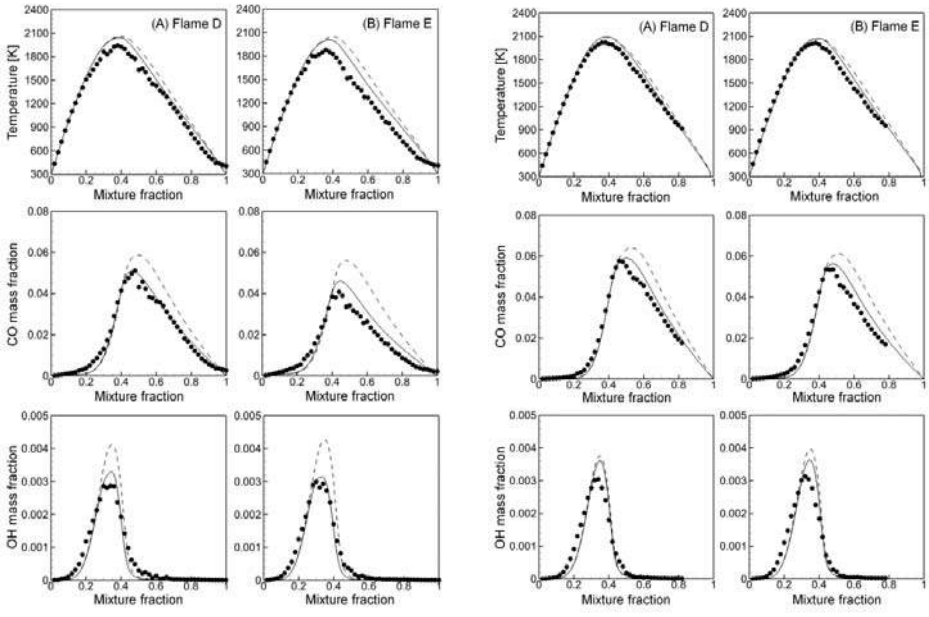
Basic Flame - Sandia Flame D & E

Results



x/d=15

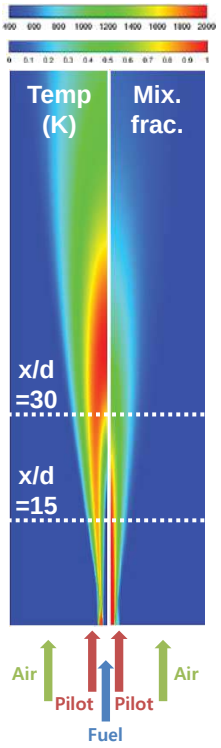
x/d=30



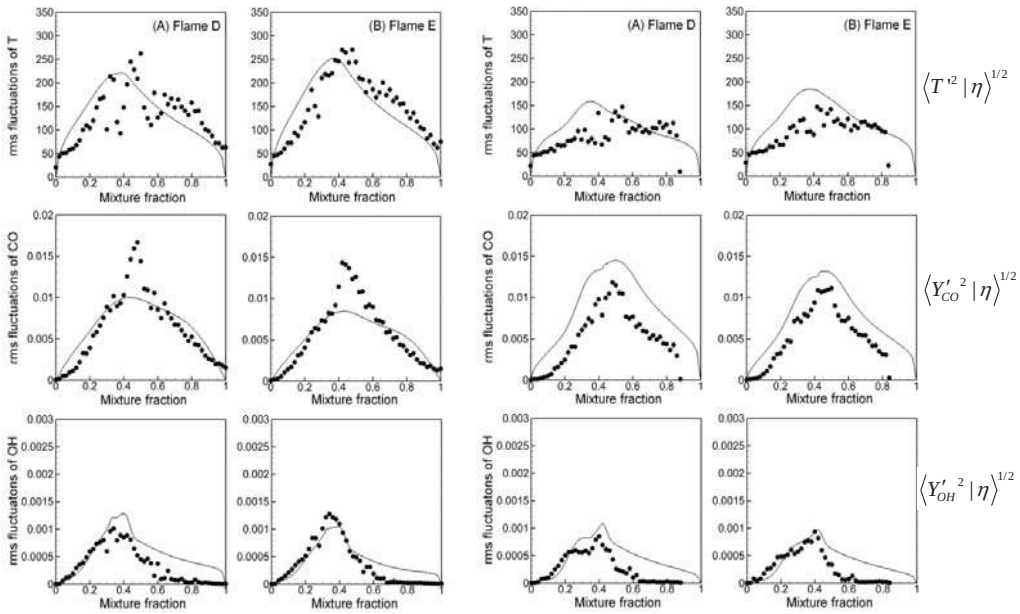
Measured and predicted T, conditional mass fractions at x/d=15, 30 in Flame D and E (symbols, measurement; lines, prediction)

Basic Flame - Sandia Flame D & E

Results



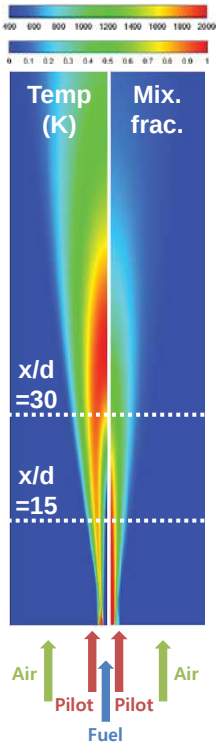
x/d=15 x/d=30



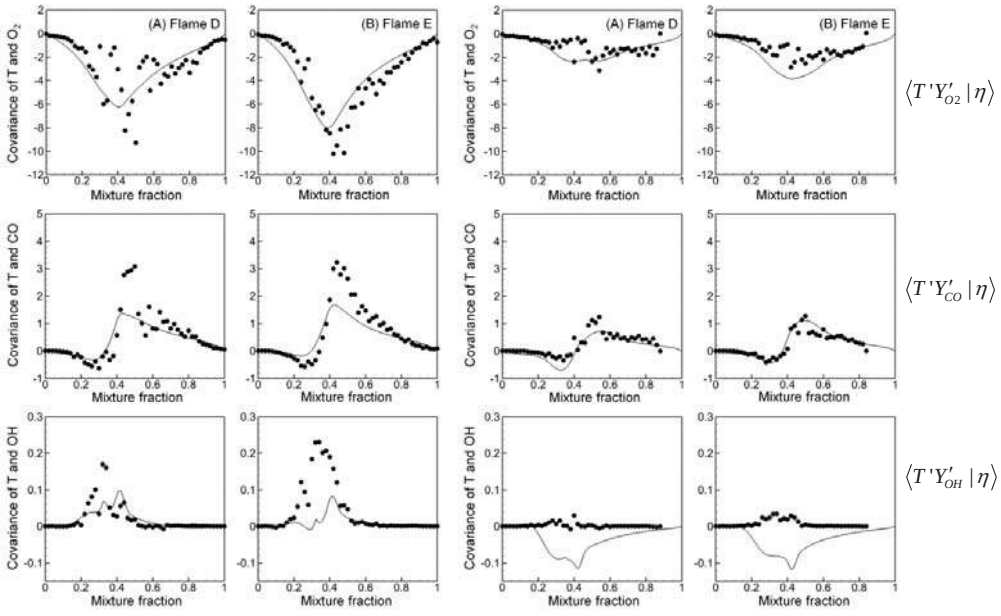
Measured and predicted conditional rms fluctuations at x/d=15, 30 in Flame D and E (symbols, measurement; lines, prediction)

Basic Flame - Sandia Flame D & E

Results



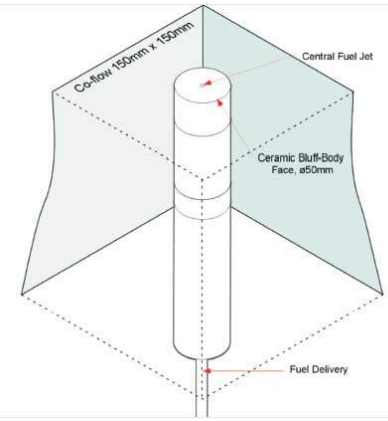
x/d=15 x/d=30



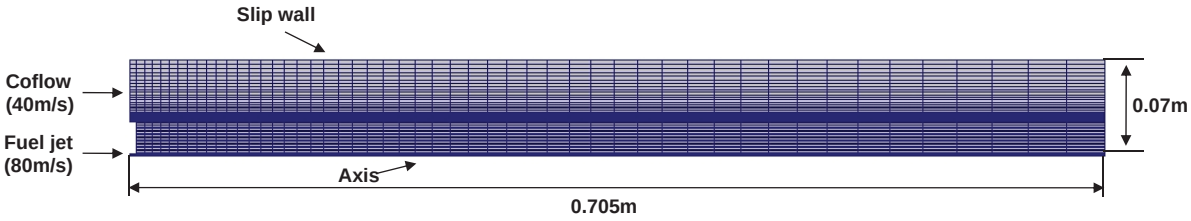
Measured and predicted conditional covariances at x/d=15, 30 in Flame D and E (symbols, measurement; lines, prediction)

Basic Flame - Bluff Body Flame

Case description



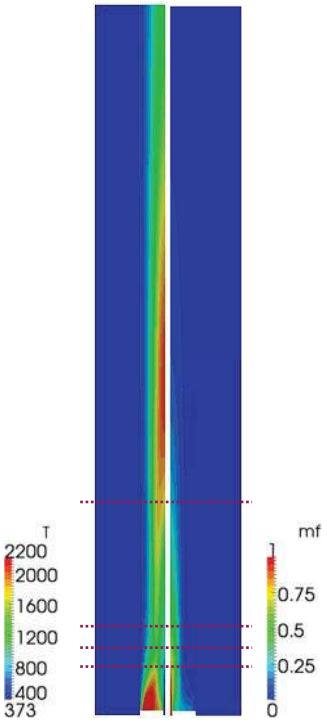
Description	Specification
Fuel	CH_3OH(methanol)
Fuel jet (mm)	1.8
Bluff body radius (mm)	25
Fuel / Air mean vel (m/s)	80 / 40
Fuel temp (K)	373
Adiabatic flame temp (K)	2260
Stoichiometric mixture fraction	0.135

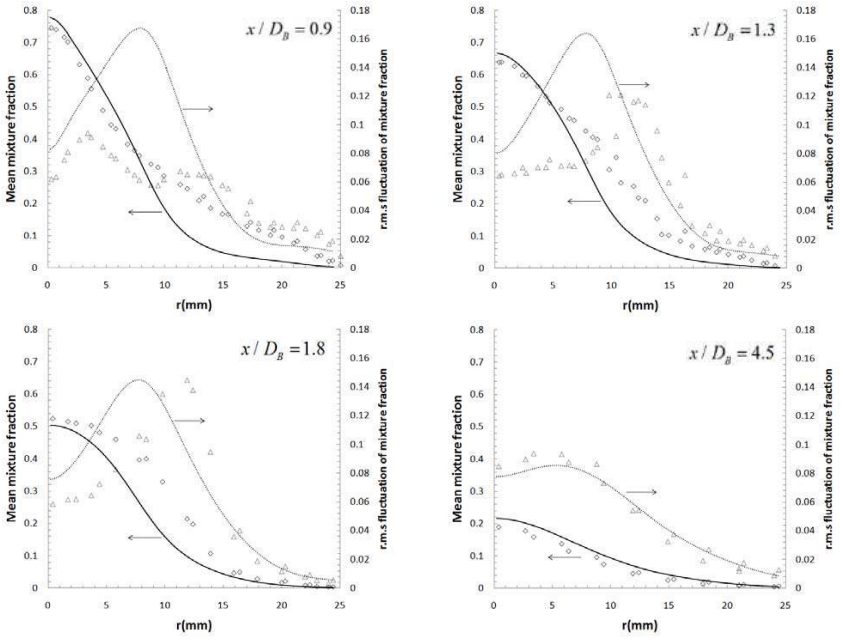


"Instantaneous and mean compositional structure of bluff body stabilized non - premixed flames", B. B. Dally, A. R. Masri, R. S. Barlow, G. J. Fiechtner, Combustion and Flame, 114:119-148 (1998)

Basic Flame - Bluff Body Flame

Results



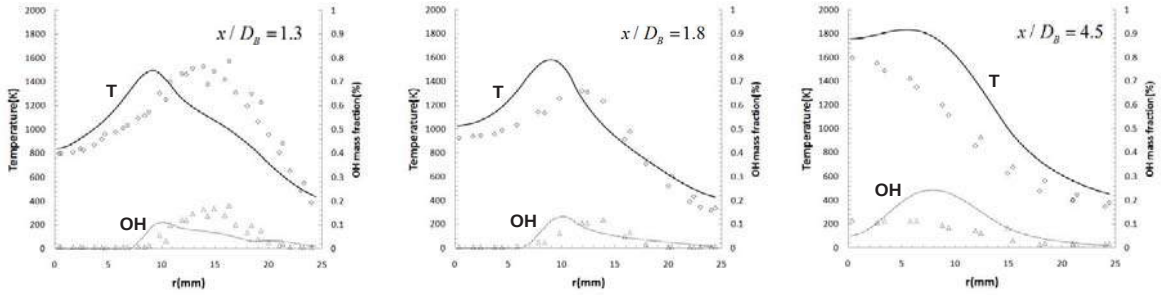


Radial distributions of the Favre mean mixture fraction and r.m.s fluctuation of the mixture fraction

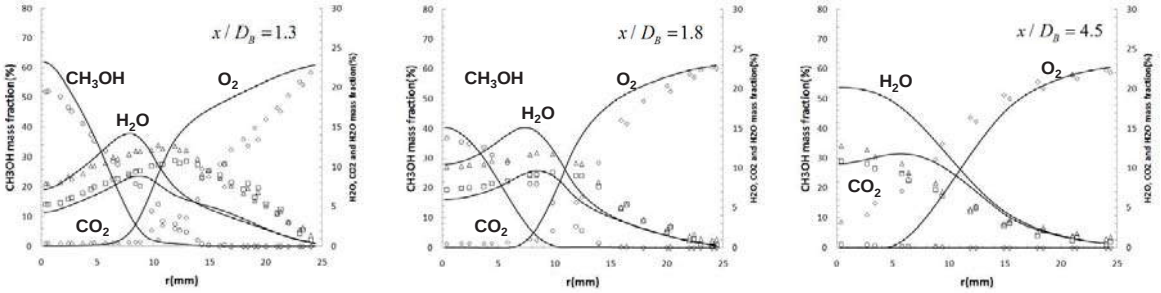
Combustion Laboratory POSTECH

Basic Flame - Bluff Body Flame

Results



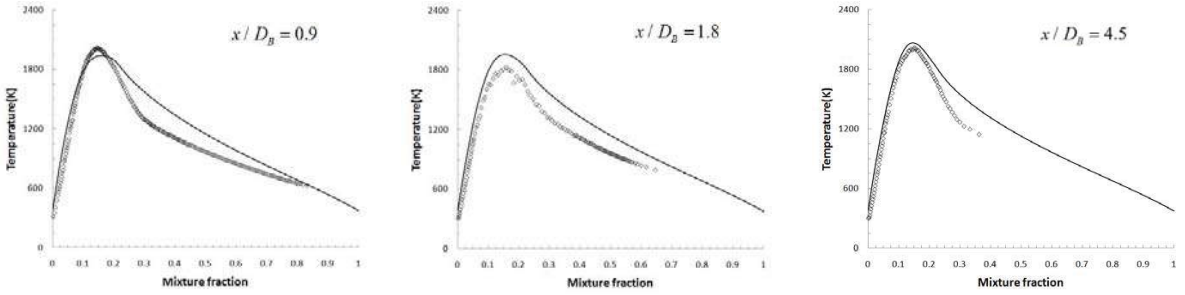
Radial distributions of the Favre mean temperature and OH mass fraction`



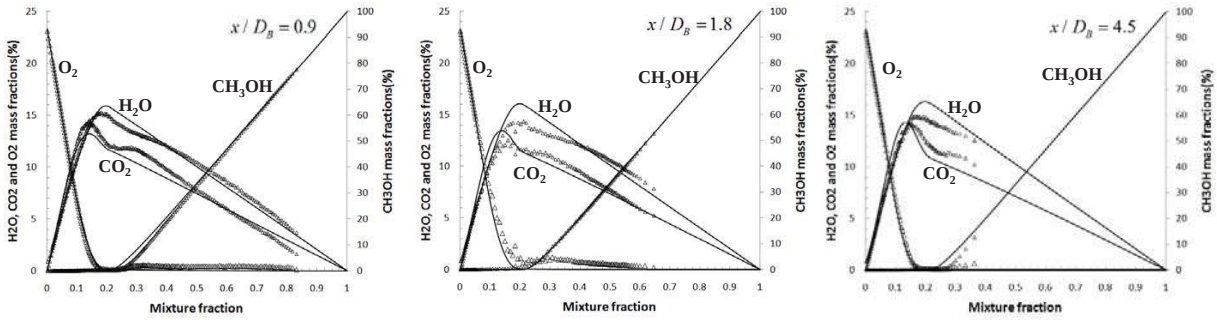
Radial distributions of the major species mass fractions

Basic Flame - Bluff Body Flame

Results



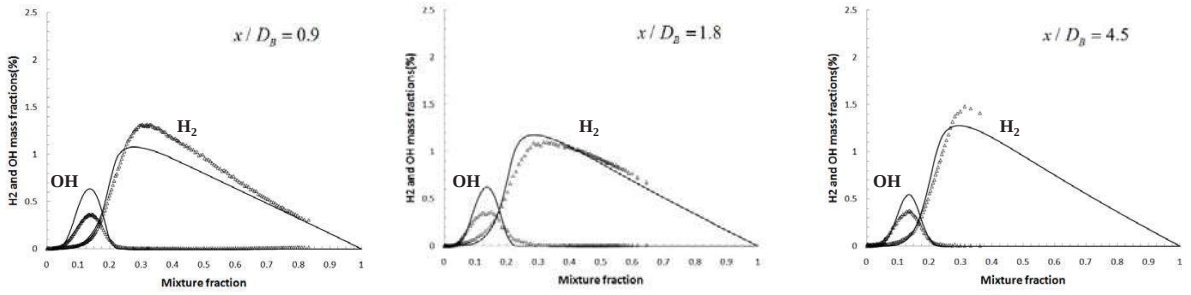
Conditional mean temperature with respect to the mixture fraction



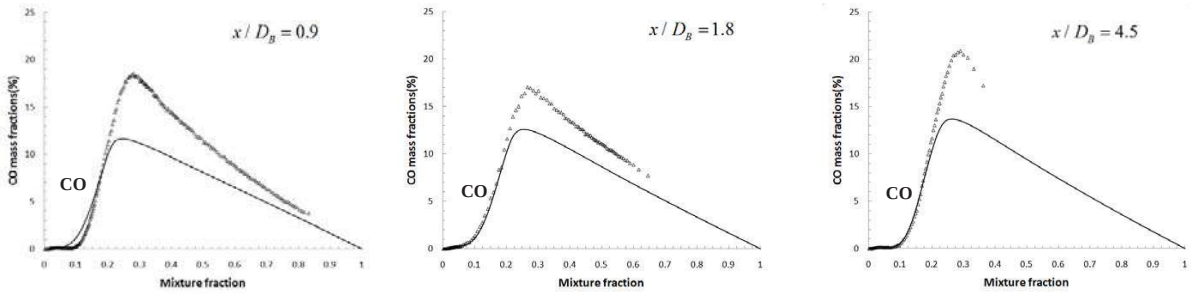
Major species mass fractions with respect to the mixture fraction

Basic Flame - Bluff Body Flame

Results



H_2 and OH mass fractions with respect to the mixture fraction



CO mass fraction with respect to the mixture fraction

Steady Laminar Flamelet Model

Model description

- Turbulent flame modeled as an ensemble of thin, laminar, locally 1-D flamelet structures
- Reacting scalars mapped from physical space to mixture fraction space

SLFM Library (Q_i, Q_h, T)

- Contains Q_i, Q_h and T distributions in mixture fraction space
- **Parameterized in terms of scalar dissipation rate (SDR)**
- Usually **pre-calculated** by in-house code or other tools (OpenFOAM?)

$$0 = \langle N|\eta\rangle \frac{\partial^2 Q_\eta}{\partial \eta^2} + \langle \dot{w}_\eta|\eta\rangle$$

PDF Library

- Contains probability density function i
- **Parameterized in terms of mixture fraction and its variance**
- **Pre-calculated** or calculate on the fly

$$\tilde{P}(\eta) = \frac{\zeta^{\alpha-1}(1-\zeta)^{\beta-1}}{\Gamma(\alpha)\Gamma(\beta)}\Gamma(\alpha+\beta)$$

- Integrate scalars (Yi, T, h …) from SLFM and PDF library, make 3D Look-up table (mf, mfVar, SDR) before calculation
- Solve transport eqns for mixture fraction and its variance
- Find and interpolate scalars from 3D Look-up table for given mixture fraction, variance and SDR
- Correct thermodynamic properties at local position

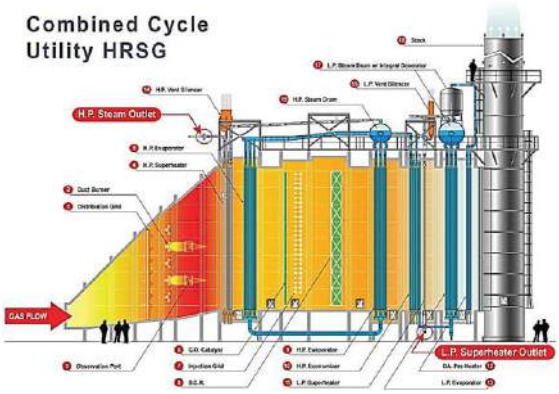
SLFMFoam

Industrial Furnace - Heat Recovery Steam Generator

Case description

- The HRSG is an energy recovery heat exchanger that recovers heat from a hot gas stream
  It produces steam that can be used in a process (cogeneration)
- The HRSG includes supplemental, or duct firing. These additional burners provide additional energy to the HRSG, which produces more steam and increases the output of the steam turbine
- Generally, duct firing provides electrical output at lower capital cost
  It is therefore often utilized for peaking operations

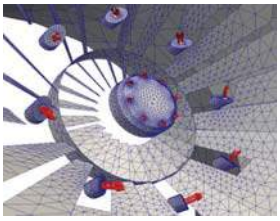
- **Main components of HRSG**
  - **Silencer**
    Attenuates noise level to meet government and site requirements
  - **Integral Deaerator**
    Uses low temperature heat to deaerate feed-water for improved thermal efficiency
  - **CO Catalyst**
    Reduces carbon monoxide in the flue gas
  - **Diverter Valve**
    Modulates steam production in the bypass systems
  - **SCR Catalyst**
    Reduces nitrous oxides in the flue gas
  - **Duct Burner**
    Provides supplementary firing of turbine exhaust to increase unfired steam production



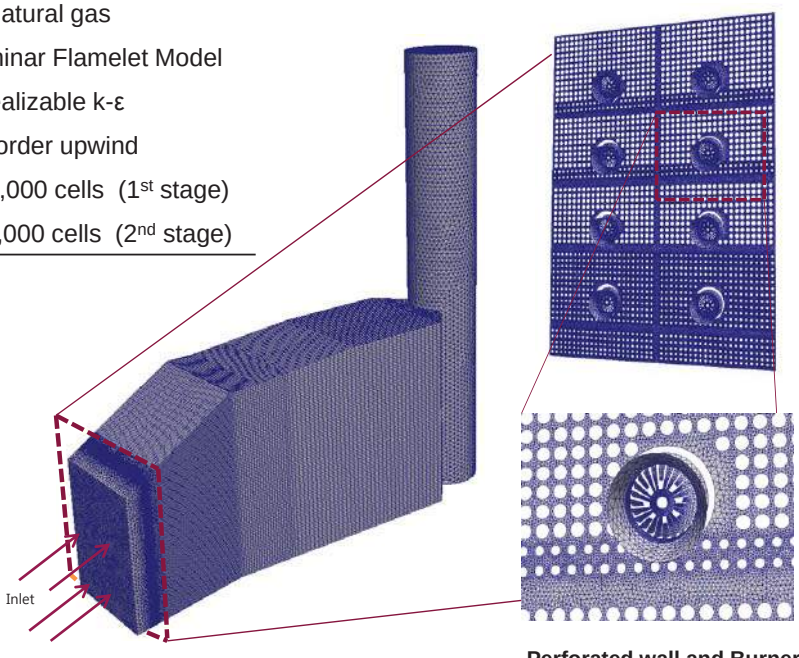
Industrial Furnace - Heat Recovery Steam Generator

Case description

Description	Specification
Fuel	Natural gas
Combustion model	Steady Laminar Flamelet Model
Turbulence model	Realizable k-ε
Discretization	2nd order upwind
Mesh	About 7,000,000 cells (1st stage) About 9,000,000 cells (2nd stage)



Fuel nozzle



Computational mesh

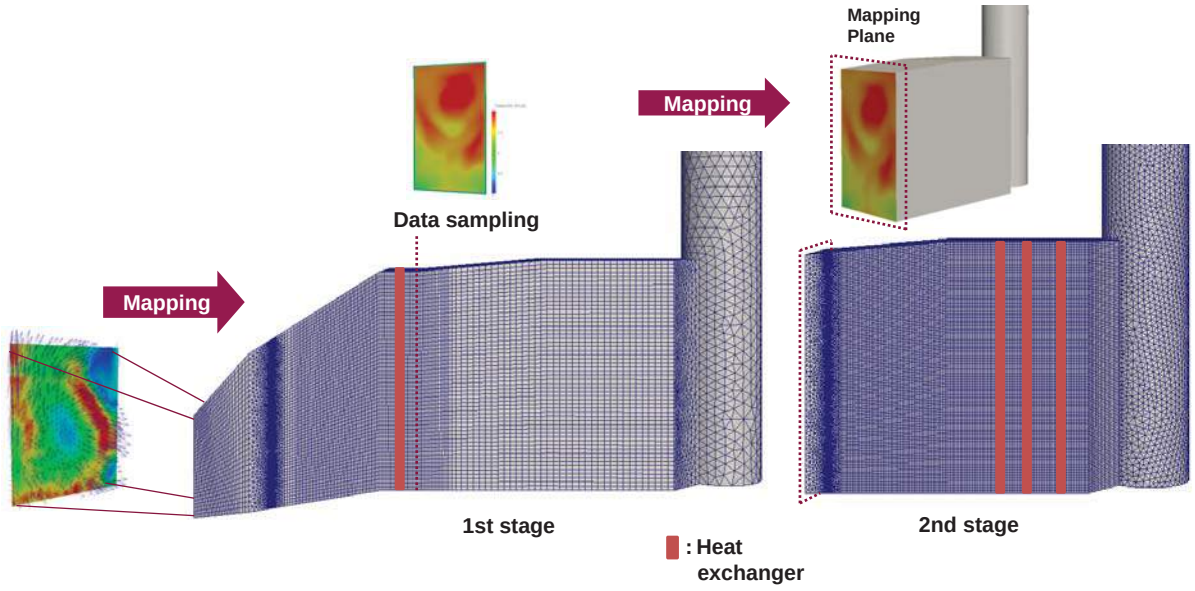
Perforated wall and Burner

Combustion Laboratory POSTECH

Industrial Furnace - Heat Recovery Steam Generator

Case description

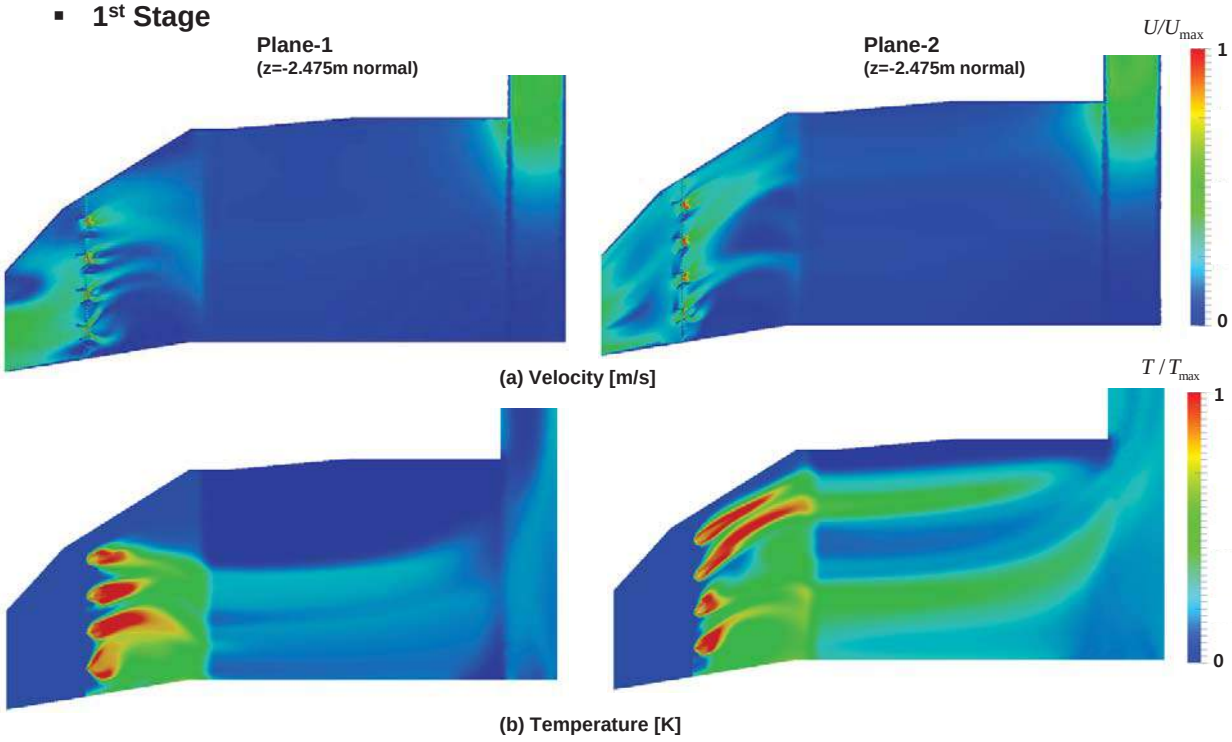
- It is very difficult to analyze a whole system of HRSG at a time
- A whole system is divided into two stages; 1st stage and 2nd stage
- A input of 2nd stage use sampled data from 1st stage result



Industrial Furnace - Heat Recovery Steam Generator

Results

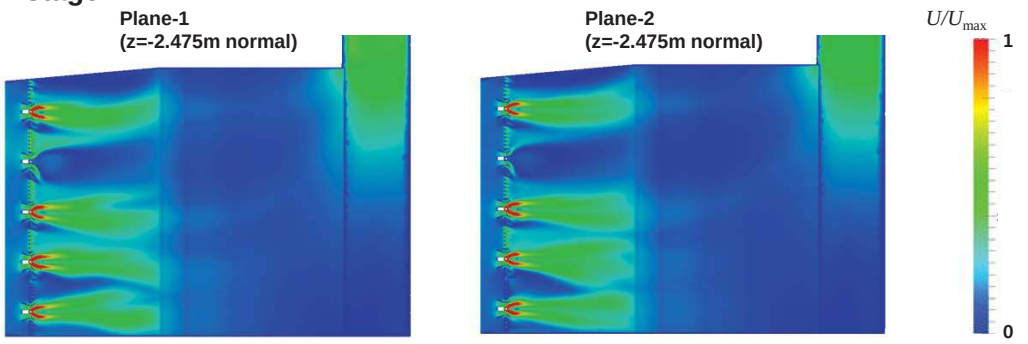
- 1st Stage



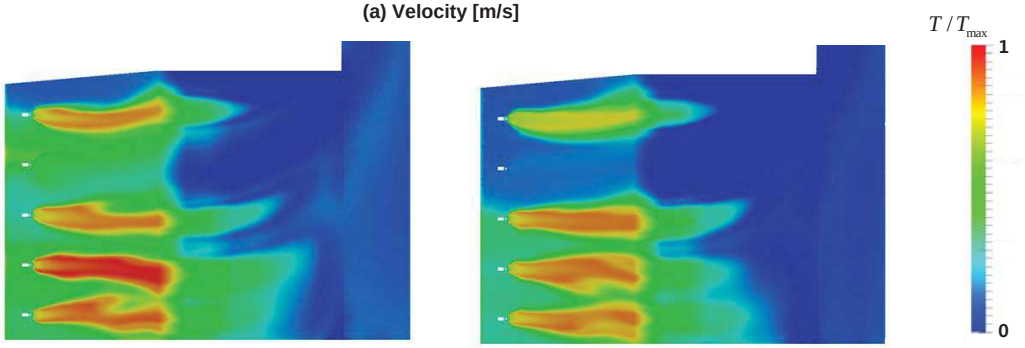
Industrial Furnace - Heat Recovery Steam Generator

Results

- **2nd stage**



(a) Velocity [m/s]



(b) Temperature [K]

Industrial Furnace - Heat Recovery Steam Generator

Results

- **Comparison with the result for FLUENT**

Case		Fluent	OpenFOAM	Measured / Performance	Ratio	
	Units	Outlet	Outlet	Outlet	Fluent	OpenFOAM
Temperature	K				0.96	1.12
Composition	mass fraction				0.94	0.92
					1.05	1.13
					1.07	0.99
					1.00	1.00
					0.99	0.97
	ppm vd @actual O2				0.49	0.75
					0.26	1.36

Turbulent Partially Premixed Flames

Modified Weller(FSD) Model

Model description

- **XiFoam (basic solver)**
  - Solid/fluid interface
  - Reasonable combustion model for partially premixed flame using Weller wrinkling factor(Xi)
  - Steady solver(transient calculation takes 10 times longer than steady calculation)

- **XiFlameletsFoam (advanced solver)**
  - Steady solver newly implemented combining modified Weller and LFM
  - Premixed process by modified Weller, nonpremixed by LFM
  - CO and NO prediction method

Algebraic equation for flame wrinkling factor

$$\Xi_{eq}^{*} = 1 + 0.62\sqrt{\frac{u'}{S_L}}R_\eta$$

$$\Xi_{eq} = 1 + 2\tilde{c}(\Xi_{eq}^{*} - 1)$$

Transport equation of progress variable

$$\frac{\partial \langle\rho\rangle \tilde{c}}{\partial t} + \frac{\partial \langle\rho\rangle \tilde{U}_i \tilde{c}}{\partial x_i} = \frac{\partial}{\partial x_i}\left(\Gamma_c \frac{\partial \tilde{c}}{\partial x_i}\right) + \tilde{\omega}_c + 2\frac{\Gamma_c}{(\tilde{Y}_{fu} - \tilde{Y}_{fb})}\frac{\partial \tilde{c}}{\partial x_i}\frac{\partial (\tilde{Y}_{fu} - \tilde{Y}_{fb})}{\partial x_i}$$

Modified for partially premixed combustion

$$\tilde{c} = \frac{\tilde{Y}_{Fu} - \tilde{Y}_F}{\tilde{Y}_{Fu} - \tilde{Y}_{Fb}} \qquad \tilde{\omega}_c = \rho_u S_L \Xi |\nabla c|$$

Unburned state

$$\tilde{\Phi}_u(\tilde{f}; \underline{x}, t) = \Phi_O + \tilde{f}(\Phi_F - \Phi_O)$$

Burned state

$$\tilde{\Phi}_b(\tilde{f}, \tilde{\chi}; \underline{x}, t) = \iint \Phi(f, \chi) P(f, \chi) df d\chi$$

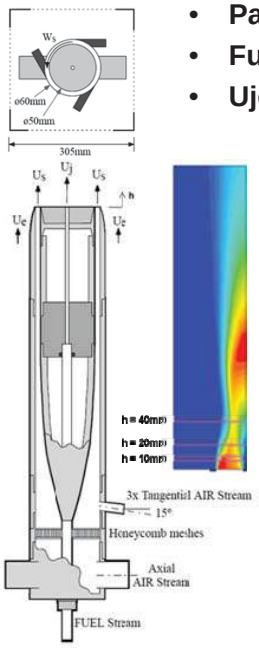
Mean conserved scalar

$$\tilde{\Phi}(\tilde{f}, \tilde{c}, \tilde{\chi}; \underline{x}, t) = (1 - \tilde{c})\tilde{\Phi}_u(\tilde{f}; \underline{x}, t) + \tilde{c}\tilde{\Phi}_b(\tilde{f}, \tilde{\chi}; \underline{x}, t)$$

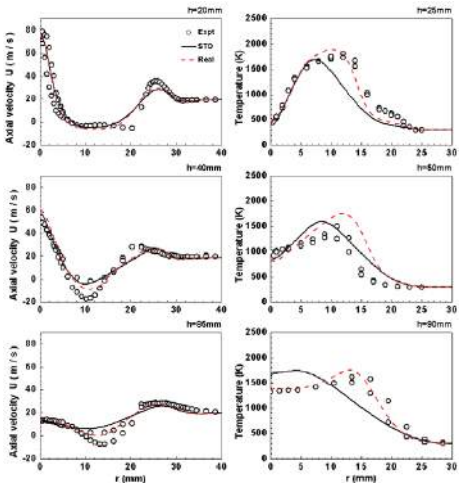
Basic Flame – Sydney Swirl Flame SMA1

Results

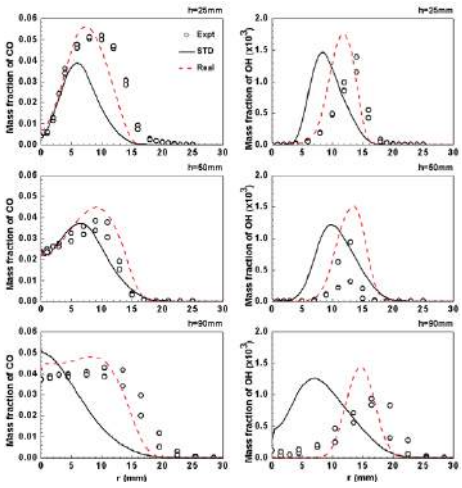
- **Partially premixed swirl flame SMA1 with XiFlameletsFoam**
- **Fuel = CH_4/air (1:2), Coflow = air**
- **Ujet = 66.3 m/s, Us = 32.9 m/s, Swirl ratio = 0.7**



Schematic of the burner



Axial velocity and temperature



Distributions of CO and OH

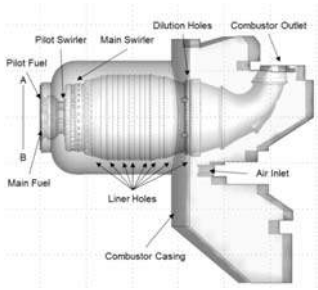
Industrial Furnace – 5MWe Micro Gas Turbine

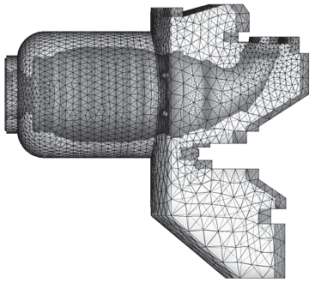
Case description

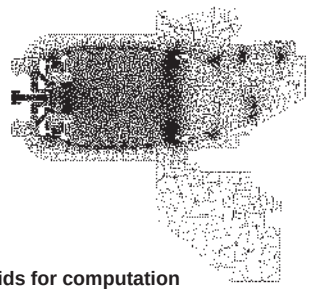
- Reverse-flow semi silo type with compressed air entering through pilot and main nozzles
- Two coaxial annular nozzles with radial swirlers
- Fuel is injected to be partially premixed with air in both nozzles

- **Numerical Method**
  - Pressure-velocity coupling based on SIMPLE algorithm
  - Gauss upwind scheme for spatial discretization of convection term
  - Mass flow inlet B.C with zero gradient pressure and fixed temperature
  - normalized residual of P < 10^{-3}, the other < 10^{-6} for convergence

- **Grid generation**
  - 14 MM tetrahedral cells converted by STAR_CCM+(5.04)



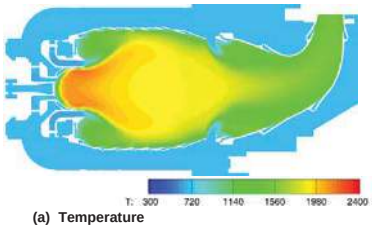




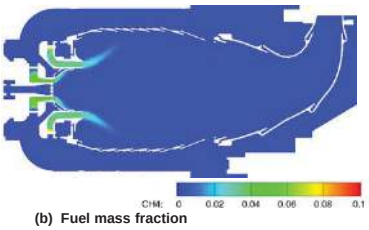
Grids for computation

Industrial Furnace – 5MWe Micro Gas Turbine

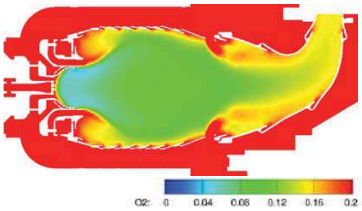
Results



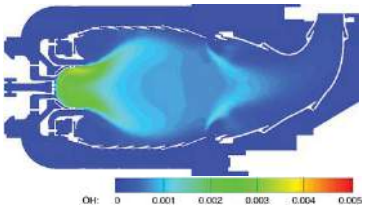
(a) Temperature



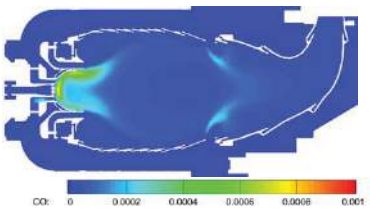
(b) Fuel mass fraction



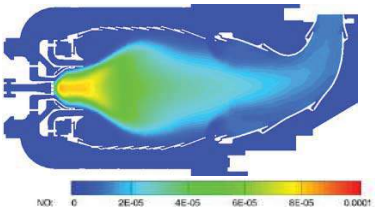
(c) O2 mass fraction



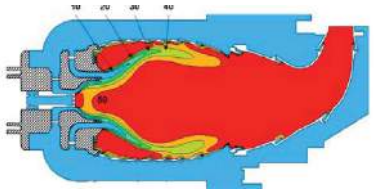
(d) OH mass fraction



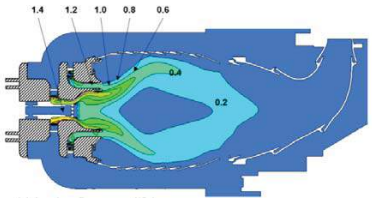
(e) CO mass fraction



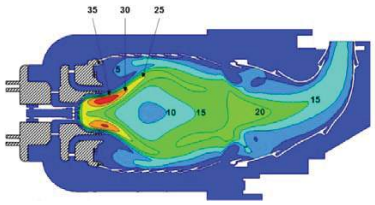
(f) NO mass fraction



(g) Wrinkling factor(Ξ)



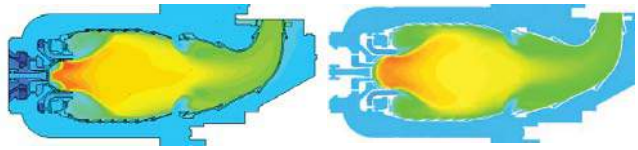
(h) Laminar flame speed(S_L)



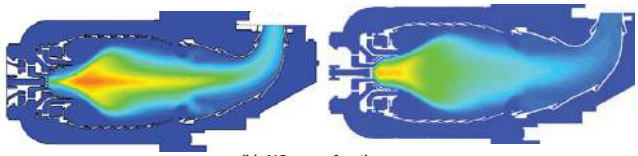
Industrial Furnace – 5MWe Micro Gas Turbine

Results

- **PCFM(STAR-CCM+) provides insight for complex reacting flow fields but**
  - **Non-premixed combustion region is calculated by equilibrium PPDF (Over-prediction of CO)**
  - **Thermal NO only considered**
- **Margin of Errors of XiFlameletsFoam(OpenFOAM)**
  - **T < 0.8%, O2 < 6%**
  - **reasonable order of degree for CO**



(a) Temperature



(b) NO mass fraction

PCFM (STAR-CCM+) **XiFlameletsFOAM (OpenFOAM)**

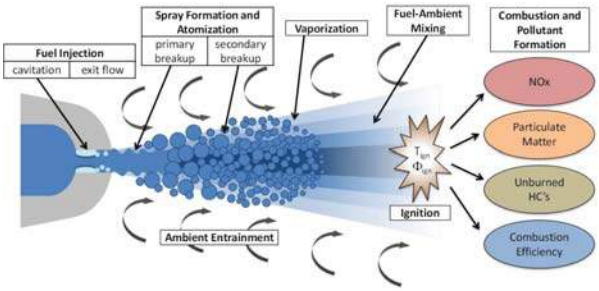
Specification	Measured	PCFM (STAR-CCM+)	XiFlamelet (OpenFOAM)
Temperature			
O_2			
CO (@15%O_2)			

Measured and calculated scalars at the outlet

Spray Combustion Modeling

Spray Combustion Modeling

CMC equation with spray



CMC $\frac{\partial Q_\eta}{\partial t} = \langle N|\eta\rangle \frac{\partial^2 Q_\eta}{\partial \eta^2} + \langle \dot{w}_\eta|\eta\rangle$

Mixture fraction $\xi = \frac{Z_i - Z_{i,oxi}}{Z_{i,fuel} - Z_{i,oxi}}$

Assumed beta-function PDF

$$\tilde{P}(\eta) = \frac{\zeta^{\alpha-1}(1-\zeta)^{\beta-1}}{\Gamma(\alpha)\Gamma(\beta)}\Gamma(\alpha+\beta)$$

where $\alpha=\tilde{\zeta}\gamma$, $\beta=(1-\tilde{\zeta})\gamma$, $\gamma = \frac{\tilde{\zeta}(1-\tilde{\zeta})}{\widetilde{\zeta''^2}}$

Mixture fraction $\frac{\partial(\bar{\rho}\tilde{\xi})}{\partial t} + \nabla\cdot(\bar{\rho}\tilde{\boldsymbol{u}}\tilde{\xi}) = \nabla\cdot\left[\frac{\mu_t}{Sc_{\tilde{\xi}}}\nabla\tilde{\xi}\right] + \bar{\rho}\tilde{s_\xi}$

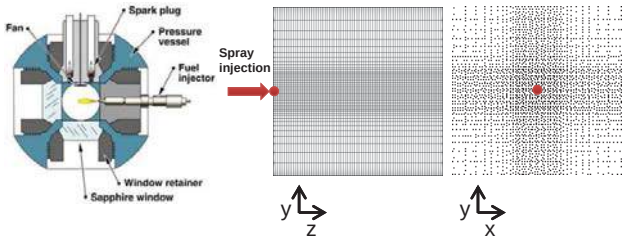
Mixture fraction variance $\frac{\partial(\bar{\rho}\widetilde{\xi''^2})}{\partial t} + \nabla\cdot(\bar{\rho}\tilde{\boldsymbol{u}}\widetilde{\xi''^2}) = \nabla\cdot\left[\frac{\mu_t}{Sc_{\widetilde{\xi''^2}}}\nabla\widetilde{\xi''^2}\right] + \frac{2\mu_t}{Sc_{\widetilde{\xi''^2}}}(\nabla\tilde{\xi})^2$

$$-2\bar{\rho}\widetilde{\xi''s_\xi}(1-\tilde{\xi}) - \bar{\rho}\widetilde{\xi''s_\xi} - \bar{\rho}\tilde{\chi}$$

ECN

Case description

- **Library of recent well-documented spray experiments**
- **Includes parametric variation of oxygen concentration, ambient temperature, ambient density, fuel type, fuel temperature, injection duration, etc.**
- **Website : http://public.ca.sandia.gov/ECN/**



Constant volume combustion chamber (Experimental setup and computational grid)

- **Injector Specification**

Parameters	Values
Injector type	Common-rail injector
Nozzle outlet diameter [mm]	0.1
Nozzle K factor	1.5
Nozzle shaping	Hydro-eroded
Discharge Coefficient	0.86
Fuel injection Pressure [MPa]	150

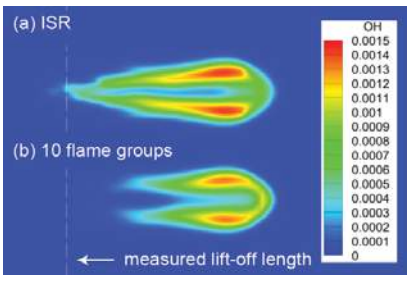
- **Simulation Cases (n-heptane spray)**

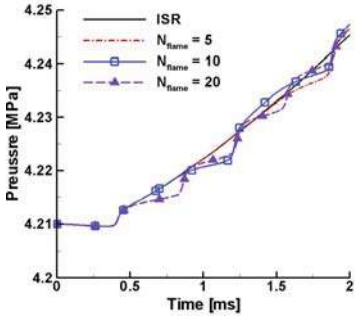
Case	O_2 [%]	Ambient Temperature [K]	Ambient Density [kg/m³]	Measured Lift-off Length [mm]
1	21	1000	14.8	17.0
2	21	1100	14.8	13.0
3	21	1200	14.8	10.0
4	15	1000	14.8	23.4
5	12	1000	14.8	29.2

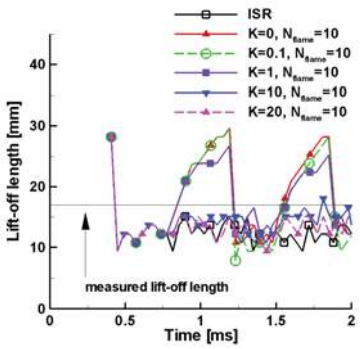
ECN

Results

- **Total fuel injected is divided into the given number of groups of equal mass according to evaporation sequence.**
- **The pressure shows a regular pattern of oscillation with abrupt rise at the moment of ignition of sequential flame groups.**
- **Without flame group interaction newly introduced flame groups undergo their own ignition delay period and sequential independent auto-ignition, even though there exist neighboring flame groups already ignited at the same location**

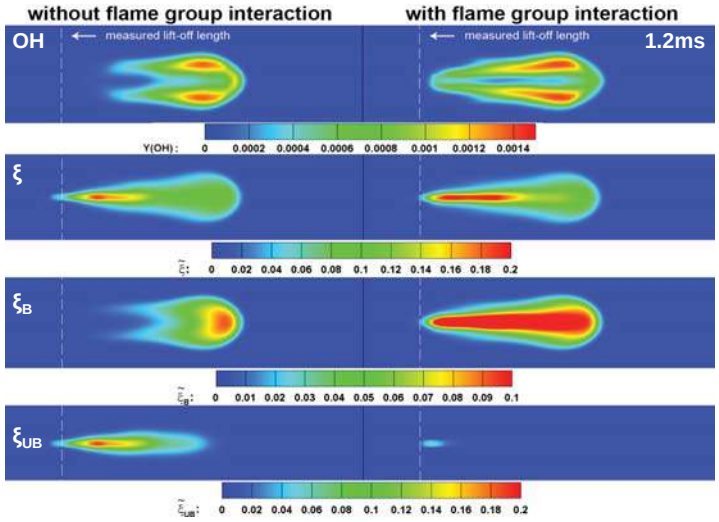






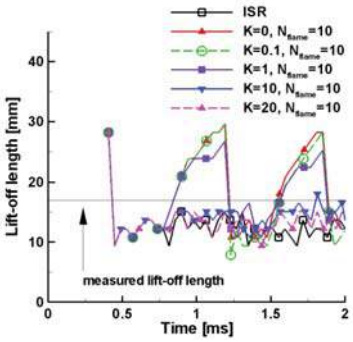
ECN

Results



$$\frac{\partial\left(\bar{\rho}\tilde{\xi}_j\right)}{\partial t}+\nabla\cdot\left(\rho\tilde{\mathbf{v}}\tilde{\xi}_j\right)=\nabla\cdot\left[\frac{\mu_t}{Sc_{\tilde{\xi}_j}}\nabla\tilde{\xi}_j\right]+\bar{\rho}\tilde{s}_{\xi,j}+\bar{\rho}\tilde{\xi}\tilde{K}\frac{(1-\tilde{c})}{\tau_t}$$ for the burned state

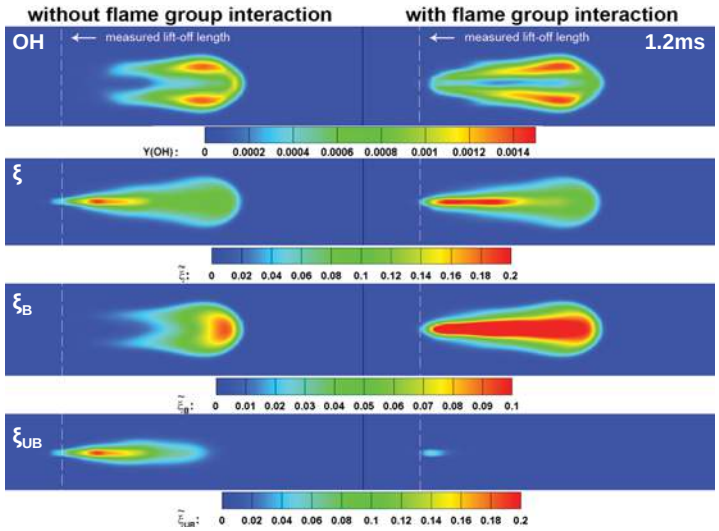
$$\frac{\partial\left(\bar{\rho}\tilde{\xi}_j\right)}{\partial t}+\nabla\cdot\left(\rho\tilde{\mathbf{v}}\tilde{\xi}_j\right)=\nabla\cdot\left[\frac{\mu_t}{Sc_{\tilde{\xi}_j}}\nabla\tilde{\xi}_j\right]+\bar{\rho}\tilde{s}_{\xi,j}-\bar{\rho}\tilde{\xi}\tilde{K}\frac{\tilde{c}}{\tau_t}$$ for the unburned state



Case	Lift-off length [mm]
Measured	17.0
ISR	12.2
K=0 (without flame group interaction)	7.9 / 29.6
K=0.1	7.9 / 29.6
K=1	9.3 / 25.2
K=10	15.1
K=20	13.6

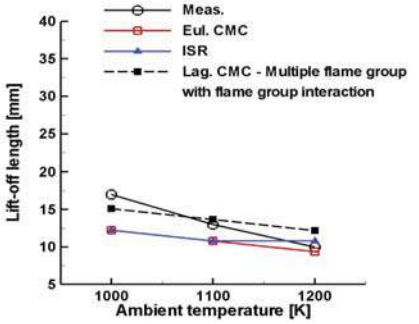
ECN

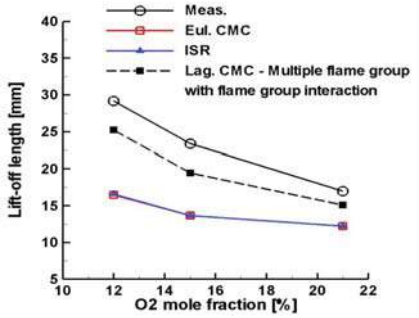
Results



$$\frac{\partial\left(\bar{\rho}\tilde{\xi}_j\right)}{\partial t}+\nabla\cdot\left(\rho\tilde{\mathbf{v}}\tilde{\xi}_j\right)=\nabla\cdot\left[\frac{\mu_t}{Sc_{\tilde{\xi}_j}}\nabla\tilde{\xi}_j\right]+\bar{\rho}\tilde{s}_{\xi,j}+\bar{\rho}\tilde{\xi}\tilde{K}\frac{(1-\tilde{c})}{\tau_t}$$ for the burned state

$$\frac{\partial\left(\bar{\rho}\tilde{\xi}_j\right)}{\partial t}+\nabla\cdot\left(\rho\tilde{\mathbf{v}}\tilde{\xi}_j\right)=\nabla\cdot\left[\frac{\mu_t}{Sc_{\tilde{\xi}_j}}\nabla\tilde{\xi}_j\right]+\bar{\rho}\tilde{s}_{\xi,j}-\bar{\rho}\tilde{\xi}\tilde{K}\frac{\tilde{c}}{\tau_t}$$ for the unburned state

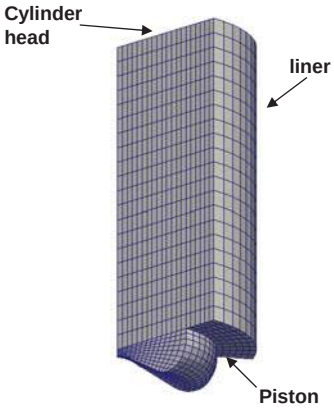




Diesel Engine – ERC

Case description

- **Geometry**



- ✓ **Fuel spray**
  Initial droplet size is determined by Rosin-Rammler distribution function with the SMD of 14 micron
- ✓ **Injected fuel temp : 311K**
- ✓ **Skeletal mechanism for n-heptane 44 species and 114 elementary steps**

- **Engine specification**

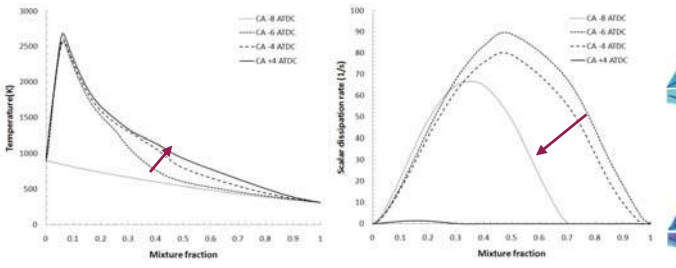
Description	Specification
Engine	Caterpillar 3401E
Engine speed (rpm)	821
Bore (mm) x Stroke (mm)	137.2 x 165.1
Compression ratio	16.1
Displacement (Liters)	2.44
Combustion chamber geometry	In-piston Mexican Hat with sharp edged crater
Max injection pressure (MPa)	190
Number of nozzle	6
Nozzle hole diameter (mm)	0.214
Spray angle (deg)	125

- **Operating condition**

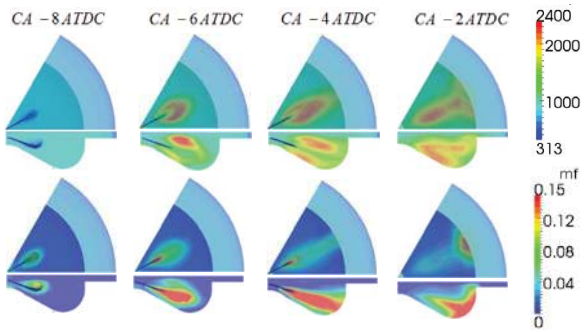
Operating conditions	
EGR level (%)	7, 27, 40
SOI timings (ATDC)	-20, -15, -10, -5 , 0, 5
Injection duration (deg)	6.5

Diesel Engine – ERC

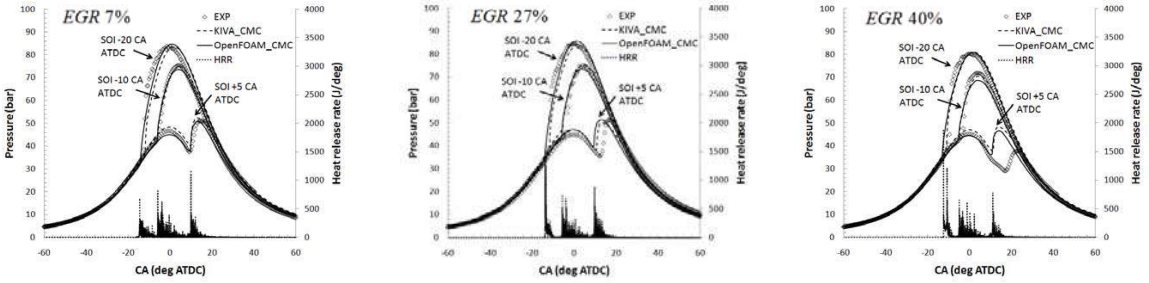
Results



Conditional mean temperature and scalar dissipation rate with respect to the mixture fraction



Spatial distributions of the temperature and mean mixture fraction



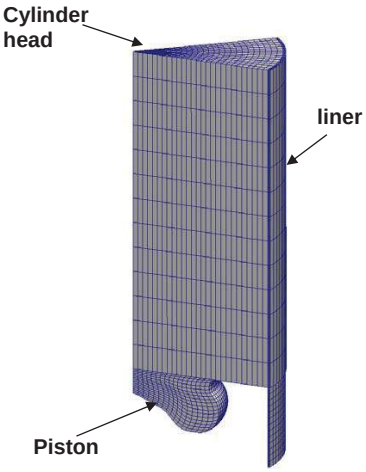
Pressure trace w.r.t different EGR (%)

Combustion Laboratory POSTECH

Diesel Engine – D1

Case description

- **Geometry**



3-D sector mesh of 45° with periodic boundary condition

- ✓ **D1 diesel engine**
- ✓ **Multiple fuel injection**

- **Engine specification**

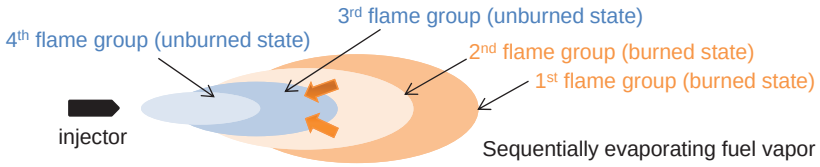
Description	Specification
Engine	D1
Engine speed (rpm)	
Bore (mm) x Stroke (mm)	
Compression ratio	
Displacement (cm^3)	
Max injection pressure (MPa)	
Number of nozzle	
Nozzle hole diameter (mm)	

- **Operating condition**

Operating conditions
SOI timings (ATDC)
Injection duration (deg)

Diesel Engine - D1

Case description



- **Flame group interaction is modeled as propagating premixed combustion by the EBU model**
- **The mean reaction progress variable is defined as**

$$\tilde{c} \equiv \tilde{\xi}_B / \tilde{\xi} \quad \text{where} \quad \tilde{\xi}_B + \tilde{\xi}_{UB} = \tilde{\xi} \quad \text{and} \quad (1-\tilde{c})\tilde{\xi} = \sum_{all\ j\ unburned} \tilde{\xi}_j$$

- **The source term for $\tilde{c}$ is given by the EBU model for premixed combustion as**

$$\tilde{w}_c = K \frac{\tilde{c}(1-\tilde{c})}{\tau_t} \quad \text{where integral time scale} \quad \tau_t = \tilde{k}/\tilde{\varepsilon}$$

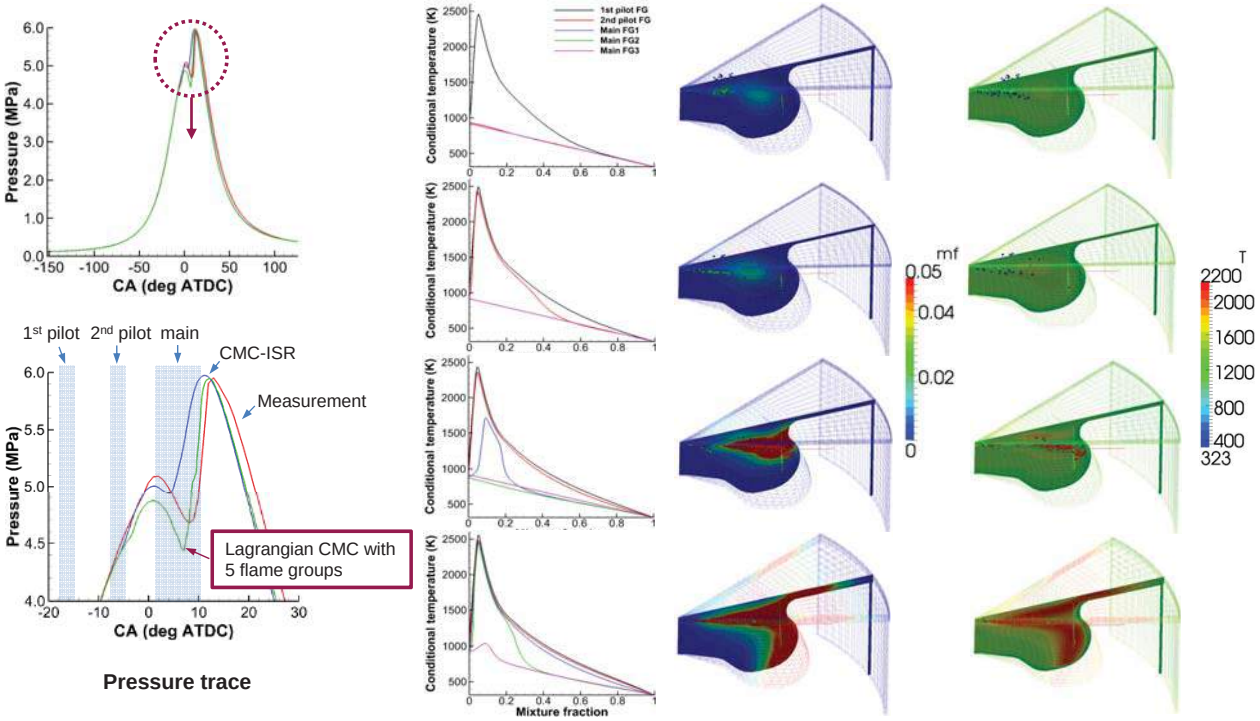
- **Transport equation for fuel fraction of the j-th fuel group**

$$\frac{\partial(\bar{\rho}\tilde{\xi}_j)}{\partial t} + \nabla\cdot(\rho\tilde{\mathbf{v}}\tilde{\xi}_j) = \nabla\cdot\left[\frac{\mu_t}{Sc_{\tilde{\xi}_j}}\nabla\tilde{\xi}_j\right] + \bar{\rho}\tilde{s}_{\xi,j} + \bar{\rho}\tilde{\xi}_j K\frac{(1-\tilde{c})}{\tau_t} \quad \text{for flame groups in the burned state}$$

$$\frac{\partial(\bar{\rho}\tilde{\xi}_j)}{\partial t} + \nabla\cdot(\rho\tilde{\mathbf{v}}\tilde{\xi}_j) = \nabla\cdot\left[\frac{\mu_t}{Sc_{\tilde{\xi}_j}}\nabla\tilde{\xi}_j\right] + \bar{\rho}\tilde{s}_{\xi,j} - \bar{\rho}\tilde{\xi}_j K\frac{\tilde{c}}{\tau_t} \quad \text{for flame groups in the unburned state}$$

Diesel Engine - D1

Results

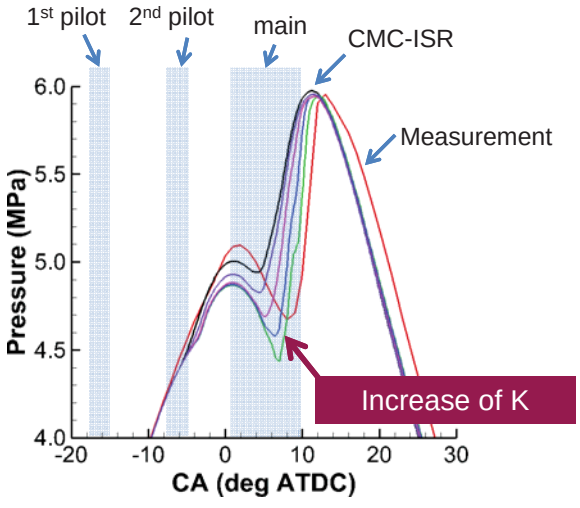


Pressure trace

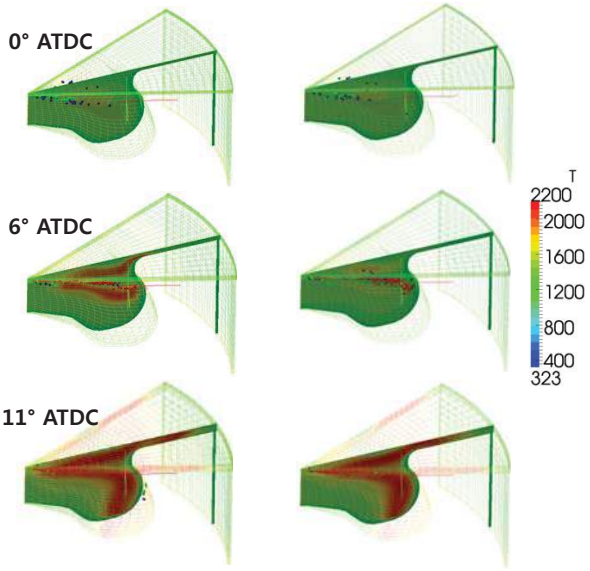
Diesel Engine - D1

Results

- **The first peak pressure is increased by higher constant K which is corresponding to intensive flame propagation between flame groups**



Pressure trace w.r.t different EBU constant K
(K = 0.1 / 1 / 2 / 5 / 20)



Spatial distributions of temperature w.r.t different K
(left K = 5, right K = 0.1)

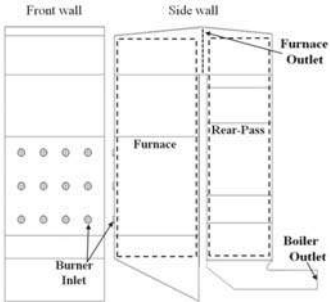
Heavy Oil Furnace - Full Scale

Case description

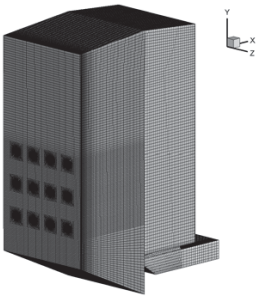
- Fuel (Heavy fuel oil) is injected from 12 burners into a furnace
- Computational domain covers from downstream of burner swirlers to the boiler outlet
- Incoming flow at each burner inlet has the swirl number

- **Numerical Method and Models**
  - Pressure-velocity coupling based on SIMPLE algorithm
  - Gauss upwind scheme for spatial discretization of convection term
  - k-ε model is employed with the wall function method
  - Fuel burning rate is given by the EDM (Eddy-dissipation Model)

- **Grid generation**
  - Hexahedral structured mesh with about 4 million elements for RANS simulation

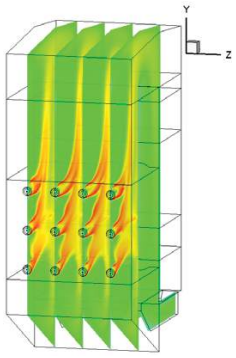


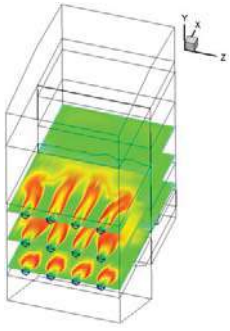
Schematic of the oil boiler

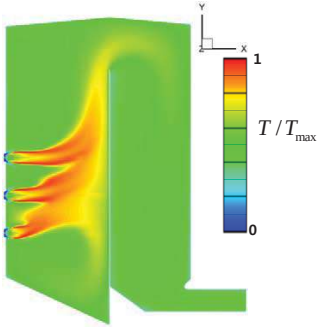

Grids for computation

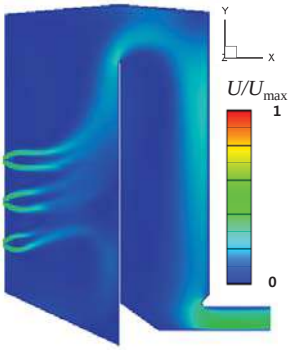
Heavy Oil Furnace - Full Scale

Results

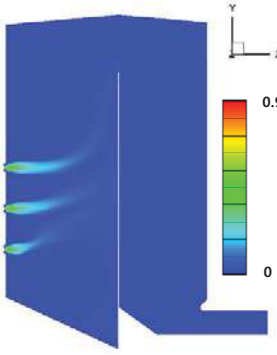



Temperature

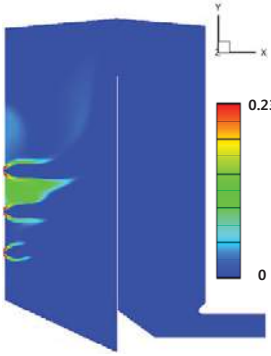




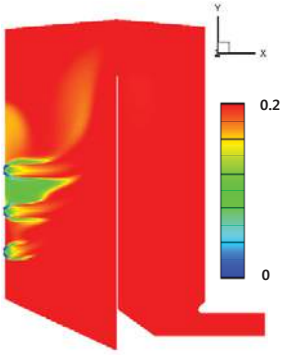
Velocity magnitude



Fuel mass fraction



O_2 mass fraction



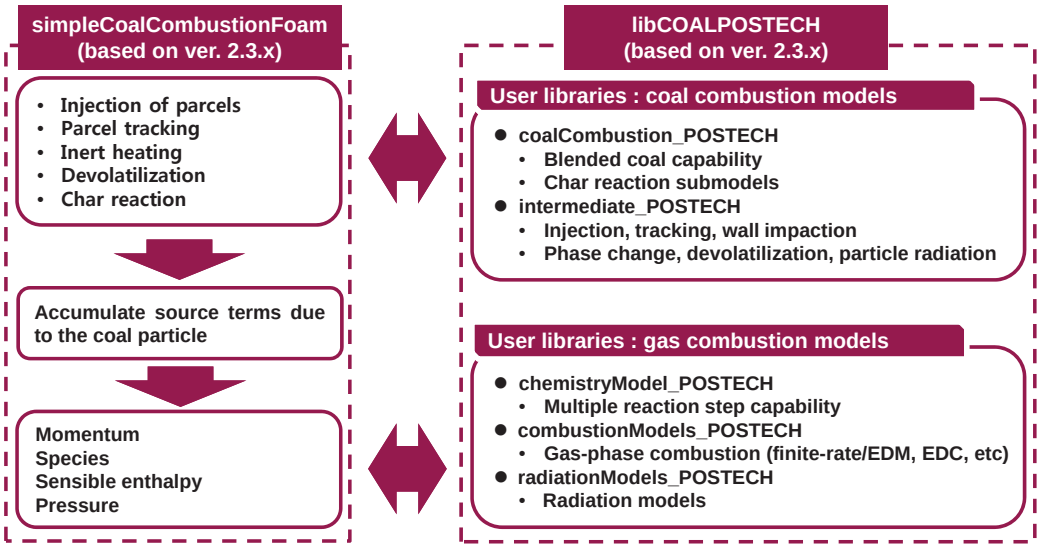
CO_2 mass fraction

Solid Combustion Modeling

Solid Combustion Modeling

simpleCoalCombustionFoam

- Steady state blended or single coal combustion solver
- Developed based on OpenFOAM ver. 2.3.x
- Includes various improved devolatilization, char surface reaction and gas combustion models

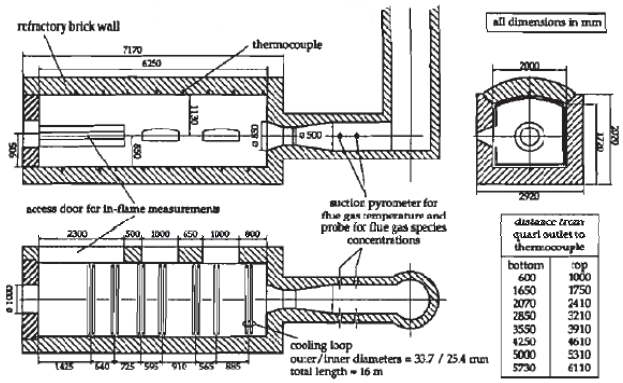


The schematic diagram of the simpleCoalCombustionFoam

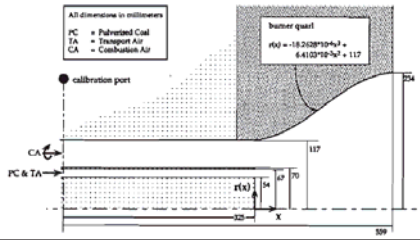
IFRF MMF 5-2 Flame

Case description

- Square cross section 2 m x 2 m, 6.25 m long with 7 cooling loops
- Measurement location(250, 500, 850, 1250, 1950 mm – V, T, O2, CO2, CO, NO)
- Air-staged burner(coal+transport air, combustion air)
- Saar hvBb coal
- Operating pressure : 1 atm
- Coal mass flow rate : 263 kg/h for 2.165 MW
- 22% Excess air
- Swirl number of combustion air(momentum ratio) : 0.923



R. Webber et al., IFRF Doc. No.F36/y/20.

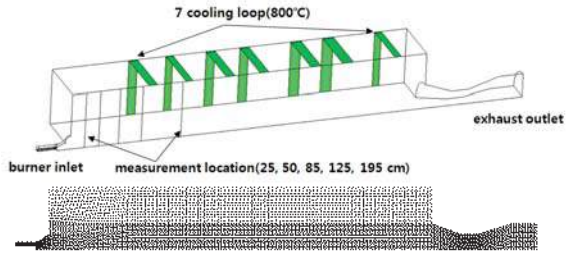


	Mass flow rate(kg/h)	V(m/s)		T(℃)
Pulverized Coal	263	23		70
Transport Air	421	23		70
Combustion Air	2670	Axial 43.9	Tan 49.4	300

IFRF MMF 5-2 Flame

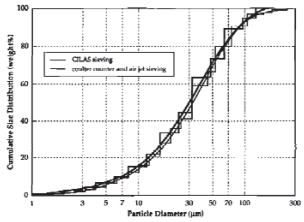
Case description

- Coal HCV(J/kg) = 2.79e+07 (AR)
- Vaporization T(K) = 773
- VM molecular weight(kg/kmol) = 45.6
- P1 radiation (α : 0.1, $E_{particle}$ = 0.8, $S_{particle}$ = 0.5)
- Standard k-e model with 1st order upwind
- Stochastic particle dispersion
- 2-step eddy dissipation
- ¼ sector periodic B.C, 50,000 structured cells



Proximate analysis (weight %)	
VM	37
Fixed Carbon	52.5
Ash	8.5
Moisture	2
Ultimate analysis (weight %, daf)	
C	79.3
H	4.7
O	13.7
N	2.3

Coal particle property	
Density(kg/m³)	1101
Cp(J/kg·K)	1990
Min size(μm)	10
Max size(μm)	150
Mean size(μm)	60
Spread parameter	1.13
Swelling Index	1
Vaporization T(℃)	500
Kinetics limited, A	6.7
Kinetics limited, Ea(MJ/kmol)	113.82
Single rate, A	2e+05
Single rate, Ea(J/kmol)	7.4e+07



Rosin-Rammler Distribution

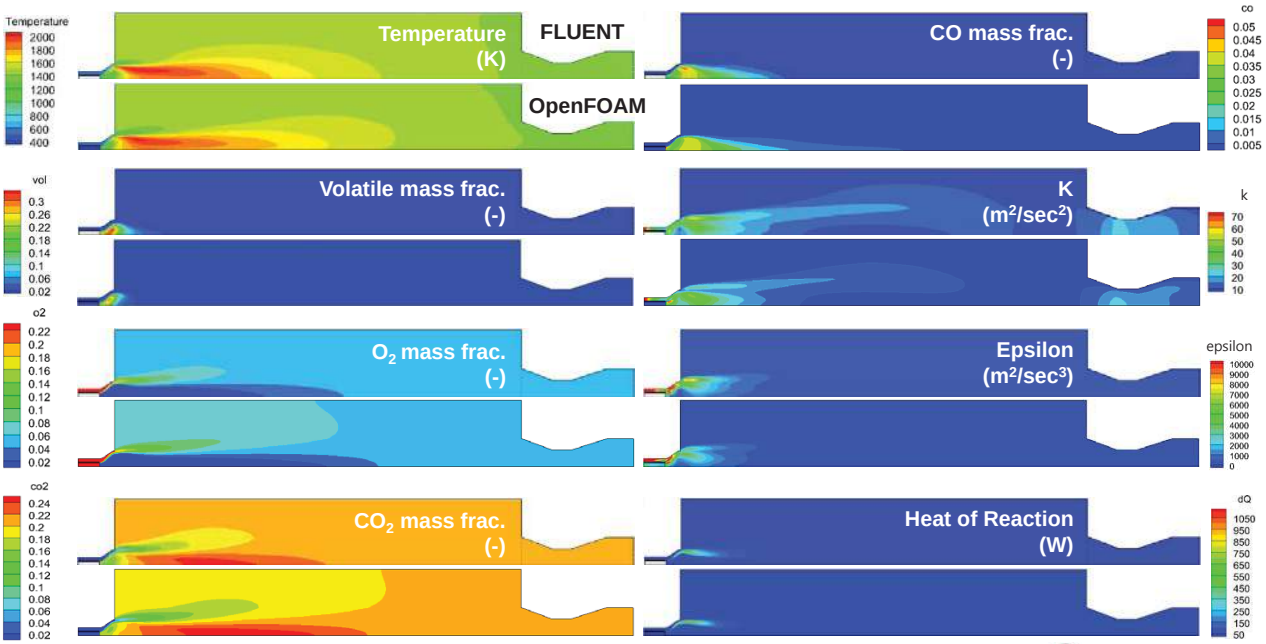
kinetic/diffusion limited char oxidation (product is assumed as CO)

single rate devolatilization

IFRF MMF 5-2 Flame

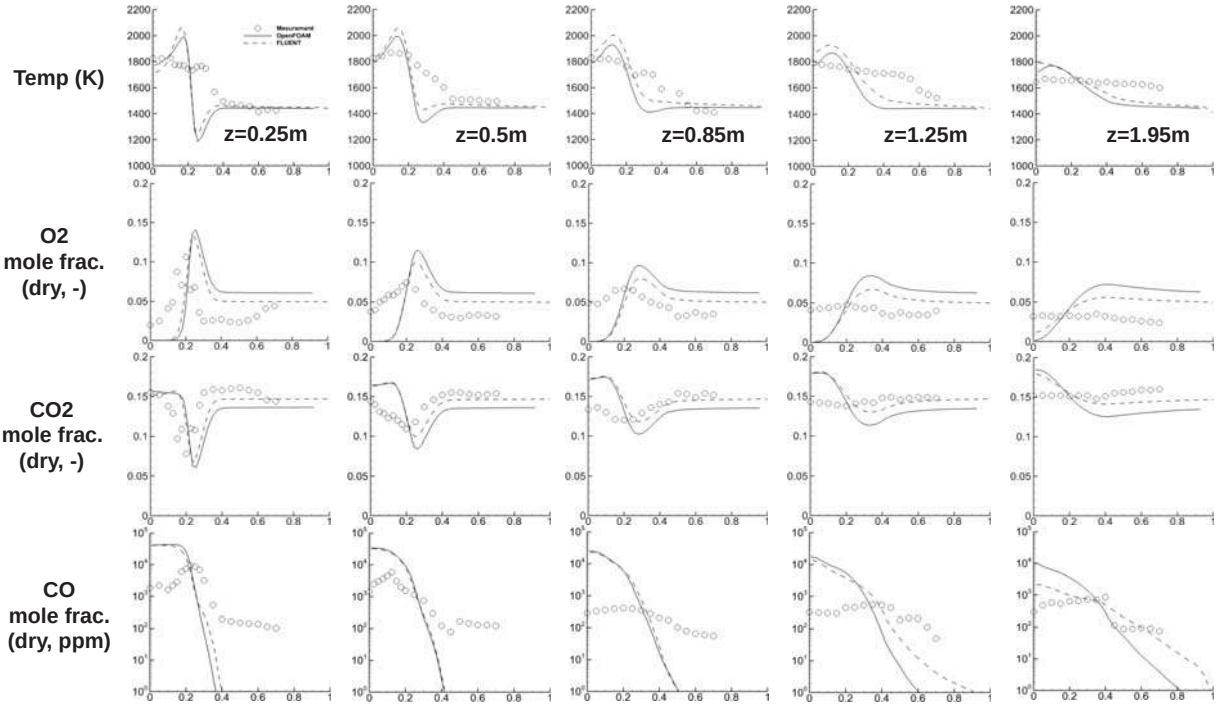
Results

- Single step kinetics devolatilization with 2-step EDM
- Kinetic diffusion limited model (also called Field's model)
- Radiation is considered by P1 model



IFRF MMF 5-2 Flame

Results



500MWe Tangentially fired pulverized-coal boiler

Case description

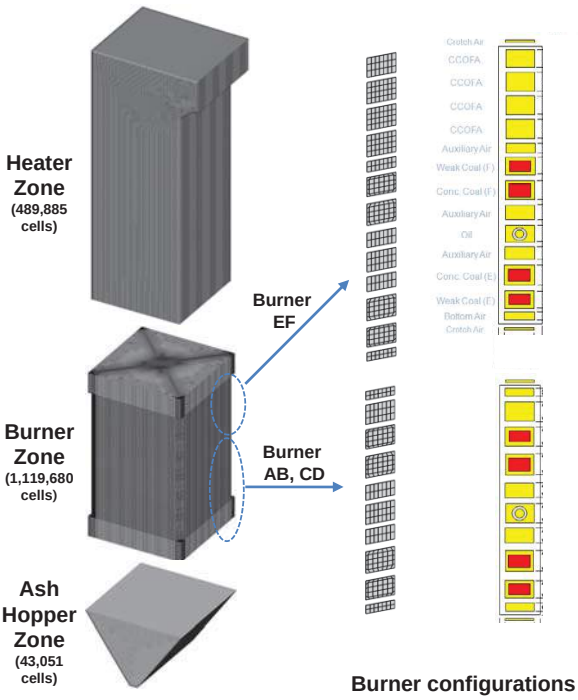
- 500MWe tangetially fired pulverized coal boiler
  - Coal HCV(J/kg) = 2.68e+07 (AR)
  - VM molecular weight(kg/kmol) = 41.3
  - P1 radiation (α : 0.1, $E_{particle}$ = 0.8, $S_{particle}$ = 0.1)
  - Standard k-e model with 1st order upwind
  - Stochastic particle dispersion
  - 2-step eddy dissipation model

Proximate analysis (weight %)	
VM	28
Fixed Carbon	52
Ash	15
Moisture	5
Ultimate analysis (weight %, daf)	
C	82
H	5.1
O	10.3
N	1.7
S	0.9

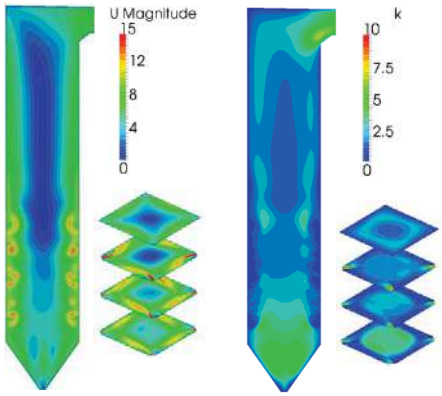
Coal particle property	
Density(kg/m^3)	1184
Cp(J/kg·K)	1960
Min size(μm)	10
Max size(μm)	150
Mean size(μm)	60
Spread parameter	1.13
Swelling Index	1
Vaporization T(℃)	500
Kinetics limited, A	6.7
Kinetics limited, Ea(MJ/kmol)	113.82
Single rate, A	3.6e+05
Single rate, Ea(J/kmol)	3.8e+07

500MWe Tangentially fired pulverized-coal boiler

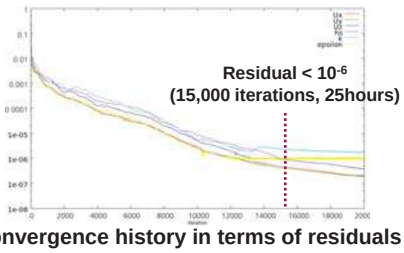
Results



Burner configurations



Distributions of velocity magnitude and turbulent kinetic energy



Convergence history in terms of residuals

Combustion Laboratory POSTECH

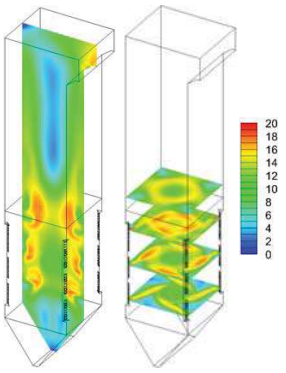
500MWe Tangentially fired pulverized-coal boiler

Results

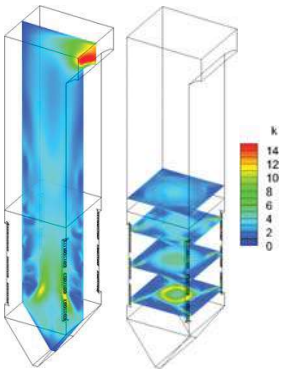
Heater Zone (489,885 cells)

Burner Zone (1,119,680 cells)

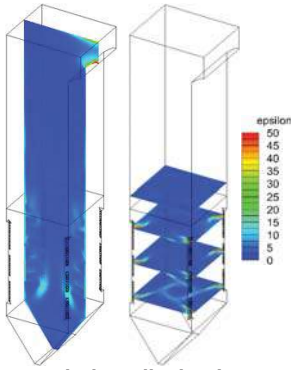
Ash Hopper Zone (43,051 cells)



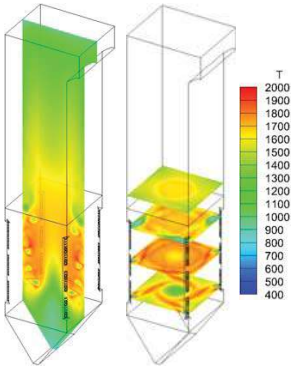
Velocity magnitude



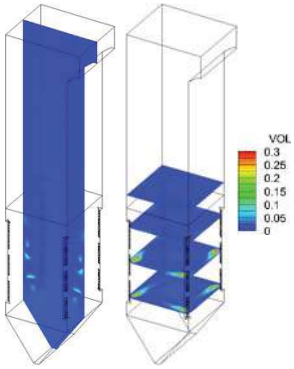
Turbulent kinetic energy



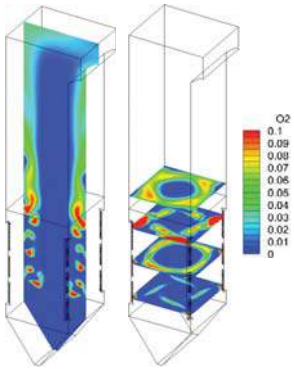
Turbulent dissipation rate



Temperature



Volatile mass fraction



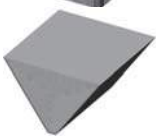
O_2 mass fraction

500MWe Tangentially fired pulverized-coal boiler

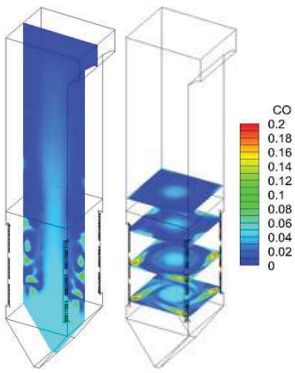
Results

Heater Zone (489,885 cells)

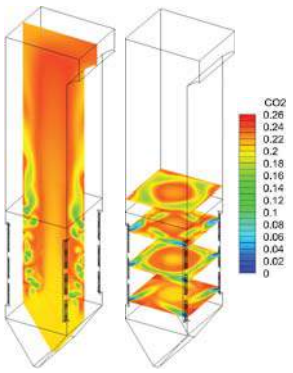
Burner Zone (1,119,680 cells)



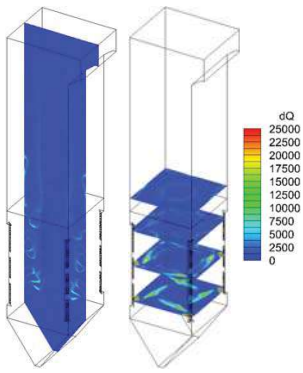
Ash Hopper Zone (43,051 cells)



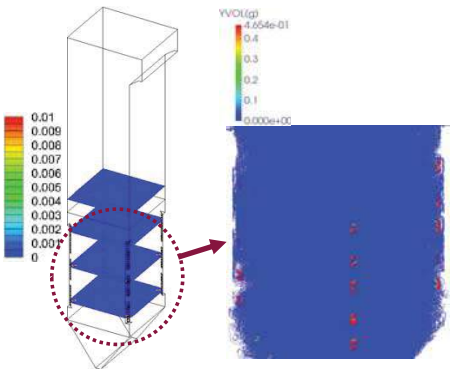
CO mass fraction



CO2 mass fraction

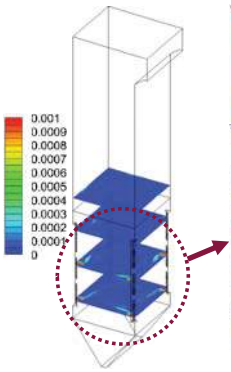


Heat release rate

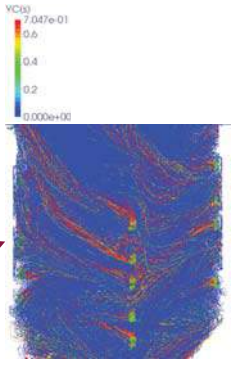


Devolatilization rate

Particle volatile mass frac.



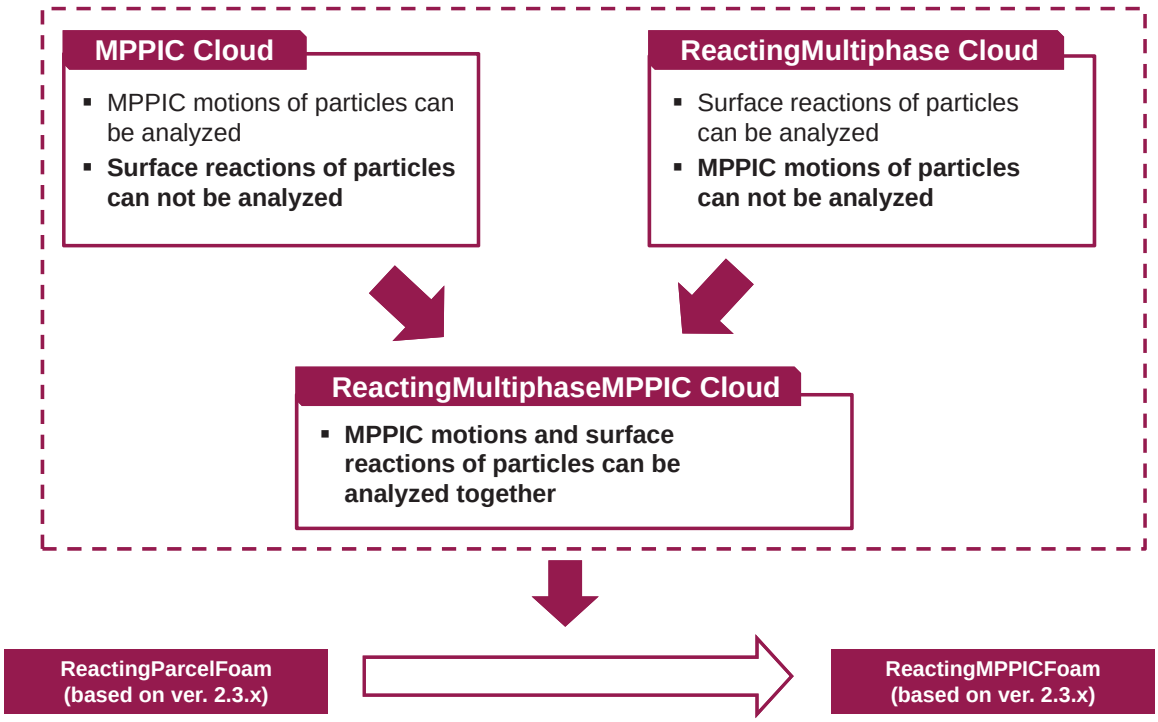
Char burning rate



Particle char mass frac.

Material Processing Furnace

ReactingMPPICFoam

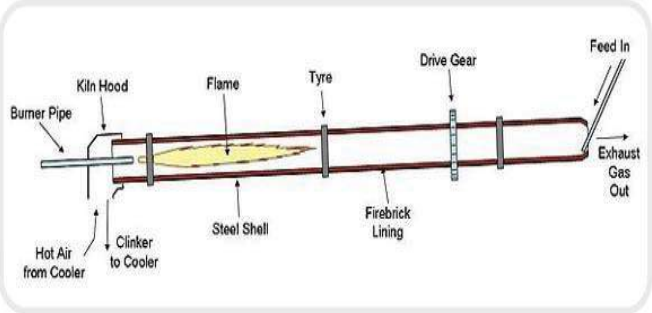


Rotary Klin

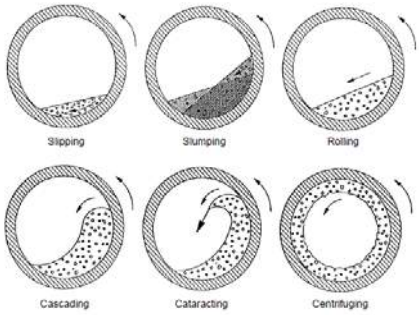
Case description

- A rotary kiln is fundamentally a heat exchanger from which energy from a hot gas phase is transferred to the bed material
- Carry out a wide range of operations such as reduction of oxide ore, reclamation of hydrated lime, calcination of petroleum coke and reclamation of hazardous waste
- Material within the kiln is heated to high temperature so that chemical reactions can take place

- Major Features
  - Material motion through the cylindrical workspace
  - Mass transfer between gas and solid phase
  - Heat transfer and chemical reaction



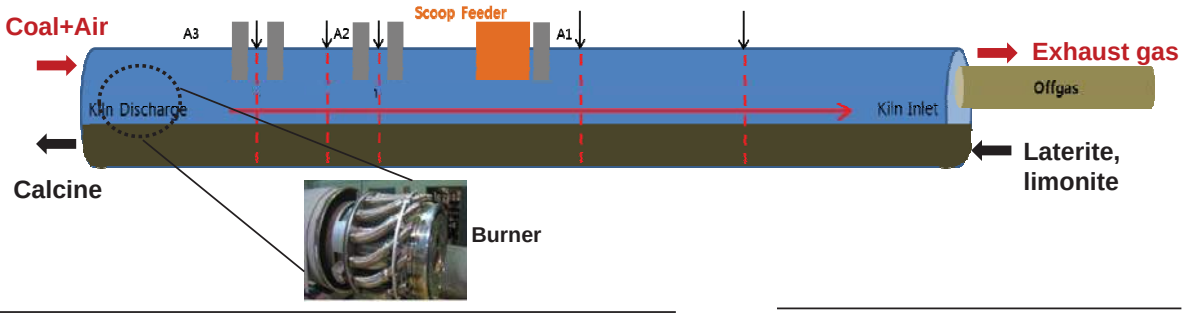
A Schematic diagram of direct heated rotary kiln



Different modes of operation in the transversal mixing plane

Rotary Klin

Case description



Description		Specification
Bed type		Moving Bed
Solid material		Laterite or limonite or Iron ore + coal
Heat source		Pulverized Coal coal burner
Physical change		Ore properties
Reaction	Solid-gas	Char reaction / Ore reduction
	Gaseous	Volatile combustion
Heat transfer		Conduction / Convection / Radiation Heat exchange among solid phases Heat transfer between gas-solid-wall

Characteristics of rotary kiln

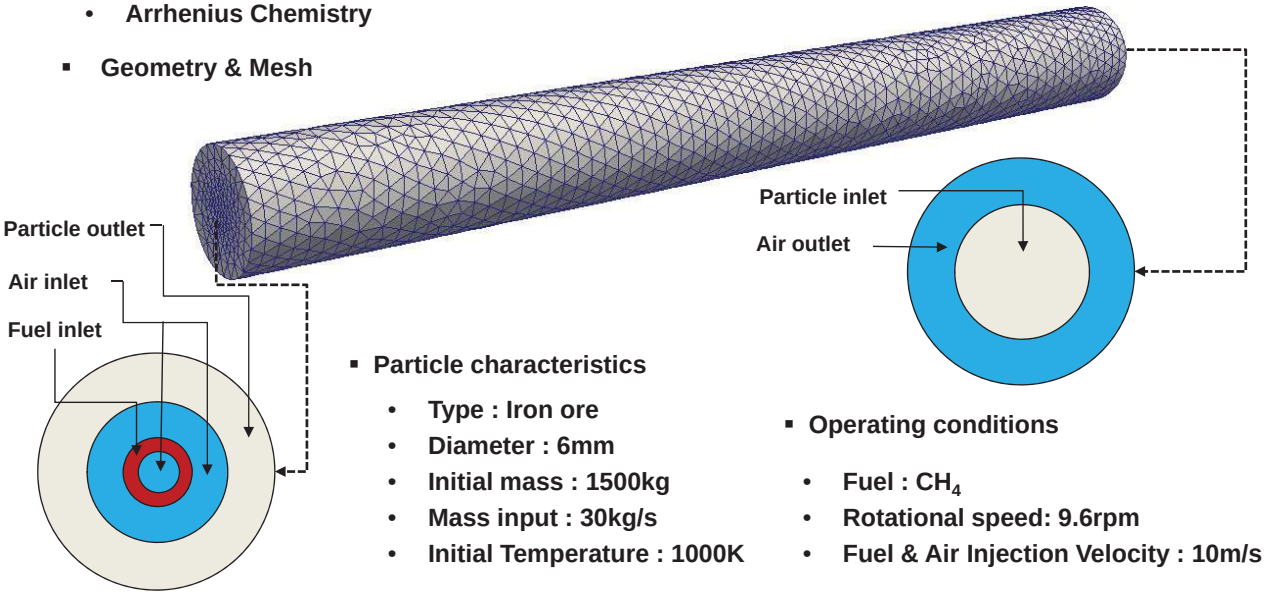
(wt %)	Laterite	Calcine
Ni	2.0	
T_Fe	13.8	
SiO_2	39.8	
MgO	22.7	
Al_2O_3	1.2	
IL	10.4	
Moisture	20.8	

Composition of material

Rotary Klin

Test Case

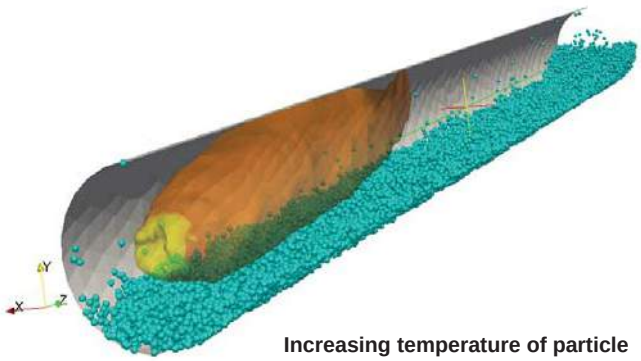
- **Numerical Method**
  - Eulerian(gas phase) - Lagrangian(solid phase) approach
  - MP-PIC (Multi Phase – Particle in Cell) method for particle motion
  - Arrhenius Chemistry
- **Geometry & Mesh**

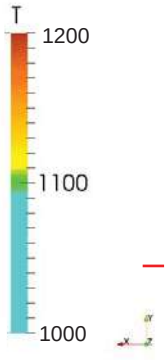


- **Particle characteristics**
  - Type : Iron ore
  - Diameter : 6mm
  - Initial mass : 1500kg
  - Mass input : 30kg/s
  - Initial Temperature : 1000K
- **Operating conditions**
  - Fuel : CH_4
  - Rotational speed: 9.6rpm
  - Fuel & Air Injection Velocity : 10m/s

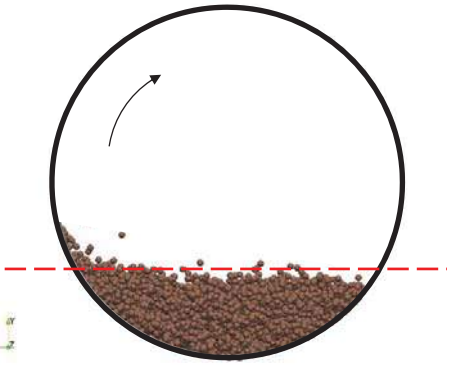
Rotary Klin

Results

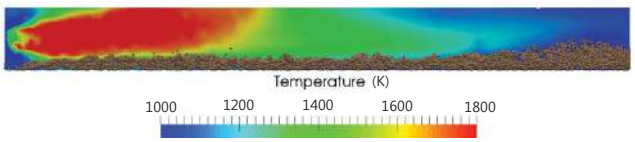


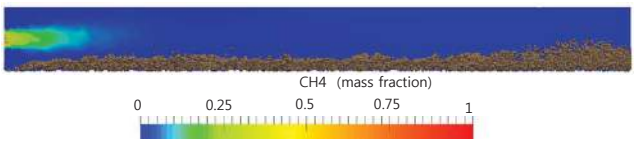


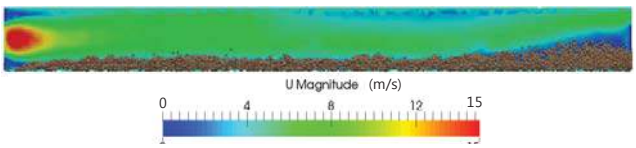
Increasing temperature of particle

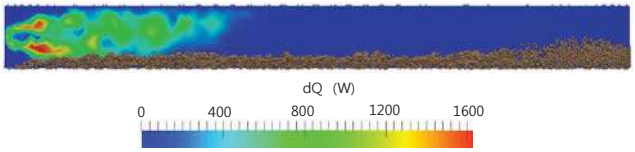


Rotating wall effect





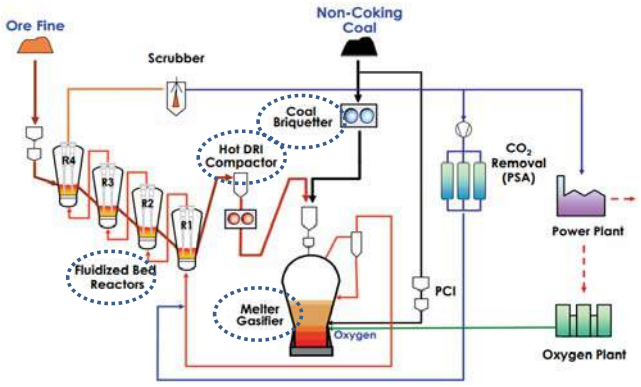




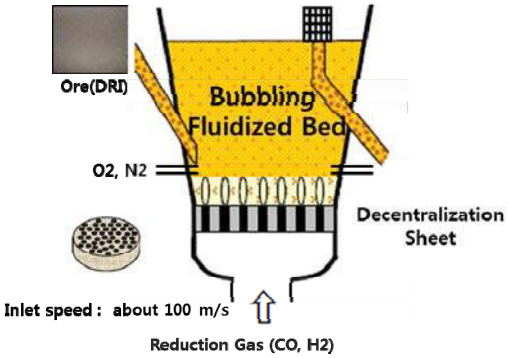
FINEX: R2

Case description

- A part of the iron-making process
- The biggest fluidized bed reactor in the world (12m toll)
- Multiphase flow: Gases(CO, H_2, N_2 ···) – Particles (HCI) flow
- Chemical reaction: reducing process, non-premixed combustion
  Both homogeneous and heterogeneous reactions are included
- Some tricky phenomenon such as sticking around O_2 nozzle
- Wide size distribution of particles



Whole process flow of the FINEX

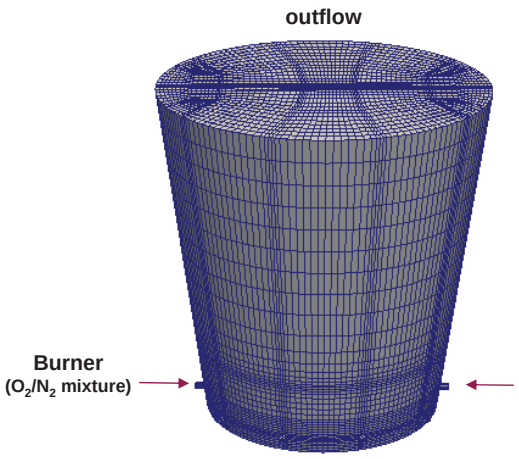


The second reactor

FINEX: R2

Test Case

- Numerical Method
  - Eulerian(gas phase) - Lagrangian(solid phase) approach
  - MP-PIC (Multi Phase – Particle in Cell) method for particle motion
  - EDM combustion model
- Geometry & Mesh

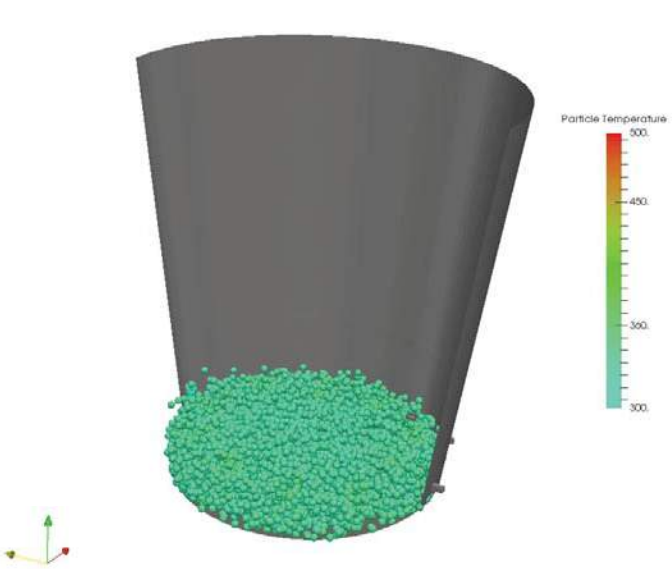


Distributing sheet
(CO/N_2 mixture)

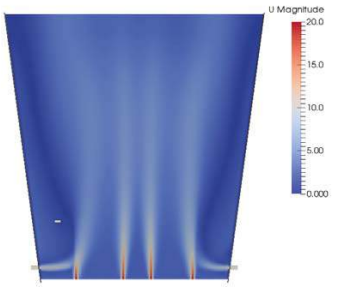
- Domain description
  - Fluidized bed reactor (particle I/O included)
  - Combustion (O_2 burner 10 pcs)
  - Distributing sheet
  - Height : 3m
- Particle characteristics
  - Type : Iron ore
  - Diameter : 1 mm (expectation)
  - Parcel Number : ~ 30000
  - Initial Temperature : 300K
- Operating conditions
  - CO/N_2 : 20 m/s through distributing sheet
  - O_2/N_2 : 10 m/s at burner nozzle

FINEX: R2

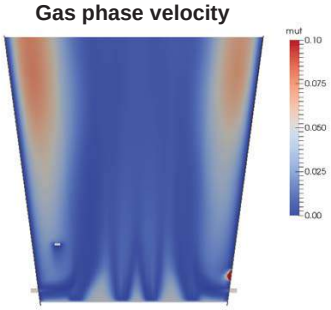
Results



Increasing temperature of particle



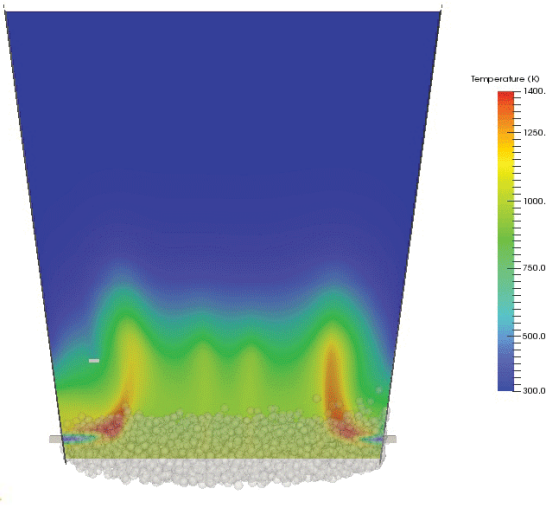
Gas phase velocity



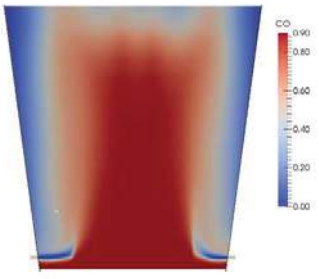
Turbulent viscosity

FINEX: R2

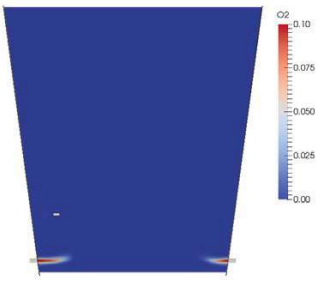
Results



Temperature contour



CO mass fraction



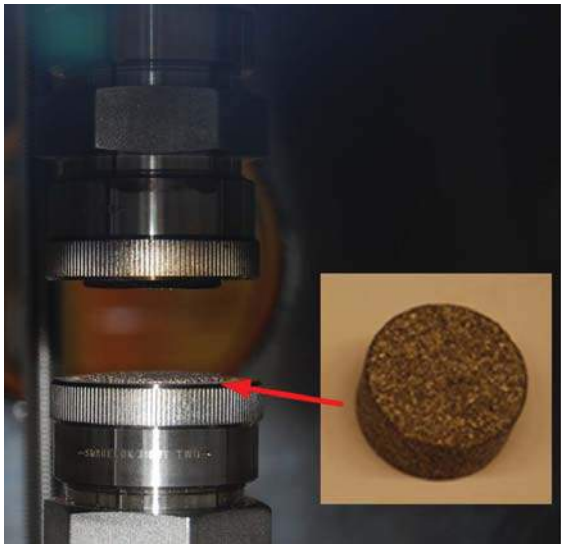
O_2 mass fraction

Plasma Assisted Combustion

Case description

- **Counterflow Burner with Stainless Steel Porous Electrodes**
  - **Diffusion flame between fuel stream and oxidizer stream**
  - **Fuel stream and oxidizer stream are CH_4 and O_2 diluted with He.**
  - **Pressure: 72 Torr**
  - **Inlet Temperature: 650 K and 600 K at oxidizer side and fuel side respectively.**
  - **Strain Rate: 400 1/s**

- **ns Pulsed Discharge**
  - **Polarity: +(oxidizer side), - (fuel side)**
  - **Pulse Duration : 12 ns (FWHM)**
  - **Pulse Voltage : 7.6 kV**
  - **Pulse Energy : 0.73 mJ/pulse**
  - **Frequency : 24 kHz**
  - **Power : 17.5 W**



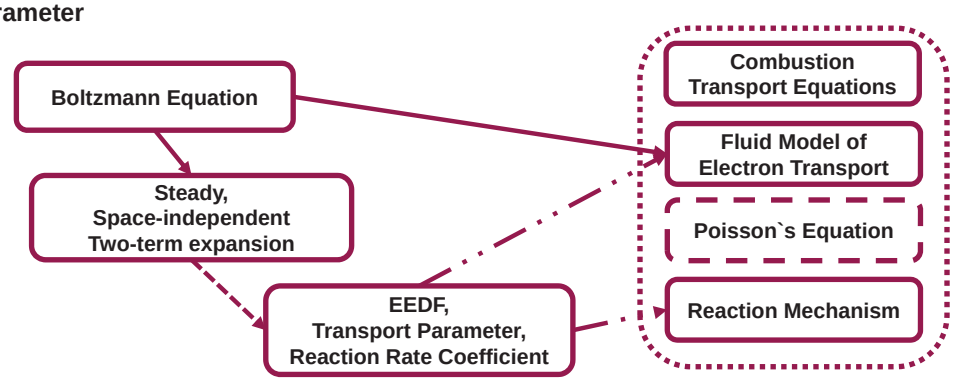
Counter flow burner with stainless steel porous electrodes (photograph)

W. Sun. (2013). *Non-equilibrium plasma-assisted combustion* (Doctoral dissertation)

Plasma Assisted Combustion

Case description

- **Opposed flow diffusion flame**
- **Transport of Electron and Electron Energy**
  - **Transport parameters of electron are calculated in-time with steady two-expansion Boltzmann equation solver (Instead of tabulation)**
- **Kinetic Model**
  - **Air-plasma model(M. Uddi, PROCI, 2009) (~ 450 reactions)**
    + USC Mech II (111 species, 784 reactions)
    + Additional reactions involving excited particles and electrons (~ 50 reactions)
- **Do not solve the Poisson equation for electric field. Rather, electric field is given as a parameter**

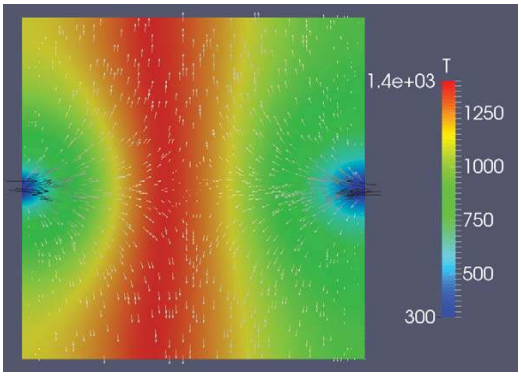


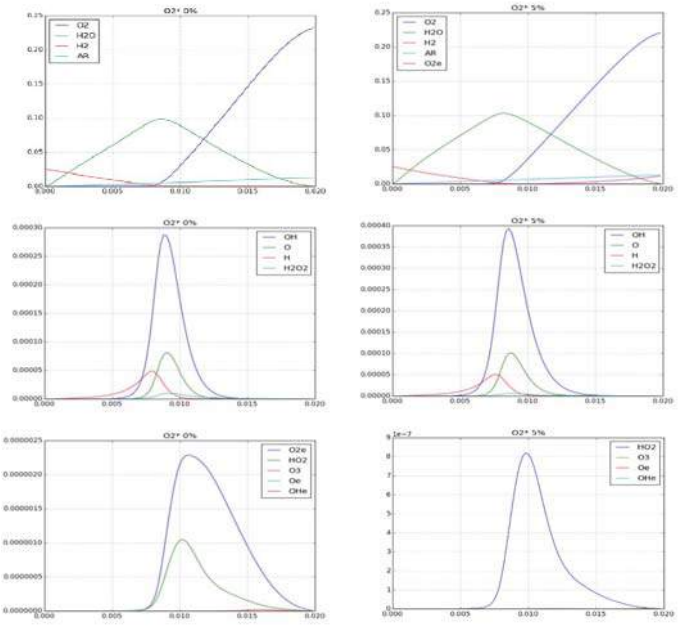
A Schematic diagram of plasma assisted combustion

Plasma Assisted Combustion

Results

- **H_2 is diluted with N_2 ($Y_{H_2} = 2.5\%$)**
- **Oxidizer is air**
- **Discharge is applied before oxidizer nozzle**
- **5% O_2 is electronically excited.**
- **Nozzle geometry is 1mm x 38mm slit**
- **Simulation flow rate is 0.1 m/s**





Mass fractions along center line

BOURIG, A., THÉVENIN, D., MARTIN, J.-., JANIGA, G. and ZÄHRINGER, K., 2009. Numerical modeling of H2-O2 flames involving electronically-excited species O2(a1Δg), O(1D) and OH(2Σ+). Proceedings of the Combustion Institute, 32 II, pp. 3171-3179

Conclusion

(1) OpenFOAM is an open source program package useful for simulation of various industrial combustion devices involving complicated multiphase physics.

(2) Turbulent combustion models are reviewed in the perspective of practical CFD application for gaseous fuel (Premixed / Non-premixed), liquid fuel (Spray) and solid fuel (Fixed, Fludized and Entrained Bed).

(3) CFD simulation is now established as a useful design and analysis tool for complicated industrial combustion devices. Extensive industrial interests shown.

(4) Further work is required for validation and implementation of more advanced and reliable turbulent combustion models to improve accuracy of the simulation results.

Implement New Combustion Models

Implement New Combustion Models

I Structure of reactingParcelFoam and PaSR model

- **reactingParcelFoam**
- **Partially stirred reactor model**

II How to implement New Combustion Models

- **Gas phase**
  - Eddy dissipation Model
  - Steady laminar flamelet model
- **Solid phase**
  - Char reaction model (Hurt-Mitchell model)

Implement New Combustion Models

I Structure of reactingParcelFoam and PaSR model

- **reactingParcelFoam**
- Partially stirred reactor model

II How to implement New Combustion Models

- Gas phase
  - Eddy dissipation Model
  - Steady laminar flamelet model
- Solid phase
  - Char reaction model (Hurt-Mitchell model)

Structure of reactingParcelFoam

reactingParcelFoam

Transient PIMPLE solver for compressible, laminar or turbulent flow with reacting multiphase Lagrangian parcels (also includes gas-phase combustion)

```
#include "fvCFD.H"
#include "turbulenceModel.H"
#include "basicReactingMultiphaseCloud.H"
#include "rhoCombustionModel.H"
#include "radiationModel.H"
#include "fvIOoptionList.H"
#include "SLGThermo.H"
#include "pimpleControl.H"

int main(int argc, char *argv[])
{
    #include "setRootCase.H"

    #include "createTime.H"
    #include "createMesh.H"
    #include "readGravitationalAcceleration.H"

    pimpleControl pimple(mesh);

    #include "createFields.H"
    #include "createRadiationModel.H"
    #include "createClouds.H"
    #include "createFvOptions.H"
```

```
Info<< "Creating combustion model\n" << endl;

autoPtr<combustionModels::rhoCombustionModel> combustion
(
    combustionModels::rhoCombustionModel::New(mesh)
);
```

1. Read **combustionProperties**
2. Constructing appropriate **'rhoCombustionModel Object'** using runtime selection mechanism
3. Auto-pointer **'combustion'** refers to above **Object**

reactingParcelFoam.C

```
parcels.evolve();

#include "rhoEqn.H"

// --- Pressure-velocity PIMPLE corrector loop
while (pimple.loop())
{
    #include "UEqn.H"
    #include "YEqn.H"
    #include "EEqn.H"

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

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

Need source term due to combustion

Structure of reactingParcelFoam

- Combustion-re… sensible enthalpy equations
- Member functi… viously constructed) are located in src/combustionModels

① Calculate source terms for Y and hs Eqns

YEqn.H

```
combustion->correct();

dQ = combustion->dQ();

label inertIndex = -1;
volScalarField Yt(0.0*Y[0]);

/* Species loc
    if (Y[i].r
    {
        volScalarField& Yi = Y[i];

        fvScalarMatrix YEqn
        (
          /* Temporal + Convection
             + Diffusion terms */
         ==
            parcels.SYi(i, Yi)
          + combustion->R(Yi)
          + fvOptions(rho, Yi)
        );

        YEqn.re
        fvOptio
        YEqn.so
        fvOptions.correct(Yi);

        Yi.max(0.0);
        Yt += Yi;
    }
```

② Heat release rate for post-processing

③ Species source term due to combustion

EEqn.H

```
volScalarField& he = thermo.he();

fvScalarMatrix EEqn
(
  /* Temporal + Convection
     + Diffusion terms */
 ==
    parcels.Sh(he)
  + radiation->Sh(thermo)
  + combustion->Sh()
  + fvOptions(rho, he)
);

EEqn.relax();
fvOptions.constrain(EEqn);
EEqn.solve();
fvOptions.correct(he);

thermo.correct();
radiation->correct();

Info<< "T gas min/max   = " << min(T).value() << ", "
    << max(T).value() << endl;
```

④ Sensible enthalpy source term due to combustion

⑤ Update thermodynamic properties for given species and sensible enthalpy field

Implement New Combustion Models

I Structure of reactingParcelFoam and PaSR model

- reactingParcelFoam
- **Partially stirred reactor model**

II How to implement New Combustion Models

- Gas phase
  - Eddy dissipation Model
  - Steady laminar flamelet model
- Solid phase
  - Char reaction model (Hurt-Mitchell model)

Structure of PaSR model

PaSR (Partially Stirred Reactor) (Golovitchev 2001)

- A computational cell is divided into the reacting and the non-reacting part.
- The reaction zone is treated as a perfectly stirred reactor.
- Reaction rates are given by

$$\overline{\rho\dot{\omega}_i} = \kappa \frac{C_{i,1} - C_{i,0}}{\Delta t} \qquad \kappa = \frac{\tau_{chemical}}{\tau_{chemical} + \tau_{mixing}}$$

src/combustionModels

```
combustionModels_POSTECH
├ ─ ─ EDM
├ ─ ─ FSD
├ ─ ─ Make
├ ─ ─ PaSR
├ ─ ─ combustionMod
├ ─ ─ diffusion
├ ─ ─ infinitelyFastChemistry
├ ─ ─ laminar
├ ─ ─ lnInclude
├ ─ ─ noCombustion
├ ─ ─ psiCombustionModel
├ ─ ─ rhoCombustionModel
└ ─ ─ singleStepCombustion
```

```
PaSR/
├ ─ ─ PaSR.C
├ ─ ─ PaSR.H
├ ─ ─ PaSRs.C
└ ─ ─ PaSRs.dep
```

src/thermophysicalModel/chemistryModel

```
chemistryModel_POSTECH
├ ─ ─ Make
├ ─ ─ chemistryModel
├ ─ ─ chemistrySolver
└ ─ ─ lnInclude
```

combustionModels/Make/files

```
combustionModel/combustionModel.C

/* Some models in here */

diffusion/diffusions.C

infinitelyFastChemistry/infinitelyFastChemistrys.C

EDM/EDMs.C

PaSR/PaSRs.C

laminar/laminars.C

/* Some models in here */

FSD/FSDs.C

noCombustion/noCombustions.C

LIB = $(FOAM_USER_LIBBIN)/libcombustionModels_POSTECH
```


Structure of PaSR model

```
template<class Type>
class PaSR
:
    public laminar<Type>
{
```

class **PaSR** inherit data and functions from class **laminar (PaSR.H)**

PaSR.C

```
template<class Type>
void Foam::combustionModels::PaSR<Type>::correct()
{
    if (this->active())
    {
        laminar<Type>::correct();

        if (turbulentReaction_)
        {
            /* Read and set turbulent properties in here */

            tmp<volScalarField> ttc(this->tc());
            const volScalarField& tc = ttc();

            forAll(epsilon, i)
            {
                scalar tk =
                    Cmix_*Foam::sqrt(muEff[i]/rho[i]/(epsilon[i] + SMALL));

                if (tk > SMALL)
                {
                    kappa_[i] = tc[i]/(tc[i] + tk);
                }
                else
                {
                    kappa_[i] = 1.0;
                }
            }
        }
```

① **Update laminar reaction rate at each cell**

$$\overline{\rho\dot{\omega}_i} = \kappa \frac{C_{i,1} - C_{i,0}}{\Delta t}$$

$$\kappa = \frac{\tau_{chemical}}{\tau_{chemical} + \tau_{mixing}}$$

Update kappa(mixing parameter) at each cell

Structure of PaSR model

PaSR.C

```
template<class Type>
Foam::tmp<Foam::fvScalarMatrix>
Foam::combustionModels::PaSR<Type>::R(volScalarField& Y) ②
{
    return kappa_*laminar<Type>::R(Y);
}


template<class Type>
Foam::tmp<Foam::volScalarField>
Foam::combustionModels::PaSR<Type>::dQ() ③
{
    return
        tmp<volScalarField>
        (
            new volScalarField("PaSR:dQ", kappa_*laminar<Type>::dQ())
        );
}


template<class Type>
Foam::tmp<Foam::volScalarField>
Foam::combustionModels::PaSR<Type>::Sh() ④
{
    return
        tmp<volScalarField>
        (
            new volScalarField("PaSR:Sh", kappa_*laminar<Type>::Sh())
        );
}
```

Return calculated source terms for species and sensible enthalpy equations

Structure of PaSR model

laminar.C

```
template<class Type>
void Foam::combustionModels::laminar<Type>::correct() ①'
{
    if (this->active())
    {
        if (integrateReactionRate_)
        {
            /* Some codes in here */

            this->chemistryPtr_->solve(this->mesh().time().deltaTValue());
        }
        else
        {
            this->chemistryPtr_->calculate();
        }
    }
}
```

Update laminar reaction rate at each cell

src/combustionModels/rhoCombustionModel/rhoChemistryCombustion

```
class rhoChemistryCombustion
:
    public rhoCombustionModel
{
    // Private Member Functions

protected:

    // Protected data

        //- Pointer to chemistry model
        autoPtr<rhoChemistryModel> chemistryPtr_;
```

```
Foam::combustionModels::rhoChemistryCombustion::rhoChemistryCombustion
(
    const word& modelType,
    const fvMesh& mesh
)
:
    rhoCombustionModel(modelType, mesh),
    chemistryPtr_(rhoChemistryModel::New(mesh))
{}
```

1. Read **chemistryProperties, thermophysicalProperties**
2. Constructing appropriate **'rhoChemistryModel Object'** using runtime selection mechanism
3. Auto-pointer **'chemistryPtr_'** refers to above **Object**

Structure of PaSR model

laminar.C

```
template<class Type>
void Foam::combustionModels::laminar<Type>::correct()  ①'
{
    if (this->active())
    {
        if (integrateReactionRate_)
        {
            /* Some codes in here */

            this->chemistryPtr_->solve(this->mesh().time().deltaTValue());
        }
        else
        {
            thi
        }
    }
}
```

Update laminar reaction rate at each cell

chemistryModel.C

```
template<class CompType, class ThermoType>
template<class DeltaTType>
Foam::scalar Foam::chemistryModel<CompType, ThermoType>::solve(const DeltaTType& deltaT)
{
    /* Some codes */
    forAll(rho, celli)
    {
        for (label i=0; i<nSpecie_; i++)
        {
            c[i] = rhoi*Y_[i][celli]/specieThermo_[i].W();
            c0[i] = c[i];
        }
        /* Some codes */
        deltaTMin = min(this->deltaTChem_[celli], deltaTMin);
        for (label i=0; i<nSpecie_; i++)
        {
            RR_[i][celli] = (c[i] - c0[i])*specieThermo_[i].W()/deltaT[celli];
        }
    }
    return deltaTMin;
}
```

$$\overline{\rho\dot{\omega}_i} = \kappa\frac{C_{i,1} - C_{i,0}}{\Delta t}$$

Structure of PaSR model

laminar.C

```
template<class Type>
Foam::tmp<Foam::fvScalarMatrix>
Foam::combustionModels::laminar<Type>::R(volScalarField& Y)  ②'
{
    tmp<fvScalarMatrix> tSu(new fvScalarMatrix(Y, dimMass/dimTime));

    fvScalarMatrix& Su = tSu();

    if (this->active())
    {
        const label specieI = this->thermo().composition().species()[Y.name()];

        Su += this->chemistryPtr_->RR(specieI);
    }

    return tSu;
}
```

Return laminar reaction source terms at each cell

```
template<class Type>
Foam::tmp<Foam::volScalarField>
Foam::combustionModels::laminar<Type>::dQ()  ③'
{
    /* Some codes in here */

    if (this->active())
    {
        tdQ() = this->chemistryPtr_->dQ();
    }

    return tdQ;
}
```

```
template<class Type>
Foam::tmp<Foam::volScalarField>
Foam::combustionModels::laminar<Type>::Sh()  ④'
{
    /* Some codes in here */

    if (this->active())
    {
        tSh() = this->chemistryPtr_->Sh();
    }

    return tSh;
}
```

Structure of PaSR model

① PaSR::correct()
→ laminar::correct()
→ chemistryModel::solve(△t)

YEqn.H

```
combustion->correct();

dQ = combustion->dQ();

label inertIndex = -1;
volScalarField Yt(0.0*Y[0]);

/* Species loc
    if (Y[i].r
    {
        volScalarField& Yi = Y[i];

        fvScalarMatrix YEqn
        (
          /* Temporal + Convection
             + Diffusion terms */
         ==
            parcels.SYi(i, Yi)
          + combustion->R(Yi)
          + fvOptions(rho, Yi)
        );

        YEqn.re
        fvOptic
        YEqn.so
        fvOptic

        Yi.max(0.0);
        Yt += Yi;
    }
```

② PaSR::dQ()
→ laminar::dQ()
→ chemistryModel::dQ()

③ PaSR::R(Y)
→ laminar::R(Y)
→ chemistryModel::RR(i)

EEqn.H

```
volScalarField& he = thermo.he();

fvScalarMatrix EEqn
(
  /* Temporal + Convection
     + Diffusion terms */
 ==
    parcels.Sh(he)
  + radiation->Sh(thermo)
  + combustion->Sh()
  + fvOptions(rho, he)
);

EEqn.relax();
fvOptions.constrain(EEqn);
EEqn.solve();
fvOptions.correct(he);

thermo.correct();
radiation->correct();

Info<< "T gas min/max   = " << min(T).value() << ", "
    << max(T).value() << endl;
```

④ PaSR::Sh()
→ laminar::Sh()
→ chemistryModel::Sh()

⑤ Update thermodynamic properties for given species and sensible enthalpy field

Structure of PaSR model

```
template<class BasicPsiThermo, class MixtureType>
void Foam::heRhoThermo<BasicPsiThermo, MixtureType>::correct()
{
    if (debug)
    {
        Info<< "entering heRhoThermo<Mix
    }

    calculate();

    if (debug)
    {
        Info<< "exiting heRhoThermo<Mixt
    }
}
```

⑤

heRhoThermo.C

```
template<class BasicPsiThermo, class MixtureType>
void Foam::heRhoThermo<BasicPsiThermo, MixtureType>::calculate()
{
    const scalarField& hCells = this->he().internalField();
    const scalarField& pCells = this->p_.internalField();

    scalarField& TCells = this->T_.internalField();
    scalarField& psiCells = this->psi_.inte
    scalarField& rhoCells = this->rho_.inte
    scalarField& muCells = this->mu_.intern
    scalarField& alphaCells = this->alpha_.

    forAll(TCells, celli)
    {
        const typename MixtureType::thermoType& mixture_ =
            this->cellMixture(celli);

        TCells[celli] = mixture_.THE
        (
            hCells[celli],
            pCells[celli],
            TCells[celli]
        );

        psiCells[celli] = mixture_.psi(pCells[celli], TCells[celli]);
        rhoCells[celli] = mixture_.rho(pCells[celli], TCells[celli]);

        muCells[celli] = mixture_.mu(pCells[celli], TCells[celli]);
        alphaCells[celli] = mixture_.alphah(pCells[celli], TCells[celli]);
    }

    forAll(this->T_.boundaryField(), patchi)
    {
        /* Some Codes in here */
    }
```

⑤'

Read and set sensible enthalpy and species mass fraction fields

Update thermodynamic properties at cell

Update thermodynamic properties at boundary

Implement New Combustion Models

I Structure of reactingParcelFoam and PaSR model

- reactingParcelFoam
- Partially stirred reactor model

II How to implement New Combustion Models

- **Gas phase**
  - Eddy dissipation Model
  - Steady laminar flamelet model
- Solid phase
  - Char reaction model (Hurt-Mitchell model)

How to Implement New Combustion Models (EDM)

EDM (Eddy Dissipation Method) (Magnussen et al.)

$$R_f = A \cdot \bar{Y}_f (\varepsilon / k)$$
$$R_f = A(\bar{Y}_{O2} / r_f)(\varepsilon / k)$$
$$R_f = A \cdot B\{\bar{Y}_P / (1+r_f)\}(\varepsilon / k)$$

Minimum R_f → Local mean rate of combustion

A, B : constants

r_f : stoichiometric oxygen requirement

- **The mean reaction rate is controlled by the turbulent mixing rate.**
- **The reaction rate is limited by the deficient species of reactants or product**

Finite Rate / EDM

EDM

$$R_{i,r} = \nu'_{i,r} MW_i A \rho \frac{\varepsilon}{k} \min\left(\frac{Y_{\text{Reactant}}}{\nu'_{\text{Reactant},r} MW_{\text{Reactant}}}\right)$$
$$R_{i,r} = \nu'_{i,r} MW_i AB \rho \frac{\varepsilon}{k} \min\left(\frac{\sum Y_{\text{Product}}}{\sum \nu''_{\text{Product},r} MW_{\text{Product}}}\right)$$

Arrhenius

$$R_{i,r} = \Gamma(\nu''_{i,r} - \nu'_{i,r})\left(k_{f,r}\prod_{j=1}^{N}[C_{j,r}]^{\nu'_j} - k_{b,r}\prod_{j=1}^{N}[C_{j,r}]^{\nu''_j}\right)$$

$k_r = A_r T^{b_r} exp(-E_r / RT)$

for species i and reaction r

- **The mean reaction rate is determined by the minimum $R_{i,r}$**

How to Implement New Combustion Models (EDM)

- Make new folder (EDM) under src/combustionModels
- Copy existing files (PaSR.H, PaSR.C and PaSRs.C) to new folder (EDM)
- Rename files (PaSR.H …) to EDM.H …
- Modify Make/files

```
combustionModels_POSTECH
├ – – EDM
│     ├ – – EDM.C
│     ├ – – EDM.H
│     ├ – – EDMs.C
│     └ – – EDMs.dep
├ – – FSD
├ – – Make
├ – – PaSR
├ – – combustionModel
├ – – diffusion
├ – – infinitelyFastChemistry
├ – – laminar
├ – – lnInclude
├ – – noCombustion
├ – – psiCombustionModel
├ – – rhoCombustionModel
└ – – singleStepCombustion
```

```
chemistryModel_POSTECH
├ – – Make
├ – – chemistryModel
├ – – chemistrySolver
└ – – lnInclude
```

Make/files

```
combustionModel/combustionModel.C

/* Some models in here */

diffusion/diffusions.C

infinitelyFastChemistry/infinitelyFastChemistrys.C

EDM/EDMs.C

PaSR/PaSRs.C

laminar/l

/* Some m

FSD/FSDs.

noCombust

LIB = $(F
```

```
#include "makeCombustionTypes.H"
#include "psiChemistryCombustion.H"
#include "rhoChemistryCombustion.H"
#include "EDM.H"

namespace Foam
{
namespace combustionModels
{
    makeCombustionTypes
      (EDM, psiChemistryCombustion, psiCombustionModel);
    makeCombustionTypes
      (EDM, rhoChemistryCombustion, rhoCombustionModel);
}
}
```


How to Implement New Combustion Models (EDM)

- Change all existing class and type name (PaSR or laminar) to new name (EDM)

laminar.H

```
#ifndef laminar_H
#define laminar_H

namespace Foam
{
namespace combustionModels
{
template<class Type>
class laminar
:
    public Type
{
    // Private data
        bool integrateReactionRate_;

protected:

private:

    // Private Member Functions
        //- Disallow copy construct
        laminar(const laminar&);
        //- Disallow default bitwise assignment
        void operator=(const laminar&);

public:

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

    // Constructors

        //- Construct from components
        laminar(const word& modelType, const fvMesh& mesh);
```

EDM.H

```
#ifndef EDM_H
#define EDM_H

namespace Foam
{
namespace combustionModels
{
template<class Type>
class EDM
:
    public Type
{
    // Private data
        //- Select EDM or finite-rate/EDM
        bool finiteRate_;
        scalar A_;
        scalar B_;

protected:

private:

    // Private Member Functions
        //- Disallow copy construct
        EDM(const EDM&);
        //- Disallow default bitwise assignment
        void operator=(const EDM&);

public:

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

    // Constructors

        //- Construct from components
        EDM(const word& modelType, const fvMesh& mesh);
```

How to Implement New Combustion Models (EDM)

Constructor at EDM.C

```
template<class Type>
Foam::combustionModels::EDM<Type>::EDM
(
    const word& modelType,
    const fvMesh& mesh
)
:
    Type(modelType, mesh),
    finiteRate_
    (
        this->coeffs().lookupOrDefault("finiteRate", false)
    ),
    A_
    (
        this->coeffs().lookupOrDefault("A", 4.0)
    ),
    B_
    (
        this->coeffs().lookupOrDefault("B", 0.5)
    )
{
    if (finiteRate_)
    {
        Info<< "    using Finite-rate/Eddy Dissipation Model" << endl;
        Info<< "    A = "<<A_<< endl;
        Info<< "    B = "<<B_<< endl;
    }
    else
    {
        Info<< "    using Eddy Dissipation Model" << endl;
        Info<< "    A = "<<A_<< endl;
        Info<< "    B = "<<B_<< endl;
    }
}
```

- **Add flag or modeling coefficient**

$$R_{i,r} = \nu'_{i,r} MW_i A \rho \frac{\varepsilon}{k} \min\left(\frac{Y_{\text{Reactant}}}{\nu'_{\text{Reactant},r} MW_{\text{Reactant}}}\right)$$

$$R_{i,r} = \nu'_{i,r} MW_i AB \rho \frac{\varepsilon}{k} \min\left(\frac{\sum Y_{\text{Product}}}{\sum \nu''_{\text{Product},r} MW_{\text{Product}}}\right)$$

How to Implement New Combustion Models (EDM)

① EDM::correct()
→ chemistryModel::calculateEDM(bool, A, B)

YEqn.H

```
combustion->correct();
dQ = combustion->dQ();

label inertIndex = -1;
volScalarField Yt(0.0*Y[0]);

/* Species loc
    if (Y[i].r
    {
        volScalarField& Yi = Y[i];

        fvScalarMatrix YEqn
        (
          /* Temporal + Convection
             + Diffusion terms */
         ==
            parcels.SYi(i, Yi)
          + combustion->R(Yi)
          + fvOptions(rho, Yi)
        );

        YEqn.re
        fvOptio
        YEqn.so
        fvOptions.correct(Yi);

        Yi.max(0.0);
        Yt += Yi;
    }
```

② EDM::dQ()
→ chemistryModel::dQ()

③ EDM::R(Y)
→ chemistryModel::RR(i)

EEqn.H

```
volScalarField& he = thermo.he();

fvScalarMatrix EEqn
(
  /* Temporal + Convection
     + Diffusion terms */
 ==
    parcels.Sh(he)
  + radiation->Sh(thermo)
  + combustion->Sh()
  + fvOptions(rho, he)
);

EEqn.relax();
fvOptions.constrain(EEqn);
EEqn.solve();
fvOptions.correct(he);

thermo.correct();
radiation->correct();

Info<< "T gas min/max   = " << min(T).value() << ", "
    << max(T).value() << endl;
```

④ EDM::Sh()
→ chemistryModel::Sh()

⑤ Update thermodynamic properties for given species and sensible enthalpy field

- **Add or modify functions**

  or Re-use exsiting functions

How to Implement New Combustion Models (EDM)

EDM.C

```
template<class Type>
Foam::tmp<Foam::fvScalarMatrix>
Foam::combustionModels::EDM<Type>::R(volScalarField& Y) const
{
    tmp<fvScalarMatrix> tSu(new fvScalarMatrix(Y, dimMass/dimTime));

    fvScalarMatrix& Su = tSu();

    if (this->active())
    {
        const label specieI = this->thermo().composition().species()[Y.name()];

        Su += this->chemistryPtr_->RR(specieI);
    }

    return tSu;
}
```

Re-used functions

```
template<class Type>
Foam::tmp<Foam::volScalarField>
Foam::combustionModels::EDM<Type>::Sh() const
{
    /* Some codes */

    if (this->active())
    {
        tSh() = this->chemistryPtr_->Sh();
    }

    return tSh;
}
```

```
template<class Type>
Foam::tmp<Foam::volScalarField>
Foam::combustionModels::EDM<Type>::dQ() const
{
    /* Some codes */

    if (this->active())
    {
        tdQ() = this->chemistryPtr_->dQ();
    }

    return tdQ;
}
```


How to Implement New Combustion Models (EDM)

EDM.C

```
template<class Type>
void Foam::combustionModels::EDM<Type>::correct()
{
    if (this->active())
    {
        this->chemistryPtr_->calculateEDM(finiteRate_, A_, B_);
    }
}
```

chemistryModel.H

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

    /* Some functions */

    //- Calculates the reaction rates (EDM or finite-rate/EDM)
    //- 12.Dec.2014 Karam Han
    virtual void calculateEDM(bool finiteRate, const scalar A, const scalar B);
```

basicChemistryModel.H

```
//- Calculates the reaction rates (EDM or finite-rate/EDM)
//- 12.Dec.2014 Karam Han
virtual void calculateEDM(bool finiteRate, const scalar A, const scalar B) = 0;
```

How to Implement New Combustion Models (EDM)

chemistryModel.C

```
template<class CompType, class ThermoType>
void Foam::chemistryModel<CompType, ThermoType>::calculateEDM
(
    bool finiteRate, const scalar A, const scalar B
)
{
    /* Some codes */

    const volScalarField& rho = this->thermo().rho();
    const volScalarField& k = this->db().objectRegistry::lookupObject<volScalarField>("k");
    const volScalarField& epsilon = this->db().objectRegistry::lookupObject<volScalarField>("epsilon");

    scalar EDMrate(0.0);
    forAll(rho, celli)
    {
        scalarField RRtemp(nSpecie_, 0.0);

        forAll(reactions_, reactioni)
        {
            const Reaction<ThermoType>& R = reactions_[reactioni];

            scalar Ymin(1.0);
            forAll(R.lhs(), lhs_speciei)
            {
                const label speciei = R.lhs()[lhs_speciei].index;
                const scalar stoiCoeff = R.lhs()[lhs_speciei].stoichCoeff;
                if(stoiCoeff != 0)
                {
                    Ymin = min(Ymin, Y_[speciei][celli]/stoiCoeff/specieThermo_[speciei].W());
                }
            }
            /* Some Codes */
```

Read variables from object registry (rho, k, epsilon)

Cell loop

Reactions loop

$CH_4 + 1.5O_2 \rightarrow CO + 2H_2O$

$CO + 0.5O_2 \rightarrow CO_2$

Species loop (Left hand side)

$$\min\left(\frac{Y_{Reactant}}{\nu'_{Reactant,r} MW_{Reactant}}\right)$$

How to Implement New Combustion Models (EDM)

```
            forAll(R.rhs(), rhs_speciei)
            {
                const label speciei = R.rhs()[rhs_speciei].index;
                const scalar stoiCoeff = R.rhs()[rhs_speciei].stoichCoeff;
                if(stoiCoeff != 0)
                {
                    Num += Y_[speciei][celli];
                    Den += stoiCoeff*specieThermo_[speciei].W();
                }
            }
            YPmin = Num / Den;
            Ymin = min(Ymin, B*YPmin);

            EDMrate = A*rho[celli]*epsilon[celli]/k[celli]*Ymin;

            forAll(R.lhs(), lhs_speciei)
            {
                const label speciei = R.lhs()[lhs_speciei].index;
                const scalar stoiCoeff = R.lhs()[lhs_speciei].stoichCoeff;

                RRtemp[speciei] -= stoiCoeff*specieThermo_[speciei].W()*EDMrate;
            }
            forAll(R.rhs(), rhs_speciei)
            {
                const label speciei = R.rhs()[rhs_speciei].index;
                const scalar stoiCoeff = R.rhs()[rhs_speciei].stoichCoeff;

                RRtemp[speciei] += stoiCoeff*specieThermo_[speciei].W()*EDMrate;
            }
        }
        for (label i=0; i<nSpecie_; i++)
        {
            RR_[i][celli] = RRtemp[i];
        }
    }
}
```

Species loop (Right hand side)

$$\min\left(\frac{\sum Y_{Product}}{\sum \nu''_{Product,r} MW_{Product}}\right)$$

$$A\rho\frac{\varepsilon}{k}\min\left(\frac{Y_{Reactant}}{\nu'_{Reactant,r} MW_{Reactant}}\right)$$

or

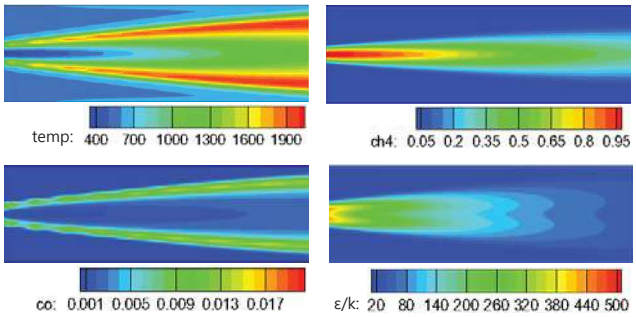
$$AB\rho\frac{\varepsilon}{k}\min\left(\frac{\sum Y_{Product}}{\sum \nu''_{Product,r} MW_{Product}}\right)$$

Apply reaction source term for each species

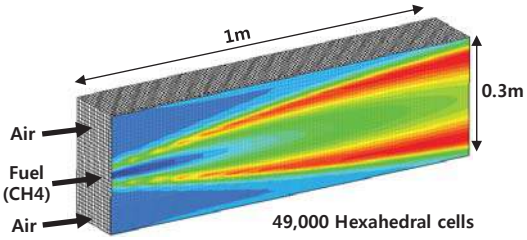
How to Implement New Combustion Models (EDM)

```
/*--------------------------------*- C++ -*----------------------------------*\
| =========                 |                                                 |
| \\      /  F ield         | OpenFOAM: The Open Source CFD Toolbox           |
|  \\    /   O peration     | Version:  2.3.x                                 |
|   \\  /    A nd           | Web:      www.OpenFOAM.org                      |
|    \\/     M anipulation  |                                                 |
\*---------------------------------------------------------------------------*/
FoamFile
{
    version     2.0;
    format      ascii;
    class       dictionary;
    location    "constant";
    object      combustionProperties;
}
// * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * //

active      true;

combustionModel EDM<rhoChemistryCombustion>;

EDMCoeffs
{
    A 4.0;
    B 0.5; //1e15;
    finiteRate false;
}

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

CombustionProperties



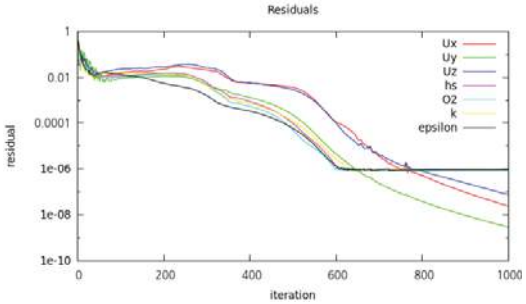
Distributions of temperature, CH4, CO



49,000 Hexahedral cells

Operating Conditions	Fuel / Air
Inlet velocity (m/s)	50 / 5
Inlet temperature (K)	355.15 / 588.15
K (m^2/s^2)	37.5 / 0.375
Epsilon (m^2/s^3)	17968.4 / 1.7968

Geometry and operating conditions



Convergence history in terms of residuals

Implement New Combustion Models

I Structure of reactingParcelFoam and PaSR model

- reactingParcelFoam
- Partially stirred reactor model

II How to implement New Combustion Models

- **Gas phase**
  - Eddy dissipation Model
  - Steady laminar flamelet model
- Solid phase
  - Char reaction model (Hurt-Mitchell model)

How to Implement New Combustion Models (SLFM)

SLFM (Steady Laminar Flamelet Model)

- Turbulent flame modeled as an ensemble of thin, laminar, locally 1-D flamelet structures
- Reacting scalars mapped from physical space to mixture fraction space

SLFM Library (Q_i, Q_h, T)

- Contains Q_i, Q_h and T distributions in mixture fraction space
- **Parameterized in terms of scalar dissipation rate (SDR)**
- Usually **pre-calculated** by in-house code or other tools (OpenFOAM?)

$$0 = \langle N|\eta\rangle \frac{\partial^2 Q_\eta}{\partial \eta^2} + \langle \dot{w}_\eta|\eta\rangle$$

PDF Library

- Contains probability density function i
- **Parameterized in terms of mixture fraction and its variance**
- **Pre-calculated** or calculate on the fly

$$\tilde{P}(\eta) = \frac{\zeta^{\alpha-1}(1-\zeta)^{\beta-1}}{\Gamma(\alpha)\Gamma(\beta)}\Gamma(\alpha+\beta)$$

- Integrate scalars (Yi, T, h …) from SLFM and PDF library, make 3D Look-up table (mf, mfVar, SDR) before calculation
- Solve transport eqns for mixture fraction and its variance
- Find and interpolate scalars from 3D Look-up table for given mixture fraction, variance and SDR
- Correct thermodynamic properties at local position

SLFMFoam

How to Implement New Combustion Models (SLFM)

SLFMFoam.C

```
#include "fvCFD.H"
#include "turbulenceModel.H"
#include "basicReactingMultiphaseCloud.H"
#include "rhoCombustionModel.H"
#include "radiationModel.H"
#include "fvIOoptionList.H"
#include "SLGThermo.H"
#include "pimpleControl.H"

int main(int argc, char *argv[])
{
    #include "setRootCase.H"

    #include "createTime.H"
    #include "createMesh.H"
    #include "readGravitationalAcceleration.H"

    pimpleControl pimple(mesh);

    #include "createFields.H"
    #include "createRadiationModel.H"
    #include "createC
    #include "createFv
    #include "initCont
    #include "readTime
    #include "compressi
    #include "setInitialDeltaT.H"
```

Add new volScalarfield **mf**, **mfVar**

```
parcels.evolve();


// --- Pressure-velocity PIMPLE corrector loop
while (pimple.loop())
{
    #include "UEqn.H"
    #include "YEqn.H"
    #include "EEqn.H"

    // --- Pressure corrector loop
    while (pimple.correct())
    {
        #i
    }

    if (pi
    {
        tu
    }
```

No need to be solved

Add new transport equations for mf and mfVar

#include “MixtureFraction.H”
#include “MixtureFractionVar.H”

How to Implement New Combustion Models (SLFM)

createField.H

```
volScalarField mf
(
    IOobject
    (
        "mf",
        runTime.timeName(),
        mesh,
        IOobject::MUST_READ,
        IOobject::AUTO_WRITE
    ),
    mesh
);

Info<<"Reading field mfVar\n"<<endl;
volScalarField mfVar
(
    IOobject
    (
        "mfVar",
        runTime.timeName(),
        mesh,
        IOobject::MUST_READ,
        IOobject::AUTO_WRITE
    ),
    mesh
);
```

Mixturefraction.H

```
fvScalarMatrix mfEqn
(

    + mvConvection->fvmDiv(phi, mf)
    - fvm::laplacian(1.47*turbulence->mut(), mf)
    ==
    Sevap
);
```

MixturefractionVar.H

```
SDR =  turbulence->epsilon() * mfVar / turbulence->k();
volVectorField Gradmf = fvc::grad(mf);

fvScalarMatrix mfVarEqn
(

    + mvConvection->fvmDiv(phi, mfVar)
    - fvm::laplacian(1.47*turbulence->mut(), mfVar)
    ==
    + 2*(1.47*turbulence->mut())*(Gradmf & Gradmf)
    + 2*rho*SDR
    + 2*(0.5)*sqrt(mag(mfVar))*Sevap*(1-mf)
);
```


How to Implement New Combustion Models (SLFM)

SLFMFoam.C

```
#include "fvCFD.H"
#include "turbulenceModel.H"
#include "basicReactingMultiphaseCloud.H"
#include "rhoCombustionModel.H"
#include "radiationModel.H"
#include "fvIOoptionList.H"
#include "SLGThermo.H"
#include "pimpleControl.H"
    #include "setRootCase.H"
    #include "readGravitationalAcceleration.H"

    #include "createFields.H"
    #include "createRadiationModel.H"
    #include "createClouds.H"
    #include "createFvOptions.H"
    #include "initContinuityErrs.H"
    #include "readTimeControls.H"
    #include "compressibleCourantNo.H"
    #include "setInitialDeltaT.H"
```

Solve transport eqns for mixture fraction and its variance

Find and interpolate scalars from 3D Look-up table for given mixture fraction, variance and SDR

Correct thermodynamic properties at local position

Add new volScalarfield **mf**, **mfVar**

```
parcels.evolve();


// --- Pressure-velocity PIMPLE corrector loop
while (pimple.loop())
{
    #include "UEqn.H"
    #include "Mixturefraction.H"
    #include "MixturefractionVar.H"

    #include "SLFMLookup.H"

    thermo.correct();

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

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

Implement New Combustion Models

I Structure of reactingParcelFoam and PaSR model

- reactingParcelFoam
- Partially stirred reactor model

II How to implement New Combustion Models

- Gas phase
  - Eddy dissipation Model
  - Steady laminar flamelet model
- **Solid phase**
  - Char reaction model (Hurt-Mitchell model)

How to Implement New Combustion Models (Char Reaction)

- **Make new folder (COxidationHM) under src/lagrangian/coalCombustion/submodels**
- **Copy existing files (COxidationKineticDiff.H …) to new folder (COxidationHM)**
- **Rename files (COxidationKineticDiff.H …) to COxidationHM.H …**
- **Modify makeCoalParcelSurfaceReactionModels.H**

```
#ifndef makeCoalParcelSurfaceReactionModels_H
#define makeCoalParcelSurfaceReactionModels_H

#include "NoSurfaceReaction.H"
#include "COxidationDiffusionLimitedRate.H"
#include "COxidationIntrinsicRate.H"
#include "COxidationKineticDiffusionLimitedRate.H"
#include "COxidationHM.H"
#include "COxidationHurtMitchell.H"
#include "COxidationMurphyShaddix.H"

#define makeCoalParcelSurfaceReactionModels(CloudType)                \
                                                                      \
    makeSurfaceReactionModelType(COxidationDiffusionLimitedRate, CloudType); \
    makeSurfaceReactionModelType                                      \
    (                                                                 \
        COxidationKineticDiffusionLimitedRate,                        \
        CloudType                                                     \
    );                                                                \
    makeSurfaceReactionModelType(COxidationHM, CloudType);            \
    makeSurfaceReactionModelType(COxidationIntrinsicRate, CloudType); \
    makeSurfaceReactionModelType(COxidationHurtMitchell, CloudType);  \
    makeSurfaceReactionModelType(COxidationMurphyShaddix, CloudType);

#endif
```

```
coalCombustion/submodels/
└ – –  surfaceReactionModel
     ├ – –  COxidationDiffusionLimitedRate
     ├ – –  COxidationHM  → ├ – –  COxidationHM.C
     ├ – –  COxidationHurtM └ – –  COxidationHM.H
     ├ – –  COxidationIntrinsicRate
     ├ – –  COxidationKineticDiffusionLimitedRate
     └ – –  COxidationMurphyShaddix
```

How to Implement New Combustion Models (Char Reaction)

- **Change all existing class and type name (COxidationKine…) to new name**

COxidationKinetic DiffusionLimited.H

```
#include "SurfaceReactionM

namespace Foam
{
// Forward class declarations
template<class CloudType>
class COxidationKineticDiffusionLimitedRate;

template<class CloudType>
class COxidationKineticDiffusionLimitedRate
:
    public SurfaceReactionModel<CloudType>
{
    // Private data

        // Model constants
            //- Mass diffusion limited rate constan
            const scalar C1_;

        // Addressing
            label CsLocalId_;
            label O2GlobalId_;
            label CO2GlobalId_;

        // Local copies of thermo properties
            scalar WC_;
            scalar WO2_;
            scalar HcCO2_;
public:

    //- Runtime type information
    TypeName("COxidationKineticDiffusionLimitedRate");
```

COxidationHM.H

```
#include "SurfaceReactionModel.H"

namespace Foam
{
// Forward class declarations
template<class CloudType>
class COxidationHM;

template<class CloudType>
class COxidationHM
:
    public SurfaceReactionModel<CloudType>
{
    // Private data

        // Model constants
            // Carbon content in dry ash free based [wt%]
            const scalar Cdaf_;

        // Addressing
            label CsLocalId_;
            label ashLocalId_;
            label O2GlobalId_;
            label COGlobalId_;

        // Local copies of thermo properties
            scalar WC_;
            scalar WO2_;
            scalar HcCO_;

public:

    //- Runtime type information
    TypeName("COxidationHM");
```


How to Implement New Combustion Models (Char Reaction)

- **Add or modify functions according to the char reaction model**

COxidationHM.H

```
// Member Functions
    //- Update surface reactions
    virtual scalar calculate
    (
        const scalar dt,
        const label cellI,
        const scalar d,
        const scalar T,
        const scalar Tc,
        const scalar pc,
        const scalar rhoc,
        const scalar mass,
        const scalarField& YGas,
        const scalarField& YLiquid,
        const scalarField& YSolid,
        const scalarField& YMixture,
        const scalar N,
        scalarField& dMassGas,
        scalarField& dMassLiquid,
        scalarField& dMassSolid,
        scalarField& dMassSRCarrier
    ) const;
```

COxidationHM.C

```
template<class CloudType>
Foam::scalar Foam::COxidationHM<CloudType>::calculate
( /* Some scalar and field */ ) const
{
    /* Some functions */

    scalar kD = 5e-12/d*pow(0.5*(T + Tc), 0.75);
    scalar kC = ach*exp(-ech/Rgas/T);

    /* Calculate char reaction rate here
       according to Hurt-Mitchell Model */

    //Change in C mass [kg]
    scalar dmC = Ap * rate * dt;

    // Molar consumption
    const scalar dOmega = dmC/WC_;
    const scalar dmO2 = dOmega*Sb_*WO2_;
    const scalar dmCO = dOmega*(WC_ + Sb_*WO2_);

    // Update local particle C mass
    dMassSolid[CsLocalId_] += dOmega*WC_;
    dMassSRCarrier[O2GlobalId_] -= dmO2;
    dMassSRCarrier[COGlobalId_] += dmCO;

    // Heat of reaction [J]
    return dmC*HsC - dmCO*HcCO_;
```

Thank you
for your kind attention

OpenFOAM: Code Development, Debugging and Trouble-Shooting

Hrvoje Jasak

hrvoje.jasak@fsb.hr, h.jasak@wikki.co.uk

University of Zagreb, Croatia and

Wikki Ltd, United Kingdom

Outline

Objective

- Illustrate basic steps in creating custom applications, adding on-the-fly post-processing and extending capabilities of the library

Topics

- Organise your work with OpenFOAM
- `wmake` build system
- Programming guidelines
- Enforcing consistent style
- Create your own application: simple modifications in top-level code
- Adding a boundary condition
- Debugging OpenFOAM with gdb
- Release and debug version; environment variables and porting
- Some programming techniques
- Working with `git`
- Stability of discretisation: handling source and sink terms

OpenFOAM Work Space

Organise Your Work with OpenFOAM

- OpenFOAM is a library of tools, not a monolithic single-executable
- Most changes do not require surgery on the library level: code is developed in local work space
- In most cases, groups of users rely on a central installation. If changes are necessary, take out only files to be changed
- Environment variables and library structure control the location of the library, external packages (*e.g.* gcc, ParaView) and work space
- For model development, start by copying a model and changing its name: library functionality is unaffected

Local Work Space

- **Run directory**: `$FOAM_RUN`. Ready-to-run cases and results, test loop *etc.* May contain case-specific setup tools, solvers and utilities
- **Local work space**: `/home/hjasak/OpenFOAM/hjasak-2.3`. Contains applications, libraries and personal library and executable space

Build System

`wmake` Build System

- Wrapper around `make`
- Automatic creation of dependency files
- Multi-machine, portable system; control files in `OpenFOAM-2.3/wmake/rules`
- Clear separation between
  - Application source files (`Make/files`): source files to compile and name and location of executable to produce
  - Include files followed recursively to automatically create dependencies `*.dep`
  - Application specific options (`Make/options`): include paths and link libraries
  - Language- and machine-specific options
- Supports multiple versions, controlled by environment variables
- Some user setting in `OpenFOAM-2.3/etc/cshrc` and `bashrc`

Main Build Commands

- `wclean` Remove .o files, .dep files
- `wmake` Build application
- `wmake libso` Build shared library

Programming Guidelines

OpenFOAM And Object-Orientation

- OpenFOAM library tools are strictly object-oriented: trying hard to weed out the hacks, tricks and work-arounds
- Adhering to standard is critical for quality software development in C++: ISO/IEC 14882-2003 incorporating the latest Addendum notes

Writing C in C++

- C++ compiler supports the complete C syntax: writing procedural programming in C is very tempting for beginners
- Object Orientation represents a paradigm shift: the way the problem is approached needs to be changed, not just the programming language. This is not easy
- Some benefits of C++ (like data protection and avoiding code duplication) may seem a bit esoteric, but they represent a real qualitative advantage
  1. Work to understand why C++ forces you to do things
  2. Adhere to the style even if not completely obvious: ask questions, discuss
  3. Play games: minimum amount of code to check for debugging :-)
  4. Analyse and rewrite your own work: more understanding leads to better code
  5. Try porting or regularly use multiple compilers
  6. Do not tolerate warning messages: they are really errors!

Enforcing Consistent Style

Writing Software In OpenFOAM Style

- OpenFOAM library tools are strictly object-oriented; top-level codes are more in functional style, unless implementation is wrapped into model libraries
- OpenFOAM uses ALL features of C++ to the maximum benefit: you will need to learn it. Also, the code is an example of **good C++**: study and understand it

Enforcing Consistent Style

- Source code style in OpenFOAM is remarkably consistent:
  - Code separation into files
  - Comment and indentation style
  - Approach to common problems, e.g. I/O, construction of objects, stream support, handling function parameters, const and non-const access
  - Blank lines, no trailing whitespace, no spaces around brackets
- **Using file stubs**: `foamNew` script
  - `foamNew H exampleClass`: new header file
  - `foamNew C exampleClass`: new implementation file
  - `foamNew I exampleClass`: new inline function file
  - `foamNew IO exampleClass`: new IO section file
  - `foamNew App exampleClass`: new application file

Creating Your Applications

Customising Solver and Utilities

- Source code in OpenFOAM serves 2 functions
  - Efficient and customised top-level solver for class of physics. Ready to run in a manner of commercial CFD/CCM software
  - Example of OpenFOAM classes and library functionality in use
- Modifications can be simple, *e.g.* additional post-processing or customised solver output or a completely new model or solver

Creating Your Applications

1. Find appropriate code in OpenFOAM which is closest to the new use or provides a starting point
2. Copy into local work space and rename
3. Change file name and location of library/executable: Make/files
4. Environment variables point to local work space applications and libraries: `$FOAM_PROJECT_USER_DIR`, `$FOAM_USER_APPBIN` and `$FOAM_USER_LIBBIN`
5. Change the code

Example: `scalarTransportFoam`

- Manipulating input/output; changing inlet condition; adding a probe

New Boundary Condition

Run-Time Selection Table Functionality

- In many cases, OpenFOAM provides functionality selectable at run-time which needs to be changed for the purpose. Example: viscosity model; ramped fixed value boundary conditions
- New functionality should be run-time selectable (like implemented models)
- . . . but should not interfere with existing code! There is no need to change existing library functionality unless we have found bugs
- For the new choice to become available, it needs to be instantiated and linked with the executable.

Boundary Condition: Ramped Fixed Value

- Find closest similar boundary condition: `oscillatingFixedValue`
- Copy, rename, change input/output and functionality. Follow existing code patterns
- Compile and link executable; consider relocating into a library
- Beware of the `defaultFvPatchField` problem: verify code with print statements

Debugging OpenFOAM

Build and Debug Libraries

- Release build optimised for speed of execution; Debug build provides additional run-time checking and detailed trace-back capability
- Debug switches
  - Each set of classes or class hierarchy provides own debug stream
  - . . . but complete flow of messages would be overwhelming!
  - Choosing debug message source: `OpenFOAM-2.3/etc/controlDict`
- Using trace-back on failure
  - `gdb icoFoam`: start debugger on `icoFoam` executable
  - `r -case <case>`: perform the run from the debugger
  - `where` provides full trace-back with function names, file and line numbers
  - Similar tricks for debugging parallel runs: attach gdb to a running process
- Valgrind: `valgrind.org`
  - Memory allocation, deallocation and access checking tool
  - `valgrind --tool=memcheck icoFoam`
  - Slow, but reliable: every allocation, access and deallocation is tracked
  - Debug version of OpenFOAM has built-in array bound checking, but comes with a loss of performance of approx factor of 3

OpenFOAM Environment

Environment Variables and Porting

- Software was developed on multiple platforms and ported regularly: better quality and adherence to standard
- Switching environment must be made easy: source single dot-file
- All tools, compiler versions and paths can be controlled with environment variables
- **Environment variables**
  - Environment setting support one installation on multiple machines
  - User environment: $HOME/OpenFOAM-2.3/etc/cshrc
  - OpenFOAM tools: OpenFOAM-2.3/settings.csh
    - Standard layout, *e.g.* FOAM_SRC, FOAM_RUN
    - Compiler and library settings, communications library *etc.*
- Additional setting
  - `FOAM_ABORT`: behaviour on abort
  - `FOAM_SIGFPE`: handling floating point exceptions
  - `FOAM_SETNAN`: set all memory to invalid on initialisation

Some Basic Rules

Basic Rules and Naming Conventions

- "Make your code look like OpenFOAM"
- All names in camel-back case: `polyMesh`
- Reasonable name abbreviations: `fvMesh`
- Four-space indentation
- Lots of comments and blank lines: 2 lines after function definition
- Group members and data by functionality:

```
const vectorField& cellCentres() const;
const vectorField& faceCentres() const;
const scalarField& cellVolumes() const;
const vectorField& faceAreas() const;
```

- Doxygen-style comments on all data and member functions in the header: `//-`
- Claim authorship of files and add extended comment on functionality

Some Basic Rules

Basic Rules and Naming Conventions

- Private data members have names with trailing underscore

```
class fvPatch
{
    // Private data

        //- Reference to the underlying polyPatch
        const polyPatch& polyPatch_;

        //- Reference to boundary mesh
        const fvBoundaryMesh& boundaryMesh_;

    ...
};
```

- **Strict data protection**
  - No public data objects or public pointers: use const and non-const reference
  - Strict enforcement of const and non-const interface
  - Class functionality will enforce the nature of the interface, not internal operation of the class

Some Basic Rules

Basic Rules and Naming Conventions

- **Strict data protection**
  - No public data objects or public pointers: use const and non-const reference
  - Strict enforcement of const and non-const interface
  - Class functionality will enforce the nature of the interface, not internal operation of the class
- Some “standard” protection against unintended copying

```
// Private Member Functions

    //- Disallow default bitwise copy construct
    exampleClass(const exampleClass&);

    //- Disallow default bitwise assignment
    void operator=(const exampleClass&);
```

- Carefully consider static member functions, class vs. namespace, use of virtual functions and run-time selection etc.
- All compiler warnings on all platforms should be fixed!

Dictionary Data Input

Data Input in Dictionary Format

- Main input system: **dictionary**
  - List of keyword – value pairs delimited by semicolon
  - Order of entries is irrelevant
  - Embedded dictionary levels allowed
  - Field data also in dictionary format: uniform or expanded data, using “super-tokens” for efficient access to raw stream

```
dimensions      [0 1 -1 0 0 0 0];
internalField   uniform (0 0 0);

boundaryField
{
    movingWall
    {
        type            fixedValue;
        value           uniform (1 0 0);
    }
    ...
}
```

Dictionary Data Input

Access to Dictionary Data

- For class object, dictionary construction is trivial: constructor from `Istream`

```
//- Construct from Istream.
explicit dimensioned(Istream&);

template <class Type>
dimensioned<Type>::dimensioned
(
    Istream& is
)
:
    name_(is),
    dimensions_(is),
    value_(pTraits<Type>(is))
{}
```

and

```
dimensionedScalar DT
(
    transportProperties.lookup("DT")
);
```


Dictionary Data Input

Access to Dictionary Data

- Problem: primitive data types are not classes: no `Istream` constructor
- Using helper function: take a stream and return a primitive

```
scalar maxCo
(
    readScalar(runTime.controlDict().lookup("maxCo"))
);

label nCorr(readLabel(piso.lookup("nCorrectors")));
```

- Access embedded dictionary: `subDict`

```
laplacianSchemes_ = dict.subDict("laplacianSchemes");
```

Virtual Function Mechanism

Virtual Base Class Interface: Class Families

- Virtual functions are the "first" kingpin of OpenFOAM
- The role of virtual base classes is two-fold
  - Collect related objects into containers. A container holds a homogenous set of objects, eg. `IOobject` of `fvPatchField`
  - **Virtual function dispatch**: access specific functionality based on actual object type at **run-time**
- Example: `fvPatchField::evaluate()` to `zeroGradientFvPatchField::evaluate()`
- Designing virtual interfaces requires extra care and knowledge, especially regarding the argument list
- A virtual function needs to answer a large variety of needs:
  - Definition should take account ALL the classes we can possibly envisage
  - Beware of large and unmanageable virtual interfaces: this is not the only way of passing data to a function
- Re-connecting class derivation graphs should be avoided: sometimes a sign of over-engineered class structure

Virtual Function Mechanism

Virtual Function Mechanism

- Principle: User refers to a base class for a selection of classes which answer to the same interface but have type-specific behaviour
- Each derived type implements the specific behaviour of the base class
- Actual object type determined at run-time
- As a result, the class definition of derived types does not need to be known at compile time: excellent!
- Unfortunately, virtual functions do not handle the construction problem: this is handled by the run-time selection
- OpenFOAM does not rely completely on the C++ standard **Run-Time Type Identification (RTTI)** because of I/O and portability issues

Alternative Ways of Passing Data: Example in Free Stream

- Recover from one of existing arguments
- Recover from parent classes
- Provide at construction time for the class
- Consider going up the hierarchy to where data is available
- If all else fails, consider querying the `objectRegistry`

Run-Time Selection

WIKKI

Run-Time Selection Mechanism

- Construction Problem: By definition, virtual functions are not operational until the object is constructed: cannot have a virtual constructor!
- Therefore, all is well once all patch fields are constructed
- If a geometric field is to be copied, patch fields need to be copied as well: cannot have a virtual copy constructor. Hmm.
- Solution clone() virtual function:

```
virtual tmp<fvPatchField<Type> > clone() const = 0;
```

- What about construction from `Istream`?
  1. Open Istream
  2. Read type of patch field, eg. `fixedValue`
  3. Create object of the type specified by string and return as pointer to base class. Additional data depending on type can be read from the stream or a dictionary

Run-Time Selection

WIKKI

Run-Time Selection Mechanism

- We wish to keep the derived virtual base classes in complete isolation
- We could register the constructor into the table because all constructors need to have the identical signature
- The lookup key is the class type name: all derived classes are named:
- Solution: **Factory Mechanism**

```
//- Return a pointer to a new patchField created on
//  freestore given patch and internal field
static tmp<fvPatchField<Type> > New
(
    const word&,
    const fvPatch&,
    const Field<Type>&
);
```

- The second component is **automatic registration** of objects onto the lookup table using the static initialisation mechanism in C++
- Example: construction of `fvPatchField`s from a dictionary
- (Do you really want to know how this is done?)

Lazy Evaluation

Programming Principle of Lazy Evaluation

1. What I do not know, should not hurt me
   - Do not wish to pay memory overhead
   - Do not wish to waste CPU time
2. Delay actual calculation until data is required
3. Delay memory allocation until it is necessary
4. Do not try to anticipate needs: answer to requests

Programming Technique: `primitiveMesh` Example

- `primitiveMesh` needs to provide cell volumes:

```
const scalarField& cellVolumes() const;
```

- This is clearly private data of `primitiveMesh`
- However, if cell volumes are not required, they should not be calculated
- Recognise that cells volumes should not/will not be used one cell at a time (at least through this interface):

```
mesh.cells()[553].mag(mesh.points(), mesh.faces())
```


Lazy Evaluation

Implementation of Lazy Evaluation

1. Private member pointer for cells volumes, set to NULL on construction:

```
mutable scalarField* cellVolumesPtr_;
```

2. Calculation function for cell volumes:

```
void primitiveMesh::calcCellCentresAndVols() const
```

3. Access function checks if volumes are already calculated
   - If not calculated, call `calcCellCentresAndVols()`
   - Return const reference to array

```
const scalarField& primitiveMesh::cellVolumes() const
{
    if (!cellVolumesPtr_)
    {
        calcCellCentresAndVols();
    }

    return *cellVolumesPtr_;
}
```

Lazy Evaluation

Benefit from Laziness: Consider Moving Mesh Calculation

- When mesh moves, cell volumes out of date
- Delete pointer, set to `NULL` and do nothing!
- When volumes are needed again, they will be calculated on demand
- Perfect example of `mutable` keyword: calculation of cell volumes does not change the actual `primitiveMesh`, but the algorithm we use changes the state of its private data member

Function Objects

Implementation and Use of Function Objects

- Top-level solver code is typically modified for trivial reasons
  - Add sampling points for solution fields
  - Calculate integral values, eg. forces concentration, flow rates
  - Follow `min-max` field values for data monitoring or debugging
- This is tedious: hundreds of copies, multiple solvers etc.
- Solution: **function objects** embedded in time loop
  - `Time` class holds a table of virtual base class objects with stubs
  - User writes stubs as derived versions of `functionObject` class
  - Function object library is loaded at run-time using dynamic libraries
  - User-defined function object is created using run-time selection, with data read from `system/controlDict` file
  - `functionObject`s are initialised at first call and executed at `runTime++`
- Contents of function object table and function object parameters can be updated while the simulation is running

Function Objects

Writing Function Objects

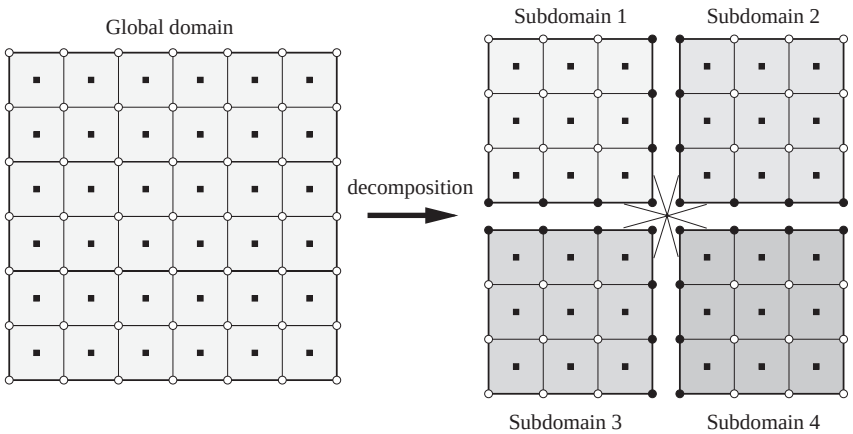
- Dictionary constructor
- Data access using `objectRegistry`
- Functionality in virtual functions: start, execute (every time-step), read

```
class functionObject
{
public:
    static autoPtr<functionObject> New
    (
        const word& name,
        const Time&,
        const dictionary&
    );

    virtual bool start() = 0;

    virtual bool execute() = 0;

    virtual bool read(const dictionary& dict) = 0;
};
```


Consistent Execution Path

Parallelisation by Consistent Execution Path

- In OpenFOAM, parallelisation is established in **domain decomposition mode**
- A basic premise is that the same set of operations is performed over all mesh components, where each processor is responsible for a part of the domain
- **Synchronisation points** are defined by data exchange between processors, which can be pairwise and global
- As the system relies on sequential global order of synchronisation, if is essential that all processors **follow the same execution patch**



Working with `git`

`git`: A Distributed Version Control System

- Keep track of your code development, know currect changes
- Preserve change history and check-points
- Collaborate with developers without compromising the code
- Distribute code and bug-fixes in a reliable way

Basic `git` Actions

- Create repository: `git init`
- Edit some files, make changes
- Create a new branch: `git checkout -b NAME`
- Select changes to store: `git add FILENAME`
- Check that changes selected: `git status`
- Store changes: `git commit`
- Update repository or pull in changes: `git push`, `git pull`
- See effect of switching branches: `git checkout NAME`
- Merge branch: `git merge --no-ff BRANCHNAME`

Stability of Discretisation

Live derivation and hands-on exercise

Overview: stability of implementation of sources, sinks and boundary conditions

$$\alpha = 3$$

$$\frac{\partial \alpha}{\partial t} = 3$$

$$\frac{\partial \alpha}{\partial t} = \frac{\alpha^* - \alpha}{\tau}$$

$$\frac{\partial \alpha}{\partial t} + \mathbf{u} \bullet \nabla \alpha = 0, \quad \nabla \bullet \mathbf{u} \neq 0$$

$$\frac{\partial \alpha}{\partial t} + \nabla \bullet (\mathbf{u}\, \alpha) = 0, \quad \nabla \bullet \mathbf{u} = 0$$

$$\frac{\partial \alpha}{\partial t} + \nabla \bullet (\mathbf{u}\, \alpha) - \nabla \bullet (\gamma \nabla \alpha) = \frac{\alpha^* - \alpha}{\tau}$$

Summary

OpenFOAM Programming

- OpenFOAM is a good and complete example of use of object orientation and C++
- Code layout designed for multiple users sharing a central installation and developing tools in local workspace
- Consistent style and some programming guidelines available through file stubs: `foamNew` script for new code layout
- Most (good) development starts from existing code and extends its capabilities

Programming Techniques

- Strictly defined variable and class naming conventions should be adhered to: OpenFOAM has consistent look-and-feel
- `dictionary`: main mechanism for data I/O
- Virtual function mechanism and run-time selection are a basis of OpenFOAM architecture should be adhered to on most user-defined selection points
  - Encapsulation of class families under common interfaces
  - Flexible and non-intrusive addition of functionality
- Lazy evaluation is critical for code efficiency and memory management
- Function objects allow the user to embed own code without recompiling

Incompressible, steady-state turbulent flow simulations in OpenFOAM®

Tommaso Lucchini

Department of Energy, Politecnico di Milano

Topics

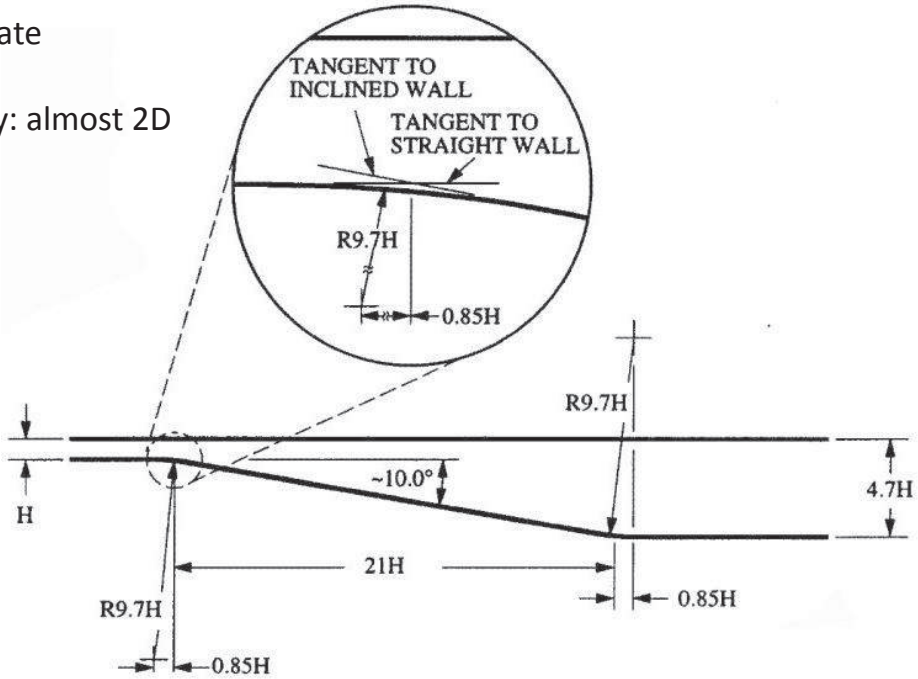
- The problem: flow simulation through an asymmetric planar diffuser.
- The solver: `simpleFoam`
- Case setup: mesh conversion, definition of initial conditions, dictionaries
- Effects of turbulence models and wall functions
- Validation with experimental data : velocity field, …

The slides are based on the `OpenFOAM-2.3.1` distribution.

Flow problem

Flow through an asymmetric plane diffuser

- Steady-state
- Fluid: air
- Geometry: almost 2D



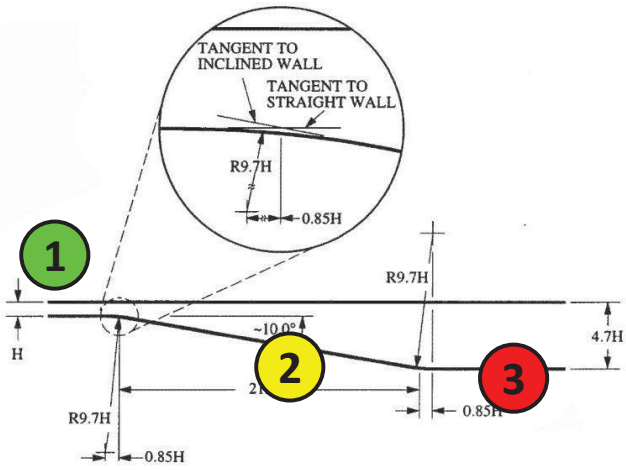
Detailed description of the problem and full set of experimental data available here

Flow problem

Flow through an asymmetric plane diffuser

This flow includes three important features:

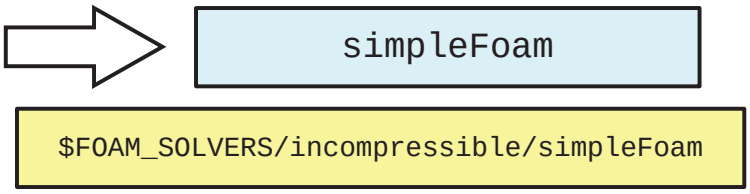
1) The well defined inlet conditions. The inlet channel flow is turbulent and fully-developed with a *Re*=20,000 based on the centreline velocity and the channel height.
2) A smooth-wall separation due to an adverse pressure gradient is involved. The prediction of the separation point and the extent of the recirculation region is particularly challenging for computational models.
3) This flow includes reattachment and redevelopment of the downstream boundary layer. The physics of reattachment and recovery are still not well understood and continue to pose a challenge to fluid mechanicians.



T. Lucchini: Incompressible, steady-state turbulent flow simulations in OpenFOAM®

Which solver?

- Look at the user guide of OpenFOAM (here) or to the provided documentation:
  1) Incompressible flow
  2) Steady-state flow
  3) Turbulent flow



A quick (very quick) look at the source code will help us in understanding:

- How the solver works
- Which equations are solved
- Which terms need discretization
- How turbulence is handled

simpleFoam.C

```
int main(int argc, char *argv[])
{
    #include "setRootCase.H"
    #include "createTime.H"
    #include "createMesh.H"
    #include "createFields.H"
    #include "createFvOptions.H"
    #include "initContinuityErrs.H"
    simpleControl simple(mesh);
    Info<< "\nStarting time loop\n" << endl;
    while (simple.loop())
    {
        Info<< "Time = " << runTime.timeName() << nl << endl;
        // --- Pressure-velocity SIMPLE corrector
        {
            #include "UEqn.H"
            #include "pEqn.H"
        }
        turbulence->correct();
        runTime.write();
        Info<< "ExecutionTime = " << runTime.elapsedCpuTime() << " s"
            << "  ClockTime = " << runTime.elapsedClockTime() << " s"
            << nl << endl;
    }
    Info<< "End\n" << endl;
    return 0;
}
```

Identification of case directory

simpleFoam.C

```
int main(int argc, char *argv[])
{
    #include "setRootCase.H"
    #include "createTime.H"
    #include "createMesh.H"
    #include "createFields.H"
    #include "createFvOptions.H"
    #include "initContinuityErrs.H"
    simpleControl simple(mesh);
    Info<< "\nStarting time loop\n" << endl;
    while (simple.loop())
    {
        Info<< "Time = " << runTime.timeName() << nl << endl;
        // --- Pressure-velocity SIMPLE corrector
        {
            #include "UEqn.H"
            #include "pEqn.H"
        }
        turbulence->correct();
        runTime.write();
        Info<< "ExecutionTime = " << runTime.elapsedCpuTime() << " s"
            << "  ClockTime = " << runTime.elapsedClockTime() << " s"
            << nl << endl;
    }
    Info<< "End\n" << endl;
    return 0;
}
```

- Generate time object (start, end, time step, …)
- Creation of the mesh from the `polyMesh` directory

simpleFoam.C

```
int main(int argc, char *argv[])
{
    #include "setRootCase.H"
    #include "createTime.H"
    #include "createMesh.H"
    #include "createFields.H"
    #include "createFvOptions.H"
    #include "initContinuityErrs.H"
    simpleControl simple(mesh);
    Info<< "\nStarting time loop\n" << endl;
    while (simple.loop())
    {
        Info<< "Time = " << runTime.timeName() << nl << endl;
        // --- Pressure-velocity SIMPLE corrector
        {
            #include "UEqn.H"
            #include "pEqn.H"
        }
        turbulence->correct();
        runTime.write();
        Info<< "ExecutionTime = " << runTime.elapsedCpuTime() << " s"
            << "  ClockTime = " << runTime.elapsedClockTime() << " s"
            << nl << endl;
    }
    Info<< "End\n" << endl;
    return 0;
}
```

Creates all the fields required for the simulation: Pressure, velocity, flux, turbulence fields,…

T. Lucchini: Incompressible, steady-state turbulent flow simulations in OpenFOAM®

simpleFoam.C

```
int main(int argc, char *argv[])
{
    #include "setRootCase.H"
    #include "createTime.H"
    #include "createMesh.H"
    #include "createFields.H"
    #include "createFvOptions.H"
    #include "initContinuityErrs.H"
    simpleControl simple(mesh);
    Info<< "\nStarting time loop\n" << endl
    while (simple.loop())
    {
        Info<< "Time = " << runTime.timeName() << nl << endl;
        // --- Pressure-velocity SIMPLE corrector
        {
            #include "UEqn.H"
            #include "pEqn.H"
        }
        turbulence->correct();
        runTime.write();
        Info<< "ExecutionTime = " << runTime.elapsedCpuTime() << " s"
            << "  ClockTime = " << runTime.elapsedClockTime() << " s"
            << nl << endl;
    }
    Info<< "End\n" << endl;
    return 0;
}
```

Reads proper data required by the SIMPLE loop

SIMPLE stands for *Semi-Implicit Method for Pressure Linked Equations*

simpleFoam.C

```
int main(int argc, char *argv
{
    #include "setRootCase.H"
    #include "createTime.H"
    #include "createMesh.H"
    #include "createFields.H"
    #include "createFvOptions
    #include "initContinuityEr
    simpleControl simple(mesh);
    Info<< "\nStarting time loop\n" << endl;
    while (simple.loop())
    {
        Info<< "Time = " << runTime.timeName() << nl << endl;
        // --- Pressure-velocity SIMPLE corrector
        {
            #include "UEqn.H"
            #include "pEqn.H"
        }
        turbulence->correct();
        runTime.write();
        Info<< "ExecutionTime = " << runTime.elapsedCpuTime() << " s"
            << "  ClockTime = " << runTime.elapsedClockTime() << " s"
            << nl << endl;
    }
    Info<< "End\n" << endl;
    return 0;
}
```

For every iteration until the end - `while(simple.loop())` - :

1) Build momentum equation, under-relax it, solve it (`UEqn.H`)
2) Do pressure-velocity coupling (`pEqn.H`)
3) Solve turbulence fields according to the chosen model (`turbulence->correct()`)
4) Write fields according to the `writeInterval` option in `controlDict` file

simpleFoam solver, createFields.H

```
Info<< "Reading field p\n" << endl;
volScalarField p
(
    IOobject
    (
        "p",
        runTime.timeName(),
        mesh,
        IOobject::MUST_READ,
        IOobject::AUTO_WRITE
    ),
    mesh
);
Info<< "Reading field U\n" << endl;
volVectorField U
(
    IOobject
    (
        "U",
        runTime.timeName(),
        mesh,
        IOobject::MUST_READ,
        IOobject::AUTO_WRITE
    ),
    mesh
);
#include "createPhi.H"
label pRefCell = 0;
scalar pRefValue = 0.0;
setRefCell(p, mesh.solutionDict().subDict("SIMPLE"), pRefCell, pRefValue);
singlePhaseTransportModel laminarTransport(U, phi);
autoPtr<incompressible::RASModel> turbulence
(
    incompressible::RASModel::New(U, phi, laminarTransport)
);
```

Pressure field (scalar): identified by a file called p, which must be placed in the `startTime` directory of the case. It should have as much boundary conditions as the boundary of the mesh.

simpleFoam solver, createFields.H

```
Info<< "Reading field p\n" << endl;
volScalarField p
(
    IOobject
    (
        "p",
        runTime.timeName(),
        mesh,
        IOobject::MUST_READ,
        IOobject::AUTO_WRITE
    ),
    mesh
);
Info<< "Reading field U\n" << endl;
volVectorField U
(
    IOobject
    (
        "U",
        runTime.timeName(),
        mesh,
        IOobject::MUST_READ,
        IOobject::AUTO_WRITE
    ),
    mesh
);
#include "createPhi.H"
label pRefCell = 0;
scalar pRefValue = 0.0;
setRefCell(p, mesh.solutionDict().subDict("SIMPLE"), pRefCell, pRefValue);
singlePhaseTransportModel laminarTransport(U, phi);
autoPtr<incompressible::RASModel> turbulence
(
    incompressible::RASModel::New(U, phi, laminarTransport)
);
```

Velocity field (vector): identified by a file called U, which must be placed in the `startTime` directory of the case. It should have as much boundary conditions as the boundary of the mesh.

enFOAM®

simpleFoam solver, createFields.H

```
Info<< "Reading field p\n" << endl;
volScalarField p
(
    IOobject
    (
        "p",
        runTime.timeName(),
        mesh,
        IOobject::MUST_READ,
        IOobject::AUTO_WRITE
    ),
    mesh
);
Info<< "Reading field U\n" << endl;
volVectorField U
(
    IOobject
    (
        "U",
        runTime.timeName(),
        mesh,
        IOobject::MUST_READ,
        IOobject::AUTO_WRITE
    ),
    mesh
);
#include "createPhi.H"
label pRefCell = 0;
scalar pRefValue = 0.0;
setRefCell(p, mesh.solutionDict().subDict("SIMPLE"), pRefCell, pRefValue);
singlePhaseTransportModel laminarTransport(U, phi);
autoPtr<incompressible::RASModel> turbulence
(
    incompressible::RASModel::New(U, phi, laminarTransport)
);
```

This file creates the flux field: $\vec{\mathbf{U}} \cdot \vec{\mathbf{n}} S_f$

simpleFoam solver, createFields.H

```
Info<< "Reading field p\n" << endl;
volScalarField p
(
    IOobject
    (
        "p",
        runTime.timeName(),
        mesh,
        IOobject::MUST_READ,
        IOobject::AUTO_WRITE
    ),
    mesh
);
Info<< "Reading field U\n" << endl;
volVectorField U
(
    IOobject
    (
        "U",
        runTime.timeName(),
        mesh,
        IOobject::MUST_READ,
        IOobject::AUTO_WRITE
    ),
    mesh
);
#include "createPhi.H"
label pRefCell = 0;
scalar pRefValue = 0.0;
setRefCell(p, mesh.solutionDict().subDict("SIMPLE"), pRe
singlePhaseTransportModel laminarTransport(U, phi);
autoPtr<incompressible::RASModel> turbulence
(
    incompressible::RASModel::New(U, phi, laminarTransport)
);
```

Transport properties (kinematic viscosity) is read from the `transportProperties` dictionary in `constant` directory of the case

...enFOAM®

simpleFoam solver

```
Info<< "Reading field p\n" << endl;
volScalarField p
(
    IOobject
    (
        "p",
        runTime.timeName(),
        mesh,
        IOobject::MUST_READ,
        IOobject::AUTO_WRITE
    ),
    mesh
);
Info<< "Reading field U\n" << endl;
volVectorField U
(
```

Creation of the RANS turbulence model RASModel (run-time selectable).

Available turbulence models implemented in:
$FOAM_SRC/turbulenceModels/incompressible/RAS

```
kEpsilon
kkLOmega
kOmega
kOmegaSST
LamBremhorstKE
laminar
LaunderGibsonRSTM
LaunderSharmaKE
LienCubicKE
LienCubicKELowRe
LienLeschzinerLowRe
LRR
NonlinearKEShih
qZeta
realizableKE
RNGkEpsilon
SpalartAllmaras
v2f
```

```
#include "createPhi.H"
label pRefCell = 0;
scalar pRefValue = 0.0;
setRefCell(p, mesh.solutionDict().subDict("SIMPLE"), pRefCell, pRefValue);
singlePhaseTransportModel laminarTransport(U, phi);
autoPtr<incompressible::RASModel> turbulence
(
    incompressible::RASModel::New(U, phi, laminarTransport)
);
```

In the .H file of each model implemantation related references are reported.

simpleFoam solver, createFields.H

- So far we have understood what we have in:
  - 0 directory: fields which will be solved by equations (p, U, k, epsilon, omega,…)
  - constant directory : mesh, fluid properties, turbulence properties
- Now discretization-related aspects will be illustrated as well. Quick look to:
  - UEqn.H
  - pEqn.H

simpleFoam solver, UEqn.H

- Steady-state momentum equation for an incompressible fluid: $\nabla \cdot \mathbf{U} + \nabla \cdot \boldsymbol{\tau} = -\nabla p$

```
// Momentum predictor

    tmp<fvVectorMatrix> UEqn
    (
        fvm::div(phi, U)
      + turbulence->divDevReff(U)
      ==
        fvOptions(U)
    );

    UEqn().relax();

    fvOptions.constrain(UEqn());

    solve(UEqn() == -fvc::grad(p));

    fvOptions.correct(U);
```

- $\mathbf{U}$: velocity
- $\boldsymbol{\tau}$: Reynolds stress tensor (viscous + turbulence)
- p : pressure

What do we need to do?

1) Discretize in a suitable way convection, stress and gradient terms
   - `div(phi, U)`
   - `turbulence->divDevReff(U)`
   - `grad(p)`
2) Provide proper under-relaxation factors for momentum equation: `UEqn().relax()`
3) Define suitable parameters for solution of momentum equation:
   - `solve(UEqn() == -fvc::grad(p));`

simpleFoam solver, pEqn.H

```
{
    volScalarField rAU(1.0/UEqn().A());
    volVectorField HbyA("HbyA", U);
    HbyA = rAU*UEqn().H();
    UEqn.clear();
    surfaceScalarField phiHbyA("phiHbyA", fvc::interpolate(HbyA) & mesh.Sf());
    fvOptions.makeRelative(phiHbyA);
    adjustPhi(phiHbyA, U, p);
    // Non-orthogonal pressure corrector loop
    while (simple.correctNonOrthogonal())
    {
        fvScalarMatrix pEqn
        (
            fvm::laplacian(rAU, p) == fvc::div(phiHbyA)
        );
        pEqn.setReference(pRefCell, pRefValue);

        pEqn.solve();

        if (simple.finalNonOrthogonalIter())
        {
            phi = phiHbyA - pEqn.flux();
        }
    }

    #include "continuityErrs.H"
    // Explicitly relax pressure for momentum corrector
    p.relax();

    // Momentum corrector
    U = HbyA - rAU*fvc::grad(p);
    U.correctBoundaryConditions();
    fvOptions.correct(U);
}
```

Estimation of velocity field (and face flux) $\tilde{u}_{i,P}^{m*}$ without pressure gradient contribution

Pressure equation is then derived:

$$\frac{\delta \tilde{u}_{i,P}^{m*}}{\delta x_i} - \frac{\delta}{\delta x_i}\left[\frac{1}{A_P^{u_i}}\left(\frac{\delta p^m}{\delta x_i}\right)^P\right] = 0$$

Relaxation for pressure equation

Correction of velocity on the basis of pressure gradient

simpleFoam solver, pEqn.H

```
{
    volScalarField rAU(1.0/UEqn().A());
    volVectorField HbyA("HbyA", U);
    HbyA = rAU*UEqn().H();
    UEqn.clear();
    surfaceScalarField phiHbyA("phiHbyA", fvc::interpolate(HbyA) & mesh.Sf());
    fvOptions.makeRelative(phiHbyA);
    adjustPhi(phiHbyA, U, p);
    // Non-orthogonal pressure corrector loop
    while (simple.correctNonOrthogonal())
    {
        fvScalarMatrix pEqn
        (
            fvm::laplacian(rAU, p) == fvc::div(phiHbyA)
        );
        pEqn.setReference(pRefCell, pRefValue);

        pEqn.solve();

        if (simple.finalNonOrthogonalIter())
        {
            phi = phiHbyA - pEqn.flux();
        }
    }

    #include "continuityErrs.H"
    // Explicitly relax pressure for momentum corrector
    p.relax();

    // Momentum corrector
    U = HbyA - rAU*fvc::grad(p);
    U.correctBoundaryConditions();
    fvOptions.correct(U);
}
```

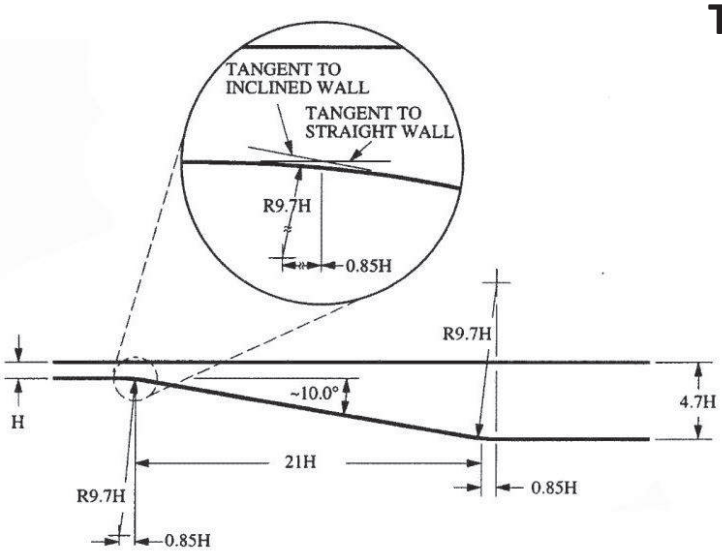
What do we need to do?

1) Discretize in a suitable laplacian term in pressure equation:
   - `fvm::laplacian(rAU, p)`
2) Provide proper underrelaxation factors for pressure field: `p.relax()`
3) Define suitable parameters for solution of momentum equation:
   - `pEqn.solve();`

simpleFoam solver

- Similar considerations can be also done for turbulence fields. In particular, it is necessary to discretize convection and diffusion terms in transport equations for k, ε, ω, and solve them in a proper manner…

Diffuser simulation: the mesh



To be considered:

1) Fully developed turbulent flow on the inlet boundary: one way is to compose a computational model of the diffuser that includes a sufficiently long inlet channel. If this method is employed, one should use sufficiently long inlet channel (110H in experiments) in order to ensure fully developed flow at the diffuser inlet.

2) Everything is non-dimensional
   - Re = 20000
   - Lengths and heights are function of H

   ⇨ Let's assume H = 1 m and compute everything accordingly.

Diffuser simulation: mesh

- Start from the `pitzDaily` tutorial of `simpleFoam` solver. Copy it in your run directory and call it `diffuserBase`

```
> run
> cp -r $FOAM_TUTORIALS/incompressible/simpleFoam/pitzDaily diffuserBase
```

- Copy in your run directory a Fluent® mesh generated with Gambit®, called `fullGeometry2.msh`. Then import the mesh using `fluentMeshToFoam`

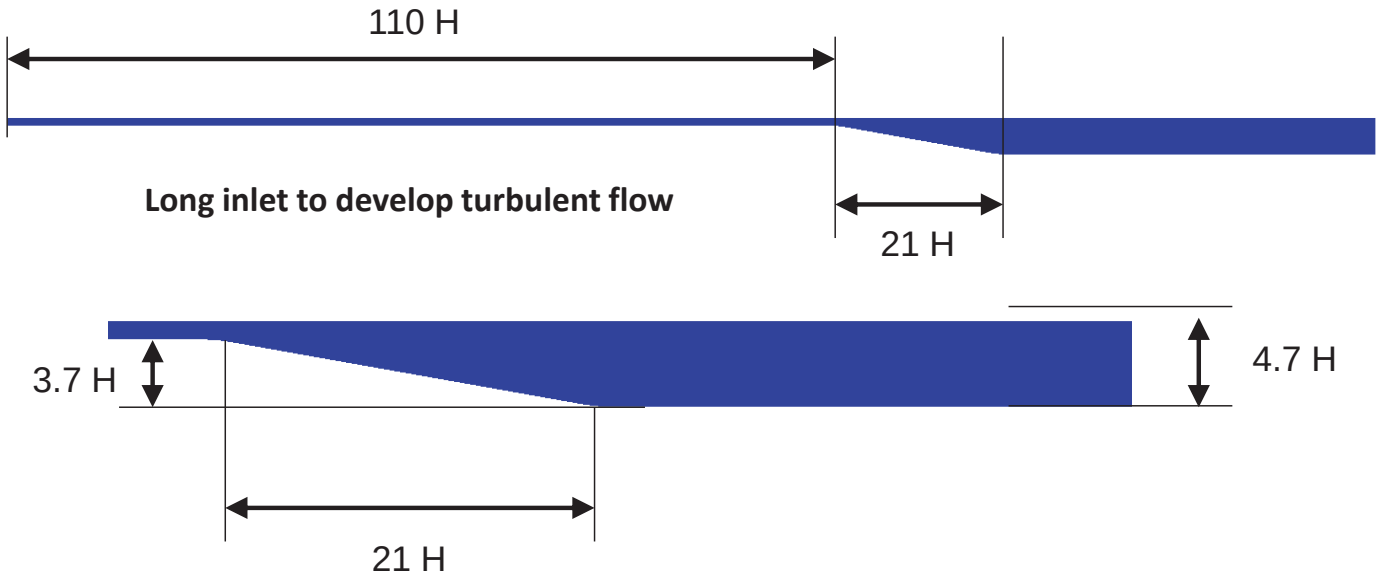
```
> fluentMeshToFoam –case diffuserBase fullGeometry2.msh
```

- Check the mesh and see the log file

```
> checkMesh –case diffuserBase
```

- Open the mesh in paraview (but de-activate all the fields since the case has not been set up properly)

Diffuser simulation: first case

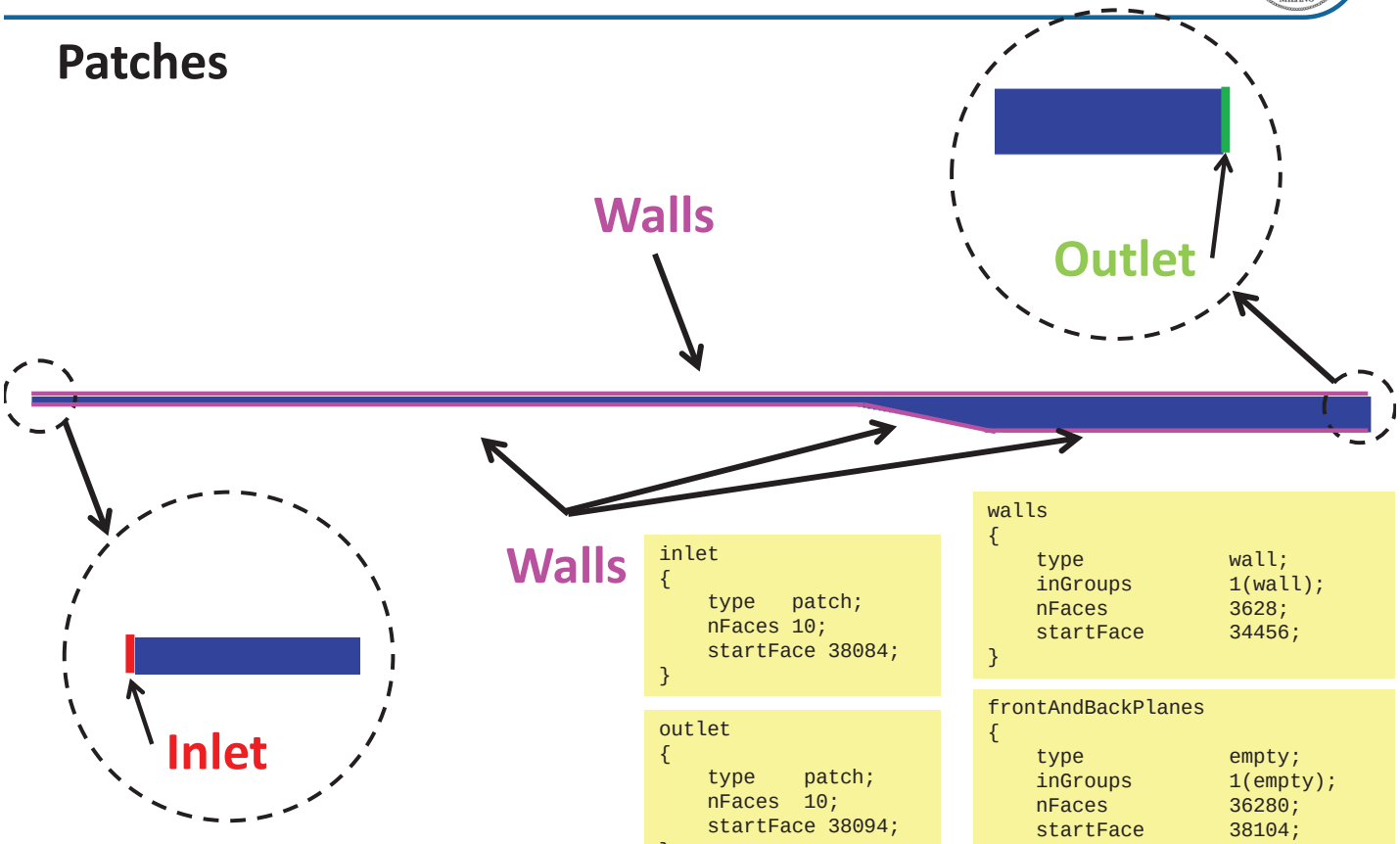


First attempt setup

- Mesh size from user experience
- Standard k-ε model for turbulence and standard wall functions
- First order convection schemes
- Standard OpenFOAM settings for equation solution

Diffuser simulation: mesh

Patches




```
inlet
{
    type    patch;
    nFaces 10;
    startFace 38084;
}
```

```
outlet
{
    type    patch;
    nFaces  10;
    startFace 38094;
}
```

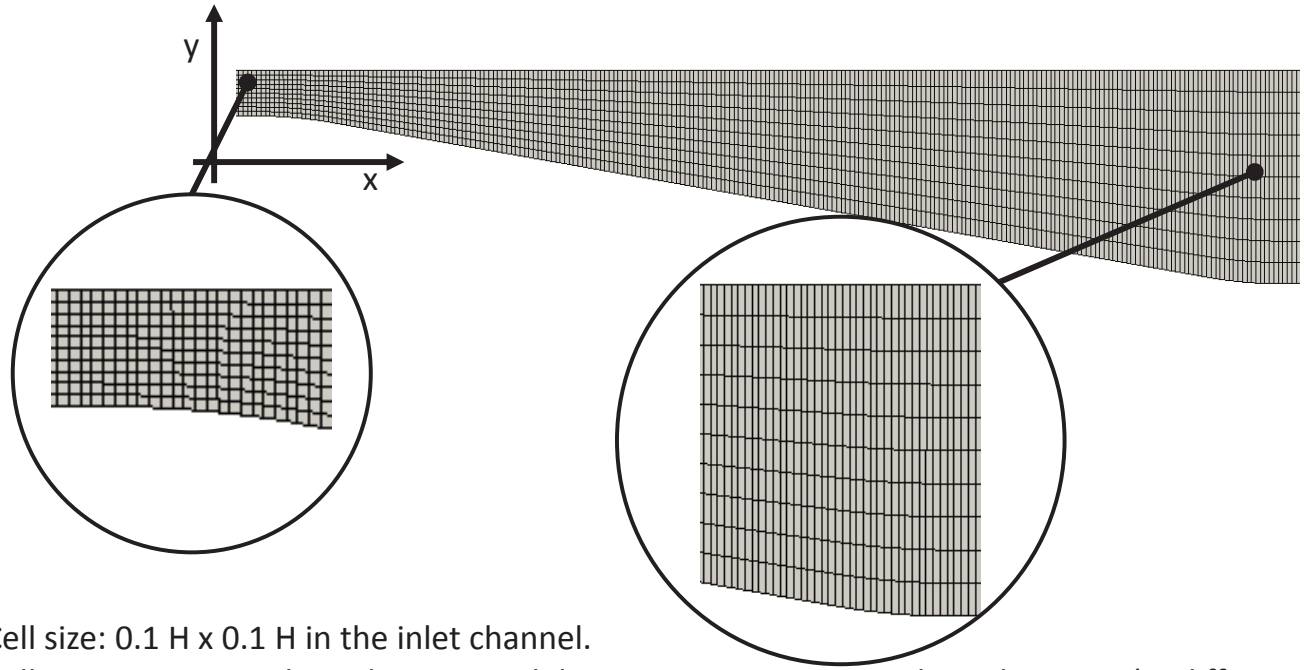
```
walls
{
    type            wall;
    inGroups        1(wall);
    nFaces          3628;
    startFace       34456;
}
```

```
frontAndBackPlanes
{
    type            empty;
    inGroups        1(empty);
    nFaces          36280;
    startFace       38104;
}
```

Diffuser simulation: mesh

Mesh in the diffuser region

Block structured grid



•Cell size: 0.1 H x 0.1 H in the inlet channel.
•Cell size is constant along the x axis, while it grows up to 0.47 H along the y axis (at diffuser exit)
•This is a preliminary mesh, let's see how it works….

Diffuser simulation: properties

- The `constant` directory of the case includes:

```
> ls diffuserBase/constant
polyMesh RASProperties transportProperties
```

- Edit the `transportProperties` dictionary to set the kinematic viscosity (entry `nu`) to the air value (10^{-4} m^2/s):

```
nu              nu [0 2 -1 0 0 0 0] 1e-04;
```

- In the `RASProperties` dictionary it is necessary to define the turbulence model which will be used:

```
RASModel kEpsilon;
turbulence on;
```

In this case, the k-ε model will be used for turbulence (`RASModel` entry) and model equations will be solved.
Set `turbulence off;` in case the user intends to keep turbulence parameters constant.
Set `RASModel laminar;` for a laminar flow simulation.

Diffuser simulation: initial conditions

- In the `0` directory we have:

```
> ls diffuserBase/0
epsilon  k  nut  nuTilda  p U
```

- We need to set proper internal values and boundary conditions for:
  - Pressure (`p`) and velocity (`U`)
  - Turbulent kinetic energy (`k`) and dissipation rate (`epsilon`)
- We also need to check the turbulent viscosity field `nut` and the wall functions applied to `k`, `epsilon` and `nut` fields.
- From Re = 20000 and $\nu_{air} = 10^{-4}$ m^2/s it is possible to estimate the inlet velocity:

$$\mathrm{Re} = \frac{UL}{\nu} \Rightarrow U = \frac{\mathrm{Re}\cdot\nu}{L} = \frac{20000\cdot 10^{-4}\ m^2/s}{1\,m} = 2\,m/s$$

Diffuser simulation: initial conditions

Velocity field U

- It is a vector field, hence the three components must be specified in the absolute cartesian coordinate system (the same of the mesh)
- Velocity is uniform and equal to 2 m/s (along the x-axis) at the inlet boundary. Then a turbulent profile will develop along the channel.
- At outlet the zero gradient boundary condition is specified
- Velocity is zero at the walls (no-slip condition)

```
dimensions      [0 1 -1 0 0 0 0];
internalField   uniform (0 0 0);
boundaryField
{
    inlet
    {
        type            fixedValue;
        value           uniform (2 0 0);
    }
    outlet
    {
        type            zeroGradient;
    }
    walls
    {
        type            fixedValue;
        value           uniform (0 0 0);
    }
    frontAndBack
    {
        type            empty;
    }
}
```

T. Lucchini: Incompressible, steady-state turbulent flow simulations in OpenFOAM®

Diffuser simulation: initial conditions

Velocity field U

Inlet

- Other possible boundary conditions where velocity field is specified at inlet (similar to `fixedValue`):
  - `surfaceNormalFixedValue`
  - `movingWallVelocity`
  - `inletOutlet`
  - `flowRateInletVelocity`
  - `outletInlet`
- With all these boundary conditions (including `fixedValue`) pressure should be set in the same patch to `zeroGradient` for consistency.

```
dimensions      [0 1 -1 0 0 0 0];
internalField   uniform (0 0 0);
boundaryField
{
    inlet
    {
        type            fixedValue;
        value           uniform (2 0 0);
    }
    outlet
    {
        type            zeroGradient;
    }
    walls
    {
        type            fixedValue;
        value           uniform (0 0 0);
    }
    frontAndBack
    {
        type            empty;
    }
}
```


Diffuser simulation: initial conditions

Velocity field U

Outlet

- `zeroGradient` is consistent with the pressure boundary condition (`fixedValue`) at outlet.
- Other velocity boundary conditions at outlet compatible with pressure fixed value:
  - `pressureDirectedInletOutletVelocity`
  - `pressureDirectedInletVelocity`
  - `pressureInletOutletParSlipVelocity`
  - `pressureInletOutletVelocity`
  - `pressureInletUniformVelocity`
  - `pressureInletVelocity`
  - `pressureNormalInletOutletVelocity`
  - …

See also:

`$FOAM_SRC/finiteVolume/fields/fvPatchFields`

```
dimensions      [0 1 -1 0 0 0 0];
internalField   uniform (0 0 0);
boundaryField
{
    inlet
    {
        type            fixedValue;
        value           uniform (2 0 0);
    }
    outlet
    {
        type            zeroGradient;
    }
    walls
    {
        type            fixedValue;
        value           uniform (0 0 0);
    }
    frontAndBack
    {
        type            empty;
    }
}
```

T. Lucchini: Incompressible, steady-state turbulent flow simulations in OpenFOAM®

Diffuser simulation: initial conditions

Pressure field p

- In incompressible solvers, OpenFOAM transports p/ρ and the pressure is relative. Which is why pressure dimensions are m^2/s^2.
- To be consistent with velocity field, at inlet pressure is set to zeroGradient.
- At outlet pressure is set to 0.
- At walls pressure is zeroGradient, consistently with velocity field.

```
dimensions      [0 2 -2 0 0 0 0];
internalField   uniform 0;
boundaryField
{
    inlet
    {
        type            zeroGradient;
    }
    outlet
    {
        type            fixedValue;
        value           uniform 0;
    }
    walls
    {
        type            zeroGradient;
    }
    frontAndBack
    {
        type            empty;
    }
}
```


Diffuser simulation: initial conditions

Pressure field p

- In case pressure has to be specified at the inlet boundary, possible solutions are:
  - `fixedValue`
  - `totalPressure`
  - `syringePressure`
  - `uniformTotalPressure`
  - `waveTransmissive`
  - …

See also:

`$FOAM_SRC/finiteVolume/fields/fvPatchFields`

```
dimensions      [0 2 -2 0 0 0 0];
internalField   uniform 0;
boundaryField
{
    inlet
    {
        type            zeroGradient;
    }
    outlet
    {
        type            fixedValue;
        value           uniform 0;
    }
    walls
    {
        type            zeroGradient;
    }
    frontAndBack
    {
        type            empty;
    }
}
```

T. Lucchini: Incompressible, steady-state turbulent flow simulations in OpenFOAM®

Diffuser simulation: initial conditions

Coupling pressure and velocity boundary conditions - 1

velocity

```
boundaryField
{
    inlet
    {
        type fixedValue;
        value uniform (1 0 0);
    }
    outlet
    {
        type zeroGradient;
    }
}
```

pressure

```
boundaryField
{
    inlet
    {
        type zeroGradient;
    }
    outlet
    {
        type fixedValue;
        value uniform 0
    }
}
```

```
boundaryField
{
    inlet
    {
        type zeroGradient;
    }
    outlet
    {
        type pressureInletOutletVelocity;
        // other entries here…
    }
}
```

```
boundaryField
{
    inlet
    {
        type totalPressure;
        p0 1;
        value uniform 0;
        // other entries here…
    }
    outlet
    {
        type fixedValue;
        value uniform 0;
    }
}
```


Diffuser simulation: initial conditions

Coupling pressure and velocity boundary conditions - 2

velocity

```
boundaryField
{
    inlet
    {
        type flowRateInletVelocity;
        volumetricFlowRate 0.3;
        value uniform (0 0 0);
    }
    outlet
    {
        type zeroGradient;
    }
}
```

pressure

```
boundaryField
{
    inlet
    {
        type zeroGradient;
    }
    outlet
    {
        type fixedValue;
        value uniform 0
    }
}
```

T. Lucchini: Incompressible, steady-state turbulent flow simulations in OpenFOAM®

Diffuser simulation: initial conditions

Turbulence fields

- The fields which are directly related with the employed turbulence model must be always specified. Hence, in case k-ε model is used, k and `epsilon` fields must be initialized.

- Specification of the `nut` (turbulent viscosity) field is optional. But in case it is specified in the start time directory, its boundary conditions (mainly at wall) must be consistent with what was specified for k and `epsilon`.

Diffuser simulation: initial conditions

Turbulent kinetic energy k

- Initial internal value: directly taken from the pitzDaily tutorial. It will not affect the final solution.
- At the `inlet` boundary, turbulent kinetic energy is computed assuming an intensity equal to 5% the velocity value.
- At `outlet`, the `zeroGradient` boundary condition is used (`inletOutlet` can sometimes increase the solution stability)

```
dimensions      [0 2 -2 0 0 0 0];
internalField   uniform 0.375;
boundaryField
{
    walls
    {
        type kqRWallFunction;
        value uniform 0.375;
    }
    inlet
    {
        type turbulentIntensityKineticEnergyInlet;
        intensity 0.05;
        value uniform 0.375;
    }
    outlet
    {
        type zeroGradient;
    }
    frontAndBackPlanes
    {
        type empty;
    }
}
```

T. Lucchini: Incompressible, steady-state turbulent flow simulations in OpenFOAM®

Diffuser simulation: initial conditions

Turbulent kinetic energy k

- Wall functions should be used for all the patches which are identified as `wall` type in the mesh boundary.
- Available wall functions for k:
  - ✓ `kLowReWallFunction`
  - ✓ `kqRWallFunction`
- `kqRWallFunction` is for high-Reynolds meshes and we will use this one for this first case.

```
dimensions      [0 2 -2 0 0 0 0];
internalField   uniform 0.375;
boundaryField
{
   walls
   {
        type kqRWallFunction;
        value uniform 0.375;
   }
    inlet
    {
        type turbulentIntensityKineticEnergyInlet;
        intensity 0.05;
        value uniform 0.375;
    }
    outlet
    {
        type zeroGradient;
    }
    frontAndBackPlanes
    {
        type empty;
    }
}
```


Diffuser simulation: initial conditions

Turbulent kinetic energy dissipation rate `epsilon`

- Initial internal value: directly taken from the pitzDaily tutorial. It will not affect the final solution.
- At the `inlet` boundary, ε is estimated from k and a mixing length equal to 1% of the channel heigth.
- At `outlet`, the `zeroGradient` boundary condition is used (inletOutlet can sometimes increase the solution stability)

```
dimensions      [0 2 -3 0 0 0 0];

internalField   uniform 14.855;

boundaryField
{
    walls
    {
        type epsilonWallFunction;
        value uniform 14.855;
        Cmu 0.09;
        kappa 0.41;
        E 9.8;
    }
    inlet
    {
        type turbulentMixingLengthDissipationRateInlet;
        mixingLength 0.01;
        value uniform 14.855;
    }
    outlet
    {
        type zeroGradient;
    }
    frontAndBackPlanes
    {
        type empty;
    }
}
```

T. Lucchini: Incompressible, steady-state turbulent flow simulations in OpenFOAM®

Diffuser simulation: initial conditions

Turbulent kinetic energy dissipation rate `epsilon`

- Wall functions should be used for all the patches which are identified as `wall` type in the mesh boundary.
- Available wall functions for k:
  - ✓ `epsilonLowReWallFunction`
  - ✓ `epsilonWallFunction`
- `epsilonWallFunction` is for high-Reynolds meshes and we will use this one for this first case. Entries `Cmu`, `kappa` and `E` are optional.

```
dimensions      [0 2 -3 0 0 0 0];
internalField   uniform 14.855;
boundaryField
{
   walls
   {
        type epsilonWallFunction;
        value uniform 14.855;
        Cmu 0.09;
        kappa 0.41;
        E 9.8;
   }
    inlet
    {
        type turbulentMixingLengthDissipationRateInlet;
        mixingLength 0.01;
        value uniform 14.855;
    }
    outlet
    {
        type zeroGradient;
    }
    frontAndBackPlanes
    {
        type empty;
    }
}
```


Diffuser simulation: initial conditions

Turbulent viscosity `nut`

- At inlet and outlet boundaries, nut is computed directly from turbulence fields and the selected turbulence model.
- At the walls, the `nutkWallFunction` is applied. This corresponds to the computation of turbulence viscosity according to high-reynolds wall functions. Available wall-functions for `nut` are:
  - `nutkAtmRoughWallFunction`
  - `nutkRoughWallFunction`
  - `nutkWallFunction`
  - `nutLowReWallFunction`
  - `nutURoughWallFunction`
  - `nutUSpaldingWallFunction`
  - `nutUTabulatedWallFunction`
  - `nutUWallFunction`
  - `nutWallFunction`

```
dimensions      [0 2 -1 0 0 0 0];

internalField   uniform 0;

boundaryField
{
    walls
    {
        type            nutkWallFunction;
        Cmu             0.09;
        kappa           0.41;
        E               9.8;
        value           uniform 0;
    }
    inlet
    {
        type            calculated;
        value           uniform 0;
    }
    outlet
    {
        type            calculated;
        value           uniform 0;
    }
    frontAndBackPlanes
    {
        type            empty;
    }
}
```

T. Lucchini: Incompressible, steady-state turbulent flow simulations in OpenFOAM®

Diffuser simulation: directory `system`

`controlDict`

- 1000 iterations are specified. Results will be written every 50 time-steps.

`fvSchemes`

```
ddtSchemes
{
    default         steadyState;
}

gradSchemes
{
    default         Gauss linear;
}

divSchemes
{
    default         none;
    div(phi,U)      bounded Gauss upwind;
    div(phi,k)      bounded Gauss upwind;
    div(phi,epsilon) bounded Gauss upwind;
    div(phi,R)      bounded Gauss upwind;
    div(R)          Gauss linear;
    div(phi,nuTilda) bounded Gauss upwind;
    div((nuEff*dev(T(grad(U))))) Gauss linear;
}
```

```
laplacianSchemes
{
    default         Gauss linear corrected;
}

interpolationSchemes
{
    default         linear;
}

snGradSchemes
{
    default         corrected;
}

fluxRequired
{
    default         no;
    p               ;
}
```


Diffuser simulation: directory `system`

`controlDict`

- 1000 iterations are specified. Results will be written every 50 time-steps.

`fvSchemes`

```
ddtSchemes
{
    default         steadyState;
}

gradSchemes
{
    default         Gauss linear;
}

divSchemes
{
    default         none;
    div(phi,U)      bounded Gauss upwind;
    div(phi,k)      bounded Gauss upwind;
    div(phi,epsilon) bounded Gauss upwind;
    div(phi,R)      bounded Gauss upwind;
    div(R)          Gauss linear;
    div(phi,nuTilda) bounded Gauss upwind;
    div((nuEff*dev(T(grad(U))))) Gauss linear;
}
```

```
laplacianSchemes
{
    default         Gauss linear corrected;
}

interpolationSchemes
{
    default         linear;
}

snGradSchemes
{
    default         corrected;
}

fluxRequired
{
    default         no;
    p               ;
}
```

T. Lucchini: Incompressible, steady-state turbulent flow simulations in OpenFOAM®

Diffuser simulation: directory `system`

`fvSchemes` - 1

```
ddtSchemes
{
    default         steadyState;
}

gradSchemes
{
    default         Gauss linear;
}

divSchemes
{
    default         none;
    div(phi,U)      bounded Gauss upwind;
    div(phi,k)      bounded Gauss upwind;
    div(phi,epsilon) bounded Gauss upwind;
    div(phi,R)      bounded Gauss upwind;
    div(R)          Gauss linear;
    div(phi,nuTilda) bounded Gauss upwind;
    div((nuEff*dev(T(grad(U))))) Gauss linear;
}
```

Steady-state discretization for **time derivative** is correct when simple foam is used. This is not applied to U and p (they do not have time derivatives), but it will be applied to turbulence fields (turbulence model equations include time derivatives)

Diffuser simulation: directory `system`

`fvSchemes` - 2

```
ddtSchemes
{
    default         steadyState;
}

gradSchemes
{
    default         Gauss linear;
}

divSchemes
{
    default         none;
    div(phi,U)      bounded Gauss upwind;
    div(phi,k)      bounded Gauss upwind;
    div(phi,epsilon) bounded Gauss upwind;
    div(phi,R)      bounded Gauss upwind;
    div(R)          Gauss linear;
    div(phi,nuTilda) bounded Gauss upwind;
    div((nuEff*dev(T(grad(U))))) Gauss linear;
}
```

Linear interpolation will be used to discretize all the **gradients** employed in the transport equations.

To improve stability on poor meshes limit the velocity gradient by adding this line to the sub-dictionary:

```
grad(U) cellLimited Gauss linear 1;
```

T. Lucchini: Incompressible, steady-state turbulent flow simulations in OpenFOAM®

Diffuser simulation: directory system

fvSchemes - 3

```
ddtSchemes
{
    default         steadyState;
}

gradSchemes
{
    default         Gauss linear;
}

divSchemes
{
    default         none;
    div(phi,U)      bounded Gauss upwind;
    div(phi,k)      bounded Gauss upwind;
    div(phi,epsilon) bounded Gauss upwind;
    div(phi,R)      bounded Gauss upwind;
    div(R)          Gauss linear;
    div(phi,nuTilda) bounded Gauss upwind;
    div((nuEff*dev(T(grad(U))))) Gauss linear;
}
```

Convection schemes are very important in terms of accuracy and stability.

Here:

- 1st order (upwind) is used for all convection terms in transport equation.
- Linear interpolation is used for source terms which are not convective but are computed as divergence (like the Reynolds stress tensor)

Diffuser simulation: directory system

fvSchemes - 4

```
ddtSchemes
{
    default         steadyState;
}

gradSchemes
{
    default         Gauss linear;
}

divSchemes
{
    default         none;
    div(phi,U)      bounded Gauss upwind;
    div(phi,k)      bounded Gauss upwind;
    div(phi,epsilon) bounded Gauss upwind;
    div(phi,R)      bounded Gauss upwind;
    div(R)          Gauss linear;
    div(phi,nuTilda) bounded Gauss upwind;
    div((nuEff*dev(T(grad(U))))) Gauss linear;
}
```

Other options for convection terms (examples):

To increase stability and keep accuracy:

```
div(phi,U)  bounded Gauss linearUpwindV grad(U);
```

Second order

```
div(phi,U)  bounded Gauss limitedLinear 1.0;
div(phi,U)  bounded Gauss SFCD;
div(phi,U)  bounded Gauss GammaV 1.0;
div(phi,k)  bounded Gauss SFCD;
div(phi,k)  bounded Gauss Gamma 1.0;
```

T. Lucchini: Incompressible, steady-state turbulent flow simulations in OpenFOAM®

Diffuser simulation: directory system

fvSchemes - 5

```
laplacianSchemes
{
    default         Gauss linear corrected;
}

interpolationSchemes
{
    default         linear;
}

snGradSchemes
{
    default         corrected;
}

fluxRequired
{
    default         no;
    p               ;
}
```

In the diffusion term, gradient is discretized as follows:

$$\mathbf{S}\cdot(\nabla\phi)_f = \underbrace{\boldsymbol{\Delta}\cdot(\nabla\phi)_f}_{\text{orthogonal contribution}} + \underbrace{\mathbf{k}\cdot(\nabla\phi)_f}_{\text{non-orthogonal correction}}$$

The first term is treated implicitly (it goes into the matrix coefficient), the second one which is the non-orthogonal part is handled as a source term and, for this reason, it can create instability.

$$\mathbf{S}\cdot(\nabla\phi)_f = \boldsymbol{\Delta}\cdot\frac{\phi_N-\phi_P}{|\mathbf{d}|} + \mathbf{k}\cdot(\nabla\phi)_f$$

Diffuser simulation: directory system

fvSchemes - 6

```
laplacianSchemes
{
    default         Gauss linear corrected;
}

interpolationSchemes
{
    default         linear;
}

snGradSchemes
{
    default         corrected;
}

fluxRequired
{
    default         no;
    p               ;
}
```

Possible choices for Laplacian schemes

- All the non-orthogonal part is retained:

```
default Gauss linear corrected;
```

- Non-ortogonal part limited to 0.5 times the orthogonal one (stability is increased by accuracy reduced):

```
default Gauss linear limited 0.5;
```

- Non-orthogonal part completely discarded:

```
default Gauss linear uncorrected;
```

- Different schemes for turbulent fields, default for other laplacians…

```
default Gauss linear limited 1.0;
laplacian(DepsilonEff, epsilon) Gauss
linear limited 0.5;
laplacian(DkEff, epsilon) Gauss linear
limited 0.5;
```

T. Lucchini: Incompressible, steady-state turbulent flow simulations in OpenFOAM®

Diffuser simulation: directory system

fvSchemes - 7

```
laplacianSchemes
{
    default         Gauss linear corrected;
}

interpolationSchemes
{
    default         linear;
}

snGradSchemes
{
    default         corrected;
}

fluxRequired
{
    default         no;
    p               ;
}
```

snGradSchemes: discretization of surface normal gradients. Whether not directly employed in transport equations, no need to use specific schemes (check the user guide).

interpolationSchemes: variable interpolation at face centres values (check the user guide).

fluxRequired: needed only for p field in the pressure-velocity coupling stage.

Diffuser simulation: directory system

fvSolution - 1

```
solvers
{

// settings for solution of transport equations

}

SIMPLE
{

// settings for the SIMPLE algorithm and convergence

}

relaxationFactors
{

// relaxation factors

}
```

The fvSolution dictionary is made by three different sub-dictionaries:

1) solvers: settings for solution of transport equations.

2) SIMPLE : settings for the SIMPLE pressure-velocity coupling and controls for automatic convergence.

3) relaxationFactors: under-relaxation factors for the fields and equations.

Diffuser simulation: directory `system`

`fvSolution` - 2

```
solvers
{
    p
    {
        solver          GAMG;
        tolerance       1e-06;
        relTol          0.1;
        smoother        GaussSeidel;
        nPreSweeps      0;
        nPostSweeps     2;
        cacheAgglomeration on;
        agglomerator    faceAreaPair;
        nCellsInCoarsestLevel 10;
        mergeLevels     1;
    }

    "(U|k|epsilon|R|nuTilda)"
    {
        solver          smoothSolver;
        smoother        symGaussSeidel;
        tolerance       1e-05;
        relTol          0.1;
    }
}
```

The `solvers` subdictionary / a

Pressure equation is solved using an algebraic multi-grid method (GAMG). The absolute tolerance is 1e-6, the relative one is 0.1. Before solving the equation, the initial residual is evaluated based on the current values of the field.

After each solver iteration the residual is re-evaluated. The solver stops if at least one of the following conditions are reached:

- the residual falls below the *solver tolerance*, `tolerance`;
- the ratio of current to initial residuals falls below the *solver relative tolerance*, `relTol`;
- the number of iterations exceeds a *maximum number of iterations*, `maxIter`;

Diffuser simulation: directory `system`

`fvSolution` - 3

```
solvers
{
    p
    {
        solver          GAMG;
        tolerance       1e-06;
        relTol          0.1;
        smoother        GaussSeidel;
        nPreSweeps      0;
        nPostSweeps     2;
        cacheAgglomeration on;
        agglomerator    faceAreaPair;
        nCellsInCoarsestLevel 10;
        mergeLevels     1;
    }

    "(U|k|epsilon|R|nuTilda)"
    {
        solver          smoothSolver;
        smoother        symGaussSeidel;
        tolerance       1e-05;
        relTol          0.1;
    }
}
```

The `solvers` subdictionary / b

Alternatives for solution of pressure equation

```
solvers
{
    p
    {
        solver          PCG;
        tolerance       1e-06;
        relTol          0.1;
        preconditioner  DIC;
    }
}
```

GAMG is much faster than PCG

T. Lucchini: Incompressible, steady-state turbulent flow simulations in OpenFOAM®

Diffuser simulation: directory system

fvSolution - 4

```
solvers
{
    p
    {
        solver          GAMG;
        tolerance       1e-06;
        relTol          0.1;
        smoother        GaussSeidel;
        nPreSweeps      0;
        nPostSweeps     2;
        cacheAgglomeration on;
        agglomerator    faceAreaPair;
        nCellsInCoarsestLevel 10;
        mergeLevels     1;
    }

    "(U|k|epsilon|R|nuTilda)"
    {
        solver          smoothSolver;
        smoother        symGaussSeidel;
        tolerance       1e-05;
        relTol          0.1;
    }
}
```

The solvers subdictionary / c

All other fields are solved using a smoother, which is in this case represented by the Gauss-Seidel method. The absolute tolerance is 1e-5, the relative is 0.1.

Another option (more stable) is to use Preconditioned bi-conjugate Choleski method with DILU preconditioner:

```
"(U|k|epsilon|R|nuTilda)"
{
    solver          PBiCG;
    preconditioner  DILU;
    tolerance       1e-05;
    relTol          0.1;
}
```


Diffuser simulation: directory system

fvSolution - 5

```
SIMPLE
{
    nNonOrthogonalCorrectors 0;

    residualControl
    {
        p               1e-2;
        U               1e-3;
        "(k|epsilon|omega)" 1e-3;
    }
}
```

The SIMPLE subdictionary

Number of non-orthogonal correctors is `nNonOrthogonalCorrectors`.

Since the pressure equation is:

$$\frac{\delta \tilde{u}_{i,P}^{m*}}{\delta x_i} - \frac{\delta}{\delta x_i}\left[\frac{1}{A_P^{u_i}}\left(\frac{\delta p^m}{\delta x_i}\right)^P\right] = 0$$

And Laplacian is decomposed into two parts:

- orthogonal: implicit
- non-ortogonal: source term

On non-orthogonal meshes, the use of non-orthogonal correctors is expected to increase the convergence of pressure equation. This is mainly true for unsteady solvers but not completely for steady-state solvers, since final solution is computed iteratively. Use 0 – 4 **`nNonOrthogonalCorrectors`** with simpleFoam, depending on mesh non-orthogonality

T. Lucchini: Incompressible, steady-state turbulent flow simulations in OpenFOAM®

Diffuser simulation: directory `system`

`fvSolution` - 6

```
SIMPLE
{
    nNonOrthogonalCorrectors 0;

    residualControl
    {
        p               1e-2;
        U               1e-3;
        "(k|epsilon|omega)" 1e-3;
    }
}
```

The `SIMPLE` subdictionary

`residualControl`: parameters for convergence control. When initial residuals of all the specified quantities in this sub-dictionary fall down the given tolerances, simulation is considered to be converged and is arrested immediately (the last iteration is written).

How to handle residual control:

1) No specification: simulation will be arrested at `endTime`, according to `controlDict` settings
2) In case `residualControl` is used: relate the convergence criteria with what was specified in `fvSolution` in the `tolerance` entry.
   - For pressure use a $10^3/10^4$ factor : `tolerance 1e-6; residualControl 1e-2`
   - For other fields use a $10^2/10^3$ factor

Diffuser simulation: directory `system`

`fvSolution` - 7

```
relaxationFactors
{
    fields
    {
        p               0.3;
    }
    equations
    {
        U               0.7;
        k               0.7;
        epsilon         0.7;
        R               0.7;
        nuTilda         0.7;
    }
}
```

The `relaxationFactors` subdictionary

- Under-relaxation is a technique used to improve the stability of a computation, mainly for steady-state problems.
- Under-relaxation works by limiting the amount which a variable changes from one iteration to the next, either by modifying the solution matrix and source prior to solving for a field or by modifying the field directly.
- An under-relaxation factor α, $0<\alpha\leq 1$, specifies the amount of under-relaxation, ranging from not at all ($\alpha = 1$) and increasing its strength as it tends to zero.

Diffuser simulation: directory `system`

`fvSolution` - 8

```
relaxationFactors
{
    fields
    {
        p               0.3;
    }
    equations
    {
        U               0.7;
        k               0.7;
        epsilon         0.7;
        R               0.7;
        nuTilda         0.7;
    }
}
```

The `relaxationFactors` subdictionary

- An optimum choice of α is the one that is small enough to ensure stable computation but large enough to move the iterative process forward quickly.

Recommendation: the sum of relaxation factors for p and U must give one!

```
relaxationFactors
{
    fields
    {
        p 0.1;
    }
    equations
    {
        U 0.9;
    }
}
```

```
relaxationFactors
{
    fields
    {
        p 0.2;
    }
    equations
    {
        U 0.8;
    }
}
```

```
relaxationFactors
{
    fields
    {
        p 0.3;
    }
    equations
    {
        U 0.7;
    }
}
```


Diffuser simulation: directory `system`

`fvSolution` - 9

```
relaxationFactors
{
    fields
    {
        p               0.1;
    }
    equations
    {
        U               0.9;
        k               0.3;
        epsilon         0.3;
        R               0.3;
        nuTilda         0.3;
    }
}
```

The `relaxationFactors` subdictionary

- Very conservative, slow convergence

Diffuser simulation: directory system

fvSolution - 10

```
relaxationFactors
{
    fields
    {
        p               0.2;
    }
    equations
    {
        U               0.8;
        k               0.5;
        epsilon         0.5;
        R               0.5;
        nuTilda         0.5;
    }
}
```

The relaxationFactors subdictionary

- Conservative, convergence ok

Diffuser simulation: directory system

fvSolution - 11

```
relaxationFactors
{
    fields
    {
        p               0.3;
    }
    equations
    {
        U               0.7;
        k               0.7;
        epsilon         0.7;
        R               0.7;
        nuTilda         0.7;
    }
}
```

The relaxationFactors subdictionary

- Fast convergence (less stable)

T. Lucchini: Incompressible, steady-state turbulent flow simulations in OpenFOAM®

Diffuser simulation: parallelization

- Except for very simple cases, all simulations need to be run in parallel in order to achieve results in a very short time. OpenFOAM is a parallel code.
- Parallelization is done by means of domain decomposition which is carried out before running the simulation. Equations are separately solved by each processor on any domain, processor boundaries are then used to exchange information among the different processors.
- Domain decomposition: `decomposePar`
  `$FOAM_UTILITIES/parallelProcessing/decomposePar`
- Domain reconstruction: `reconstructPar`
  `$FOAM_UTILITIES/parallelProcessing/reconstructPar`
- Copy the `decomposeParDict` dictionary from the utility directory and put it in the `system` directory of your case:

```
> cp $FOAM_UTILITIES/parallelProcessing/decomposePar/decomposeParDict diffuserBase/system
```


Diffuser simulation: parallelization

`decomposeParDict` dictionary

```
numberOfSubdomains  2;

method          scotch;
// method          hierarchical;
// method          simple;
// method          metis;
// method          manual;
// method          multiLevel;
// method          structured;
```

`numberOfSubdomains` : number of processor domain the case has to be decomposed. Set it to a number of processors which is lower or equal to the CPU of the machine you are using.

`method` : the geometry domain can be decomposed in different ways:

- `scotch`: attempts to minimise the number of processor boundaries
- `hierarchical`: same as simple, except the user specifies the order in which the directional split is done
- `simple`: simple geometric decomposition in which the domain is split into pieces by direction
- `manual`: the user directly specifies the allocation of each cell to a particular processor

Further information about domain decomposition (and related dictionaries) are available here

Diffuser simulation

- **Running the first case:**

1) Let's assume we will run the case on 4 processors with scotch decomposition. Edit the decomposeParDict file of the diffuserBase case by setting 4 in the numberOfSubdomains entry and scotch in the decompositionMethod entry. No other entries have to be specified.

```
numberOfSubdomains  4;

method          scotch;
```

2) Decompose the domain. Run the decomposePar utility:

```
> decomposePar –case diffuserBase
```

or, inside the `diffuserBase` directory, just run:

```
> cd diffuserBase
> decomposePar
```


Diffuser simulation

- **Running the first case:**

3) Now, inside the case you will find `0`, `constant`, `system` and all the `processor*` directories where the domain (and all the fields) were consistently decomposed. To see a portion of the domain run, inside the case:

```
> paraFoam –case processor0
```

and you will see how the domain belonging to `processor0` looks like.

4) To decompose again the domain, run `decomposePar` but with the `–force` option

```
> decomposePar –case diffuserBase -force
```

BE CAREFUL with this command. It will delete all the processor directories (and results inside them…)

Diffuser simulation

- **Running the first case:**

3) Run the `simpleFoam` solver. From your run directory:

```
> mpirun –np 4 simpleFoam –case diffuserBase –parallel > diffuserBase.log &
```

inside the case directory:

```
> mpirun –np 4 simpleFoam –parallel > diffuserBase.log &
```

remember to use the `-parallel` option when you run the case in parallel and to specify before the solver command (in this case `simpleFoam`) `mpirun` followed by `–np` and the number of processor domains (same entry as `numberOfSubdomains` in `decomposeParDict` dictionary of your case).

Diffuser simulation

- **The case is finished. What to do?**

1) Look at the log file of the case to see what happened. It ends with this line:

```
SIMPLE solution converged in 438 iterations
```

438 iterations were necessary to reach convergence according to what was specified by the user in the `fvSolution` dictionary.

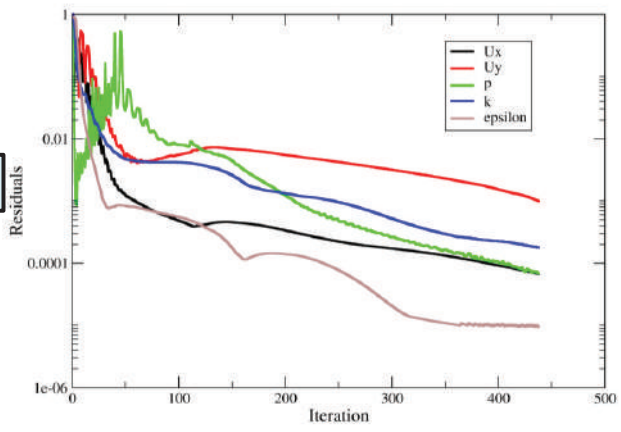
2) Check the residuals to see if they smoothly decreased for any of the solved field. Run:

```
> foamLog diffuserBase.log
```

Plot residuals inside the `log` directory

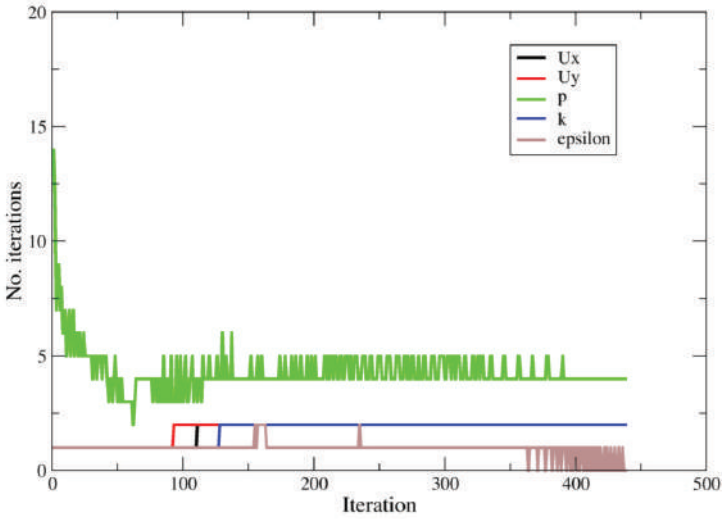
```
xmgrace -log y Ux_0 Uy_0 p_0 k_0 epsilon_0
```

Residuals went down the specified tolerance in a smooth way. `epsilon` convergence was much faster than other fields.



Diffuser simulation

By plotting the number of iterations `UxIters_0 , UyIters_0 , pIters_0,..` you can have a clearer idea of what happened in terms of equation solution. For all the equations (except epsilon in the last 100 iterations) at least one iteration was carried out. This is very important since all the fields are constantly updated together.



To post-process results, reconstruct the case with `reconstructPar`:

```
> reconstructPar
```

or

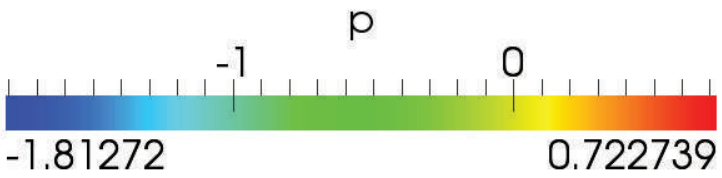
```
> reconstructPar -case diffuserBase
```

Now that we are sure that the case converged, we can look at results with `paraFoam` and make some post-processing…

Diffuser simulation

Fields

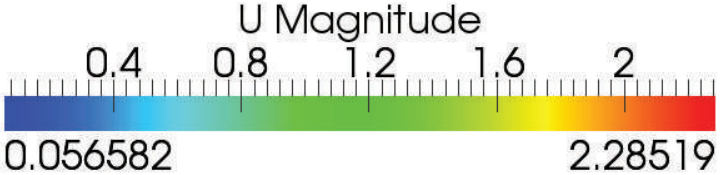
pressure



Pressure decreases inside the duct

Pressure recovery inside the diffuser

velocity



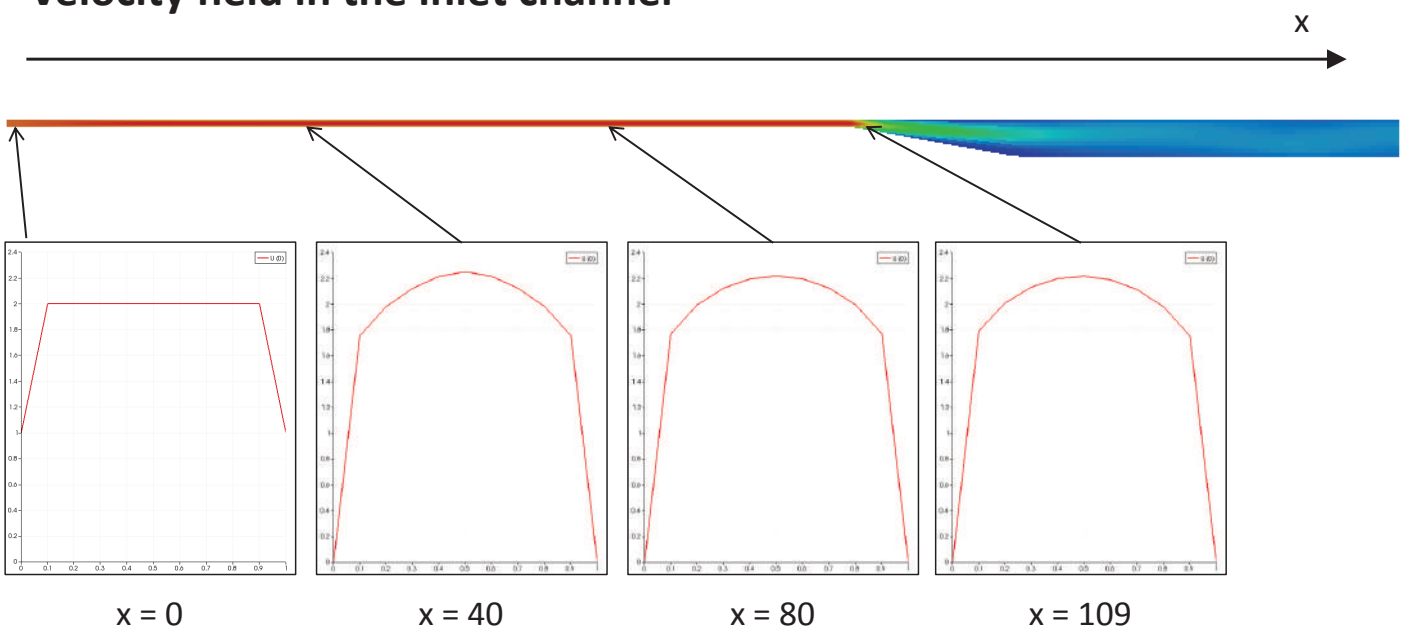
Re-development of the boundary layer

Turbulent flow development in the inlet pipe

Flow detachment due to adverse pressure gradient

Diffuser simulation

Velocity field in the inlet channel

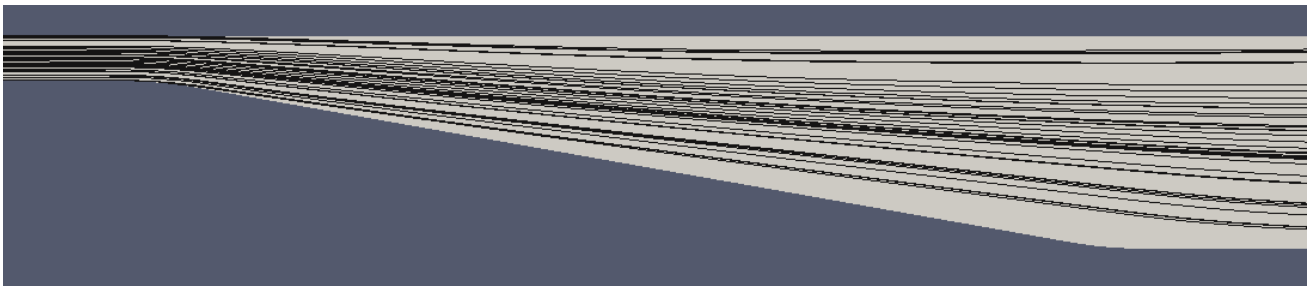


Visualize the evolution of velocity profiles in the inlet channel using the **Plot over line** filter in paraview

Diffuser simulation

Velocity field in the inlet channel

Use the **Stream tracers** filter in paraview to see the flow streamlines in the channel



What about the flow recirculation? Which actions to consider now?

1) Check the y+ values to see if they are in the proper range
   - If not rerun the simulation with a more refined mesh at the walls
2) Compare results with experimental data (if available)
3) See if results improve when using lower tolerances for residuals
4) Use high-order numerical schemes
5) Check if a refined mesh provides better results
6) Change the turbulence model

Diffuser simulation

Checking the y+ values

1) Run the `yPlusRAS` utility and see the output at the last time-step

```
> yPlusRAS -case diffuserBase
```

- The output reports for the last time-step

```
Patch 0 named walls y+ : min: 34.4434 max: 126.287 average: 62.2359
```

- y+ should be in the 5-50 range (or 10-100). However, here it looks quite high and a more refined mesh is necessary.

- We will re-run the case with this new configuration:
  - Finer mesh, refined at walls
  - Lower tolerances for residuals

Diffuser simulation

Run the case with a refined mesh and finer tolerances

- Create a new case, called `diffuserRefined` (clone the `diffuserBase` case, remove the files `y` and `yPlus` in the `0` directory of the new case)
- Import the grid called `fullGeometryRefined.msh` in the new case.
- Check the mesh (use `checkMesh`) to see new size and quality
- Modify some entries in the `controlDict` and `fvSolution` dictionaries in the new case:

`controlDict`

```
application     simpleFoam;

startFrom       latestTime;

startTime       0;

stopAt          endTime;

endTime         4000;

deltaT          1;

writeControl    timeStep;

writeInterval   1000;
```

`fvSolution`

```
solvers
{
    p
    {
        solver          GAMG;
        tolerance       1e-10;
        relTol          0.01;
        smoother        GaussSeidel;
        nPreSweeps      0;
        nPostSweeps     2;
        cacheAgglomeration on;
        agglomerator    faceAreaPair;
        nCellsInCoarsestLevel 10;
        mergeLevels     1;
    }
    "(U|k|epsilon|R|nuTilda)"
    {
        solver          smoothSolver;
        smoother        symGaussSeidel;
        tolerance       1e-10;
        relTol          0.1;
    }
}
```

```
SIMPLE
{
    nNonOrthogonalCorrectors 0;

    residualControl
    {
        p               1e-5;
        U               1e-6;
        "(k|epsilon|omega)" 1e-6;
    }
}
```

Modifications with respect to the `diffuserBase` case displayed with the **blue** color

Diffuser simulation

Run the case with a refined mesh and finer tolerances

- The case setup was changed. We use the absolute tolerance in `fvSolution` equal to 1e-10 for all the fields. All transport equations will be almost always solved.
- Convergence will be reached when initial residuals on pressure will be lower than 1e-5 and, at the same time, residuals on all the other fields will be lower than 1e-6.
- More iterations with respect to the previous case are expected. To this end, `endTime` is set to `4000` and `writeInterval` is increased accordingly to avoid writing too much time-steps.
- Decompose the case (use `decomposePar`), then re-run it with `simpleFoam`

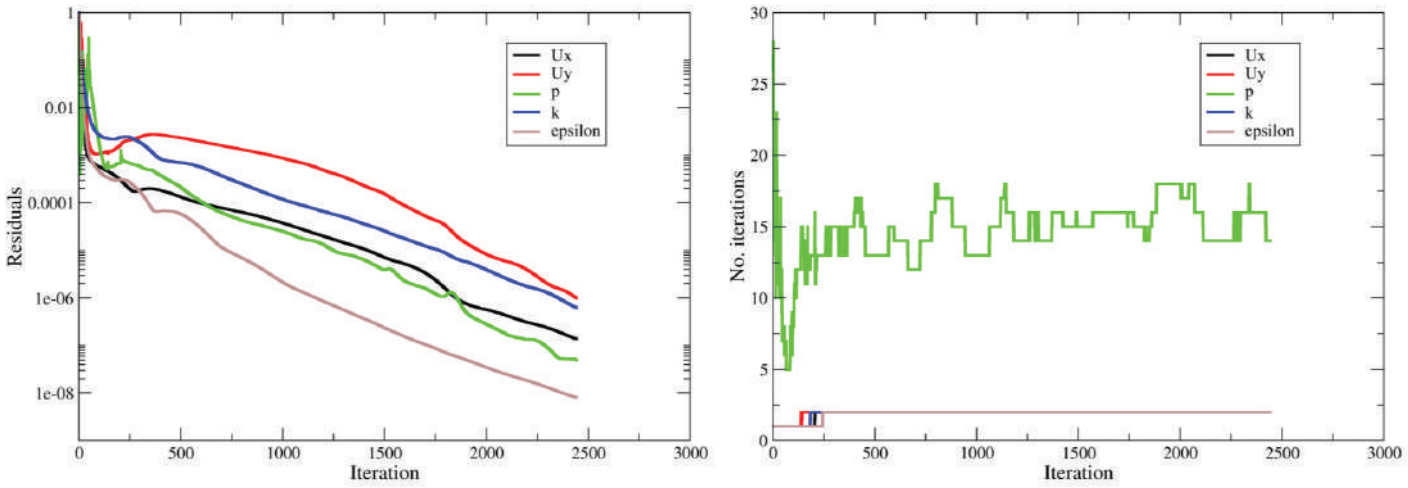
```
> cd diffuserRefine
> decomposePar
> mpirun -np 4 simpleFoam -parallel > diffuserRefine.log
```

- Initial conditions are the same as `diffuserBase` case.
  - ⇨ Use `mapFields` utility (if you want, not necessary) to initialize the flow field with the one of the latest time of the diffuserBase case. This is expected to reduce the number of iterations required to reach convergence.

Diffuser simulation

Run the case with a refined mesh and finer tolerances

- The case converged in more iterations (2442)



- Residuals are lower and all equations are solved in each iteration (also epsilon)
- Reconstruct the case (`reconstructPar`), visualize it with paraFoam.

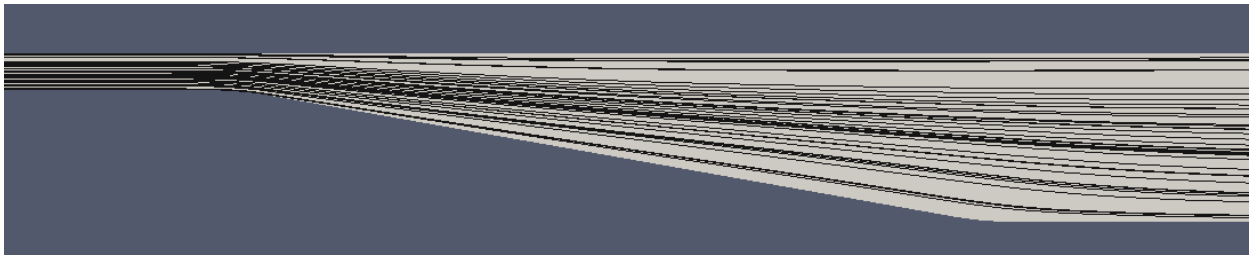
Diffuser simulation

Run the case with a refined mesh and finer tolerances

- yPlus is now inside the expected range (run `yPlusRAS` utility on the `diffuserRefined` case)

```
Patch 0 named walls y+ : min: 17.3399 max: 56.4667 average: 30.6195
```

- But still we do not have any detachment of the flow in the diffuser….



- What to do? Maybe it is a matter of accuracy (upwind is too diffusive, we know…)

Diffuser simulation

Run the `diffuserRefined` case with more accurate numerical schemes

- Change to second order (limitedLinear) the convection schemes in `fvSchemes` dictionary.
- Simulation will start from the last iteration (2442) and will be arrested (if not converged before) at iteration 8000.

```
startFrom       latestTime;

startTime       0;

stopAt          endTime;

endTime         8000;

deltaT          1;

writeControl    timeStep;

writeInterval   1000;
```

```
divSchemes
{
    default         none;
    div(phi,U)      bounded Gauss limitedLinear 1.0;
    div(phi,k)      bounded Gauss limitedLinear 1.0;
    div(phi,epsilon) bounded Gauss limitedLinear 1.0;
    div(phi,R)      bounded Gauss limitedLinear 1.0;
    div(R)          Gauss linear;
    div(phi,nuTilda) bounded Gauss limitedLinear 1.0;
    div((nuEff*dev(T(grad(U))))) Gauss linear;
}
```

T. Lucchini: Incompressible, steady-state turbulent flow simulations in OpenFOAM®

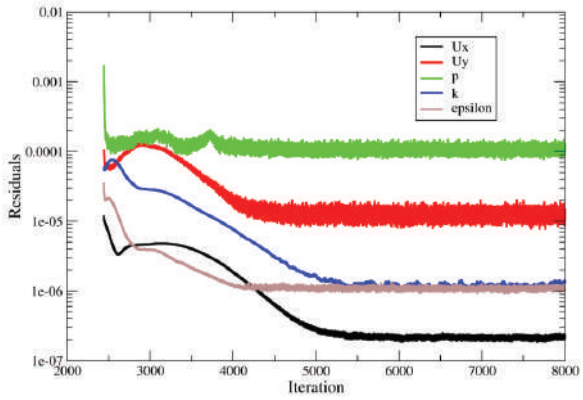
Diffuser simulation

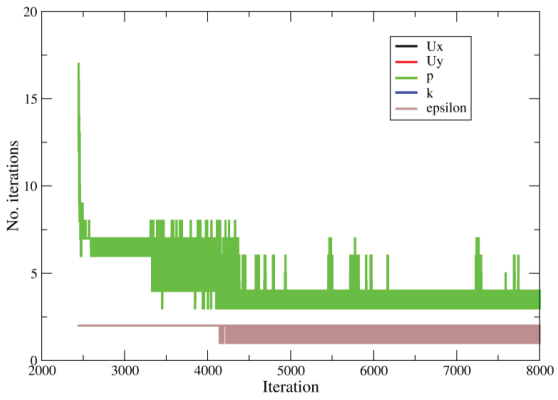
Run the `diffuserRefined` case with more accurate numerical schemes

- Run the case with `simpleFoam` (in parallel). There is no need to decompose the case again.

```
> cd diffuserRefine
> mpirun -np 4 simpleFoam -parallel > diffuserRefine.2ndOrd.log
```

- Convergence (to user tolerance) is not reached, but residuals an all the fields are stable. Run reconstructPar to visualize results at the latest time-step





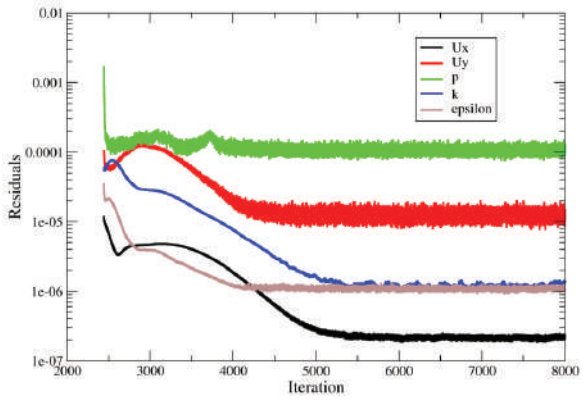
Diffuser simulation

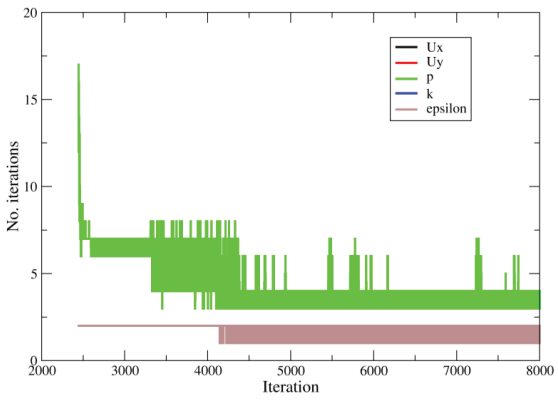
Run the `diffuserRefined` case with more accurate numerical schemes

- Run the case with `simpleFoam` (in parallel). There is no need to decompose the case again.

```
> cd diffuserRefine
> mpirun -np 4 simpleFoam -parallel > diffuserRefine.2ndOrd.log
```

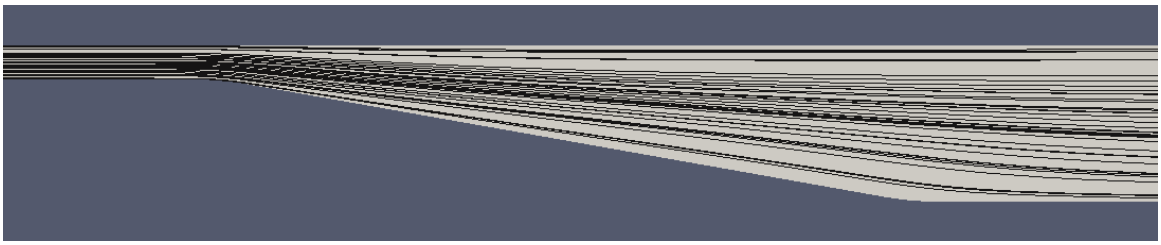
- Convergence (to user tolerance) is not reached, but residuals an all the fields are stable. Run reconstructPar to visualize results at the latest time-step





Diffuser simulation

Run the `diffuserRefined` case with more accurate numerical schemes



- Still, we do not see any flow recirculation….

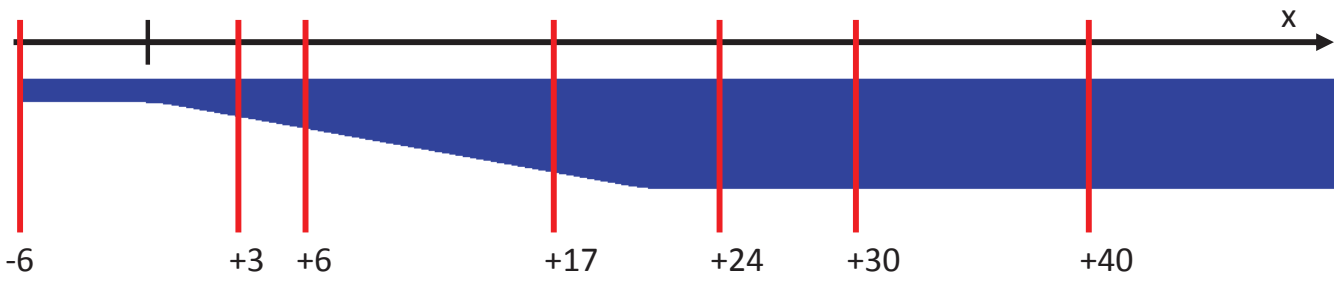
What about the flow recirculation? Which actions to consider now?

1) **Check the y+ values to see if they are in the proper range**
   - **If not rerun the simulation with a more refined mesh at the walls**
2) **Compare results with experimental data (if available)**
3) **See if results improve when using lower tolerances for residuals**
4) **Use high-order numerical schemes**
5) **Check if a refined mesh provides better results**
6) **Change the turbulence model**

Diffuser simulation

Comparison with experimental data: velocity profiles

- Axial velocity distribution available at different distances from diffuser inlet



Lots of data are available. We will focus on:

- **-6** : prediction of inlet velocity profile
- **+6** : flow at diffuser inlet
- **+17** : flow detachment
- **+30** : flow re-attachment and re-development of boundary layer
- **+40** : flow field downstream

Diffuser simulation

Comparison with experimental data: velocity profiles

- Run the `sample` utility on both the cases: `diffuserBase`, `diffuserRefine`
- The sampleDict dictionary includes lines along velocity will be sampled:

```
fields
(
    U
);

sets
(
    line-6
    {
        type        uniform;
        axis        distance;
        start       (104.13 -3.7 0);
        end         (104.13 1.0 0);
        nPoints     100;
    }
    line+6
    {
        type        uniform;
        axis        distance;
        start       (115.98 -3.7 0);
        end         (115.98 1.0 0);
        nPoints     100;
    }
```

```
    line+17
    {
        type        uniform;
        axis        distance;
        start       (126.93 -3.7 0);
        end         (126.93 1.0 0);
        nPoints     100;
    }
    line+30
    {
        type        uniform;
        axis        distance;
        start       (140.48 -3.7 0);
        end       (140.48 1.0 0);
        nPoints     100;
    }
    line+40
    {
        type        uniform;
        axis        distance;
        start       (149.85 -3.7 0);
        end         (149.85 1.0 0);
        nPoints     100;
    }
);
```


Diffuser simulation

Comparison with experimental data: velocity profiles

```
> sample -case diffuserBase -latestTime
> sample -case diffuserRefined -time '2442:8000'
```

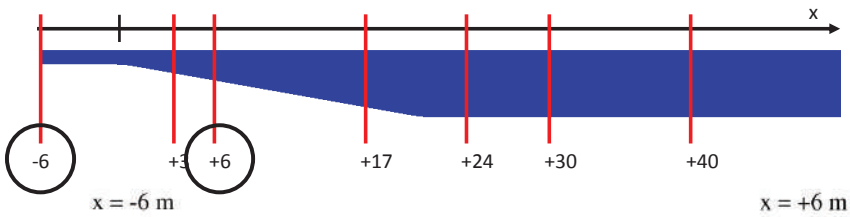
- Look now inside the `postProcessing/sets` directory of these cases:

```
> ls diffuserBase/postProcessing/sets
438
> ls diffuserBase/postProcessing/sets/438
line+17_U.xy  line+30_U.xy  line+40_U.xy  line-6_U.xy  line+6_U.xy
> ls diffuserRefined/postProcessing/sets
2442  8000
> ls diffuserBase/postProcessing/sets/2442
line+17_U.xy  line+30_U.xy  line+40_U.xy  line-6_U.xy  line+6_U.xy
> ls diffuserBase/postProcessing/sets/8000
line+17_U.xy  line+30_U.xy  line+40_U.xy  line-6_U.xy  line+6_U.xy
```

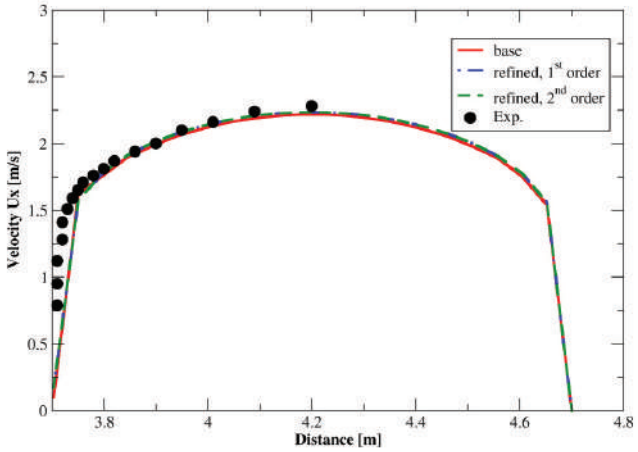
- Sub-directories with the names of the time-steps where post-processing was perfoemd appear including the sampled velocity profiles along the different lines.

Diffuser simulation

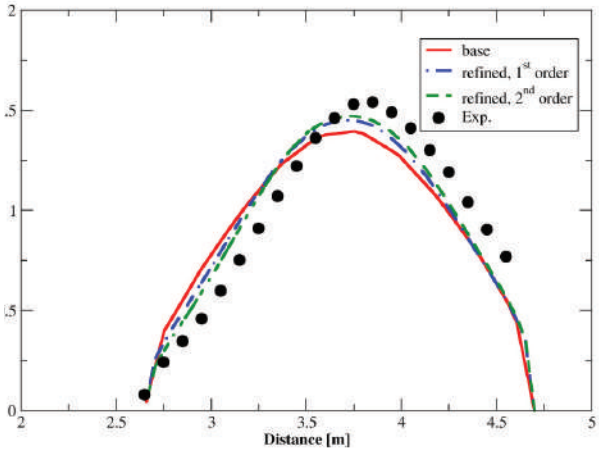
Comparison with experimental data: velocity profiles



x = -6 m

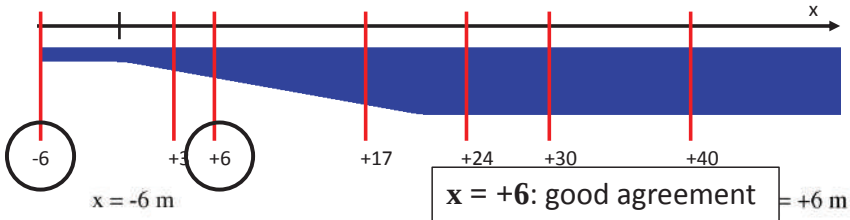


x = +6 m



Diffuser simulation

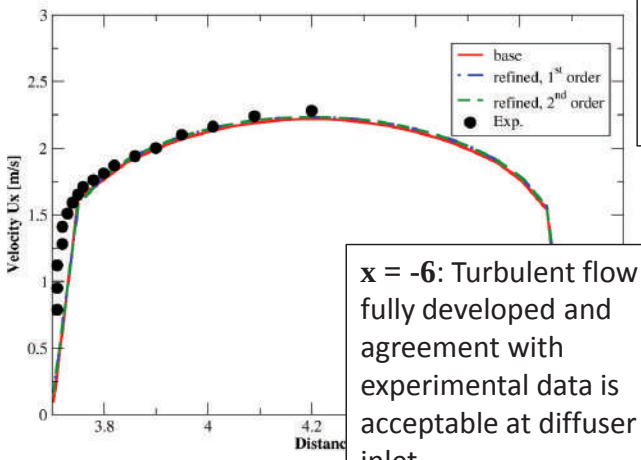
Comparison with experimental data: velocity profiles



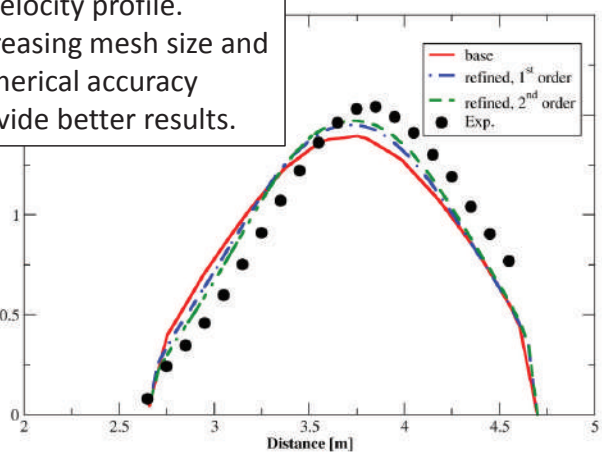
x = -6 m

x = +6: good agreement of velocity profile. Increasing mesh size and numerical accuracy provide better results.

= +6 m

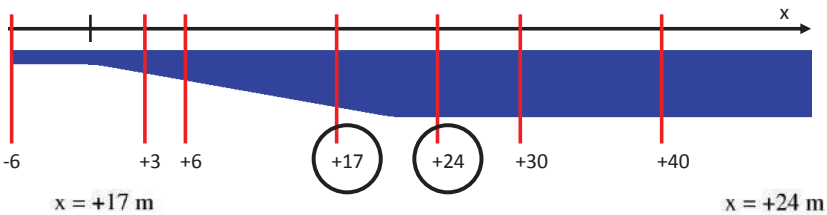


x = -6: Turbulent flow is fully developed and agreement with experimental data is acceptable at diffuser inlet

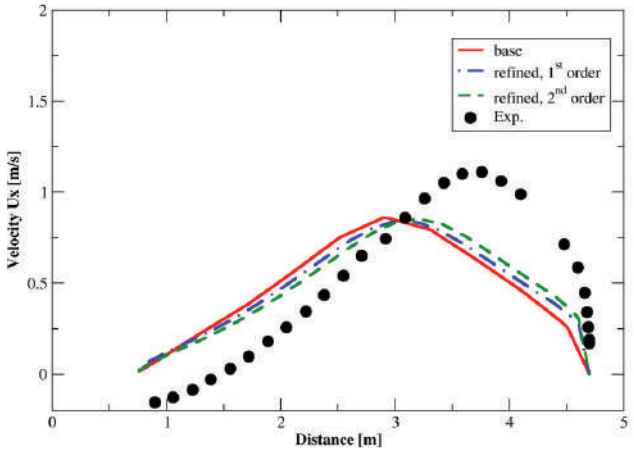


Diffuser simulation

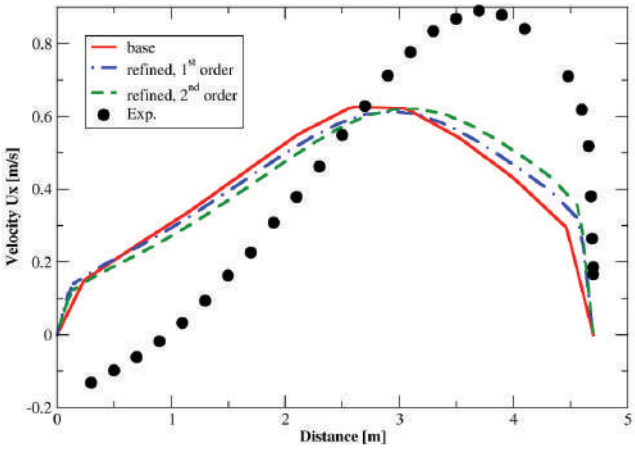
Comparison with experimental data: velocity profiles



x = +17 m

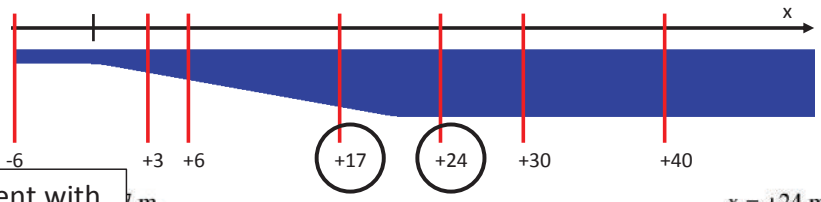


x = +24 m



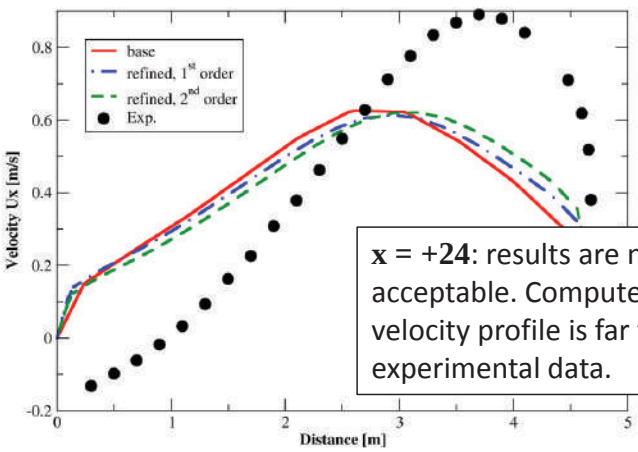
Diffuser simulation

Comparison with experimental data: velocity profiles



x = +17: agreement with experimental data is not acceptable downstream the diffuser: flow detachment (Ux < 0) is not predicted by any tested configuration.

x = +24 m

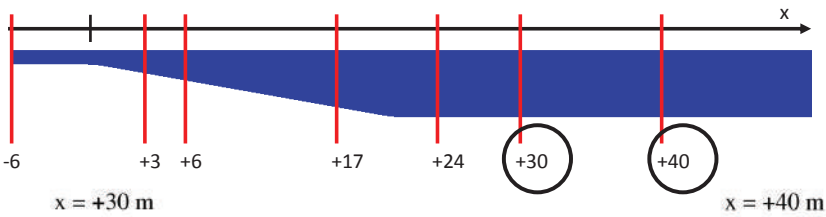


x = +24: results are not acceptable. Computed velocity profile is far from experimental data.

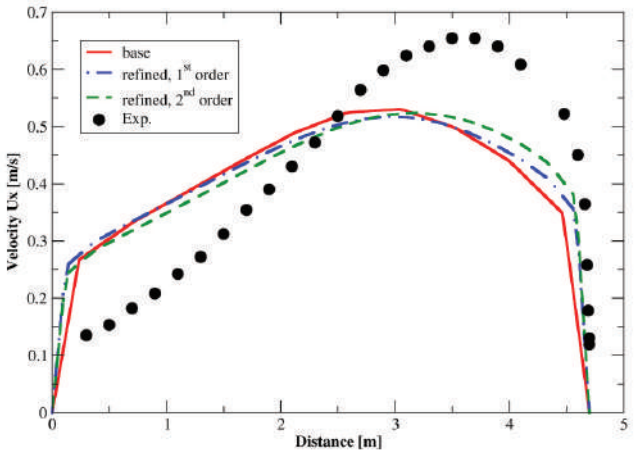
T. Lucchini: Incompressible, steady-state turbulent flow simulations in OpenFOAM®

Diffuser simulation

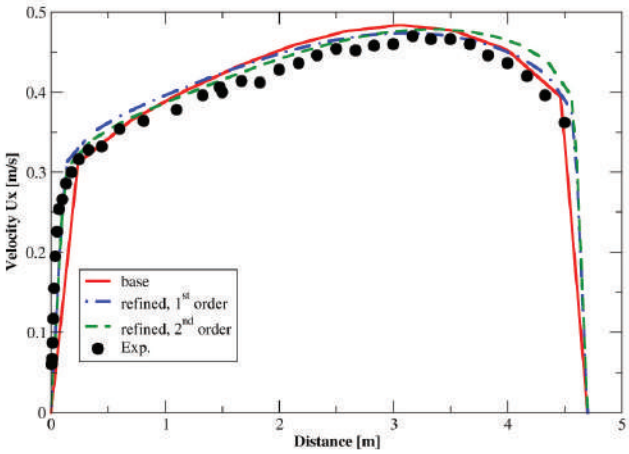
Comparison with experimental data: velocity profiles



x = +30 m

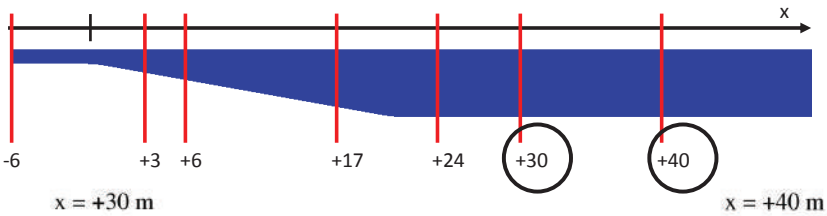


x = +40 m

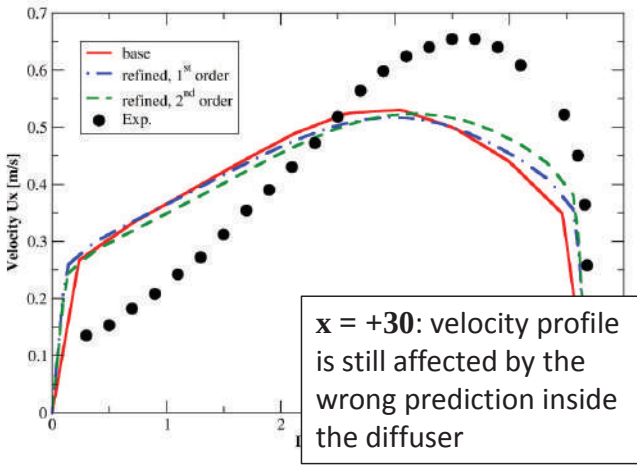


Diffuser simulation

Comparison with experimental data: velocity profiles

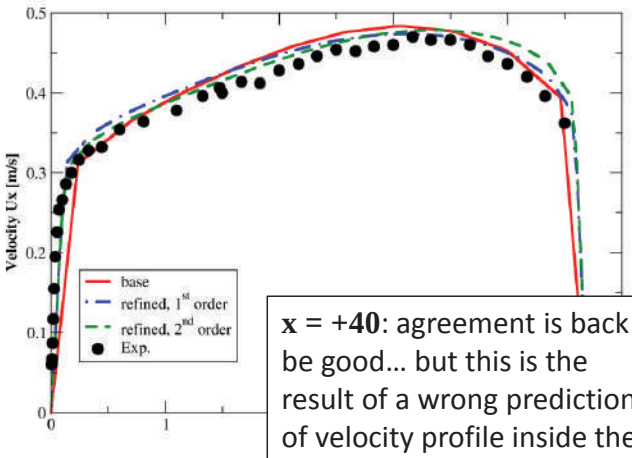


x = +30 m



x = +30: velocity profile is still affected by the wrong prediction inside the diffuser

x = +40 m



x = +40: agreement is back to be good… but this is the result of a wrong prediction of velocity profile inside the diffuser!

Diffuser simulation

Run the case with the kOmegaSST turbulence model

- From the detailed analysis carried out so far, it is worth to investigate how results change with a new turbulence model. We expect that kOmegaSST can provide better results (somebody told me…). What to do:

  1) Create a new case, called `diffuserKOmegaSST`, starting from `diffuserRefined`.

  2) In constant/RASProperties of the `diffuserKOmegaSST` case change the entry `RASModel` with `kOmegaSST` (it was `kEpsilon`).

```
RASModel        kOmegaSST;
turbulence      on;
printCoeffs     on;
```

  3) Initialize the `omega` field in the `0` directory

Diffuser simulation

Run the case with the kOmegaSST turbulence model

- **Initialization of the omega field**

  - Since $\omega = \frac{\varepsilon}{k}$, we will initialize omega with the ratio between `epsilon` and k used in the previous case.

  - At walls we will use suitable wall functions (`omegaWallFunction`)

  - At the inlet, omega will be initialized from turbulence intensity and mixing length. The boundary condition is called (`turbulentMixingLengthFrequencyInlet`)

  - No changes are needed for the k field.

Diffuser simulation

Turbulent specific dissipation rate omega

- Dimensions are inverse of time
- Initial internal value: ratio between ε and k of the previous case.
- At the `inlet` boundary, ω is estimated from k and a mixing length equal to 1% of the channel height.
- At `outlet`, the `zeroGradient` boundary condition is used (inletOutlet can sometimes increase the solution stability)

```
dimensions      [0 0 -1 0 0 0 0];
internalField   uniform 39.61;

boundaryField
{
    walls
    {
        type            omegaWallFunction;
        value           uniform 39.61;
        Cmu             0.09;
        kappa           0.41;
        E               9.8;
    }
    inlet
    {
        type        turbulentMixingLengthFrequencyInlet;
        mixingLength 0.01;
        value           uniform 39.61;
    }
    outlet
    {
        type            zeroGradient;
    }
    frontAndBackPlanes
    {
        type            empty;
    }
}
```


Diffuser simulation

Turbulent specific dissipation rate omega

- Wall functions should be used for all the patches which are identified as `wall` type in the mesh boundary.
- Available wall functions for omega:
  - ✓ `omegaWallFunction`

  which is valid for both high and low reynolds simulations

```
dimensions      [0 0 -1 0 0 0 0];
internalField   uniform 39.61;

boundaryField
{
    walls
    {
        type            omegaWallFunction;
        value           uniform 39.61;
        Cmu             0.09;
        kappa           0.41;
        E               9.8;
    }
    inlet
    {
        type        turbulentMixingLengthFrequencyInlet;
        mixingLength 0.01;
        value           uniform 39.61;
    }
    outlet
    {
        type            zeroGradient;
    }
    frontAndBackPlanes
    {
        type            empty;
    }
}
```

Diffuser simulation

Run the case with the kOmegaSST turbulence model

- **Running the case in two-steps:**

1) Get convergence with 1st order schemes:
   - Add `div(phi,omega)` in `fvSchmes` (sub-dictionary `divSchemes`) with a suitable entry, include omega field in `fvSolution (solvers, SIMPLE, relaxationFactors)`:

`fvSchemes` →

```
divSchemes
{
    default         none;
    div(phi,U)      bounded Gauss upwind;
    div(phi,k)      bounded Gauss upwind;
    div(phi,epsilon) bounded Gauss upwind;
    div(phi,R)      bounded Gauss upwind;
    div(R)          Gauss linear;
    div(phi,nuTilda) bounded Gauss upwind;
    div((nuEff*dev(T(grad(U))))) Gauss linear;
    div(phi,omega) bounded Gauss upwind;
}
```

`fvSolution` →

```
solvers
{
    // settings for p...
    "(U|k|epsilon|R|nuTilda|omega)"
    {
        solver          smoothSolver;
        smoother        symGaussSeidel;
        tolerance       1e-10;
        relTol          0.1;
    }
}
```

```
SIMPLE
{
    nNonOrthogonalCorrectors 0;

    residualControl
    {
        p               1e-5;
        U               1e-6;
        "(k|epsilon|omega)" 1e-6;
    }
}
```

```
relaxationFactors
{
    fields
    {
        p               0.3;
    }
    equations
    {
        U               0.7;
        k               0.7;
        epsilon         0.7;
        R               0.7;
        nuTilda         0.7;
        omega           0.7;
    }
}
```


Diffuser simulation

Run the case with the kOmegaSST turbulence model

- **Running the case in two-steps:**

2) Then restart the case with 2nd order schemes to improve accuracy of results (change fvSchemes before restart as follows):

```
divSchemes
{
    default         none;
    div(phi,U)      bounded Gauss limitedLinear 1.0;
    div(phi,k)      bounded Gauss limitedLinear 1.0;
    div(phi,epsilon) bounded Gauss limitedLinear 1.0;
    div(phi,R)      bounded Gauss limitedLinear 1.0;
    div(R)          Gauss linear;
    div(phi,nuTilda) bounded Gauss limitedLinear 1.0;
    div((nuEff*dev(T(grad(U))))) Gauss linear;
    div(phi,omega) bounded Gauss limitedLinear 1.0 upwind;
}
```

3) Check the residuals, reconstruct the case (`reconstructPar`) and do sampling (`sample`)

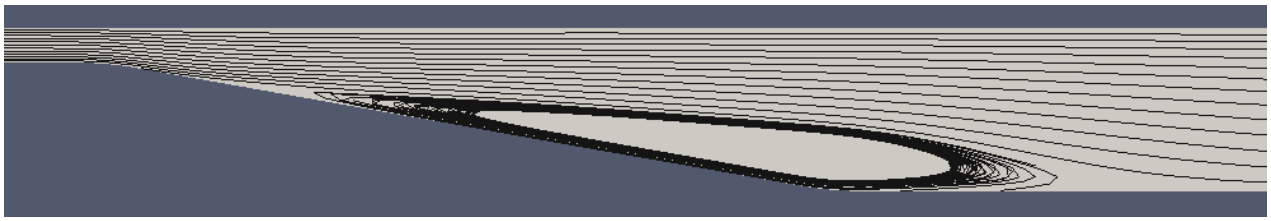
4) See if y+ is still in the proper range (use the `yPlusRas` utility)

```
Patch 0 named walls y+ : min: 16.9854 max: 65.4602 average: 31.0476
```

Diffuser simulation

Run the case with the kOmegaSST turbulence model

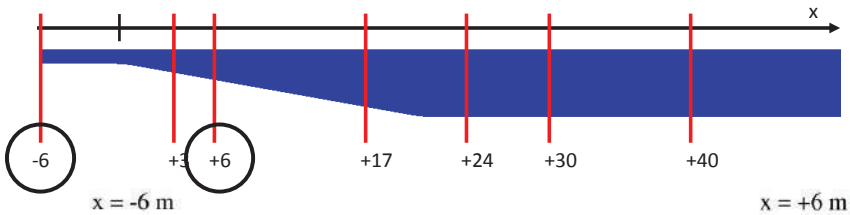
- **Post-processing with paraFoam: streamlines**



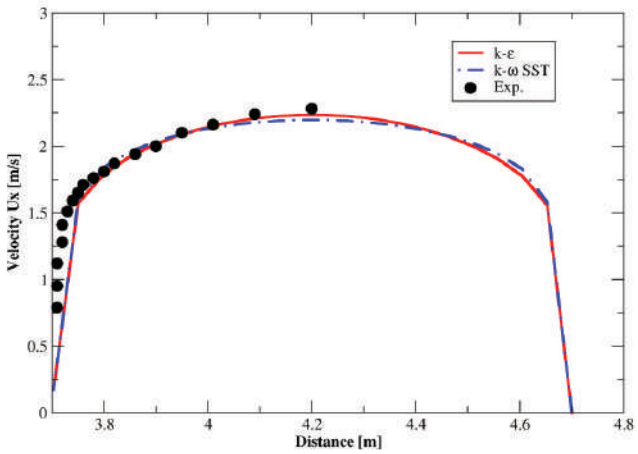
- Flow detachment and recirculation inside the diffuser are now properly predicted.
- Changing the turbulence model qualitatetively improved the computed results.
- Let's compare again computed and experimental data…
  - Standard k-ε (last iteration of the `diffuserRefined` case)
  - k-ω SST (last iteration of the `diffuserKOmegaSST` case)
  - ⇨ **Since same mesh, tolerances and numerical methods were used, the comparison is consistent.**

Diffuser simulation

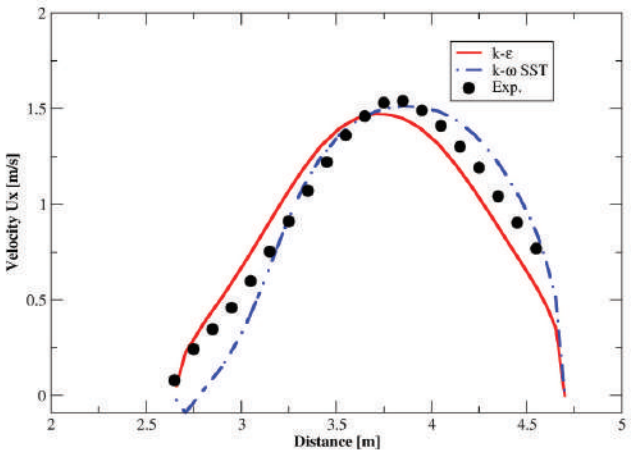
Comparison with experimental data: velocity profiles



x = -6 m



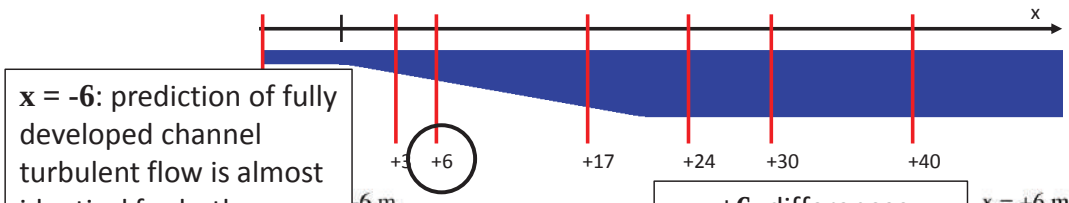
x = +6 m



T. Lucchini: Incompressible, steady-state turbulent flow simulations in OpenFOAM®

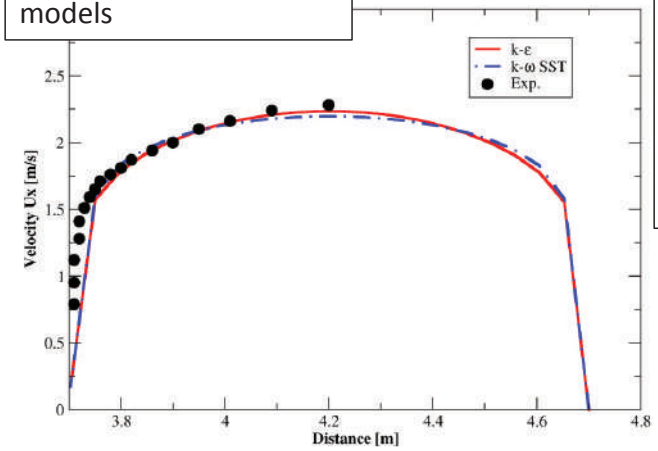
Diffuser simulation

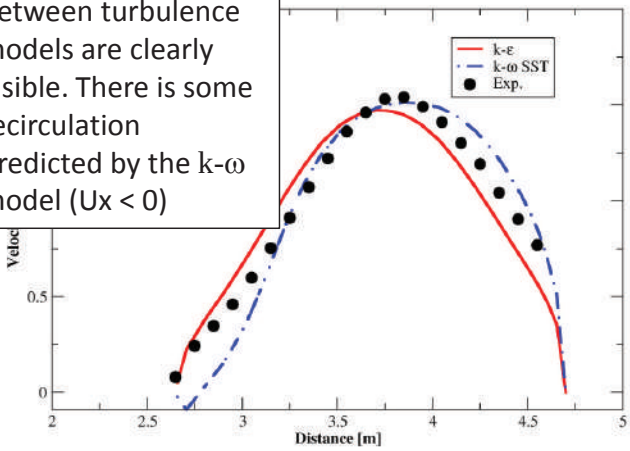
Comparison with experimental data: velocity profiles



x = -6: prediction of fully developed channel turbulent flow is almost identical for both models

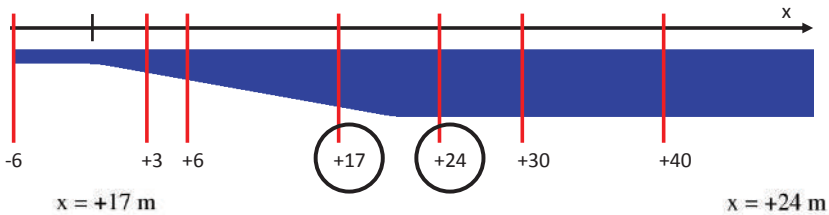
x = +6: differences between turbulence models are clearly visible. There is some recirculation predicted by the k-ω model (Ux < 0)

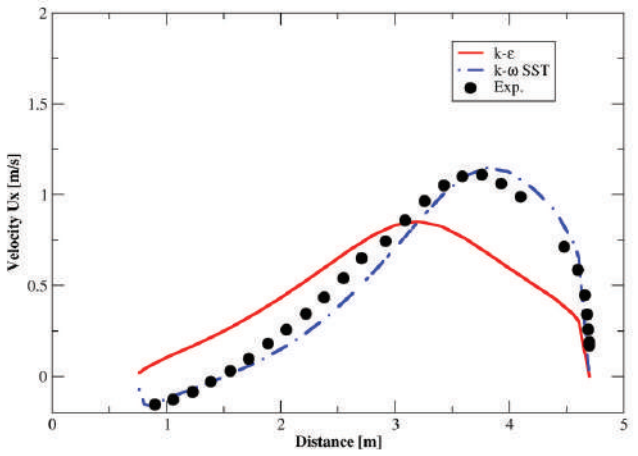


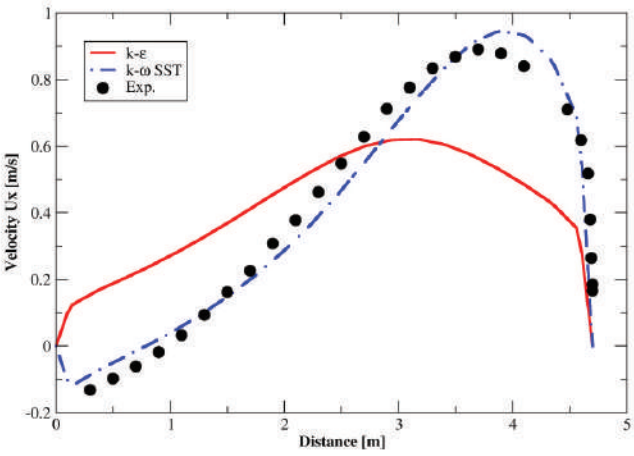


Diffuser simulation

Comparison with experimental data: velocity profiles

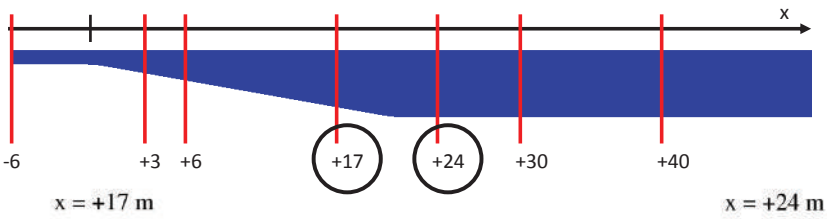






Diffuser simulation

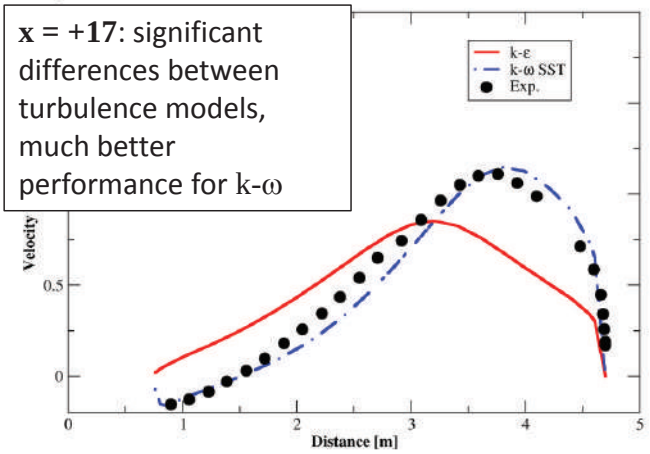
Comparison with experimental data: velocity profiles



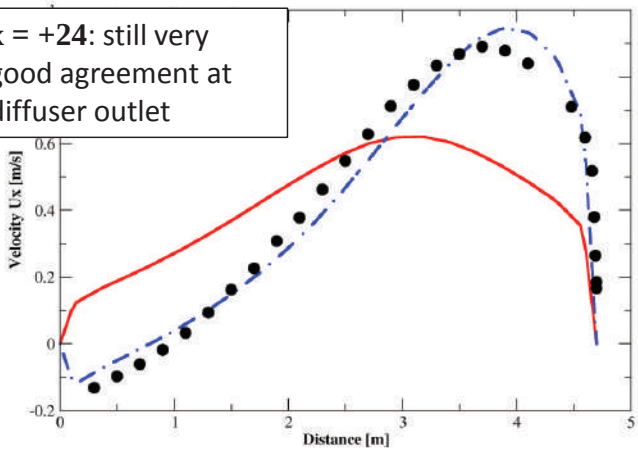
x = +17 m

x = +24 m

x = +17: significant differences between turbulence models, much better performance for k-ω

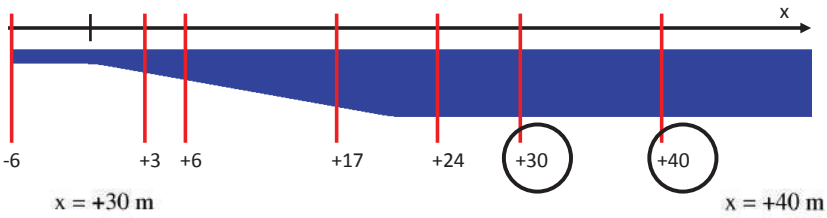


x = +24: still very good agreement at diffuser outlet



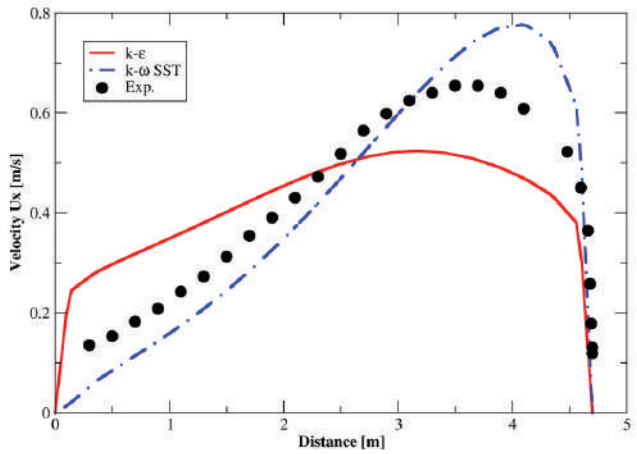
Diffuser simulation

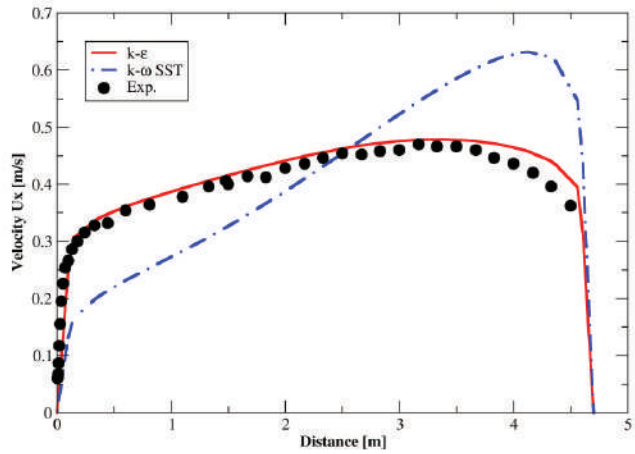
Comparison with experimental data: velocity profiles



x = +30 m

x = +40 m



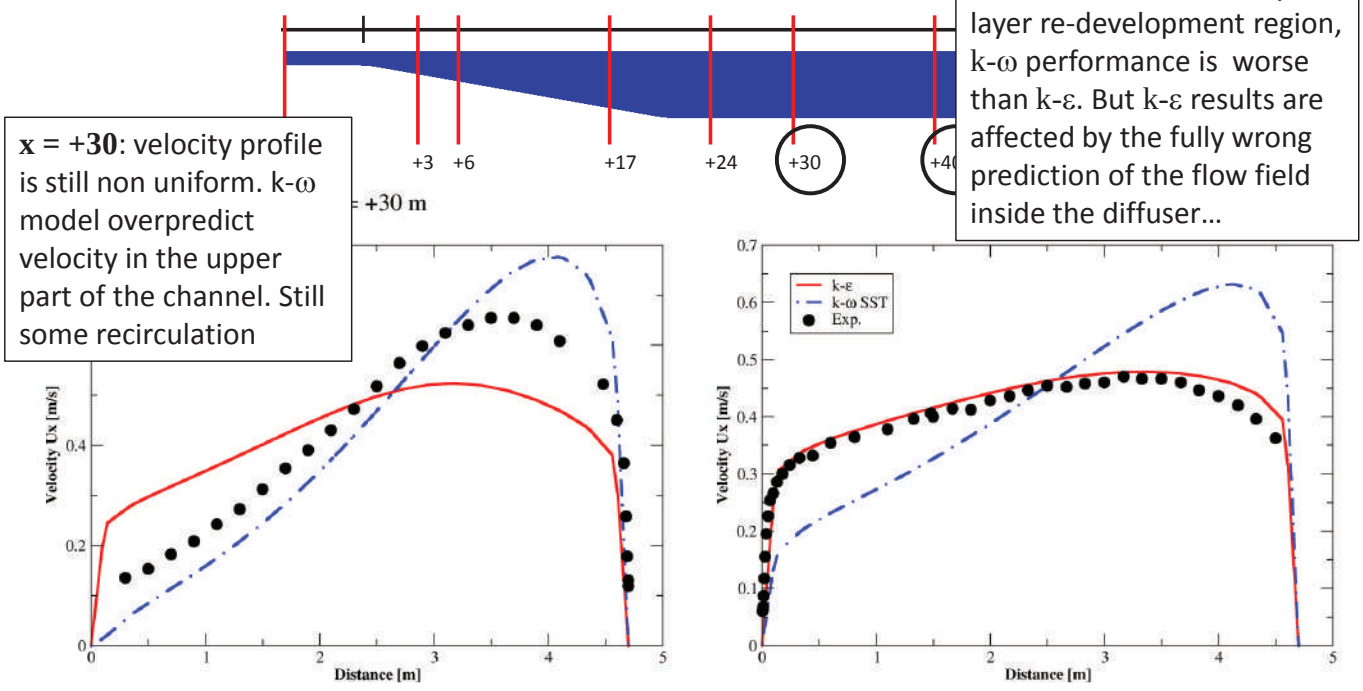


Diffuser simulation

Comparison with experimental data: velocity profiles

x = +30: velocity profile is still non uniform. k-ω model overpredict velocity in the upper part of the channel. Still some recirculation

x = +40: in the boundary layer re-development region, k-ω performance is worse than k-ε. But k-ε results are affected by the fully wrong prediction of the flow field inside the diffuser…



Diffuser simulation

Recommendation for flow simulations in OpenFOAM

- Reduce residual tolerances
- Increase the accuracy of numerical schemes when possible
- Do grid dependency studies to find the best mesh

Turbulent flows

- Do literature investigations about influence of turbulence models on flows.
- See here, for further experimental data available to investigate turbulence models and understand which model (and mesh size) fits best for your problems.

Ercoftac diffuser case: can we improve results?

- Results are good so far in the channel and diffuser flow. What about boundary layer re-development?
  - ⇨ We need to investigate more…

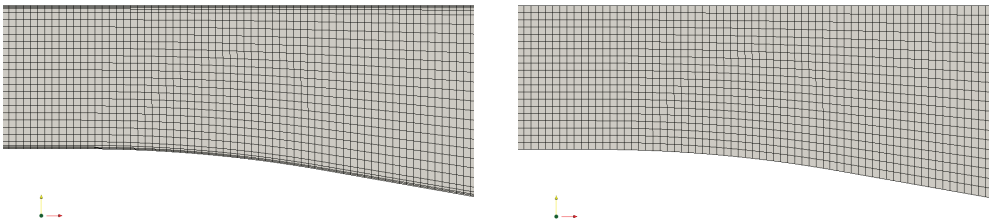
Diffuser simulation

Improving (?) results of the Ercoftac diffuser case:

- Use a more refined boundary layer:
  - Create a new case, called `diffuserKOmegaSSTBL` starting from the `diffuserKOmegaSST` one.
  - Use the `refineWallLayer` utility to refine cells next to patches in the normal direction. It requires a patch name and a weight factor. Run it three times with the `-overwrite` option. It will overwrite the `polyMesh` with a more refined one.

```
> refineWallLayer walls 0.5 -case diffuserKOmegaSSTBL  -overwrite
> refineWallLayer walls 0.5 -case diffuserKOmegaSSTBL  -overwrite
> refineWallLayer walls 0.5 -case diffuserKOmegaSSTBL  -overwrite
```

`diffuserKOmegaSSTBL` `diffuserKOmega`



Diffuser simulation

Improving (?) results of the Ercoftac diffuser case:

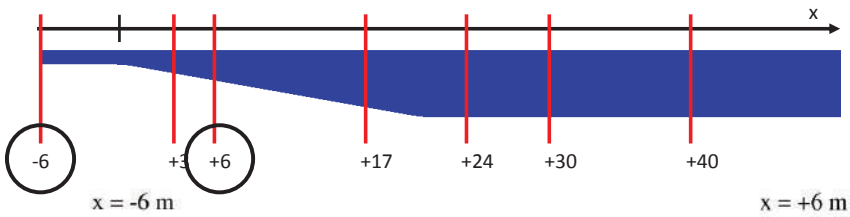
- The `refineWallLayer` utility works only on grids where the boundary layer is prismatic or hexahedral. Do not apply it to other grids, it will not work.
- **New created cells do not belong to any specific set or zone of the mesh**. In case old boundary cells were part of a set or a zone, write a `topoSetDict` that, after the `refineWallLayer` utility puts the new refined cells in the same set (then use `setToZones` utility if necessary)
- Run the new case in the usual manner. Check the y+ value at the end of the simulation:

```
Patch 0 named walls y+ : min: 0.330375 max: 4.85921 average: 1.58391
```

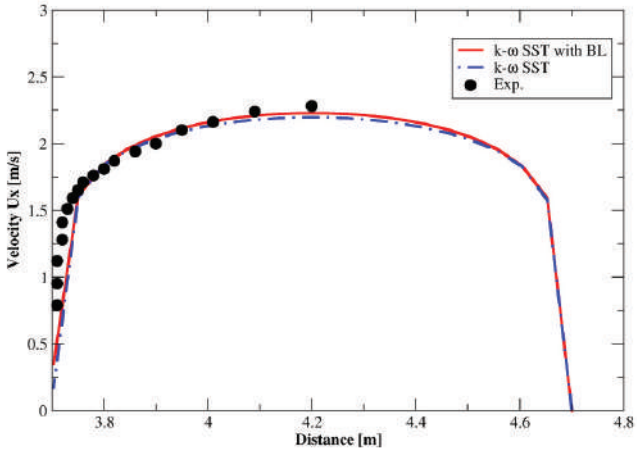
- y+ is much lower than before. Let's see the results…

Diffuser simulation

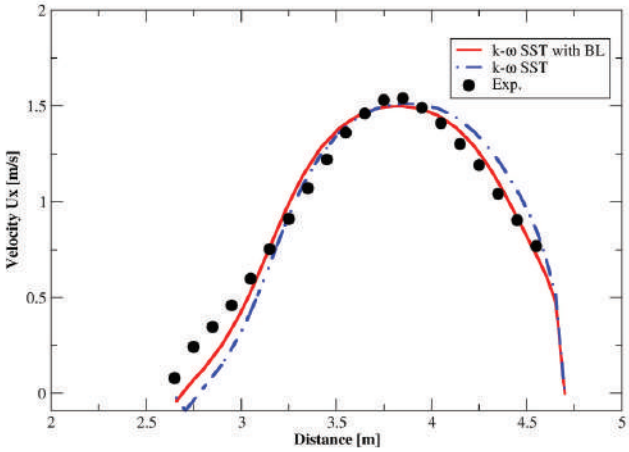
Comparison with experimental data: velocity profiles



x = -6 m

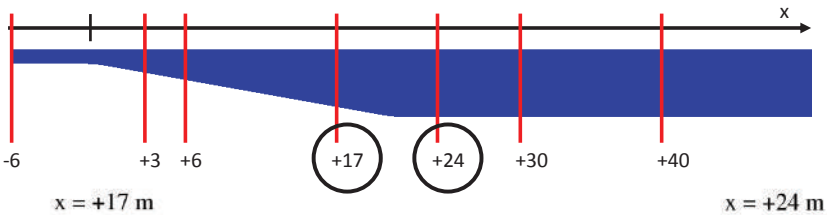


x = +6 m

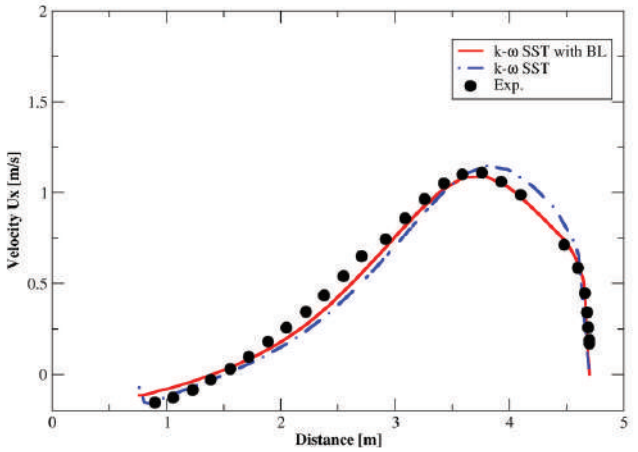


Diffuser simulation

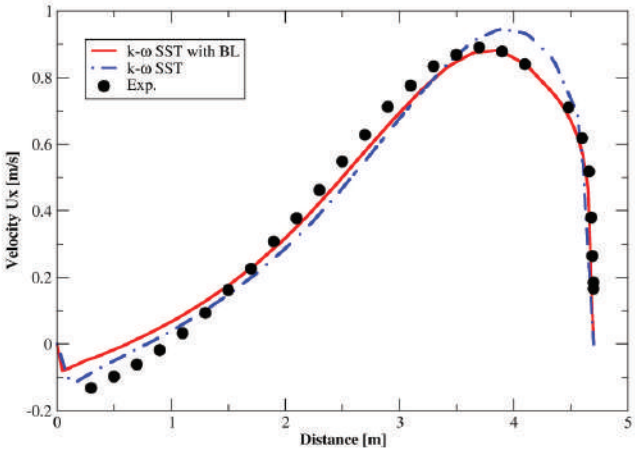
Comparison with experimental data: velocity profiles



x = +17 m

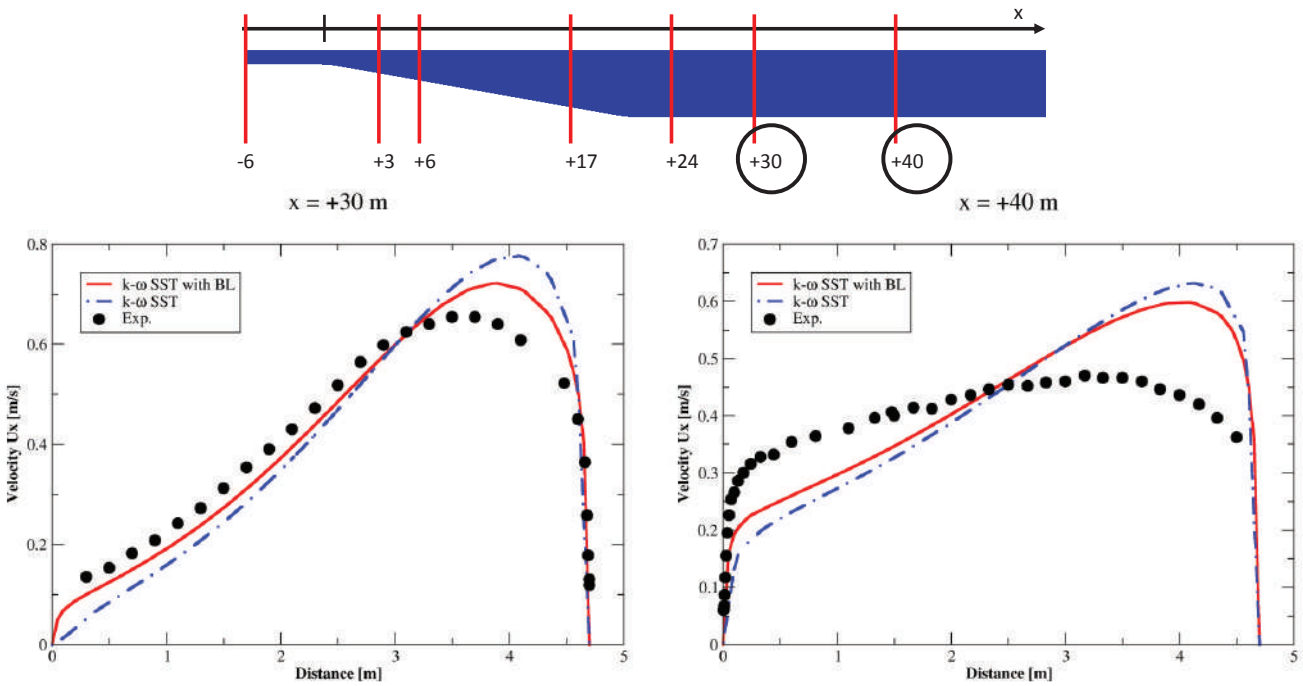


x = +24 m



Diffuser simulation

Comparison with experimental data: velocity profiles



Diffuser simulation

Improving (?) results of the Ercoftac diffuser case:

- Refinement of the wall provided a better agreement of velocity profiles. However, boundary layer re-development still requires further actions.
- **Now what to do now?**

1) Make your conclusions: find performance indexes (pressure drop, pressure coefficient, …) to understand if results are acceptable or not

2) Make further tests:

   - Low-Reynolds turbulence models
   - Other turbulence models which are neither k-ε or k-ω.

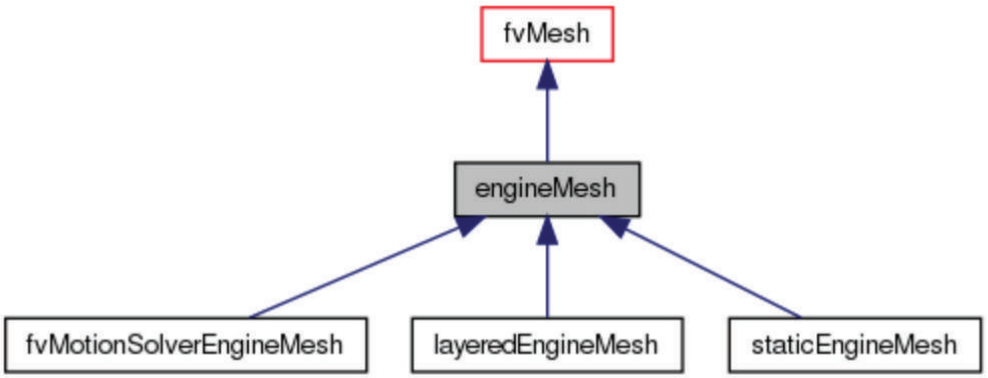
   …However boundary layer re-development is still an open-issue in turbulence models!

Customise OpenFOAM to model an engine with moving valves

Gianluca Montenegro

Example of dynamicFvMesh

- The target of this tutorial is to learn how to set up a moving mesh of cylinder with moving piston and moving valve.
- The mesh motion strategy will adopt the velocity laplacian motion solver without making use of topology changes. This will lead to a low quality mesh. A new mesh with suitable quality will be required to extend the validity of the calculation
- The developements will be based onto the enginMesh class of the OpenFOAM release.



engineMesh

```
class engineMesh
:
    public fvMesh
{
    // Private Member Functions

        //- Disallow default bitwise copy construct
        engineMesh(const engineMesh&);

        //- Disallow default bitwise assignment
        void operator=(const engineMesh&);

protected:

      const engineTime& engineDB_;

        label pistonIndex_;
        label linerIndex_;
        label cylinderHeadIndex_;

        dimensionedScalar deckHeight_;
        dimensionedScalar pistonPosition_;

        //- Select null constructed
        static autoPtr<engineMesh> New(const IOobject& io);

    //- Destructor
    virtual ~engineMesh();

   virtual void move() = 0;
```

Datalbase for conversion from time to deg and viceversa. It stores also geometrical parameters of the engine

Indexing for recognition of moving boundaries

Definition of the move function which will be specialized .

engineMesh

```
Foam::autoPtr<Foam::engineMesh> Foam::engineMesh::New
(
    const Foam::IOobject& io
)
{
    // get model name, but do not register the dictionary
    // otherwise it is registered in the database twice
    const word modelType
    (
        IOdictionary
        (
            IOobject
            (
                "engineGeometry",
                io.time().constant(),
                io.db(),
                IOobject::MUST_READ_IF_MODIFIED,
                IOobject::NO_WRITE,
                false
            )
        ).lookup("engineMesh")
    );

    Info<< "Selecting engineMesh " << modelType << endl;

    IOobjectConstructorTable::iterator cstrIter =
        IOobjectConstructorTablePtr_->find(modelType);

    return autoPtr<engineMesh>(cstrIter()(io));
}
```

engineGeometry is the dictionary that is accessed for the definition of geometry and mesh motion

engineGeometry

```
conRodLength    conRodLength [0 1 0 0 0 0 0] 0.18;

bore            bore [0 1 0 0 0 0 0] 0.1;

stroke          stroke [0 1 0 0 0 0 0] 0.09;

clearance       clearance [0 1 0 0 0 0 0] 0.000118;

rpm             rpm [0 0 -1 0 0 0 0] 1500;
```

```
engineMesh  fvMotionSolverValve;

velocityLaplacianCoeffs
{
    diffusivity uniform;
}
```

Geometrical parameters are readas imputs for the definition of the piston position

Selection of the motion algorithm used:
- motionSolver
- VelocityLaplacian
- Uniform diffusivity

fvMotionSolverEngineMesh

```
class fvMotionSolverEngineMesh
:
    public engineMesh
{
    // Private data

        dimensionedScalar pistonLayers_;

        //- Mesh-motion solver to solve for the "z" component of
the cell-centre
        //  displacements
        velocityComponentLaplacianFvMotionSolver motionSolver_;

    // Private Member Functions

        //- Disallow default bitwise copy construct
        fvMotionSolverEngineMesh(const fvMotionSolverEngineMesh&);

        //- Disallow default bitwise assignment
        void operator=(const fvMotionSolverEngineMesh&);

public:

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

    void move();
```

The velocityLaplacian motion solver is declared. No run time selection

The move() function will take care of the motion of specific boundaries such as piston and valves

fvMotionSolverEngineMesh

```
void Foam::fvMotionSolverEngineMesh::move()
{
    scalar deltaZ = engineDB_.pistonDisplacement().value();
    Info<< "deltaZ = " << deltaZ << endl;

    scalar pistonPlusLayers = pistonPosition_.value() +
pistonLayers_.value();

    scalar pistonSpeed = deltaZ/engineDB_.deltaTValue();

    motionSolver_.pointMotionU().boundaryField()[pistonIndex_] ==
pistonSpeed;

    {
        scalarField linerPoints
        (
            boundary()
[linerIndex_].patch().localPoints().component(vector::Z)
        );

        motionSolver_.pointMotionU().boundaryField()[linerIndex_] ==
            pistonSpeed*pos(deckHeight_.value() - linerPoints)
           *(deckHeight_.value() - linerPoints)
           /(deckHeight_.value() - pistonPlusLayers);
    }

    motionSolver_.solve();

    pistonPosition_.value() += deltaZ;
```

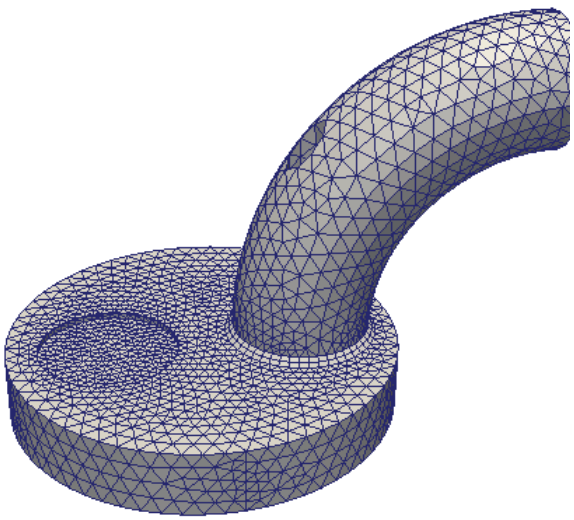
The piston patch is selected and its velocity assigned by the velocity calculated by the engineTime class

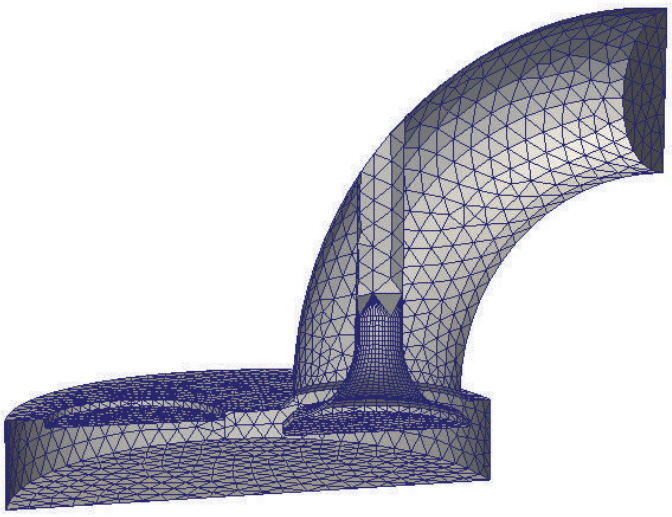
Liner points are assigned interpolating the piston motion along the piston axis

Liner points are assigned interpolating the piston motion along the piston axis

New fvMotionSolverEngineMesh class

- Adding the functionality of moving valves needs a little bit of coding.
- enginePiston and engineValve objects will be added to the mesh and some base class code will be modified to fix some bugs.





fvMotionSolverEngineMeshValve

```
class fvMotionSolverEngineMeshValve
:
    public engineMesh
{
    // Private data

        dimensionedScalar pistonLayers_;

        //- Mesh-motion solver to solve for the cell-centre
displacements
        velocityLaplacianFvMotionSolver motionSolver_;

        enginePiston piston_;

        valveBank   valves_;

    // Private Member Functions

        //- Disallow default bitwise copy construct
        fvMotionSolverEngineMeshValve(const
fvMotionSolverEngineMeshValve&);

        //- Disallow default bitwise assignment
        void operator=(const fvMotionSolverEngineMeshValve&);
```

enginePiston and valveBank are added to handle piston and valve motion.

NB: valveBank is a pointerl list to engineValve types

fvMotionSolverEngineMeshValve

```
Foam::fvMotionSolverEngineMeshValve::fvMotionSolverEngineMeshValve(con
st IOobject& io)
:
    engineMesh(io),
    pistonLayers_("pistonLayers", dimLength, 0.0),
    motionSolver_
    (
        *this,
        IOdictionary
        (
            IOobject
            (
                "dynamicMeshDict",
                time().constant(),
                *this,
                IOobject::NO_READ,
                IOobject::NO_WRITE
            ),
            engineDB_.engineDict()
        )
    ),
    piston_
    (
        *this, engineDB_.engineDict().subDict("piston")
    ),
    valves_
    (
        *this, engineDB_.engineDict().lookup("valves")
    )
```

enginePiston and valveBank needs to be added to the constructor of the class

fvMotionSolverEngineMeshValve

```
 vector pistonAxis = piston_.cs().R().e3();

    vector pistonSpeed = deltaZ/engineDB_.deltaTValue() * pistonAxis;
    motionSolver_.pointMotionU().boundaryField()[pistonIndex_] ==
pistonSpeed;
    {
        scalarField linerPoints
        (
            boundary()
[linerIndex_].patch().localPoints().component(vector::Z)
        );

        motionSolver_.pointMotionU().boundaryField()[linerIndex_] ==
            pistonSpeed*pos(deckHeight_.value() - linerPoints)
           *(deckHeight_.value() - linerPoints)
           /(deckHeight_.value() - pistonPlusLayers);
    }
    forAll(valves_, valveI)
    {
        scalar velocity = valves_[valveI].curVelocity();

label bottomPatchID = valves_[valveI].bottomPatchID().index();
        motionSolver_.pointMotionU().boundaryField()[bottomPatchID]==
velocity * valves_[valveI].cs().R().e3();

        label poppetPatchID = valves_[valveI].poppetPatchID().index();
        motionSolver_.pointMotionU().boundaryField()[poppetPatchID]==
velocity * valves_[valveI].cs().R().e3();

    }
```

Loop over the valves to assign a motion different from the piston one.

Dictionary for valve

```
valves
(
    intakeValve1
    {
        coordinateSystem
        {
            type            cartesian;
            origin          (0 0 0);
            coordinateRotation
            {
                type        localAxesRotation;
                e3          (0 0 1);
            }
        }
// Patch and zone names
        bottomPatch             intakeBottom;
        poppetPatch             intakeTop;
        stemPatch               intakeStem;
        sidePatch               no;

        detachInCylinderPatch  valveDetachInCylIn;
        detachInPortPatch      valveDetachInPortIn;

        diameter            0.04;

        liftProfile
        (
            0 0
            330 1.19777E-15
                :
            580 0
            720 0
        );
    }
```

Coordinate systems relative to the valve specifies the direction of motion (e3)

Definition of patches to be treated as valve boundaries

Lift profile is given as a graph with x and y values (angle [deg] and lift [m])

Dictionary for piston

```
piston
{
    patch               piston;

    coordinateSystem
    {
        type            cartesian;
        origin          (0 0 0);
        coordinateRotation
        {
            type        localAxesRotation;
            e3          (0 0 1);
        }
    }

    minLayer            0.001;
    maxLayer            0.0025;
}
```

Specifies the patch representing the top of the piston

Coordinate systems relative to the piston specifies the direction of motion (e3)

Code fixes for enginePiston

```
// Construct from dictionary
Foam::enginePiston::enginePiston
(
    const polyMesh& mesh,
    const dictionary& dict
)
:
    mesh_(mesh),
    engineDB_(refCast<const engineTime>(mesh_.time())),
    patchID_(dict.lookup("patch"), mesh.boundaryMesh()),
    csPtr_
    (
        coordinateSystem::New
        (
            mesh_,
            dict
        )
    ),
    minLayer_(readScalar(dict.lookup("minLayer"))),
    maxLayer_(readScalar(dict.lookup("maxLayer")))
{}
```

Fixes the reading of the dictionary for the coordinate system relative to the piston

Code fixes for engineValve

```
Foam::engineValve::engineValve
(
    const word& name,
    const polyMesh& mesh,
    const dictionary& dict
)
:
    name_(name),
    mesh_(mesh),
    engineDB_(refCast<const engineTime>(mesh_.time())),
    csPtr_
    (
        coordinateSystem::New
        (
            mesh_,
            dict
        )
    ),
    bottomPatch_(dict.lookup("bottomPatch"),
mesh.boundaryMesh()),
    poppetPatch_(dict.lookup("poppetPatch"),
mesh.boundaryMesh()),
    stemPatch_(dict.lookup("stemPatch"),
mesh.boundaryMesh()),
    curtainInPortPatch_
    (
        dict.lookup("curtainInPortPatch"),
        mesh.boundaryMesh()
    ),
```

Fixes the reading of the dictionary for the coordinate system relative to the valve

NB: While for piston the definition of the coordinate system may be redundant, for valves it is useful for cantered valve geometries

Exercise

Workshop for OpenFOAM and its Application in Industrial Combustion Devices

26-27 Feb. 2015
POSCO International Center, Pohang, Korea

Kang Y. Huh
Pohang University of Science and Technology

I simpleEdmFoam

- Case Description
- EDM (Eddy Dissipation Model)
- Code Structure
- Tutorial

II SLFMFoam

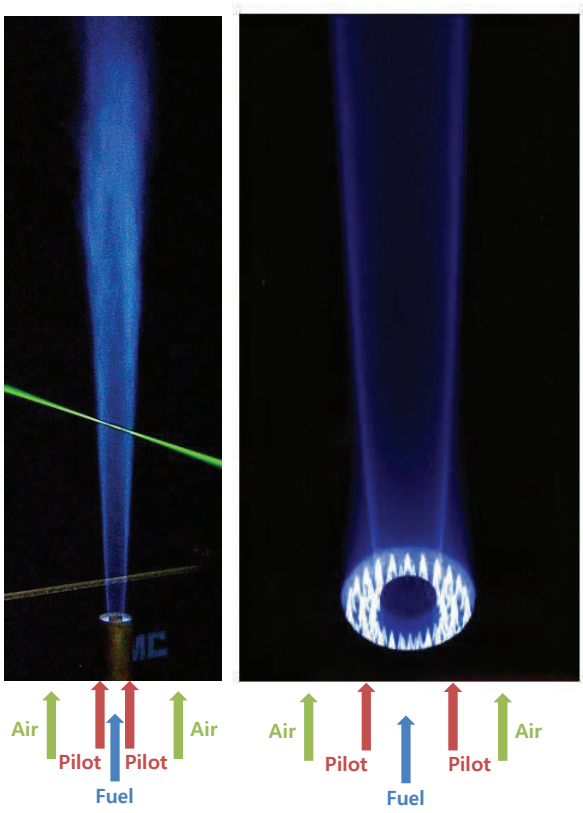
- SLFM (Stationary Laminar Flamelet Model)
- Code Structure
- Tutorial

III EdmParcelFoam

- Case Description
- Code Structure
- Tutorial

Case description

Piloted CH4/Air Flame



- **Burner Dimensions**
  - Main jet inner diameter, d = 7.2 mm
  - Pilot annulus inner diameter = 7.7 mm
  - Pilot annulus outer diameter = 18.2 mm
  - Burner outer wall diameter = 18.9 mm
  - Wind tunnel exit = 30 cm by 30 cm

- **Boundary Conditions**
  - Co-flow velocity = 0.9 m/s (291 K, 0.993 atm)
  - Main jet composition = 25% CH4, 75% dry air by volume
  - Main jet kinematic viscosity = 1.58e-05 m^2/s
  - Main jet velocity = 49.6 m/s (294K, 0.993 atm)

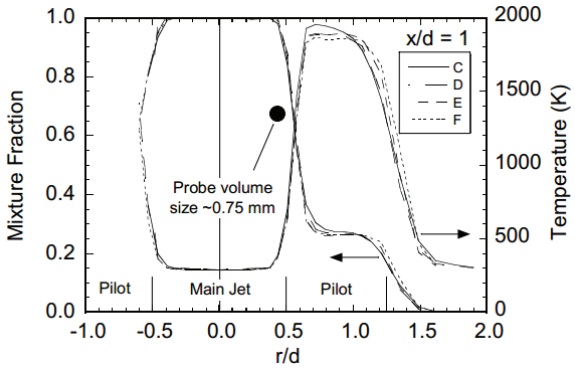
R. S. Barlow and J. H. Frank, Proc. Combust. Inst. 27:1087-1095 (1998)
R. S. Barlow, J. H. Frank, A. N. Karpetis and J. Y. Chen, Combust. Flame 143:433-449 (2005)
Ch. Schneider, A. Dreizler, J. Janicka, Combust. Flame 135:185-190 (2003)

Case description

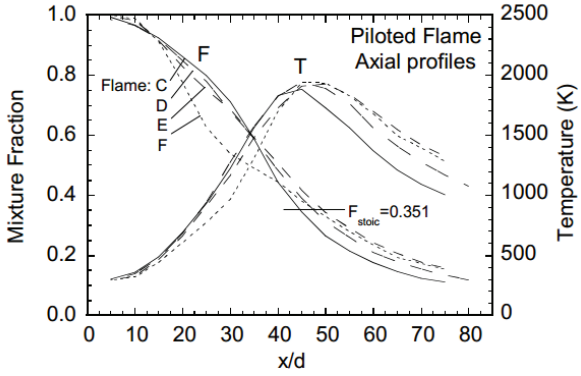
Piloted CH4/Air Flame

- **Composition**
  - Main jet (mass fraction) – N2 : 0.647, CH4 : 0.156, O2 : 0.197
  - Pilot (mass fraction) – N2 : 0.7342, O2 : 0.0540, O : 7.47e-4, H2 : 1.29e-4, H : 2.48e-5
    H2O : 0.0942, CO : 4.07e-3, CO2 : 0.1098, OH : 0.0028, NO : 4.80e-6
  - Co-flow (mass fraction) – N2 : 0.767, O2 : 0.233

- **Measured profiles**



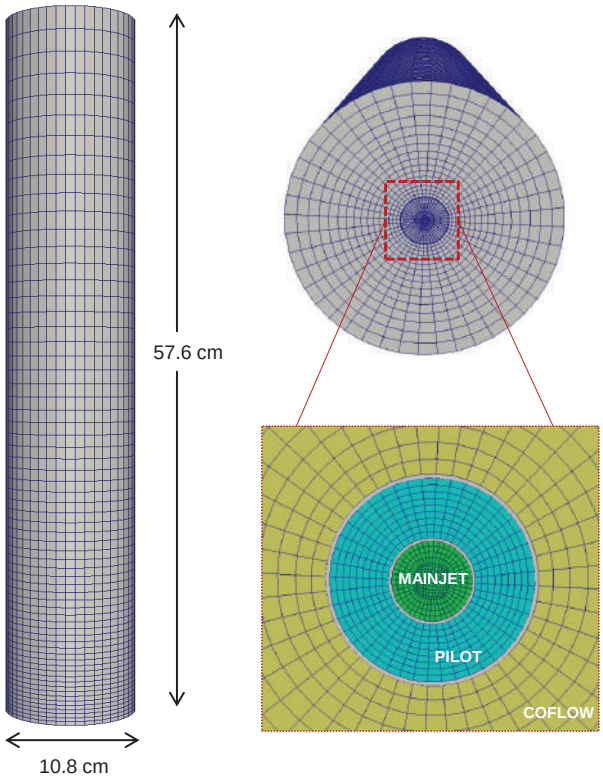
Measured radial profiles of Favre-average mixture fraction and temperature at x/d=1 in the four turbulent piloted flames



Axial profiles of measured mixture fraction and temperature (Favre average) in piloted flames C, D, E, and F

Case description

Computational grid



- **CheckMesh**

```
Mesh stats
    points:           105336
    faces:            309738
    internal faces:   303732
    cells:            102245
    faces per cell:   6
    boundary patches: 7
    point zones:      0
    face zones:       1
    cell zones:       1

Overall number of cells of each type:
    hexahedra:     102245
    prisms:        0
    wedges:        0
    pyramids:      0
    tet wedges:    0
    tetrahedra:    0
    polyhedra:     0

Checking patch topology for multiply connected surfaces...
    Patch              Faces     Points    Surface topology
    MAINJET            297       320       ok
    PILOT              396       440       ok
    COFLOW             704       748       ok
    OUTLET             1573      1596      ok
    CASING             2860      2904      ok
    INNERWALL          88        132       ok
    OUTERWALL          88        132       ok
```


simpleEdmFoam

Governing Equations

$$\frac{\partial \bar{\rho}}{\partial t} + \frac{\partial}{\partial x_k}\bar{\rho}\tilde{v}_k = 0$$

$$\frac{\partial}{\partial t}\bar{\rho}\tilde{v}_i + \frac{\partial}{\partial x_k}\bar{\rho}\boxed{v_k v_i} = -\frac{\partial \bar{p}}{\partial x_i} + \frac{\partial}{\partial x_k}\overline{\tau_{ik}} + \overline{g_i}$$

$$\frac{\partial}{\partial t}\bar{\rho}\tilde{Y}_i + \frac{\partial}{\partial x_k}\bar{\rho}\boxed{v_k Y_i} = -\frac{\partial}{\partial x_k}\overline{J_{ik}} + \boxed{\overline{\dot{w}_i}}$$

Nonlinear Reaction Term

$$\frac{\partial}{\partial t}\bar{\rho}\tilde{h} + \frac{\partial}{\partial x_k}\bar{\rho}\boxed{v_k h} = \frac{\partial \bar{p}}{\partial t} + \frac{\partial}{\partial x_k}\left[\frac{\mu}{\sigma}\frac{\partial \bar{h}}{\partial x_k} + \mu\sum_{i=1}^{N}\left(\frac{1}{Sc_i} - \frac{1}{\sigma}\right)\overline{h_i \frac{\partial Y_i}{\partial x_k}}\right]$$

Nonlinear Convection Term

$$\bar{p} = \bar{\rho}R\tilde{T}$$

$$\tilde{h} = h(\tilde{Y}_i,\ \bar{p},\ \tilde{T})$$

simpleEdmFoam

EDM(Eddy Dissipation Model)

- **EDM (Eddy Dissipation Model)**

$$R_f = A\cdot\bar{Y}_f(\varepsilon/k)$$
$$R_f = A(\bar{Y}_{O2}/r_f)(\varepsilon/k)$$
$$R_f = A\cdot B(\bar{Y}_P/(1+r_f))(\varepsilon/k)$$

Minimum R_f → Local mean rate of combustion

A, B : constants
r_f : stoichiometric oxygen requirement to burn 1kg fuel

- The mean reaction rate is controlled by the turbulent mixing rate
- The reaction rate is limited by the deficient species of reactants or product

- **Finite Rate EDM**

EDM

$$R_{i,r} = \nu'_{i,r} M_{w,i} A \frac{\rho\varepsilon}{k} \min\left(\frac{Y_R}{\nu'_{R,r} M_{w,R}}\right)$$
$$R_{i,r} = \nu'_{i,r} M_{w,i} AB \frac{\rho\varepsilon}{k} \frac{\sum_P Y_P}{\sum_j^N \nu''_{j,r} M_{w,j}}$$

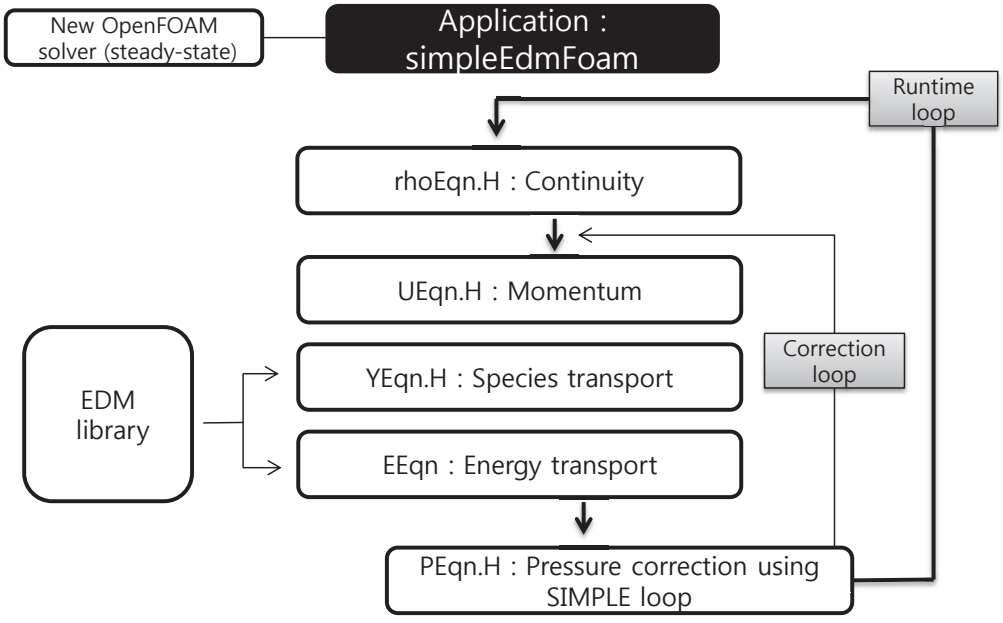
Arrhenius

$$R_{i,r} = \Gamma(\nu''_{i,r} - \nu'_{i,r})\left(k_{f,r}\prod_{j=1}^{N}[C_{j,r}]^{\nu'_j} - k_{b,r}\prod_{j=1}^{N}[C_{j,r}]^{\nu''_j}\right)$$
$$k_r = A_r T^{b_r} exp(-E_r / RT)$$

- The mean reaction rate is determined by the minimum $R_{i,r}$

simpleEdmFoam

Code Structure

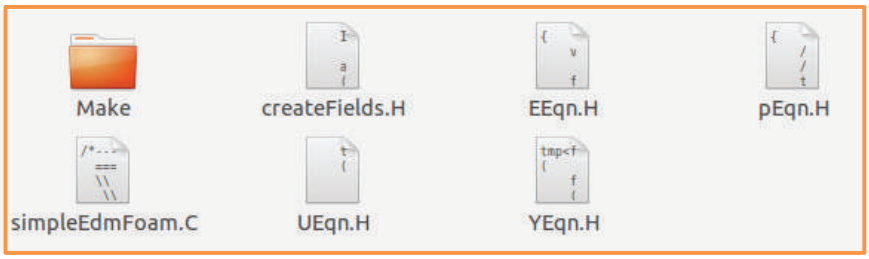


SimpleEdmFoam

simpleEdmFoam

Code Structure

/solvers/simpleEdmFoam



SimpleEdmFoam

/libs/combustionModels_POSTECH/EDM

EDM library

simpleEdmFoam

Solver

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

while (simple.loop())
{
    Info<< "Time = " << runTime.timeName() << nl << endl;

    // --- Pressure-velocity SIMPLE corrector loop
    {
        #include "UEqn.H"
        #include "YEqn.H"
        #include "EEqn.H"
        #include "pEqn.H"
    }

    turbulence->correct();

    runTime.write();

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

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

SimpleEdmFoam.C

UEqn.H

```
tmp<fvVectorMatrix> UEqn
(
    fvm::div(phi, U)
  + turbulence->divDevRhoReff(U)
 ==
    rho.dimensionedInternalField()*g
   + fvOptions(rho, U)
);
```

YEqn.H

```
volScalarField& Yi = Y[i];

fvScalarMatrix YEqn
(
    mvConvection->fvmDiv(phi, Yi)
  - fvm::laplacian(turbulence->muEff(), Yi)
 ==
   combustion->R(Yi)
 + fvOptions(rho, Yi)
);
```

EEqn.H

```
fvScalarMatrix EEqn
(
    mvConvection->fvmDiv(phi, he)
  + (
        he.name() == "e"
      ? fvc::div(phi, volScalarField("Ekp", 0.5*magSqr(U) + p/rho))
      : fvc::div(phi, volScalarField("K", 0.5*magSqr(U)))
    )
  - fvm::laplacian(turbulence->alphaEff(), he)
 ==
   radiation->Sh(thermo)
  + combustion->Sh()
 + fvOptions(rho, he)
);
```

simpleEdmFoam

EDM library

```
// Member Functions

    // Evolution

        //- Correct combustion rate
        virtual void correct();

        //- Fuel consumption rate matrix.
        virtual tmp<fvScalarMatrix> R(volScalarField& Y) const;

        //- Heat release rate calculated from fuel consumption rate matrix
        virtual tmp<volScalarField> dQ() const;

        //-  Return source for enthalpy equation [kg/m/s3]
        virtual tmp<volScalarField> Sh() const;
```

EDM.H

```
template<class Type>
Foam::tmp<Foam::fvScalarMatrix>
Foam::combustionModels::EDM<Type>::R(volScalarField& Y) const
{
    tmp<fvScalarMatrix> tSu(new fvScalarMatrix(Y, dimMass/dimTime));

    fvScalarMatrix& Su = tSu();

    if (this->active())
    {
        const label specieI = this->thermo().composition().species()[Y.name()];

        Su += this->chemistryPtr_->RR(specieI);
    }

    return tSu;
}
```

```
template<class Type>
Foam::tmp<Foam::volScalarField>
Foam::combustionModels::EDM<Type>::dQ() const
{
    tmp<volScalarField> tdQ
    (
        new volScalarField
        (
            IOobject
            (
                typeName + ":dQ",
                this->mesh().time().timeName(),
                this->mesh(),
                IOobject::NO_READ,
                IOobject::NO_WRITE,
                false
            ),
            this->mesh(),
            dimensionedScalar("dQ", dimEnergy/dimTime, 0.0),
            zeroGradientFvPatchScalarField::typeName
        )
    );

    if (this->active())
    {
        tdQ() = this->chemistryPtr_->dQ();
    }

    return tdQ;
}
template<class Type>
Foam::tmp<Foam::volScalarField>
Foam::combustionModels::EDM<Type>::Sh() const
{
    tmp<volScalarField> tSh
    (
        new volScalarField
        (
            IOobject
            (
                typeName + ":Sh",
                this->mesh().time().timeName(),
                this->mesh(),
                IOobject::NO_READ,
                IOobject::NO_WRITE,
                false
            ),
            this->mesh(),
            dimensionedScalar("zero", dimEnergy/dimTime/dimVolume, 0.0),
            zeroGradientFvPatchScalarField::typeName
        )
    );

    if (this->active())
    {
        tSh() = this->chemistryPtr_->Sh();
    }

    return tSh;
```

EDM.C

simpleEdmFoam

Code Structure

/solvers/simpleEdmFoam/Make

```
DEV_PATH=../../libs

EXE_INC = \
    -I$(LIB_SRC)/finiteVolume/lnInclude \
    -I${LIB_SRC}/meshTools/lnInclude \
    -I$(LIB_SRC)/turbulenceModels/compressible/turbulenceModel \
    -I$(LIB_SRC)/lagrangian/distributionModels/lnInclude \
    -I$(LIB_SRC)/thermophysicalModels/specie/lnInclude \
    -I$(LIB_SRC)/thermophysicalModels/basic/lnInclude \
    -I$(LIB_SRC)/thermophysicalModels/properties/liquidProperties/lnInclude \
    -I$(LIB_SRC)/thermophysicalModels/properties/liquidMixtureProperties/lnInclude \
    -I$(LIB_SRC)/thermophysicalModels/properties/solidProperties/lnInclude \
    -I$(LIB_SRC)/thermophysicalModels/properties/solidMixtureProperties/lnInclude \
    -I$(LIB_SRC)/thermophysicalModels/thermophysicalFunctions/lnInclude \
    -I$(LIB_SRC)/thermophysicalModels/reactionThermo/lnInclude \
    -I$(LIB_SRC)/thermophysicalModels/SLGThermo/lnInclude \
    -I$(LIB_SRC)/thermophysicalModels/radiationModels/lnInclude \
    -I$(LIB_SRC)/ODE/lnInclude \
    -I$(LIB_SRC)/regionModels/regionModel/lnInclude \
    -I$(LIB_SRC)/regionModels/surfaceFilmModels/lnInclude \
    -I$(LIB_SRC)/fvOptions/lnInclude \
    -I$(LIB_SRC)/sampling/lnInclude \
    -I$(FOAM_SOLVERS)/combustion/reactingFoam \
    -I$(DEV_PATH)/combustionModels_POSTECH/lnInclude \
    -I$(DEV_PATH)/chemistryModel_POSTEH/lnInclude
```

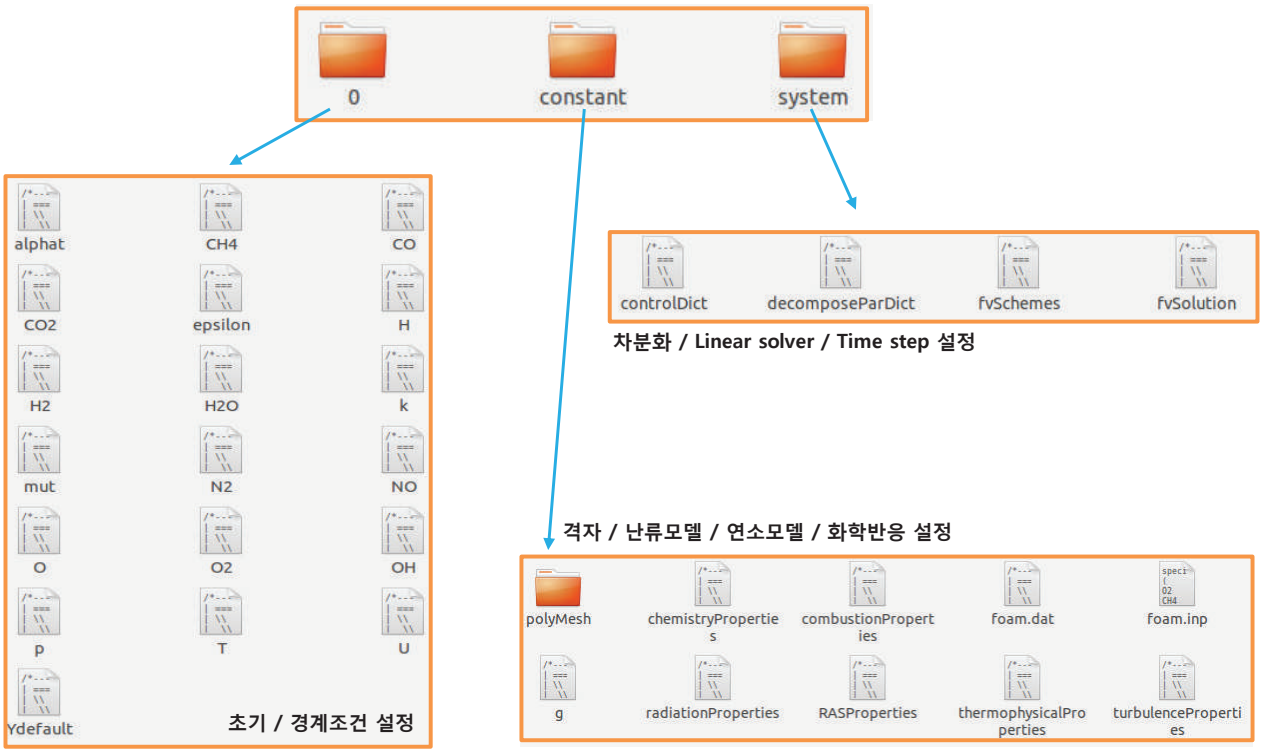
```
EXE_LIBS = \
    -L$(FOAM_USER_LIBBIN) \
            -lcombustionModels_POSTECH \
            -lchemistryModel_POSTECH \
    -lfiniteVolume \
    -lmeshTools \
    -lcompressibleTurbulenceModel \
    -lcompressibleRASModels \
    -lcompressibleLESModels \
    -lspecie \
    -lfluidThermophysicalModels \
    -lliquidProperties \
    -lliquidMixtureProperties \
    -lsolidProperties \
    -lsolidMixtureProperties \
    -lthermophysicalFunctions \
    -lreactionThermophysicalModels \
    -lSLGThermo \
    -lradiationModels \
    -lODE \
    -lregionModels \
    -lsurfaceFilmModels \
    -lfvOptions \
    -lsampling
```

options

simpleEdmFoam

Case Folder

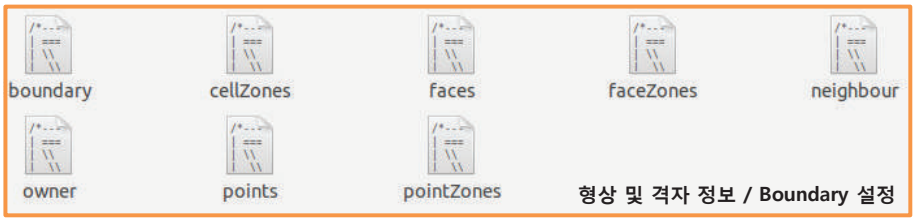
/tutorials/EDM_reacting_flow/



simpleEdmFoam

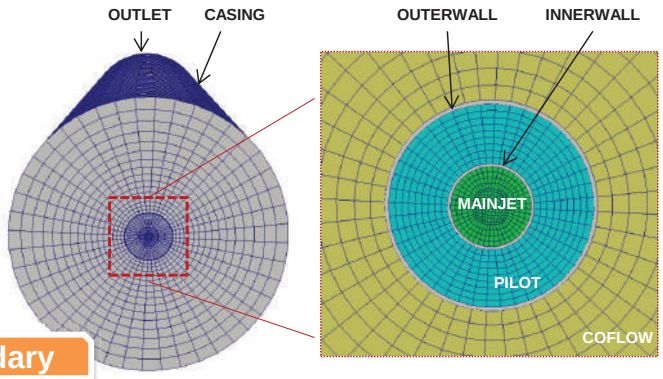
PolyMesh

/tutorials/EDM_reacting_flow/constant/polyMesh




```
MAINJET
{
    type            patch;
    inGroups        1(wall);
    nFaces          297;
    startFace       303732;
}
PILOT
{
    type            patch;
    inGroups        1(wall);
    nFaces          396;
    startFace       304029;
}
COFLOW
{
    type            patch;
    inGroups        1(wall);
    nFaces          704;
    startFace       304425;
}
OUTLET
{
    type            patch;
    nFaces          1573;
    startFace       305129;
}
CASING
{
    type            wall;
    inGroups        1(wall);
    nFaces          2860;
    startFace       306702;
}
INNERWALL
{
    type            wall;
    inGroups        1(wall);
    nFaces          88;
    startFace       309562;
}
OUTERWALL
{
    type            wall;
    inGroups        1(wall);
    nFaces          88;
    startFace       309650;
}
```

boundary



simpleEdmFoam

'0' Folder

- Species mass fraction : CH4, CO, CO2, H, H2, H2O, N2, NO, O, O2, OH

```
dimensions      [0 0 0 0 0 0 0];

internalField   uniform 0.233;

boundaryField
{
    MAINJET
    {
        type            fixedValue;
        value           uniform 0.1966623;
    }
    PILOT
    {
        type            fixedValue;
        value           uniform 0.0540244;
    }
    COFLOW
    {
        type            fixedValue;
        value           uniform 0.233;
    }
    OUTLET
    {
        type            zeroGradient;
    }
    CASING
    {
        type            slip;
    }
    INNERWALL
    {
        type            zeroGradient;
    }
    OUTERWALL
    {
        type            zeroGradient;
    }
}
```

O2

```
dimensions      [0 0 0 0 0 0 0];

internalField   uniform 0.767;

boundaryField
{
    MAINJET
    {
        type            fixedValue;
        value           uniform 0.6473821;
    }
    PILOT
    {
        type            fixedValue;
        value           uniform 0.7342;
    }
    COFLOW
    {
        type            fixedValue;
        value           uniform 0.767;
    }
    OUTLET
    {
        type            zeroGradient;
    }
    CASING
    {
        type            slip;
    }
    INNERWALL
    {
        type            zeroGradient;
    }
    OUTERWALL
    {
        type            zeroGradient;
    }
}
```

N2

```
dimensions      [0 0 0 0 0 0 0];

internalField   uniform 0.0;

boundaryField
{
    MAINJET
    {
        type            fixedValue;
        value           uniform 0.1559556;
    }
    PILOT
    {
        type            fixedValue;
        value           uniform 0.0;
    }
    COFLOW
    {
        type            fixedValue;
        value           uniform 0.0;
    }
    OUTLET
    {
        type            zeroGradient;
    }
    CASING
    {
        type            slip;
    }
    INNERWALL
    {
        type            zeroGradient;
    }
    OUTERWALL
    {
        type            zeroGradient;
    }
}
```

CH4

simpleEdmFoam

'0' Folder

- Turbulent Properties: k(turbulent kinetic energy), epsilon(energy dissipation rate), mut(turbulent viscosity), alphat(turbulent thermal diffusivity)

```
dimensions      [0 2 -2 0 0 0 0];

internalField   uniform 1.0;

boundaryField
{
    MAINJET
    {
        type        turbulentIntensityKineticEnergyInlet;
        intensity   0.0458;           // 4.58% turbulence
        value       uniform 1;      // placeholder
    }
    PILOT
    {
        type        turbulentIntensityKineticEnergyInlet;
        intensity   0.0628;           // 6.28% turbulence
        value       uniform 1;      // placeholder
    }
    COFLOW
    {
        type        turbulentIntensityKineticEnergyInlet;
        intensity   0.0471;           // 4.71% turbulence
        value       uniform 1;      // placeholder
    }
    OUTLET
    {
        type            zeroGradient;
    }
    CASING
    {
        type            slip;
    }
    INNERWALL
    {
        type            compressible::kqRWallFunction;
        value           uniform 0;
    }
    OUTERWALL
    {
        type            compressible::kqRWallFunction;
        value           uniform 0;
    }
}
```

k

```
dimensions      [0 2 -3 0 0 0 0];

internalField   uniform 10; //50.0;

boundaryField
{
    MAINJET
    {
        type            compressible::turbulentMixingLengthDissipationRateInlet;
        mixingLength    5.04e-4;        // turbulent mixing length or turbulence length scale
        value           uniform 200;
    }
    PILOT
    {
        type            compressible::turbulentMixingLengthDissipationRateInlet;
        mixingLength    7.35e-4;        // turbulent mixing length or turbulence length scale
        value           uniform 200;
    }
    COFLOW
    {
        type            compressible::turbulentMixingLengthDissipationRateInlet;
        mixingLength    0.0197;         // turbulent mixing length or turbulence length scale
        value           uniform 200;
    }
    OUTLET
    {
        type            zeroGradient;
    }
    CASING
    {
        type            slip;
    }
    INNERWALL
    {
        type            compressible::epsilonLowReWallFunction
        Cmu             0.09;
        kappa           0.41;
        E               9.8;
        value           uniform 0;
    }
    OUTERWALL
    {
        type            compressible::epsilonLowReWallFunction
        Cmu             0.09;
        kappa           0.41;
        E               9.8;
        value           uniform 0;
    }
}
```

epsilon

Combustion Laboratory POHANG UNIVERSITY OF SCIENCE AND TECHNOLOGY POSTECH

simpleEdmFoam

'0' Folder

- U(velocity), T(temperature), p(pressure)

```
dimensions      [0 1 -1 0 0 0 0];

internalField   uniform (0 0.9 0);

boundaryField
{
    MAINJET
    {
        type            fixedValue;
        value           uniform (0 49.6 0);
    }
    PILOT
    {
        type            fixedValue;
        value           uniform (0 11.4 0);
    }
    COFLOW
    {
        type            fixedValue;
        value           uniform (0 0.9 0);
    }
    OUTLET
    {
        type            zeroGradient;
    }
    CASING
    {
        type            slip;
    }
    INNERWALL
    {
        type            fixedValue;
        value           uniform (0 0 0);
    }
    OUTERWALL
    {
        type            fixedValue;
        value           uniform (0 0 0);
    }
}
```

U

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

internalField   uniform 300;

boundaryField
{
    MAINJET
    {
        type            fixedValue;
        value           uniform 294.0;
    }
    PILOT
    {
        type            fixedValue;
        value           uniform 1880.0;
    }
    COFLOW
    {
        type            fixedValue;
        value           uniform 291.0;
    }
    OUTLET
    {
        type            zeroGradient;
    }
    CASING
    {
        type            slip;
    }
    INNERWALL
    {
        type            zeroGradient;
    }
    OUTERWALL
    {
        type            zeroGradient;
    }
}
```

T

```
dimensions      [1 -1 -2 0 0 0 0];

internalField   uniform 100615.725;

boundaryField
{
    MAINJET
    {
        type            zeroGradient;
    }
    PILOT
    {
        type            zeroGradient;
    }
    COFLOW
    {
        type            zeroGradient;
    }
    OUTLET
    {
        type            fixedValue;
        value           uniform 100615.725;
    }
    CASING
    {
        type            slip;
    }
    INNERWALL
    {
        type            zeroGradient;
    }
    OUTERWALL
    {
        type            zeroGradient;
    }
}
```

p

simpleEdmFoam

'constant' Folder

- Combustion & Turbulence model

```
chemistryType
{
    chemistrySolver   noChemistrySolver;
    chemistryThermo   rho;
}

chemistry       on;

initialChemicalTimeStep 1e-7;

odeCoeffs
{
    solver          seulex;
}
```

chemistryProperties

```
active      true;

combustionModel EDM<rhoChemistryCombustion>;

EDMCoeffs
{
    A 4.0;
    B 0.5;
    finiteRate false;
}
```

combustionProperties

```
FoamFile
{
    version     2.0;
    format      ascii;
    class       dictionary;
    location    "constant";
    object      turbulenceProperties;
}
// * * * * * * * * * * * * * * * * * *

simulationType  RASModel;
```

turbulenceProperties

```
FoamFile
{
    version     2.0;
    format      ascii;
    class       dictionary;
    location    "constant";
    object      RASProperties;
}
// * * * * * * * * * * * * * * * * * *

RASModel        kEpsilon;

turbulence      on;

printCoeffs     yes
```

RASProperties

Combustion Laboratory POHANG UNIVERSITY OF SCIENCE AND TECHNOLOGY POSTECH

simpleEdmFoam

Solver

- Chemical Reactions

```
thermoType
{
    type            heRhoThermo;
    mixture         reactingMixture;
    transport       sutherland;
    thermo          janaf;
    energy          sensibleEnthalpy;
    equationOfState perfectGas;
    specie          specie;
}

chemistryReader foamChemistryReader;

foamChemistryFile "$FOAM_CASE/constant/foam.inp";

foamChemistryThermoFile "$FOAM_CASE/constant/foam.dat";

inertSpecie
```

thermophysicalProperties

```
reactions
{
    methaneReaction1
    {
        type    irreversibleArrheniusReaction;
        reaction "CH4^(0.7) + 1.5O2^(0.8) = CO + 2H2O";
        A       5.012E11; //[1/s]
        beta    0;
        Ta      24450; // = [Ea / RR] = [(J/kmole) / (J/kmole/K)] = [K]
    }
    methaneReaction2
    {
        type    irreversibleArrheniusReaction;
        reaction "CO + 0.5O2 = CO2";
        A       5.012E11; //[1/s]
        beta    0;
        Ta      24450; // = [Ea / RR] = [(J/kmole) / (J/kmole/K)] = [K]
    }
}
```

foam.inp

```
CH4
{
    specie
    {
        nMoles          1;
        molWeight       16.043;
    }
    thermodynamics
    {
        Tlow            200;
        Thigh           6000;
        Tcommon         1000;
        highCpCoeffs    ( 1.65326 0.0100263 -3.31661e-06 5.36483e-10 -3.146
        lowCpCoeffs     ( 5.14911 -0.0136622 4.91454e-05 -4.84247e-08 1.666
    }
    transport
    {
        As              1.67212e-06;
        Ts              170.672;
    }
}
CO
{
    specie
    {
        nMoles          1;
        molWeight       28.0106;
    }
    thermodynamics
    {
        Tlow            200;
        Thigh           6000;
        Tcommon         1000;
        highCpCoeffs    ( 3.04849 0.00135173 -4.85794e-07 7.88536e-11 -4.69
        lowCpCoeffs     ( 3.57953 -0.000610354 1.01681e-06 9.07006e-10 -9.0
    }
    transport
    {
        As              1.67212e-06;
        Ts              170.672;
    }
}
```

foam.dat

simpleEdmFoam

Solver

- Discretization / Linear Solver / Relaxation Factor

```
ddtSchemes
{
    default         steadyState;
}

gradSchemes
{
    default         cellLimited Gauss linear 1;
}

divSchemes
{
    div(phi,U)      bounded Gauss upwind;
    div(phid,p)     bounded Gauss upwind;
    div(phi,K)      bounded Gauss linear;
    div(phi,h)      bounded Gauss upwind;
    div(phi,hs)     bounded Gauss upwind;
    div(phi,k)      bounded Gauss upwind;
    div(phi,epsilon) bounded Gauss upwind;
    div((muEff*dev2(T(grad(U))))) Gauss linear;
    div(phi,Yi_h)  Gauss upwind;
}

laplacianSchemes
{
    default         Gauss linear corrected;
    laplacian(muEff,U) Gauss linear corrected;
    laplacian(mut,U) Gauss linear corrected;
    laplacian(DkEff,k) Gauss linear corrected;
    laplacian(DepsilonEff,epsilon) Gauss linear corrected;
    laplacian(DREff,R) Gauss linear corrected;
    laplacian((rho*(1|A(U))),p) Gauss linear corrected;
    laplacian(alphaEff,h) Gauss linear corrected;
    laplacian(alphaEff,hs) Gauss linear corrected;

}
```

fvSchemes

```
p
{
    solver          GAMG;
    tolerance       1e-08;
    relTol          0.1;
    smoother        GaussSeidel;
    cacheAgglomeration on;
    nCellsInCoarsestLevel 40;
    agglomerator    faceAreaPair;
    mergeLevels     1;
}

"(Yi|h|U|k|epsilon)"
{
    solver          PBiCG;
    preconditioner  DILU;
    tolerance       1e-06;
    relTol          0.1;
}

relaxationFactors
{
    fields
    {
        p               0.15;
        rho             1.0;
    }
    equations
    {
        U               0.3;
        h               0.5;
        Yi              0.5;
        ".*"            0.5;
    }
}
```

fvSolution

Combustion Laboratory POHANG UNIVERSITY OF SCIENCE AND TECHNOLOGY POSTECH

simpleEdmFoam

Solver

- Calculation / MPI(Message Passing Interface)

controldict

```
application     simpleEdmFoam;

startFrom       startTime; //latestTime;

startTime       0;

stopAt          endTime;

endTime         2000;

deltaT          1;

writeControl    timeStep;

writeInterval   100;

purgeWrite      0;

writeFormat     ascii;

writePrecision  10;

writeCompression off;

timeFormat      general;

timePrecision   6;

runTimeModifiable yes;

adjustTimeStep
```

decomposePardict

```
FoamFile
{
    version     2.0;
    format      ascii;
    class       dictionary;
    location    "system";
    object      decomposeParDict;
}
// * * * * * * * * * * * * * * * * *

numberOfSubdomains 4;

method          scotch;

simpleCoeffs
{
    n               ( 2 2 1 );
    delta           0.001;
}

hierarchicalCoeffs
{
    n               ( 1 1 1 );
    delta           0.001;
    order           xyz;
}

manualCoeffs
{
    dataFile        "";
}

distributed     no;

roots
```


simpleEdmFoam

Tutorial

1. **Open ‘Terminal’ : click** 

2. **~$ cd tutorials/EDM_reacting_flow**

```
:~$ cd tutorials/EDM_reacting_flow
```

3. **~$ decomposePar**

```
:~/tutorials/EDM_reacting_flow$ decomposePar
```

```
Time = 0

Processor 0: field transfer
Processor 1: field transfer
Processor 2: field transfer
Processor 3: field transfer

End.
```

4. **~$ mpirun –np 4 simpleEdmFoam -parallel**

```
:~/tutorials/EDM_reacting_flow$ mpirun -np 4 simpleEdmFoam -parallel
```

simpleEdmFoam

Tutorial

- Calculating

```
Create time

Create mesh for time = 0


Reading g

SIMPLE: no convergence criteria found. Calculations will run for 2000

Creating combustion model

Selecting combustion model EDM<rhoChemistryCombustion>
Selecting chemistry type
{
    chemistrySolver noChemistrySolver;
    chemistryThermo rho;
}

Selecting thermodynamics package
{
    type            heRhoThermo;
    mixture         reactingMixture;
    transport       sutherland;
    thermo          janaf;
    energy          sensibleEnthalpy;
    equationOfState perfectGas;
    specie          specie;
}
```

①

```
Selecting chemistryReader foamChemistryReader
chemistryModel: Number of species = 6 and reactions = 2
    using Eddy Dissipation Model
    A = 4
    B = 0.5
Creating component thermo properties:
    multi-component carrier - 6 species
    liquids - 1 components
    solids - 2 components

Reading field U

Reading/calculating face flux field phi

Creating turbulence model

Selecting turbulence model type RASModel
Selecting RAS turbulence model kEpsilon
kEpsilonCoeffs
{
    Cmu             0.09;
    C1              1.44;
    C2              1.92;
    C3              -0.33;
    sigmak          1;
    sigmaEps        1.3;
    Prt             1;
}

Creating multi-variate interpolation scheme

Selecting radiationModel none
No finite volume options present
```

②

simpleEdmFoam

Tutorial

- Calculating

Time step Linear solver Equations

```
Starting time loop

Time = 1

DILUPBiCG:  Solving for Ux, Initial residual = 0.0002960869359, Final residual = 6.870197932e-06, No Iterations 1
DILUPBiCG:  Solving for Uy, Initial residual = 6.049958403e-05, Final residual = 1.572114608e-06, No Iterations 1
DILUPBiCG:  Solving for Uz, Initial residual = 0.0002957334339, Final residual = 7.595771818e-06, No Iterations 1
DILUPBiCG:  Solving for O2, Initial residual = 0.002882037756, Final residual = 8.21132418e-05, No Iterations 1
DILUPBiCG:  Solving for CH4, Initial residual = 0.0005810339876, Final residual = 4.30096263e-05, No Iterations 1
DILUPBiCG:  Solving for CO, Initial residual = 0.09487031987, Final residual = 0.003093827664, No Iterations 1
DILUPBiCG:  Solving for CO2, Initial residual = 0.001474795321, Final residual = 9.3019924e-05, No Iterations 1
DILUPBiCG:  Solving for H2O, Initial residual = 0.006647976822, Final residual = 0.0002133022425, No Iterations 1
DILUPBiCG:  Solving for h, Initial residual = 0.005070213163, Final residual = 0.0001600538902, No Iterations 1
T gas min/max   = 290.9999959, 1880
GAMG:  Solving for p, Initial residual = 0.006066414538, Final residual = 0.0002664107105, No Iterations 2
time step continuity errors : sum local = 0.002076678972, global = 0.002075187551, cumulative = 0.002075187551
p min/max = 100585.7863, 100686.8456
DILUPBiCG:  Solving for epsilon, Initial residual = 0.0001366042733, Final residual = 4.389133915e-06, No Iterations 1
DILUPBiCG:  Solving for k, Initial residual = 8.348825295e-05, Final residual = 3.595663784e-06, No Iterations 1
ExecutionTime = 1.6 s  ClockTime = 2 s
```

③

Time min/max value Residual

Combustion Laboratory POHANG UNIVERSITY OF SCIENCE AND TECHNOLOGY POSTECH

simpleEdmFoam

Tutorial

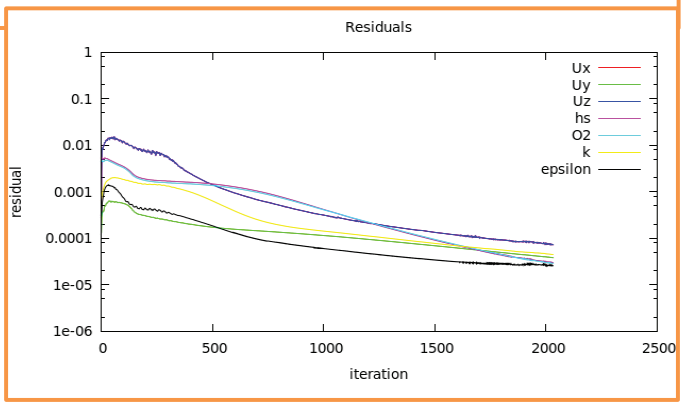
- Calculating

④

```
Time = 2000

DILUPBiCG:  Solving for Ux, Initial residual = 7.719383786e-05, Final residual = 4.62814506e-06, No Iterations 1
DILUPBiCG:  Solving for Uy, Initial residual = 3.986213421e-05, Final residual = 2.084343016e-06, No Iterations 1
DILUPBiCG:  Solving for Uz, Initial residual = 7.7177264e-05, Final residual = 4.63616661e-06, No Iterations 1
DILUPBiCG:  Solving for O2, Initial residual = 3.037889412e-05, Final residual = 1.843970396e-06, No Iterations 2
DILUPBiCG:  Solving for CH4, Initial residual = 3.73770304e-06, Final residual = 8.673845357e-07, No Iterations 1
DILUPBiCG:  Solving for CO, Initial residual = 0.000294306165, Final residual = 2.251418875e-05, No Iterations 2
DILUPBiCG:  Solving for CO2, Initial residual = 5.261531424e-05, Final residual = 1.826669566e-06, No Iterations 2
DILUPBiCG:  Solving for H2O, Initial residual = 2.473835278e-05, Final residual = 7.437427403e-07, No Iterations 2
DILUPBiCG:  Solving for h, Initial residual = 3.097368044e-05, Final residual = 2.8070519e-06, No Iterations 2
T gas min/max   = 290.9999822, 2236.181862
GAMG:  Solving for p, Initial residual = 0.001798782669, Final residual = 0.0001534781165, No Iterations 4
time step continuity errors : sum local = 1.367110234e-05, global = -1.044004129e-05, cumulative = 0.4514254368
p min/max = 100602.0955, 101074.8348
DILUPBiCG:  Solving for epsilon, Initial residual = 2.751136602e-05, Final residual = 4.41577188e-07, No Iterations 2
DILUPBiCG:  Solving for k, Initial residual = 4.66679674e-05, Final residual = 3.2940078e-07, No Iterations 3
ExecutionTime = 485.94 s  ClockTime = 487 s
```

```
relaxationFactors
{
    fields
    {
        p               0.3;
        rho             1.0;
    }
    equations
    {
        U               0.5;
        h               0.7;
        Yi              0.7;
        ".*"            0.7;
    }
}
```

fvSolution



simpleEdmFoam

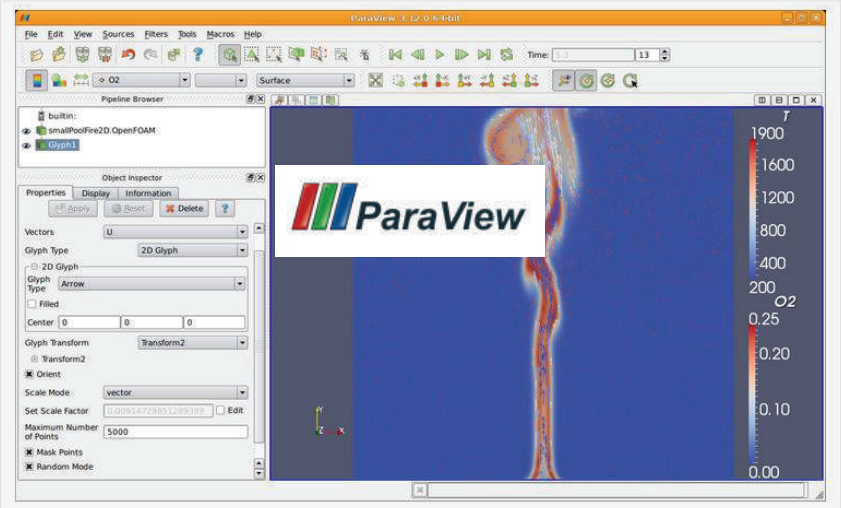
Post Processing

1. **~$ reconstructPar**

```
:~/tutorials/EDM_reacting_flow$ reconstructPar
```

2. **~$ cd tutorials/EDM_reacting_flow/paraFoam**

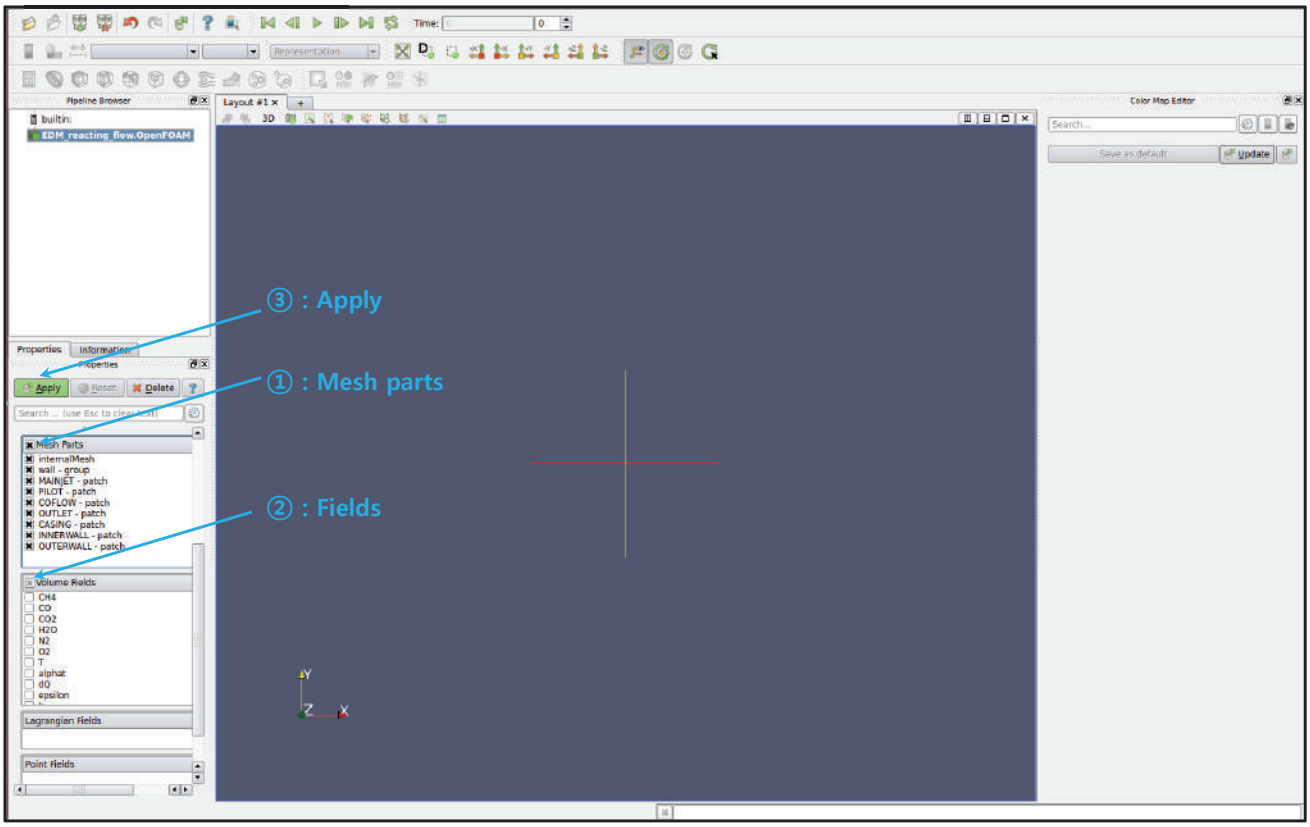
```
:~/tutorials/EDM_reacting_flow$ paraFoam
```



OpenFOAM smallPoolFire example visualized with ParaView

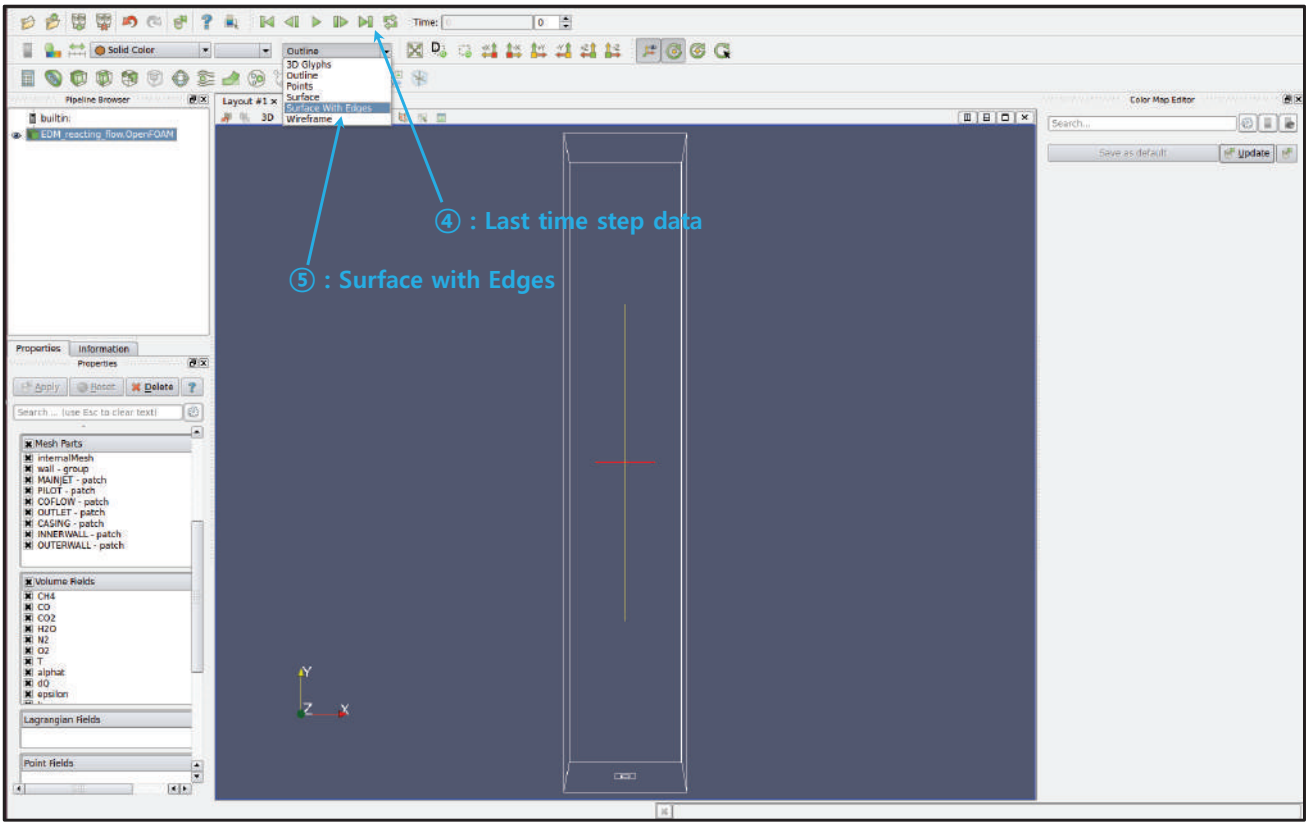
simpleEdmFoam

Post Processing



simpleEdmFoam

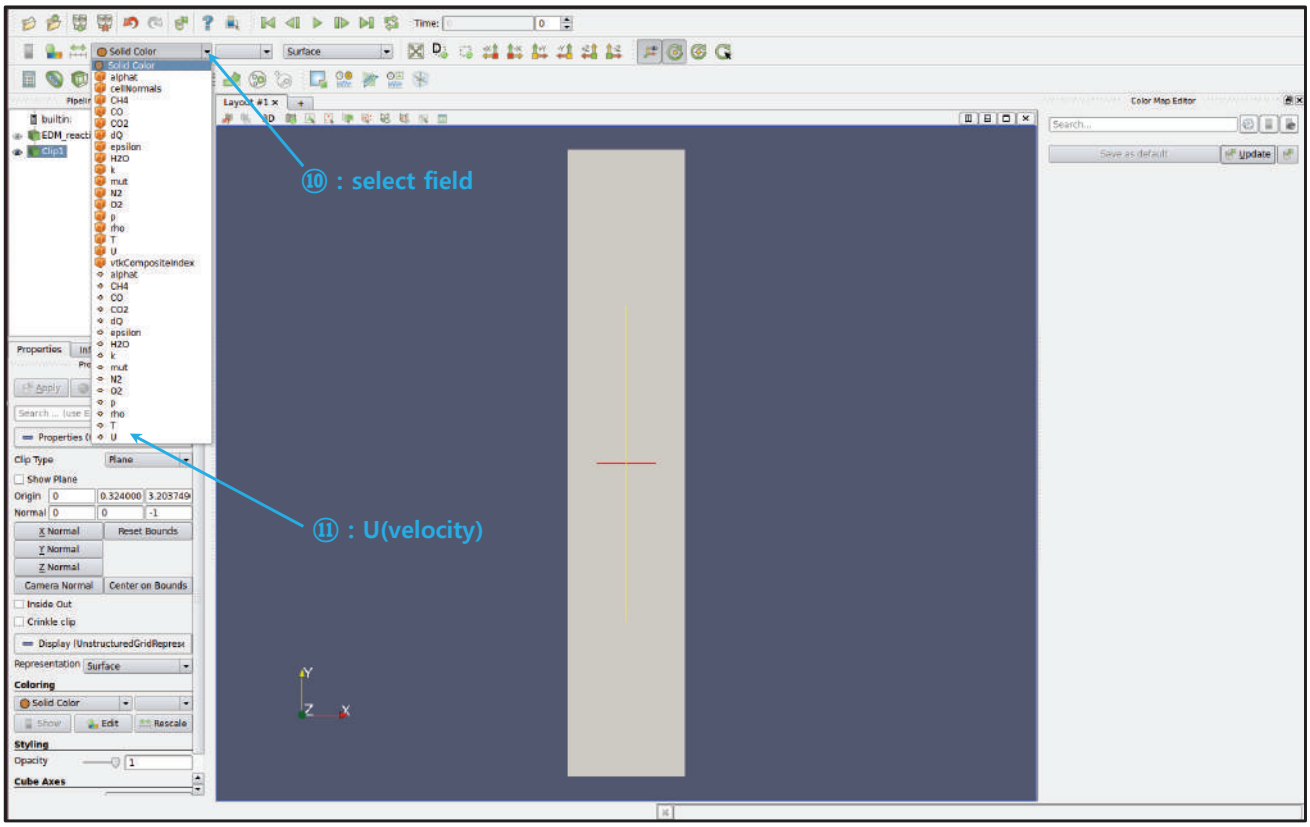
Post Processing



Combustion Laboratory
POHANG UNIVERSITY OF SCIENCE AND TECHNOLOGY
POSTECH

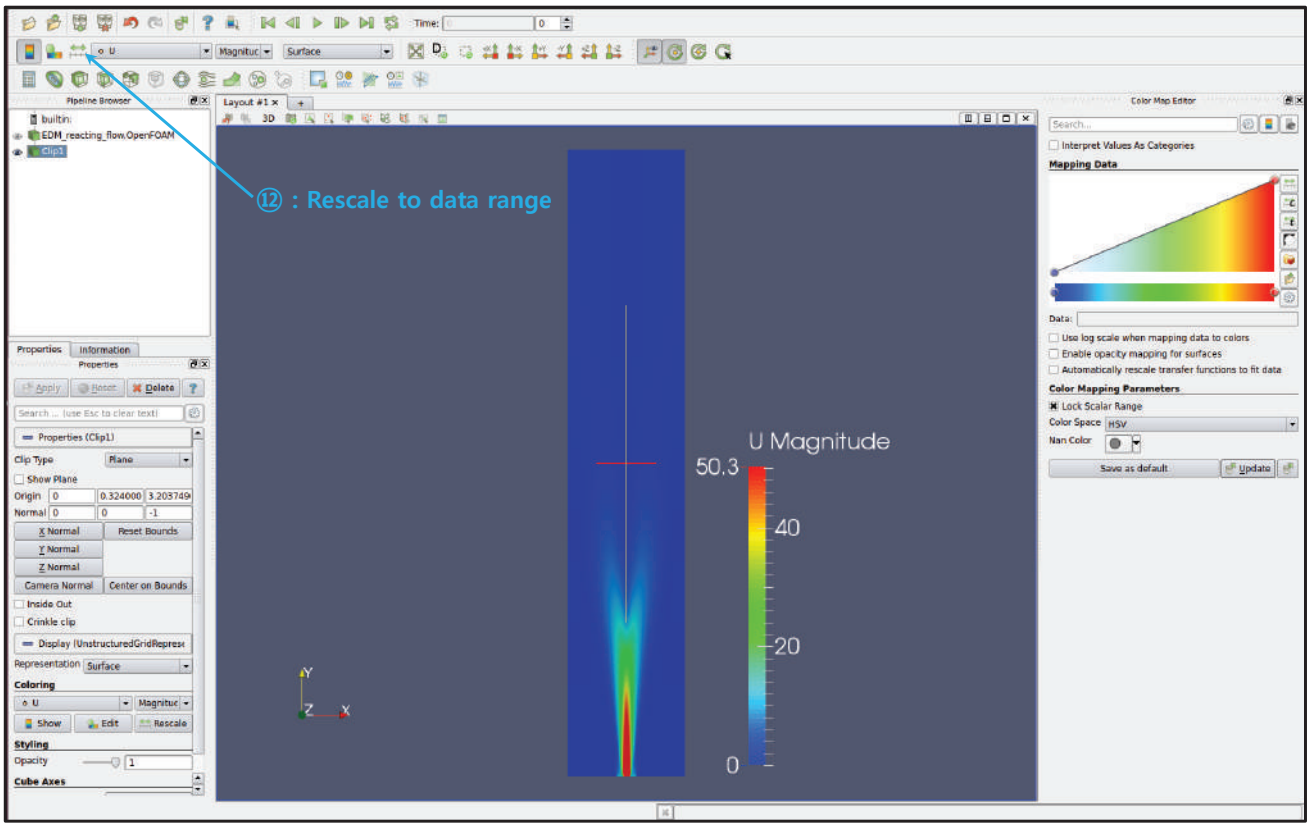
simpleEdmFoam

Post Processing



simpleEdmFoam

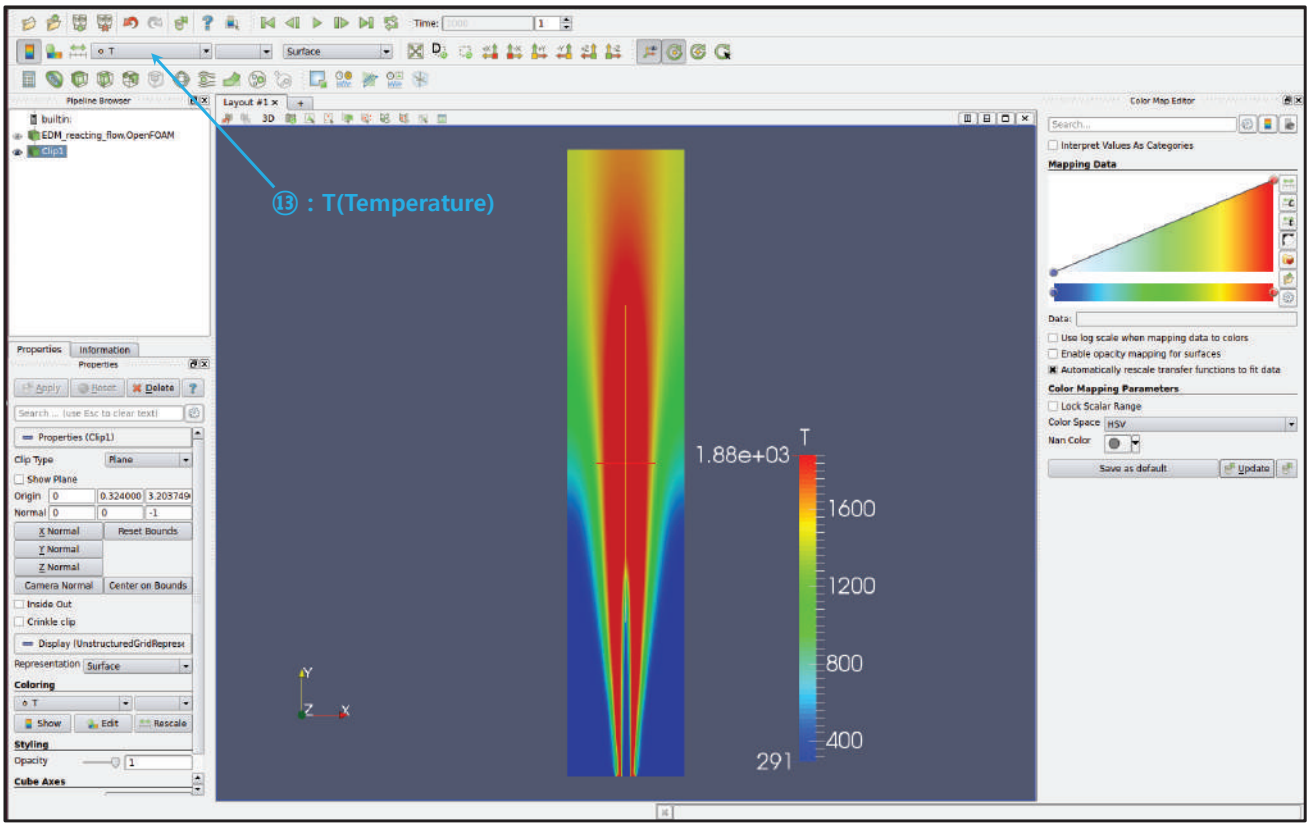
Post Processing



Combustion Laboratory
POHANG UNIVERSITY OF SCIENCE AND TECHNOLOGY
POSTECH

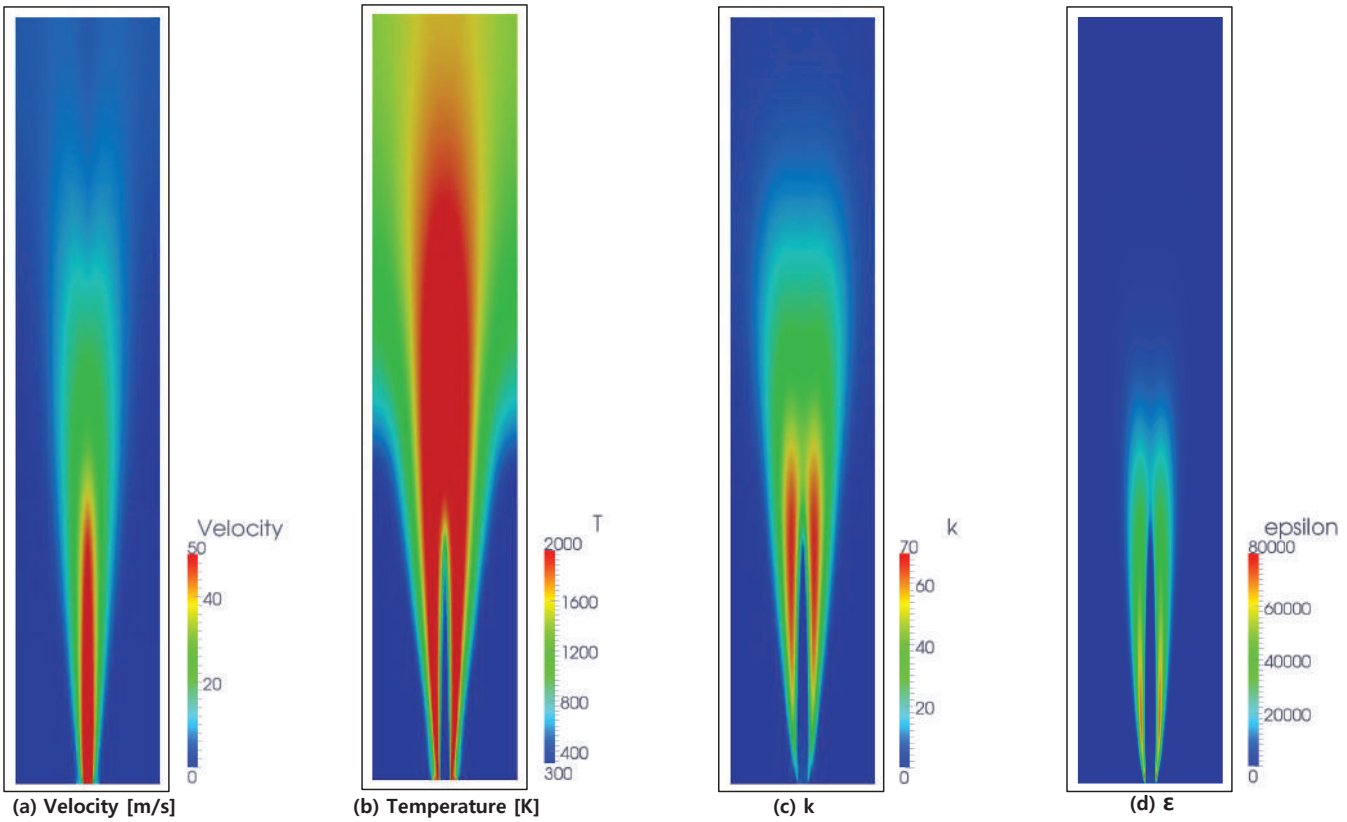
simpleEdmFoam

Post Processing



simpleEdmFoam

Results

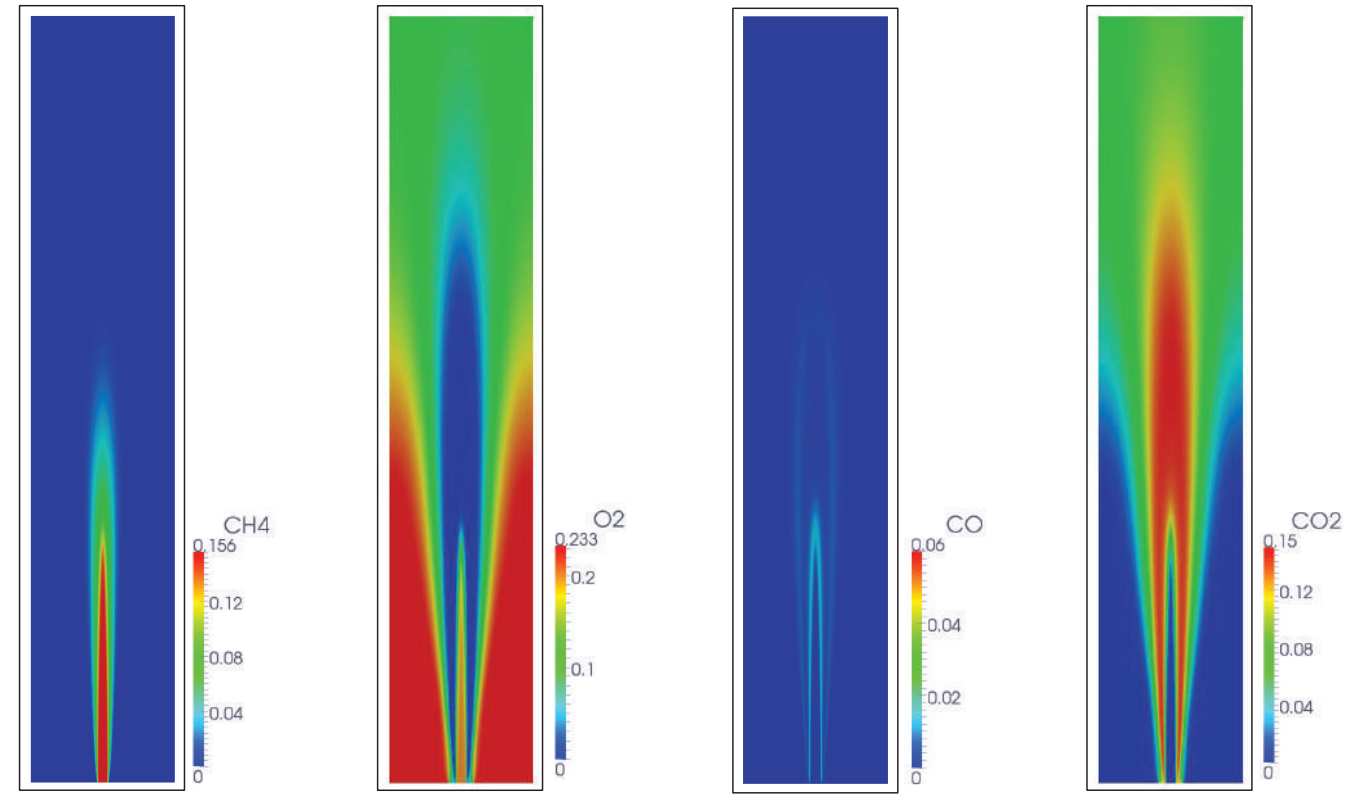


(a) Velocity [m/s] (b) Temperature [K] (c) k (d) ε

Combustion Laboratory
POHANG UNIVERSITY OF SCIENCE AND TECHNOLOGY
POSTECH

simpleEdmFoam

Results



(a) CH4 mass fraction (b) O2 mass fraction (c) CO mass fraction (d) CO2 mass fraction

SLFMFoam

Nemerical Combustion Model – Nonpremixed

- **Equilibrium Assumption – Infinitely fast chemistry**
- **Laminar Flamelet Model**
  - Steady – SLFM(Stationary Laminar Flamelet Model)
  - Transient – RIF(Representative Interactive Flamelet)

Laminar Flame Structure ↕ **Turbulence** — ***Stretch or Scalar Dissipation Rate***

- **Conditional Averaging**

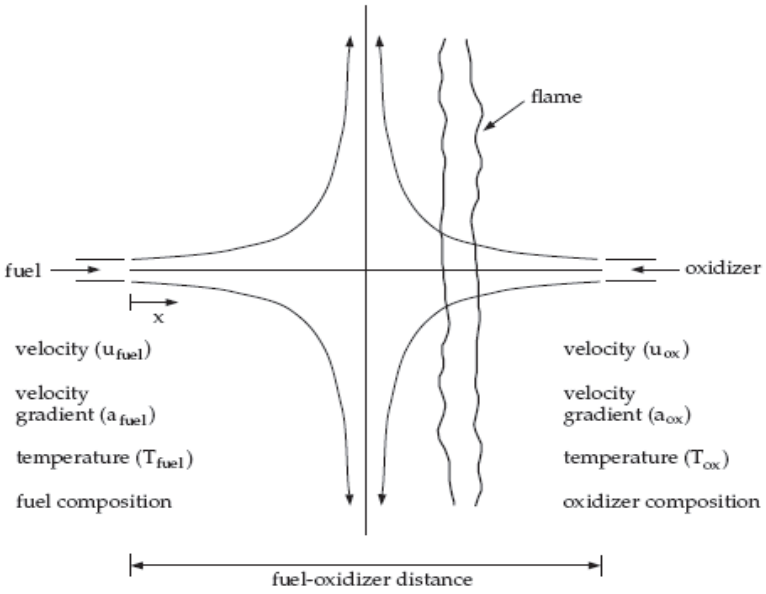
CMC(Conditional Moment Closure)

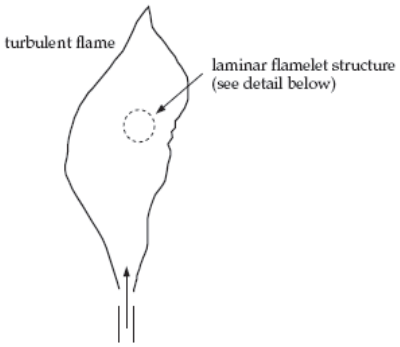
- Deterministic relationships between mixture fraction and all other reactive scalars.
- 1st order closure for chemical reaction rate.

SLFMFoam

SLFM

- Turbulent flame modeled as an ensemble of thin, laminar, locally 1-D flamelet structures
- Flame structure in terms of stoichiometric Scalar Dissipation Rate (N_{st})





$$\frac{\partial Y_i}{\partial t} = R_{i,kin} + D_i \chi \frac{\partial^2 Y_i}{\partial f^2}$$

$$\frac{\partial T}{\partial t} = -\sum_{i=1}^{N} \frac{h_i R_{i,kin}}{C_p} + \alpha\chi \frac{\partial^2 T}{\partial f^2}$$

SLFMFoam

SLFM

- Governing equation

$$0 = \langle N|\eta\rangle \frac{\partial^2 Q_\eta}{\partial \eta^2} + \langle \dot{w}_\eta|\eta\rangle$$

- Assumed beta-function PDF

$$\tilde{P}(\eta) = \frac{\zeta^{\alpha-1}(1-\zeta)^{\beta-1}}{\Gamma(\alpha)\Gamma(\beta)}\Gamma(\alpha+\beta)$$

$$\text{where } \alpha=\tilde{\zeta}\gamma,\ \ \beta=(1-\tilde{\zeta})\gamma,\ \ \gamma=\frac{\tilde{\zeta}(1-\tilde{\zeta})}{\zeta''^2}$$

- Mixture fraction

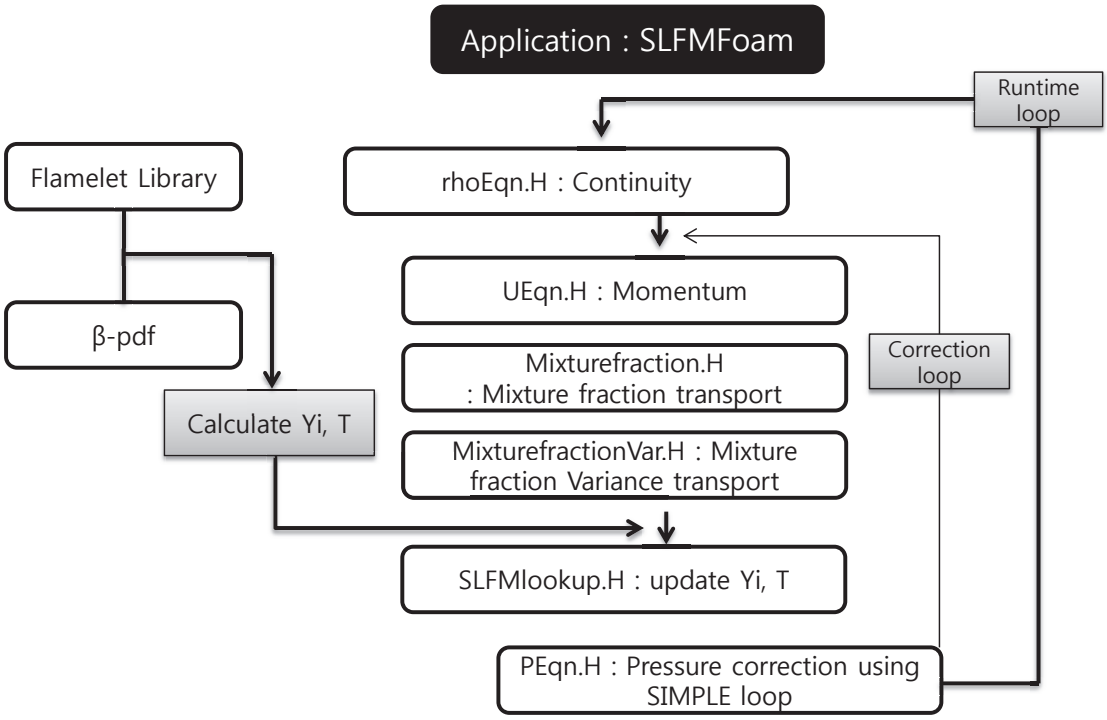
$$\frac{\partial(\bar{\rho}\tilde{\xi})}{\partial t} + \nabla\cdot\left(\bar{\rho}\tilde{\boldsymbol{u}}\tilde{\xi}\right) = \nabla\cdot\left[\frac{\mu_t}{Sc_{\tilde{\xi}}}\nabla\tilde{\xi}\right]$$

- Mixture fraction variance

$$\frac{\partial(\bar{\rho}\widetilde{\xi''^2})}{\partial t} + \nabla\cdot\left(\bar{\rho}\tilde{\boldsymbol{u}}\widetilde{\xi''^2}\right) = \nabla\cdot\left[\frac{\mu_t}{Sc_{\widetilde{\xi''^2}}}\nabla\widetilde{\xi''^2}\right] + \frac{2\mu_t}{Sc_{\widetilde{\xi''^2}}}\left(\nabla\tilde{\xi}\right)^2 - \bar{\rho}\tilde{\chi}$$

SLFMFoam

Code Structure



SLFMFoam

SLFMFoam

Code Structure

/solvers/SLFMFoam


```
#include "fvCFD.H"
#include "turbulenceModel.H"
#include "rhoCombustionModel.H"
#include "fvIOoptionList.H"
#include "SLGThermo.H"
#include "simpleControl.H"
#include "multivariateScheme.H"
#include "IFstream.H"
#include "OFstream.H"
#include "interpolateXY.H"
#include "Math.H"
#include "BetaPDF.H"
#include <sstream>
#include <iomanip>
```

SLFMFoam.C

```
    #include "setRootCase.H"
    #include "createTime.H"
    #include "createMesh.H"
    #include "readGravitationalAcceleration.H"

    simpleControl simple(mesh);

    #include "createFields.H"
    #include "createFvOptions.H"
    #include "initContinuityErrs.H"

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

    #include "readCMCProperties.H"
```

SLFMFoam

Solver

```
Info<<"Reading field mf\n"<<endl;
volScalarField mf
(
    IOobject
    (
        "mf",
        runTime.timeName(),
        mesh,
        IOobject::MUST_READ,
        IOobject::AUTO_WRITE
    ),
    mesh
);
```

Mixture fraction

```
Info<<"Reading field mfVar\n"<<endl;
volScalarField mfVar
(
    IOobject
    (
        "mfVar",
        runTime.timeName(),
        mesh,
        IOobject::MUST_READ,
        IOobject::AUTO_WRITE
    ),
    mesh
);
```

Mixture fraction variance

```
volScalarField SDR
(
    IOobject
    (
        "SDR",
        runTime.timeName(),
        mesh,
        IOobject::NO_READ,
        IOobject::AUTO_WRITE
    ),
    mesh,
    dimensionedScalar("SDR", dimEnergy/dimTime/dimEnergy, 0.0)
);
```

Scalar dissipation rate

readCMCProperties.H

```
    volScalarField deltaftn
    (
        IOobject
        (
            "deltaftn",
            runTime.timeName(),
            mesh,
            IOobject::NO_READ,
            IOobject::NO_WRITE
        ),
        mesh,
        dimensionedScalar("deltaftn", dimless, -1.0)
    );


    IOdictionary CMCdict
    (
        IOobject
        (
            "CMCdict",
            runTime.constant(),
            mesh,
            IOobject::MUST_READ,
            IOobject::NO_WRITE
        )
    );

    scalar etamax(readScalar(CMCdict.lookup("etamax")));
    scalar zetamax(200);
    scalarField etaValue(etamax+1, 0.0);
    scalarField zetaValue(zetamax+1, 0.0);
    scalarField newZetaValue;
    scalarField deta(etamax+1 ,0.0 );
    scalarField dzeta(zetamax+1 ,0.0 );

    scalar section(readScalar(CMCdict.lookup("section")));
    scalarField MFcut(section+1, 0.0);
    scalarField Neta(section+1, 0.0);

    scalarField pdf(zetamax+1,0.0);
    scalarField pdf2;
    scalarField f(zetamax+1, 0.0);
```


SLFMFoam

Solver

SLFMFoam.C

```
// --- Pressure-velocity SIMPLE corrector loop
{
    #include "UEqn.H"

        tmp<fv::convectionScheme<scalar> > mvConvection
        (
            fv::convectionScheme<scalar>::New
            (
                mesh,
                fields,
                phi,
                mesh.divScheme("div(phi,Yi_h)")
            )
        );

    #include "Mixturefraction.H"
    #include "MixturefractionVar.H"
    #include "SLFMlookup.H"
    #include "pEqn.H"
}

turbulence->correct();

runTime.write();
```

UEqn.C

```
tmp<fvVectorMatrix> UEqn
(
    fvm::div(phi, U)
  + turbulence->divDevRhoReff(U)
 ==
    rho.dimensionedInternalField()*g
  + fvOptions(rho, U)
);
```

Mixturefraction.C

```
fvScalarMatrix mfEqn
(
        (
        mvConvection->fvmDiv(phi, mf)
        - fvm::laplacian(1.47*turbulence->mut(), mf)
        + fvOptions(rho, mf)
        )
);
```

MixturefractionVar.C

```
SDR =  turbulence->epsilon() * mfVar / turbulence->k();
volVectorField Gradmf = fvc::grad(mf);


fvScalarMatrix mfVarEqn
(
        (
        mvConvection->fvmDiv(phi, mfVar)
        - fvm::laplacian(1.47*turbulence->mut(), mfVar)
        -2*(1.47*turbulence->mut())*(Gradmf & Gradmf)
        +2*rho*SDR
        + fvOptions(rho, mfVar)
        )
);
```

SLFMFoam

Solver

Flamelet library

```
for(label k=0; k<ny ; k++) //species loop in sdr*.inp
{
  sdrFile>>yname;

  label species_index(0);
  for(label ii = 0 ; ii<Ysize ; ii++)//species loop in OpenFOAM mechanism
  {
    if(yname == Y[ii].name())
    {
      species_index = ii; //find appropriate species index
    }
  }
  for(label j=0 ; j<neta ; j++)
  {
    sdrFile>>Y_SLFM_temp[(ny+1)*j + species_index];
  }
  for(label j=0 ; j<neta ; j++)
  {
    sdrFile>>gg;
  }
}
```

makeSLFMlib.H

```
for(label k=ny ; k<ny+3 ; k++)
{
  sdrFile>>yname;
  if(yname == "TEMPERATURE")
  {
    for(label j=0 ; j<neta ; j++)
    {
      sdrFile>>Y_SLFM_temp[(ny+1)*j + ny];
    }
    for(label j=0 ; j<neta ; j++)
    {
      sdrFile>>gg;
    }
  }
  else
  {
    for(label j=0 ; j<neta ; j++)
    {
      sdrFile>>gg;
    }
  }
}
```

wylib.inp (OFstream wyFile)

```
INPUT FILE FOR THE SR-CMC ROUTINES
No.GRID POINTS No. SPECIES
          75          53
PRESSURE(ATM)
  1.00000000000000
MIXTURE FRACTION GRID
 0.000000000000000E+000  1.250000000000000E-002  2.500000000000000E-002
 3.750000000000000E-002  5.000000000000000E-002  6.250000000000000E-002
 7.500000000000000E-002  8.750000000000001E-002  0.100000000000000
 0.112500000000000       0.125000000000000       0.137500000000000
                                 ⋮
H2
 0.000000000000000E+000  1.199876917356024E-010  2.404599628458580E-010
 3.634374359506292E-010  4.923059051856218E-010  6.402815132399437E-010
 8.231616025136224E-010  1.106859158062047E-009  1.566260931137226E-009
 2.474411258063766E-009  4.221179938897097E-009  7.828444157308641E-009
 1.446600851949211E-008  2.504993582664863E-008  4.365857706813568E-008
 7.641434971437619E-008  1.437508917987143E-007  2.647941346143816E-007
 4.297697281978580E-007  7.304554901008893E-007  1.227437577116229E-006
                                 ⋮
CO
 0.000000000000000E+000  2.758643139581209E-008  5.546642769447130E-008
 8.403598134785700E-008  1.135376351029554E-007  1.449717185692140E-007
 1.789526379711045E-007  2.184383358321492E-007  2.656790358058966E-007
 3.298029260745916E-007  4.225705966158698E-007  5.744696341063719E-007
 8.178988840065641E-007  1.165286898907161E-006  1.738412965004570E-006
 2.702136085506771E-006  4.581700348802778E-006  6.820507934488328E-006
 1.251701913067433E-005  2.101857752475426E-005  3.527568015458441E-005
 6.242780209130965E-005  9.626731824717643E-005  1.645046667517994E-004
 2.698135846745153E-004  3.884741848523198E-004  5.575394629928423E-004
                                 ⋮
O2
 0.233000000000000       0.224754313587326       0.216508626848526
 0.208262938613688       0.200017246870485       0.191771543120061
 0.183525817195215       0.175280033855829       0.167034166584222
 0.158788121161046       0.150541780465921       0.142294853615556
 0.134047082840520       0.127447653884125       0.120846788679216
 0.114243989786987       0.107637900827792       0.101030612932794
 9.441692407668151E-002  8.780100013979590E-002  8.118054334241118E-002
 7.455856389885569E-002  6.793579480660268E-002  6.132186250985606E-002
 5.472477529703249E-002  4.976905474115009E-002  4.484596939632648E-002
 3.996580795445329E-002  3.515712280589375E-002  3.043532021330975E-002
 2.585889376376704E-002  2.144854810691688E-002  1.731681682997562E-002
```

Flamelet library

SLFMFoam

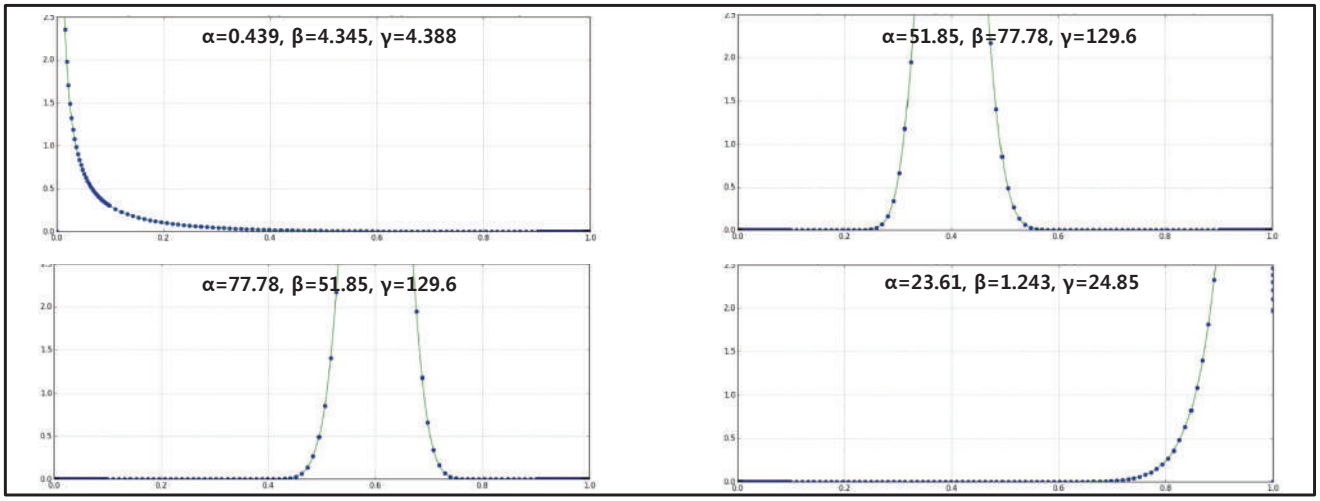
Solver

BetaPDF.H

```
private:
const scalar eta;
const scalar var_eta;
scalar gammma;
scalar alpha;
scalar beta;
scalar lndenum;
singularity s;

template <typename F, typename G>
inline scalar integrate_f_g (scalar low, scalar upper, label nInterval, F f, G g, bool debug=false)

template <typename G>
scalar beta_integration (scalar low, scalar upper, G g, Beta beta, bool debug=false)
```

```
gammma = (eta*(1.0-eta) / var_eta - 1.0);
alpha = (eta * gammma);
beta = ((1.0-eta) * gammma);
```

$$\alpha=\tilde{\zeta}\gamma,\ \ \beta=(1-\tilde{\zeta})\gamma,\ \ \gamma=\frac{\tilde{\zeta}(1-\tilde{\zeta})}{\zeta''^2}$$



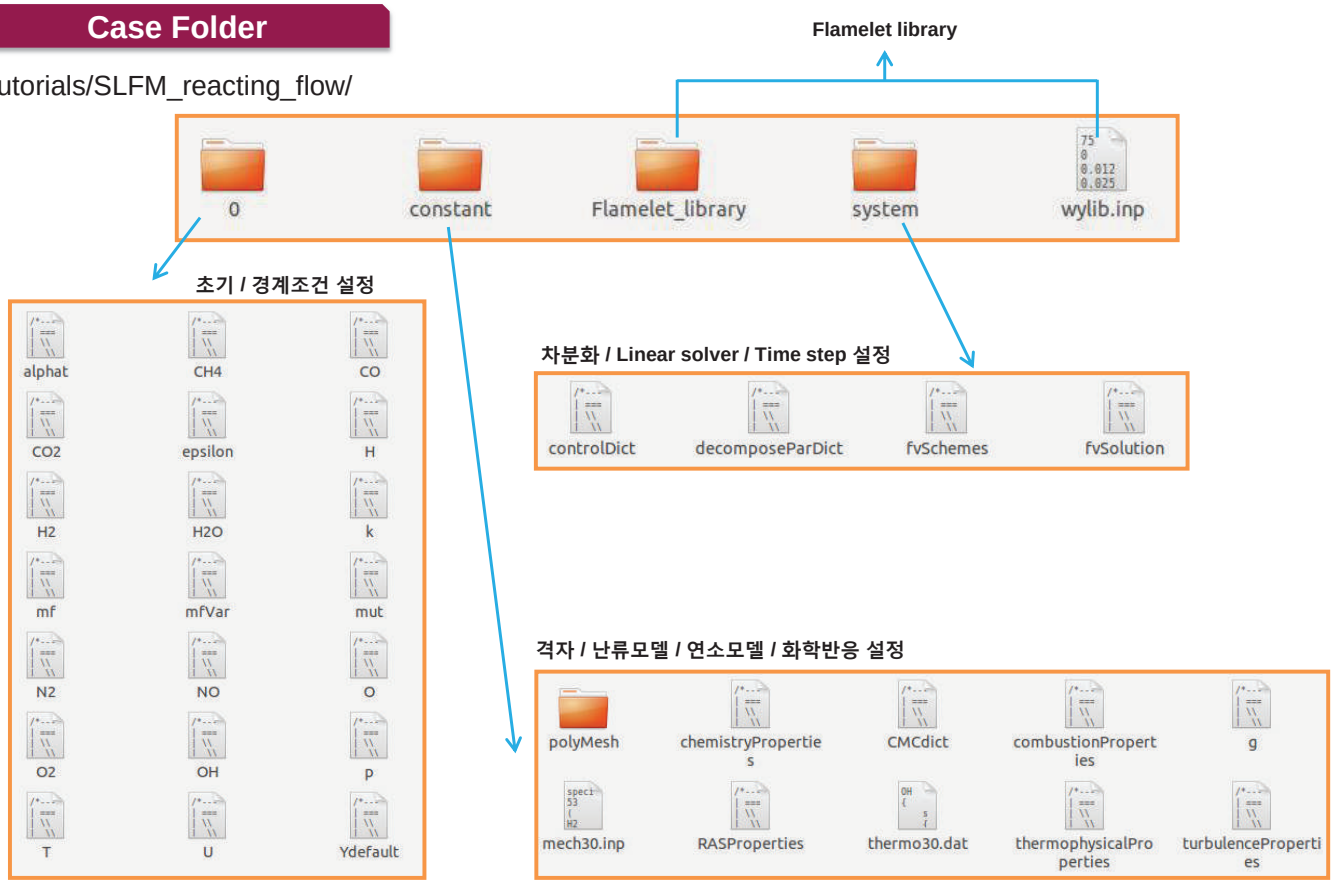
β-PDF plot

Combustion Laboratory POHANG UNIVERSITY OF SCIENCE AND TECHNOLOGY POSTECH

SLFMFoam

Case Folder

/tutorials/SLFM_reacting_flow/



SLFMFoam

'0' Folder

- Mixture fraction, Mixture fraction variance

```
FoamFile
{
    version     2.0;
    format      ascii;
    class       volScalarField;
    location    "0";
    object      mf;
}
// * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * //

dimensions      [0 0 0 0 0 0 0];

internalField   uniform 0.0;

boundaryField
{
    MAINJET
    {
        type            fixedValue;
        value           uniform 1.0;
    }
    PILOT
    {
        type            fixedValue;
        value           uniform 0.27;
    }
    COFLOW
    {
        type            fixedValue;
        value           uniform 0.0;
    }
    OUTLET
    {
        type            zeroGradient;
    }
    CASING
    {
        type            slip;
    }
    INNERWALL
    {
        type            zeroGradient;
    }
    OUTERWALL
    {
        type            zeroGradient;
    }
}
```

mf

```
FoamFile
{
    version     2.0;
    format      ascii;
    class       volScalarField;
    location    "0";
    object      mfVar;
}
// * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * //

dimensions      [0 0 0 0 0 0 0];

internalField   uniform 0.0;

boundaryField
{
    MAINJET
    {
        type            fixedValue;
        value           uniform 0.0;
    }
    PILOT
    {
        type            fixedValue;
        value           uniform 0.0;
    }
    COFLOW
    {
        type            fixedValue;
        value           uniform 0.0;
    }
    OUTLET
    {
        type            zeroGradient;
    }
    CASING
    {
        type            slip;
    }
    INNERWALL
    {
        type            zeroGradient;
    }
    OUTERWALL
    {
        type            zeroGradient;
    }
}
```

mfVar

SLFMFoam

‘constant’ Folder

- Chemical Reactions

thermophysicalProperties

```
thermoType
{
    type            heRhoThermo;
    mixture         reactingMixture;
    transport       sutherland;
    thermo          janaf;
    energy          sensibleEnthalpy;
    equationOfState perfectGas;
    specie          specie;
}

inertSpecie N2;

chemistryReader foamChemistryReader;

foamChemistryFile "$FOAM_CASE/constant/mech30.inp";

foamChemistryThermoFile "$FOAM_CASE/constant/thermo30.dat";
```

mech30.dat

```
un-named-reaction-12
{
    type            reversibleArrheniusReaction;
    reaction        "O + HCO = OH + CO";
    A               3e+10;
    beta            0;
    Ta              0;
}
un-named-reaction-13
{
    type            reversibleArrheniusReaction;
    reaction        "O + HCO = H + CO2";
    A               3e+10;
    beta            0;
    Ta              0;
}
un-named-reaction-14
{
    type            reversibleArrheniusReaction;
    reaction        "O + CH2O = OH + HCO";
    A               3.9e+10;
    beta            0;
    Ta              1781.3;
}
un-named-reaction-15
{
    type            reversibleArrheniusReaction;
    reaction        "O + CH2OH = OH + CH2O";
    A               1e+10;
    beta            0;
    Ta              0;
}
un-named-reaction-16
{
    type            reversibleArrheniusReaction;
    reaction        "O + CH3O = OH + CH2O";
    A               1e+10;
    beta            0;
    Ta              0;
}
```

- GRI-3.0 mechanism
  - 53 species
  - 325 steps

thermo30.dat

```
OH
{
    specie
    {
        nMoles          1;
        molWeight       17.0074;
    }
    thermodynamics
    {
        Tlow            200;
        Thigh           3500;
        Tcommon         1000;
        highCpCoeffs    ( 3.09289 0.00054843 1.26505e-07 -8.79462e-11 1.17412e-
        lowCpCoeffs     ( 3.99202 -0.00240132 4.61794e-06 -3.88113e-09 1.36411e
    }
    transport
    {
        As              1.67212e-06;
        Ts              170.672;
    }
}
```


SLFMFoam

‘constant’ Folder

- Mixture fraction space

CMCdict

```
uni_eta false; ①

section 4;  ②

etamax 90;  ③

nzvar 60;   ④
zvar_max 0.1;  ⑤
zvar_spacing 100; ⑥

non_uni_eta_dict  ⑦
{
        MFcut (0.0 0.1 0.25 0.5 1.0);
        Neta (0 20 40 70 90);
}

makeSLFMlib false; //true;  ⑧

SLFM true; //false;         ⑨

SLFM_dict
{
        SLFMlib_FileName wylib.inp;
        Maximum_Nst 300.1;
        Minimum_Nst 0.1;
        delta_Nst 3;
        eta_stoi 0.351;
}
```

① Uniform eta(mixture fraction) space 설정
- false : non uniform eta dict(⑦) 사용

② eta space section 수

③ eta spacing
- 90 : mixture fraction을 0부터 1까지 총 90개 구간으로 구분

④ mixture fraction variance 수

⑤ maximum mixture fraction variance 설정

⑥ spacing coefficient

⑦ non uniform eta spacing
- 0~0.1, 0.1~0.25, 0.25~0.5, 0.5~1.0 4 section(②) 에 각각 20, 20, 30, 20 개의 eta space

⑧ wylib.inp 파일 생성 여부

⑨ SLFM 연소모델 사용 여부

SLFMFoam

‘0’ Folder

- Discretization / Linear Solver / Relaxation Factor

```
ddtSchemes
{
    default         steadyState;
}

gradSchemes
{
    default         cellLimited Gauss linear 1;
}

divSchemes
{
    default         none;

    div(phi,U)      bounded Gauss upwind;
    div(phi,Yi_h)   Gauss upwind;
    div(phi,mf)   bounded Gauss upwind;
    div(phi,mfVar)   bounded Gauss upwind;
    div(phi,K)      bounded Gauss upwind;
    div(phid,p)     bounded Gauss upwind; ;
    div(phi,epsilon) bounded Gauss upwind;
    div(phi,k) bounded Gauss upwind;
    div((muEff*dev2(T(grad(U))))) Gauss linear;
}

laplacianSchemes
{
    default         Gauss linear uncorrected;
}
```

fvSchems

```
p
{
    solver          GAMG;
    tolerance       0;
    relTol          0.1;
    smoother        DICGaussSeidel;
    nPreSweeps      0;
    nPostSweeps     2;
    cacheAgglomeration true;
    nCellsInCoarsestLevel 40;
    agglomerator    faceAreaPair;
    mergeLevels     1;
};

"(U|Yi|h|k|epsilon|mf|mfVar)"
{
    solver          PBiCG;
    preconditioner  DILU;
    tolerance       1e-08;
    relTol          0.0;
}
relaxationFactors
{
    fields
    {
        p               0.15;
        rho             1.0;
    }
    equations
    {
        U               0.3;
        mf              0.3;
        mfVar           0.15;
        h               0.3;
        ".*"            0.3;
    }
}
```

fvSolution

SLFMFoam

‘0’ Folder

- Calculation / MPI(Message Passing Interface)

```
application     SLFMFoam;

startFoam       startTime; //latestTime;

startTime       0;

stopAt          endTime;

endTime         2000;

deltaT          1;

writeControl    timeStep;

writeInterval   100;

purgeWrite      0;

writeFormat     ascii;

writePrecision  10;

writeCompression uncompressed;

timeFormat      general;

timePrecision   6;

runTimeModifiable yes;
```

controlDict

```
numberOfSubdomains 4;

method          simple;

simpleCoeffs
{
    n               ( 1 2 2 );
    delta           0.001;
}

hierarchicalCoeffs
{
    n               ( 1 1 1 );
    delta           0.001;
    order           xyz;
}

scotchCoeffs
{

}

manualCoeffs
{
    dataFile        "";
}

distributed     no;

roots           ( );
```

decomposeParDict

Combustion Laboratory POHANG UNIVERSITY OF SCIENCE AND TECHNOLOGY POSTECH

SLFMFoam

Tutorial

1. **Open 'Terminal' : click** 

2. **~$ cd tutorials/SLFM_reacting_flow**

```
:~$ cd tutorials/SLFM_reacting_flow
```

3. **~$ decomposePar**

```
:~/tutorials/SLFM_reacting_flow$ decomposePar
```

```
Time = 0

Processor 0: field transfer
Processor 1: field transfer
Processor 2: field transfer
Processor 3: field transfer

End.
```

4. **~$ mpirun –np 4 SLFMFoam -parallel**

```
:~/tutorials/SLFM_reacting_flow$ mpirun -np 4 SLFMFoam -parallel
```


SLFMFoam

Tutorial

- Calculating

①

```
Selecting chemistryReader foamChemistryReader
chemistryModel: Number of species = 53 and reactions = 325
    using integrated reaction rate
Creating component thermo properties:
    multi-component carrier - 53 species
    no liquid components
    no solid components

Reading field U

Reading/calculating face flux field phi

Creating turbulence model

Selecting turbulence model type RASModel
Selecting RAS turbulence model kEpsilon
kEpsilonCoeffs
{
    Cmu             0.09;
    C1              1.44;
    C2              1.92;
    C3              -0.33;
    sigmak          1;
    sigmaEps        1.3;
    Prt             1;
}
```

②

```
Reading field mf

Reading field mfVar

Set eta space grid      false
```

③

```
mfvarloop 7     0.002324925777      18.91756731     59.09301954
mfvarloop 8     0.002698531483      16.26489843     50.80684767
mfvarloop 9     0.003083462307      14.20416091     44.36969851
mfvarloop 10    0.003480061546      12.557772       39.22685481
mfvarloop 11    0.003888682905      11.21272416     35.02531362
mfvarloop 12    0.004309690811      10.0936787      31.52973863
mfvarloop 13    0.004743460738      9.148477777     28.57720378
mfvarloop 14    0.00519037954       8.339865515     26.05133248
mfvarloop 15    0.005650845801      7.640520692     23.86678113
mfvarloop 16    0.006125270184      7.029952159     21.95954128
mfvarloop 17    0.006614075802      6.492489998     20.28066463
mfvarloop 18    0.007117698594      6.015946292     18.79207965
mfvarloop 19    0.007636587713      5.590698812     17.46372928
mfvarloop 20    0.008171205928      5.209049596     16.27156729
mfvarloop 21    0.008722030035      4.864766794     15.1961272
mfvarloop 22    0.009289551284      4.552751462     14.22148137
mfvarloop 23    0.009874275814      4.26879128      13.33447173
mfvarloop 24    0.01047672511       4.009375839     12.52413278
mfvarloop 25    0.01109743646       3.771556266     11.78125308
mfvarloop 26    0.01173696345       3.552837232     11.09803795
mfvarloop 27    0.01239587643       3.351092967     10.4678471
mfvarloop 28    0.01307476306       3.164501273     9.884988511
mfvarloop 29    0.01377422878       2.991491178     9.344554918
mfvarloop 30    0.01449489743       2.830701076     8.842293053
mfvarloop 31    0.01523741171       2.680944953     8.374498153
mfvarloop 32    0.01600243385       2.541184956     7.937928266
mfvarloop 33    0.01679064612       2.410508947     7.529734134
mfvarloop 34    0.01760275148       2.288112015     7.147401448
mfvarloop 35    0.0184394742        2.17328116      6.788703005
mfvarloop 36    0.01930156052       2.065382533     6.45165884
mfvarloop 37    0.02018977927       1.963850737     6.134502818
mfvarloop 38    0.02110482262       1.868179825     5.835654505
```

SLFMFoam

Tutorial

- Calculating

Time step

β-PDF, Flamelet library integration

```
Integration Done!

Starting time loop

Time = 1

DILUPBiCG:  Solving for Ux, Initial residual = 0.00229937001, Final residual = 8.012418588e-09, No Iterations 4
DILUPBiCG:  Solving for Uy, Initial residual = 9.164055872e-05, Final residual = 8.660325996e-11, No Iterations 4
DILUPBiCG:  Solving for Uz, Initial residual = 0.002301325921, Final residual = 6.77154281e-09, No Iterations 4
DILUPBiCG:  Solving for mf, Initial residual = 1, Final residual = 6.268664377e-09, No Iterations 6
DILUPBiCG:  Solving for mfVar, Initial residual = 1, Final residual = 1.197297392e-10, No Iterations 5
T gas min/max   = 291, 1880
GAMG:  Solving for p, Initial residual = 0.07389476442, Final residual = 0.003338744395, No Iterations 1
time step continuity errors : sum local = 0.03095055174, global = -0.03089194321, cumulative = -0.03089194321
rho min/max = 0.1786158296, 1.199758525
p min/max = 100529.164, 100716.1384
DILUPBiCG:  Solving for epsilon, Initial residual = 0.0006856716338, Final residual = 7.362426751e-09, No Iterations 4
DILUPBiCG:  Solving for k, Initial residual = 0.0003361699725, Final residual = 8.998211338e-09, No Iterations 3
ExecutionTime = 44.37 s  ClockTime = 45 s
```

④

Time

Residual

SLFMFoam

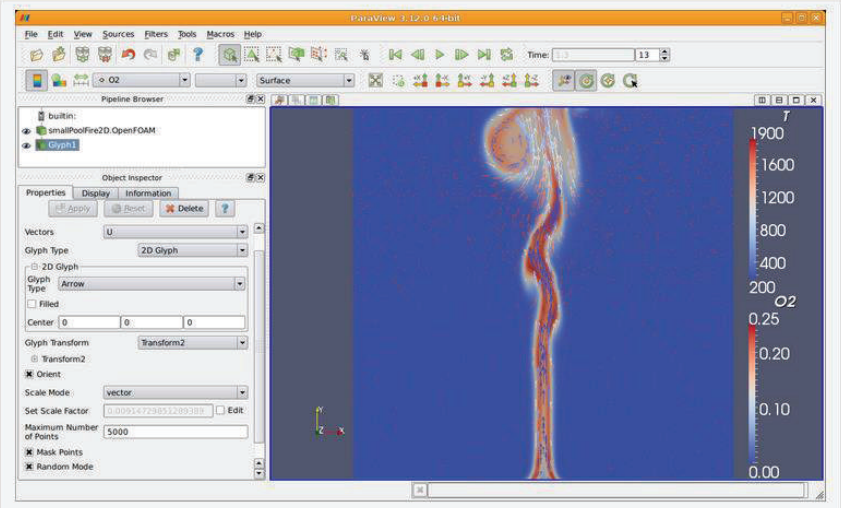
Post Processing

1. **~$ reconstructPar**

```
:~/tutorials/SLFM_reacting_flow$ reconstructPar
```

2. **~$ cd tutorials/EDM_reacting_flow/paraFoam**

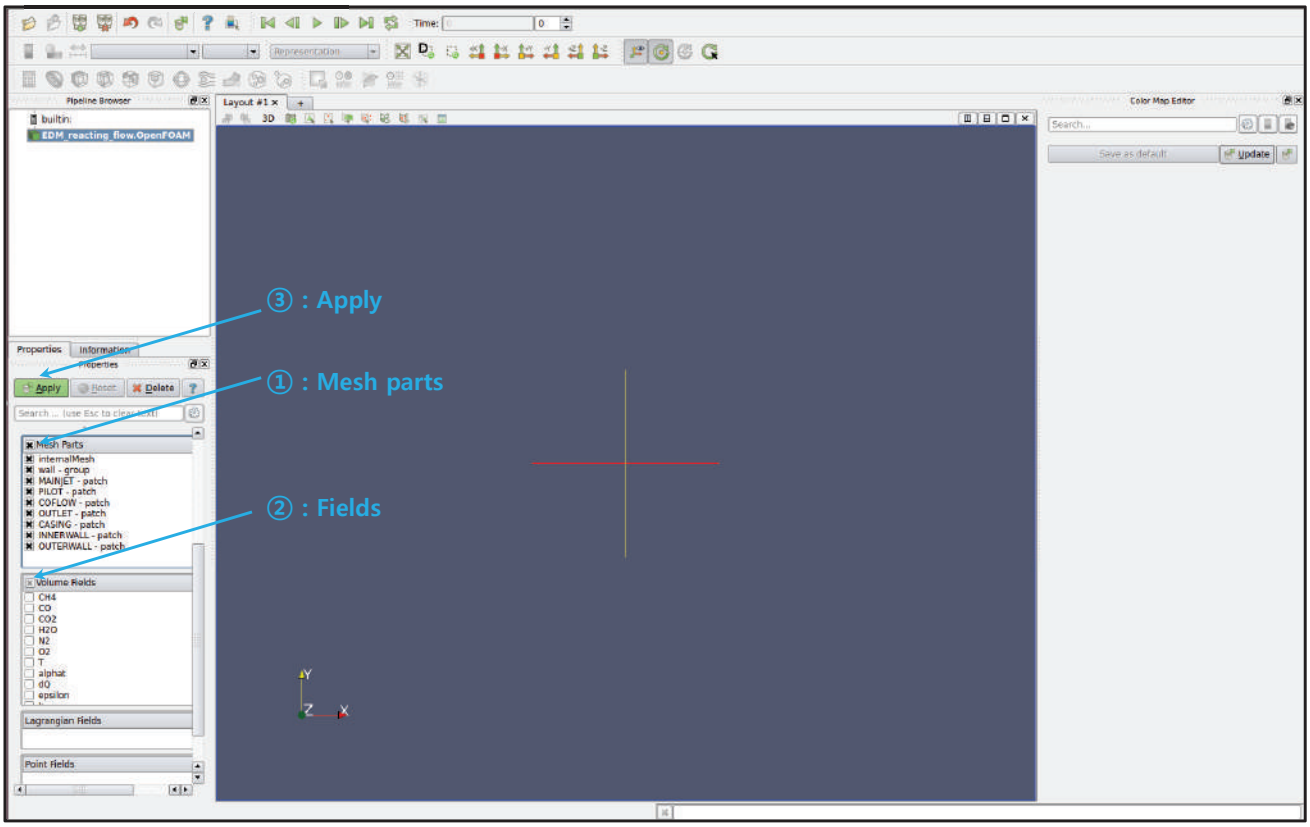
```
:~/tutorials/EDM_reacting_flow$ paraFoam
```



OpenFOAM smallPoolFire example visualized with ParaView

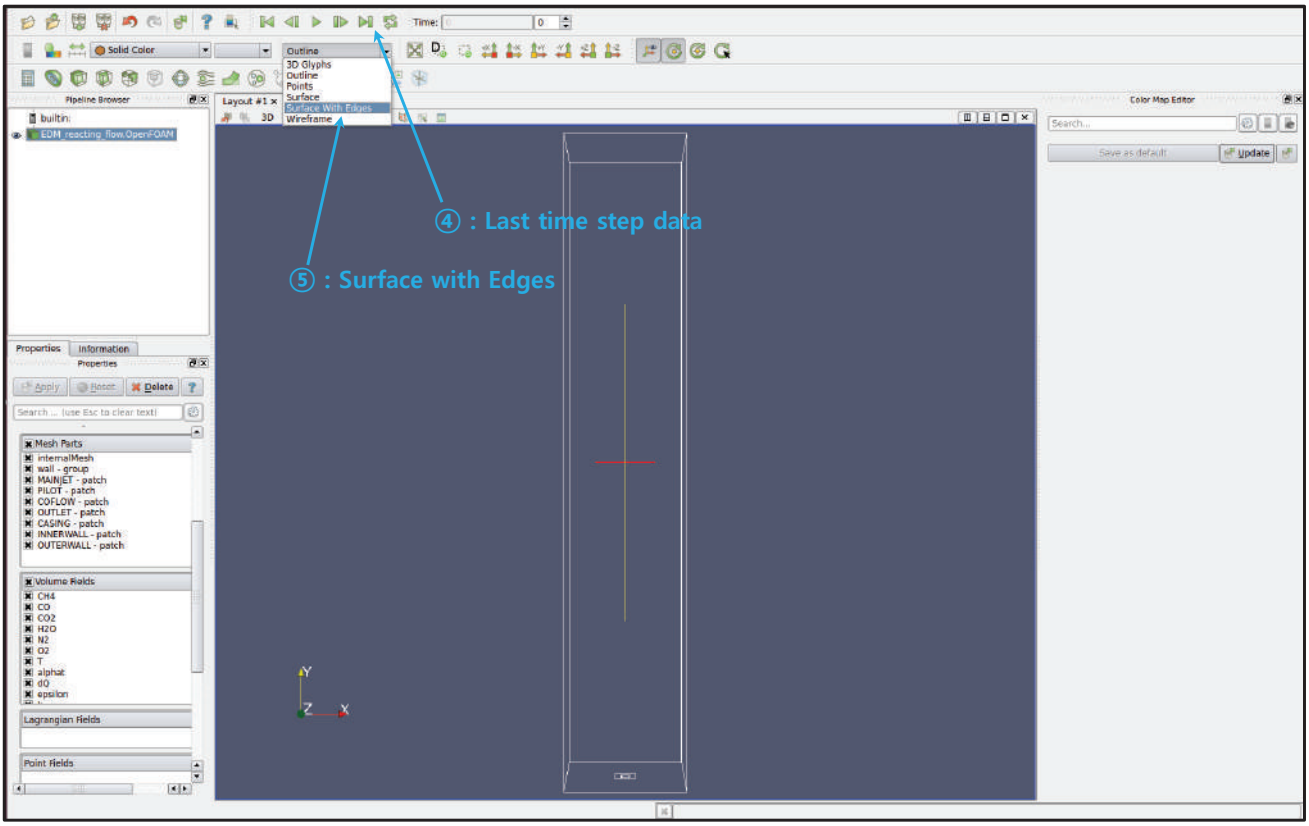
SLFMFoam

Post Processing



SLFMFoam

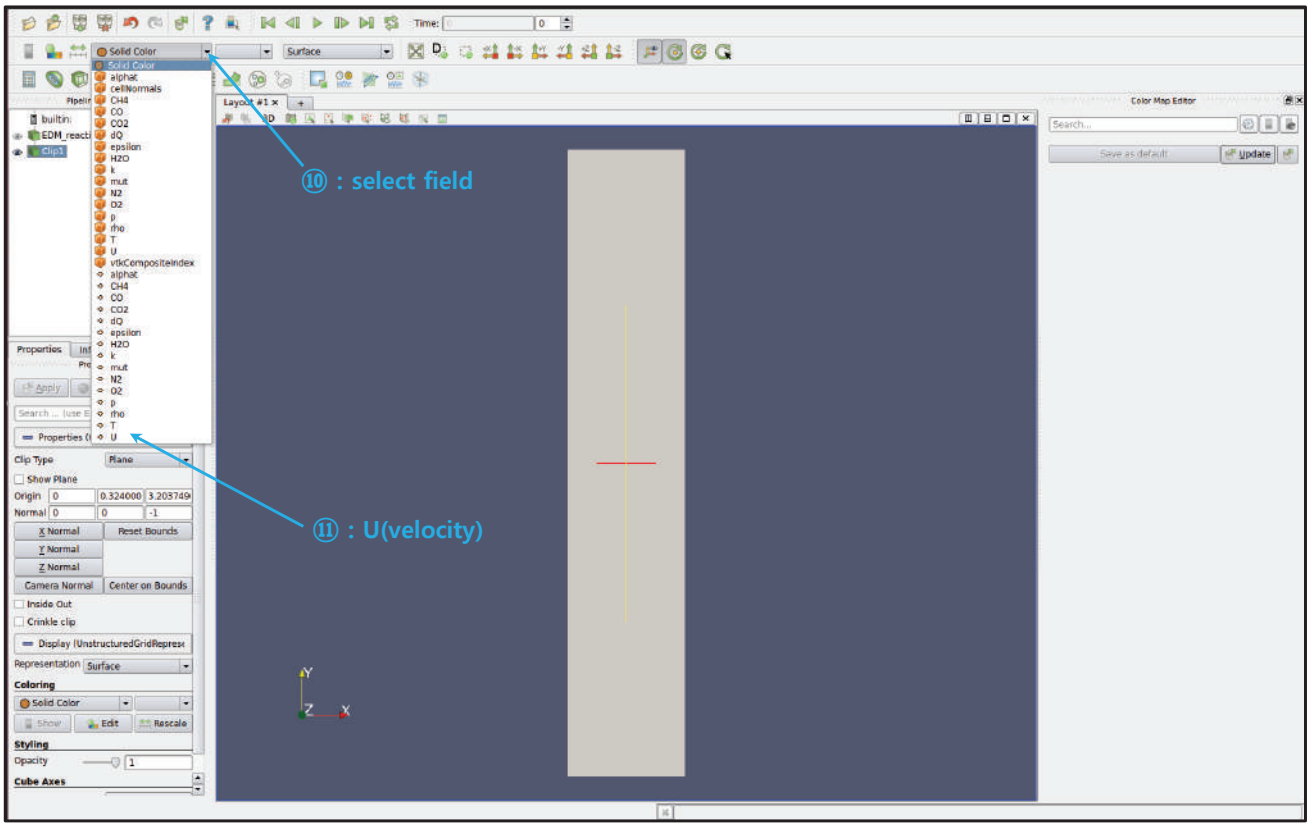
Post Processing



Combustion Laboratory
POHANG UNIVERSITY OF SCIENCE AND TECHNOLOGY
POSTECH

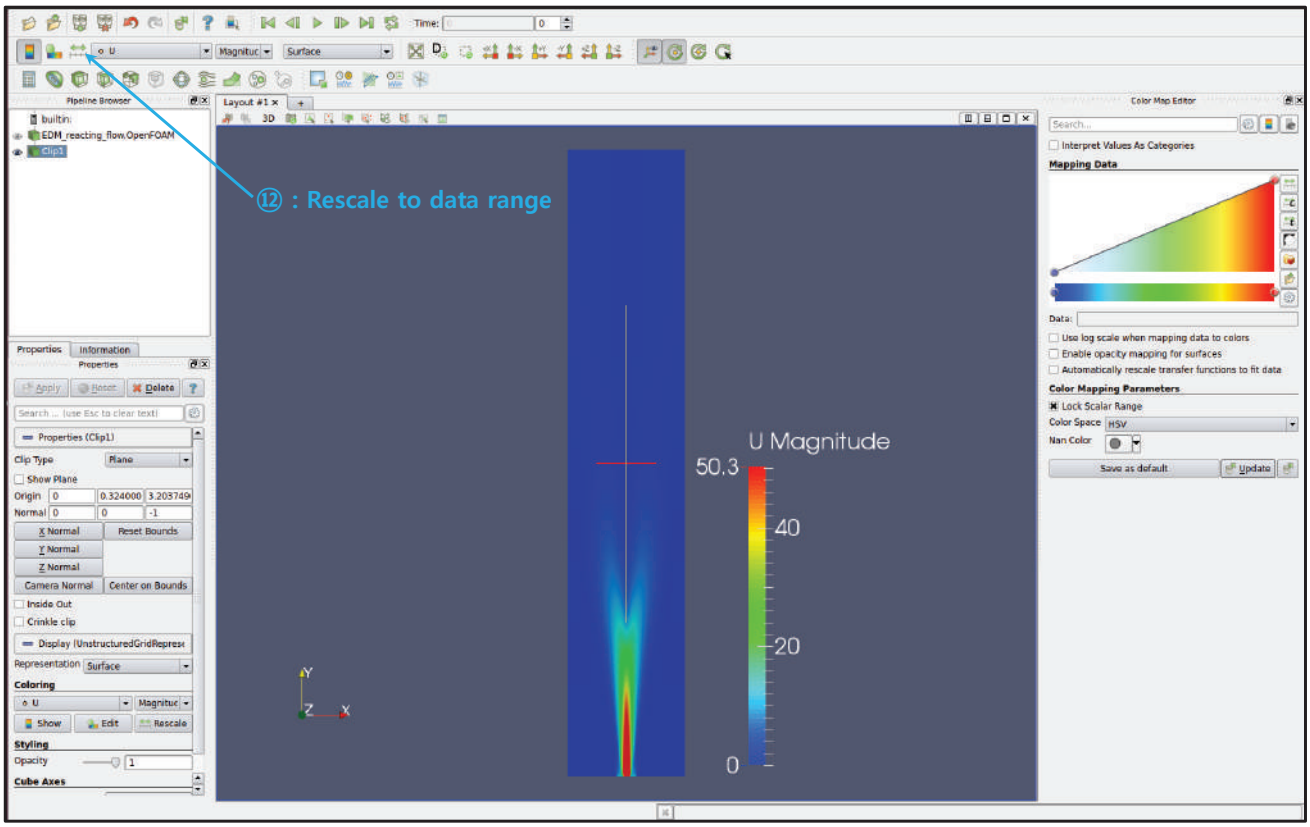
SLFMFoam

Post Processing



SLFMFoam

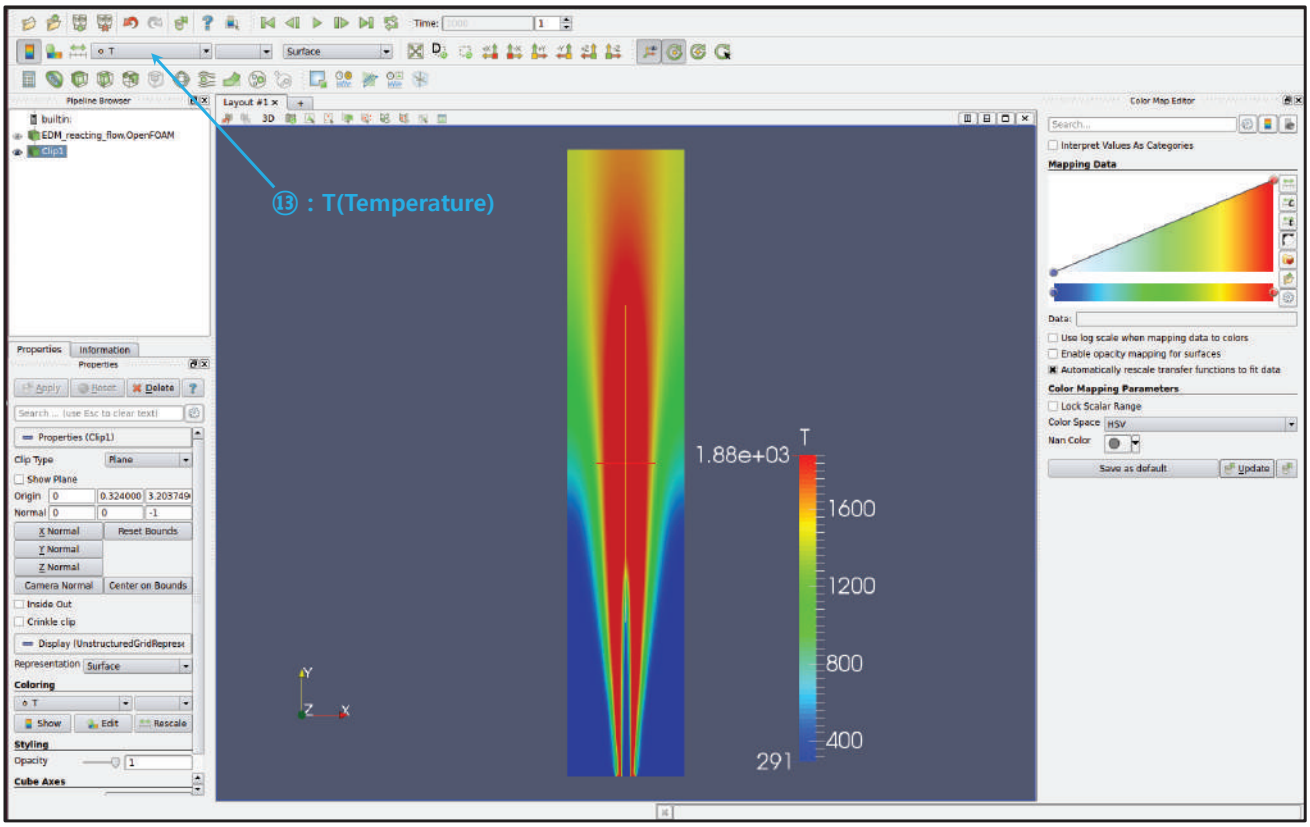
Post Processing



Combustion Laboratory
POHANG UNIVERSITY OF SCIENCE AND TECHNOLOGY
POSTECH

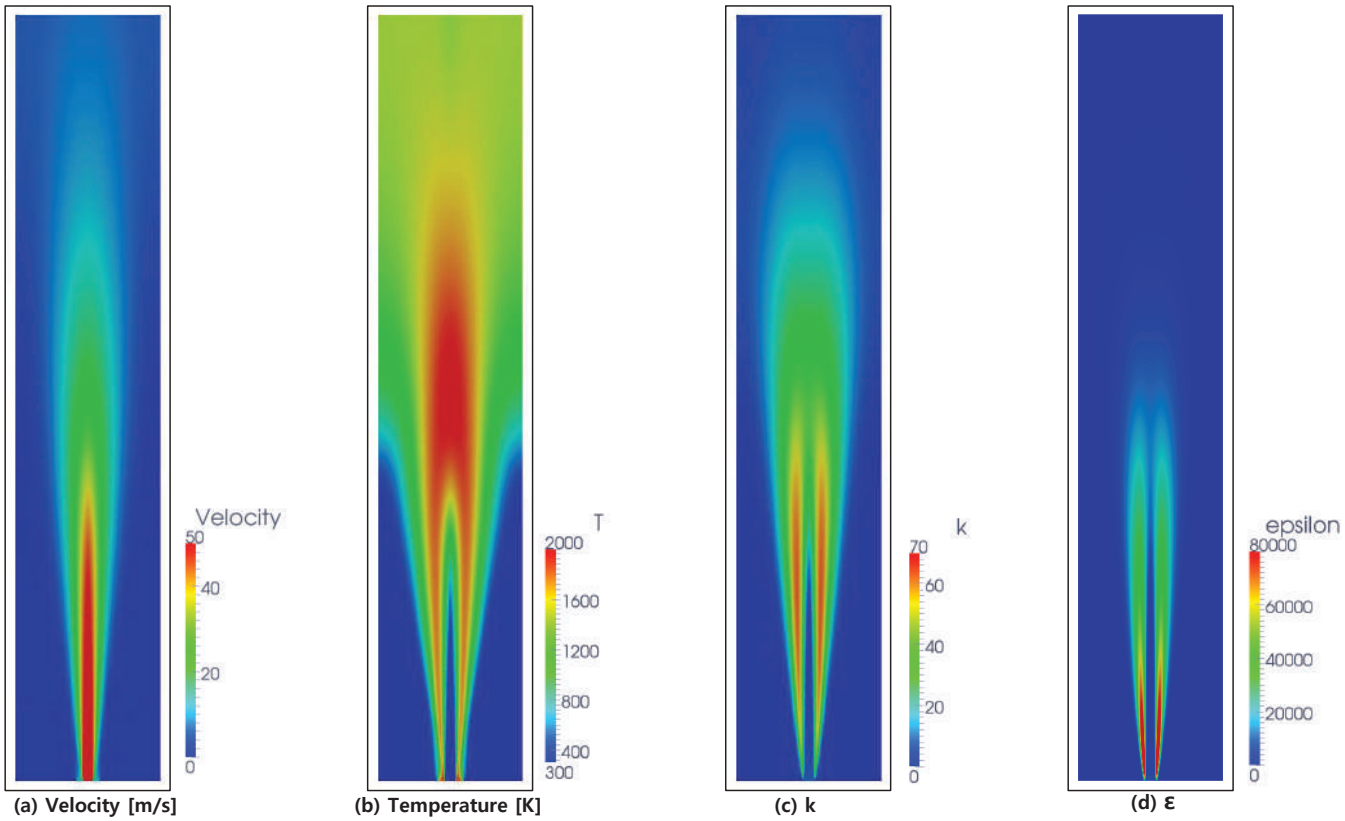
SLFMFoam

Post Processing



SLFMFoam

Results

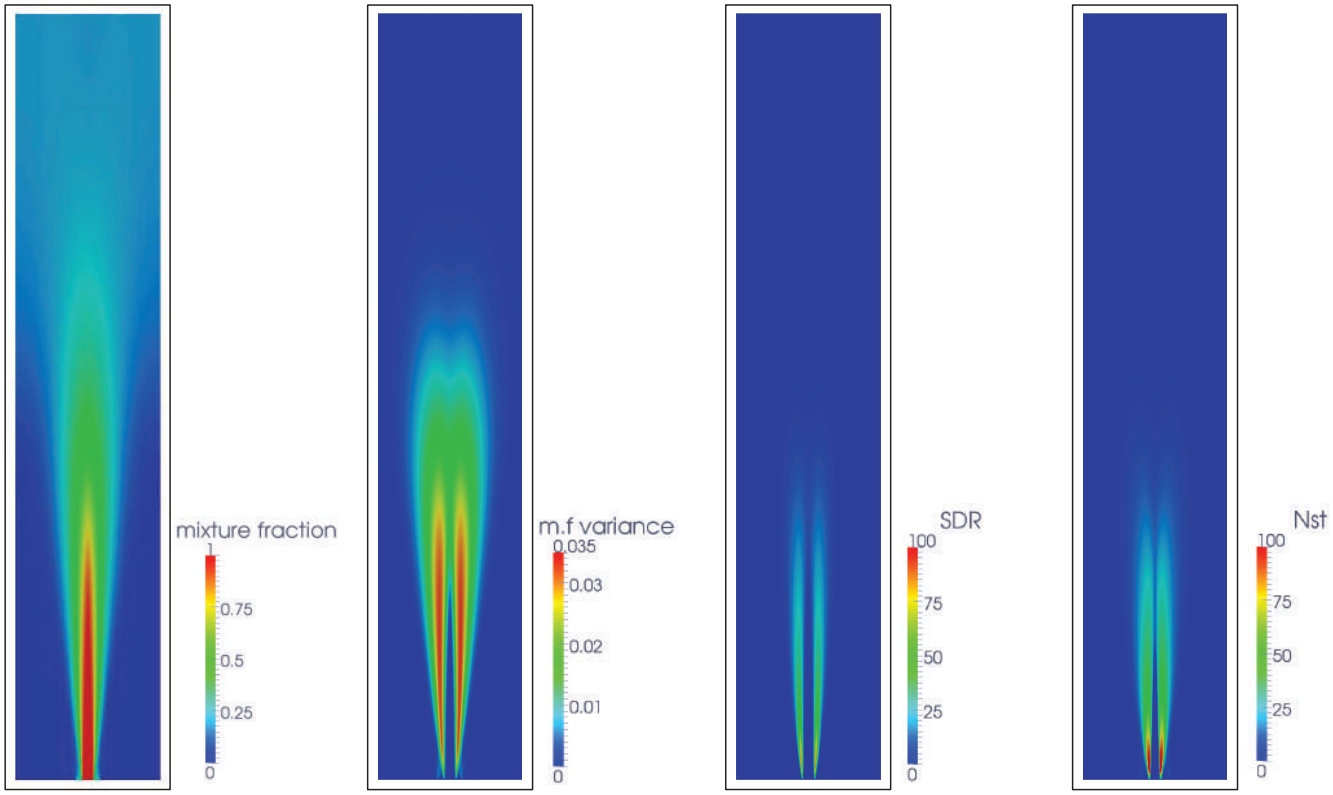


(a) Velocity [m/s] (b) Temperature [K] (c) k (d) ε

Combustion Laboratory
POHANG UNIVERSITY OF SCIENCE AND TECHNOLOGY
POSTECH

SLFMFoam

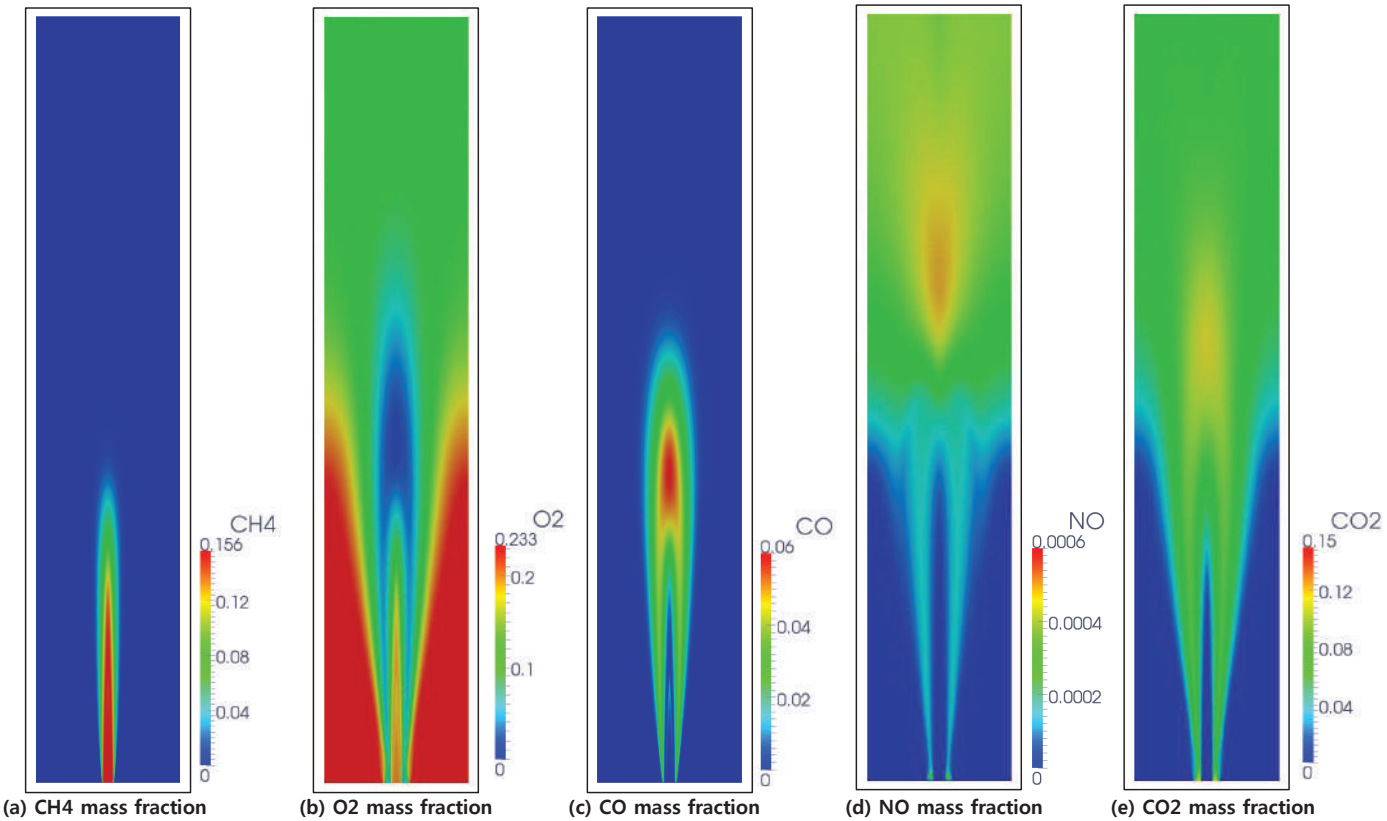
Results



(a) Mixture fraction (b) Mixture fraction variance (c) Scalar dissipation rate (d) Stoichiometric SDR

SLFMFoam

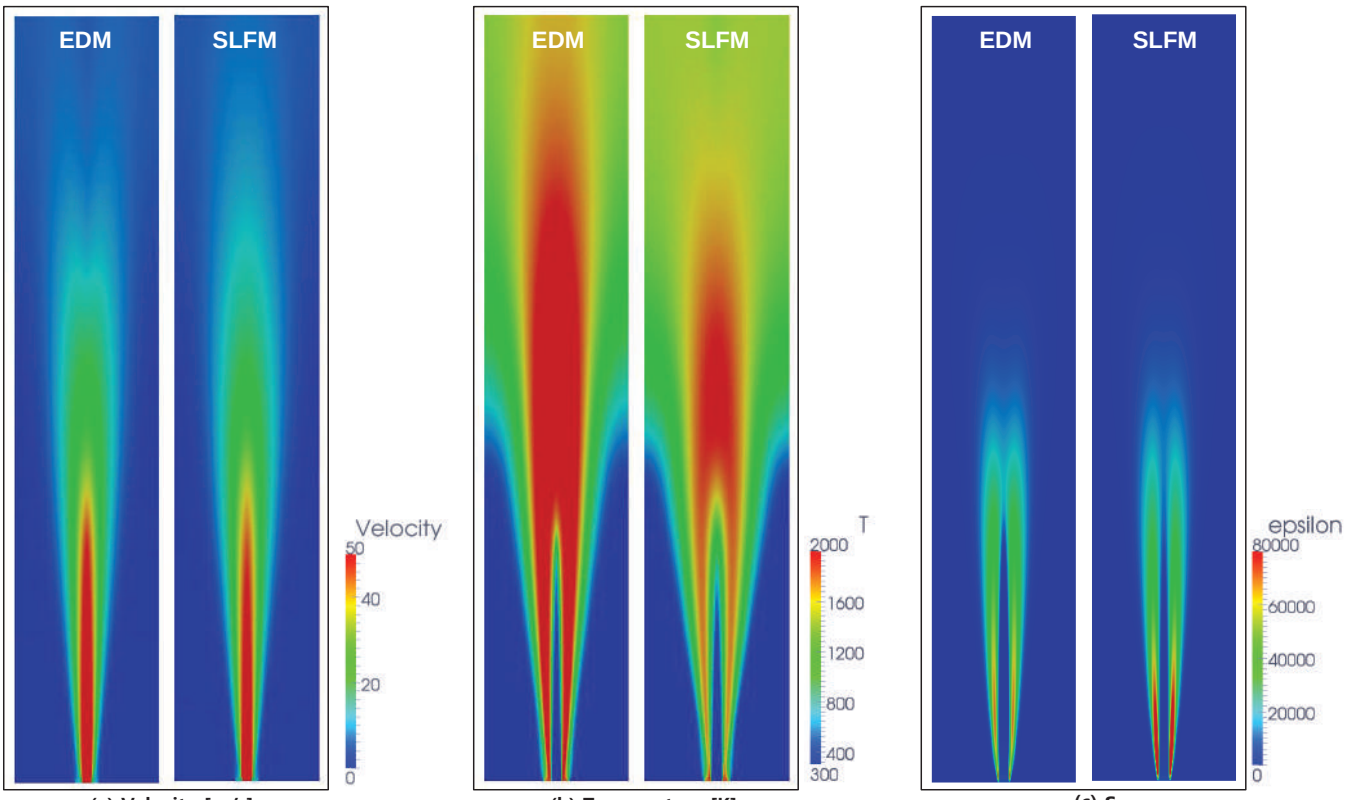
Results



(a) CH4 mass fraction (b) O2 mass fraction (c) CO mass fraction (d) NO mass fraction (e) CO2 mass fraction

SLFMFoam

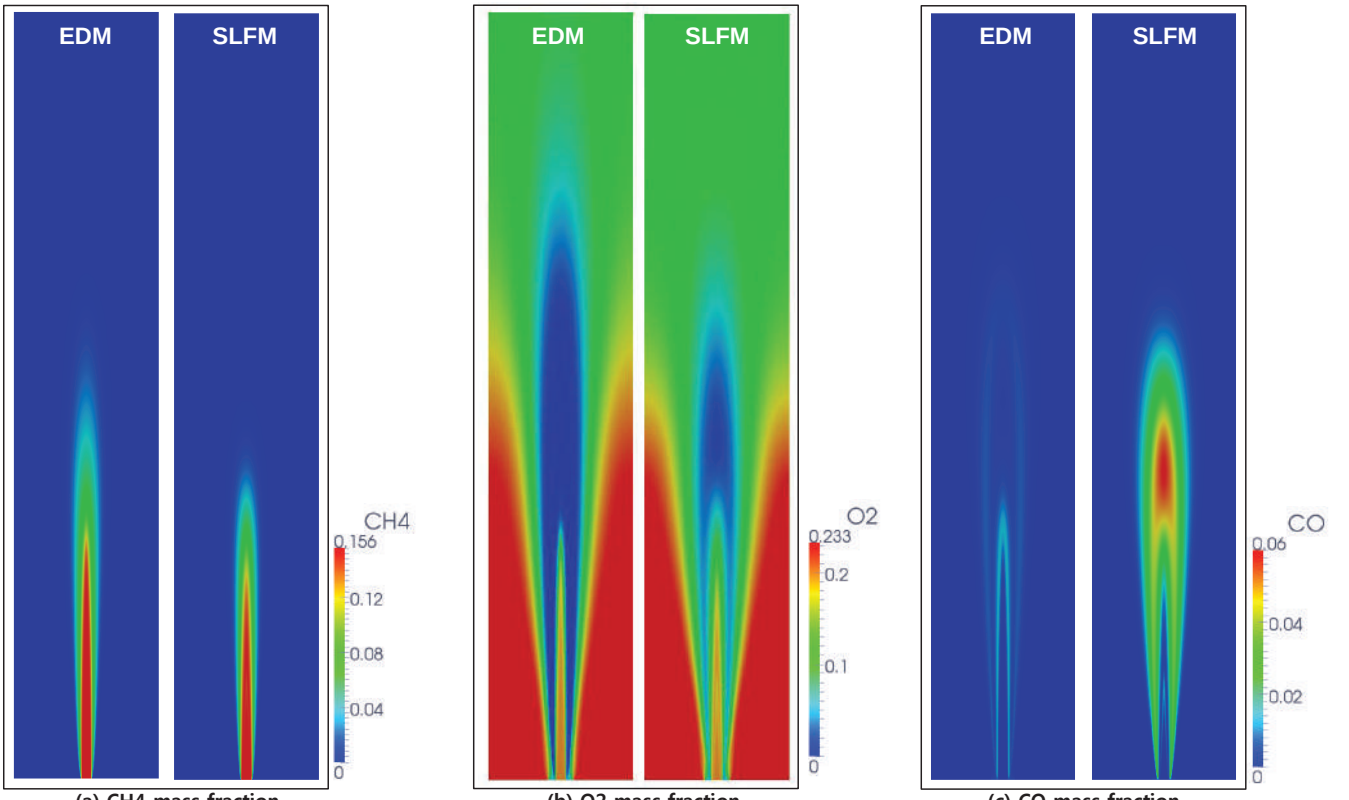
Comparison



(a) Velocity [m/s] (b) Temperature [K] (c) ε

SLFMFoam

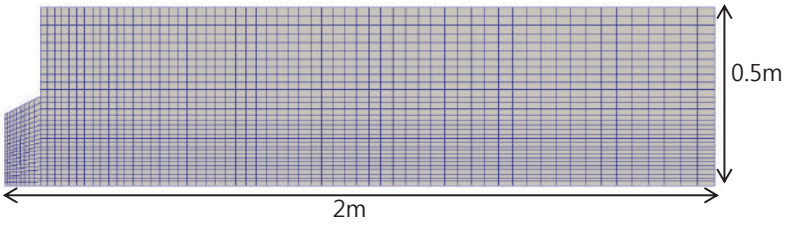
Comparison

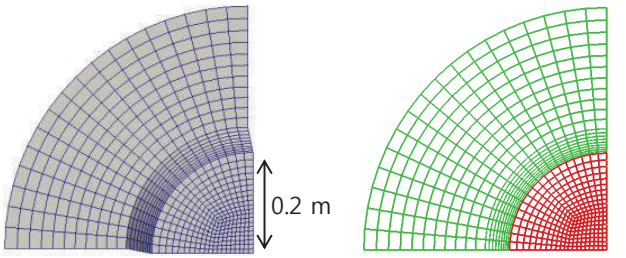


(a) CH4 mass fraction (b) O2 mass fraction (c) CO mass fraction

Case description

Computational grid





Swirl Flow Inlet

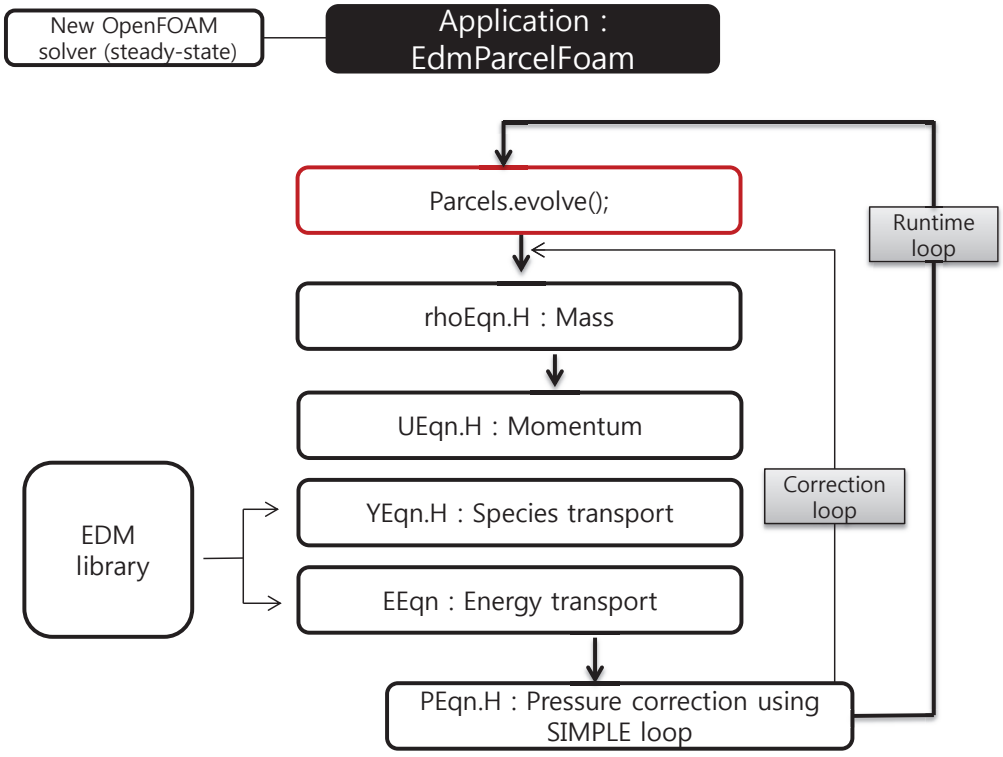
- ¼ quarter mesh
- Periodic boundary condition

- **CheckMesh**

```
Mesh stats
    points:           33861
    faces:            95060
    internal faces:   88864
    cells:            30654
    faces per cell:   6
    boundary patches: 6
    point zones:      0
    face zones:       1
    cell zones:       1

Overall number of cells of each type:
    hexahedra:     30654
    prisms:        0
    wedges:        0
    pyramids:      0
    tet wedges:    0
    tetrahedra:    0
    polyhedra:     0

Checking patch topology for multiply conne
    Patch              Faces     Points
    INLET              279       309
    WALL               360       399
    SLIPWALL           1062      1140
    OUTLET             477       518
    CYCLIC_half0       2009      2109
    CYCLIC_half1       2009      2109
```


EdmParcelFoam

Code Structure

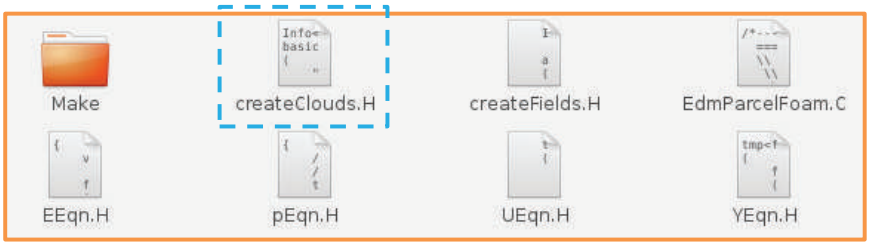


EdmParcelFoam

EdmParcelFoam

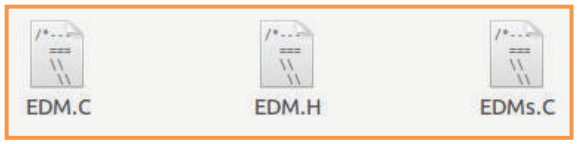
Code Structure

/solvers/EdmParcelFoam



EdmParcelFoam

/libs/combustionModels_POSTECH/EDM



EDM library

EdmParcelFoam

Solver

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

while (simple.loop())
{
    Info<< "Time = " << runTime.timeName() << nl << endl;

    parcels.evolve();

    // --- Pressure-velocity SIMPLE corrector loop
    {
        #include "UEqn.H"
        #include "YEqn.H"
        #include "EEqn.H"
        #include "pEqn.H"
    }

    turbulence->correct();

    runTime.write();

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

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

EdmParcelFoam.C

UEqn.H

```
tmp<fvVectorMatrix> UEqn
(
    fvm::div(phi, U)
  + turbulence->divDevRhoReff(U)
 ==
    rho.dimensionedInternalField()*g
  + parcels.SU(U)
  + fvOptions(rho, U)
);
```

YEqn.H

```
volScalarField& Yi = Y[i];

fvScalarMatrix YEqn
(
    mvConvection->fvmDiv(phi, Yi)
  - fvm::laplacian(turbulence->muEff(), Yi)
 ==
    parcels.SYi(i, Yi)
  + combustion->R(Yi)
 + fvOptions(rho, Yi)
);
```

EEqn.H

```
fvScalarMatrix EEqn
(
    mvConvection->fvmDiv(phi, he)
  + (
        he.name() == "e"
      ? fvc::div(phi, volScalarField("Ekp", 0.5*magSqr(U) + p/rho))
      : fvc::div(phi, volScalarField("K", 0.5*magSqr(U)))
    )
  - fvm::laplacian(turbulence->alphaEff(), he)
 ==
    parcels.Sh(he)
  + radiation->Sh(thermo)
  + combustion->Sh()
 + fvOptions(rho, he)
);
```

EdmParcelFoam

Cloud definition

/solvers/EdmParcelFoam/createCloud.H

```
Info<< "\nConstructing reacting cloud" << endl;
basicReactingMultiphaseCloud parcels
(
    "reactingCloud1",
    rho,
    U,
    g,
    slgThermo
);
```

createCloud.H

```
#ifndef basicReactingMultiphaseCloud_H
#define basicReactingMultiphaseCloud_H

#include "Cloud.H"
#include "KinematicCloud.H"
#include "ThermoCloud.H"
#include "ReactingCloud.H"
#include "ReactingMultiphaseCloud.H"
#include "basicReactingMultiphaseParcel.H"

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

namespace Foam
{
    typedef ReactingMultiphaseCloud
    <
        ReactingCloud
        <
            ThermoCloud
            <
                KinematicCloud
                <
                    Cloud
                    <
                        basicReactingMultiphaseParcel
                    >
                >
            >
        >
    > basicReactingMultiphaseCloud;
}
```

basicReactingMultiphaseCloud.H

- **ReactingMultiphaseCloud**

  Add to reacting cloud
  - multiphase composition
  - devolatilization
  - surface reactions

- **ReactingCloud**

  Add to thermodynamic cloud
  - Variable composition (single phase)
  - Phase change

- **ThermoCloud**

  Add to kinematic cloud
  - Heat transfer

- **KinematicCloud**

  Cloud function objects
  Particle forces
  - buoyancy
  - drag
  - pressure gradient, etc …

  Sub-model
  - Injection model
  - Dispersion model
  - Patch interaction model
  - Surface film model
  - Stochastic collision model

EdmParcelFoam

Code Structure

/solvers/EdmParcelFoam/Make

```
DEV_PATH=../../libs

EXE_INC = \
    -I$(LIB_SRC)/finiteVolume/lnInclude \
    -I${LIB_SRC}/meshTools/lnInclude \
    -I$(LIB_SRC)/turbulenceModels/compressible/turbulenceModel \
    -I$(LIB_SRC)/lagrangian/basic/lnInclude \
    -I$(LIB_SRC)/lagrangian/intermediate/lnInclude \
    -I$(LIB_SRC)/lagrangian/coalCombustion/lnInclude \
    -I$(LIB_SRC)/lagrangian/spray/lnInclude \
    -I$(LIB_SRC)/lagrangian/distributionModels/lnInclude \
    -I$(LIB_SRC)/thermophysicalModels/specie/lnInclude \
    -I$(LIB_SRC)/thermophysicalModels/basic/lnInclude \
    -I$(LIB_SRC)/thermophysicalModels/properties/liquidProperties/lnInclude \
    -I$(LIB_SRC)/thermophysicalModels/properties/liquidMixtureProperties/lnInclude \
    -I$(LIB_SRC)/thermophysicalModels/properties/solidProperties/lnInclude \
    -I$(LIB_SRC)/thermophysicalModels/properties/solidMixtureProperties/lnInclude \
    -I$(LIB_SRC)/thermophysicalModels/thermophysicalFunctions/lnInclude \
    -I$(LIB_SRC)/thermophysicalModels/reactionThermo/lnInclude \
    -I$(LIB_SRC)/thermophysicalModels/SLGThermo/lnInclude \
    -I$(LIB_SRC)/thermophysicalModels/radiationModels/lnInclude \
    -I$(LIB_SRC)/ODE/lnInclude \
    -I$(LIB_SRC)/regionModels/regionModel/lnInclude \
    -I$(LIB_SRC)/regionModels/surfaceFilmModels/lnInclude \
    -I$(LIB_SRC)/fvOptions/lnInclude \
    -I$(LIB_SRC)/sampling/lnInclude \
    -I$(FOAM_SOLVERS)/combustion/reactingFoam \
    -I$(DEV_PATH)/combustionModels_POSTECH/lnInclude \
    -I$(DEV_PATH)/chemistryModel_POSTEH/lnInclude
```

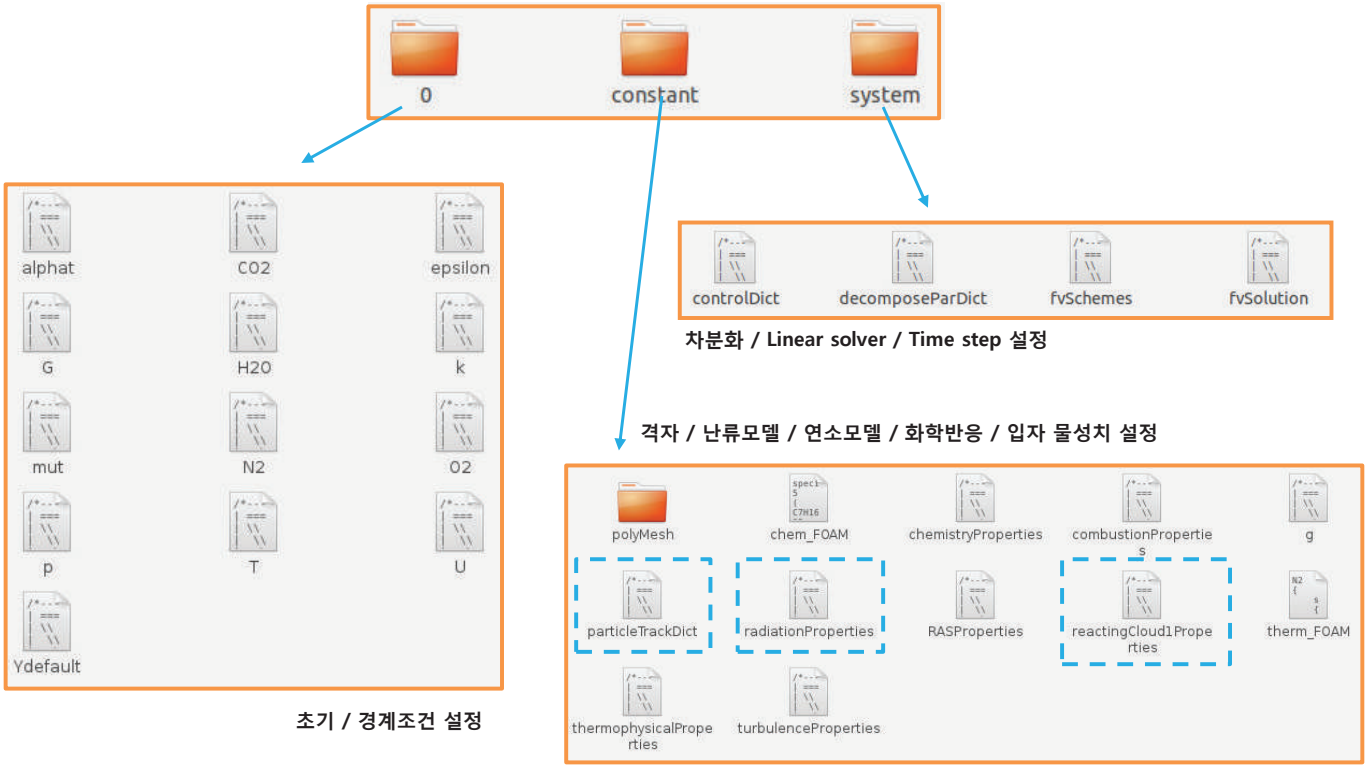
options

```
EXE_LIBS = \
    -L$(FOAM_USER_LIBBIN) \
            -lcombustionModels_POSTECH \
            -lchemistryModel_POSTECH \
    -lfiniteVolume \
    -lmeshTools \
    -lcompressibleTurbulenceModel \
    -lcompressibleRASModels \
    -lcompressibleLESModels \
    -llagrangian \
    -llagrangianIntermediate \
    -llagrangianTurbulence \
    -llagrangianSpray \
    -lspecie \
    -lfluidThermophysicalModels \
    -lliquidProperties \
    -lliquidMixtureProperties \
    -lsolidProperties \
    -lsolidMixtureProperties \
    -lthermophysicalFunctions \
    -lreactionThermophysicalModels \
    -lSLGThermo \
    -lradiationModels \
    -lODE \
    -lregionModels \
    -lsurfaceFilmModels \
    -lfvOptions \
    -lsampling
```

EdmParcelFoam

Case Folder

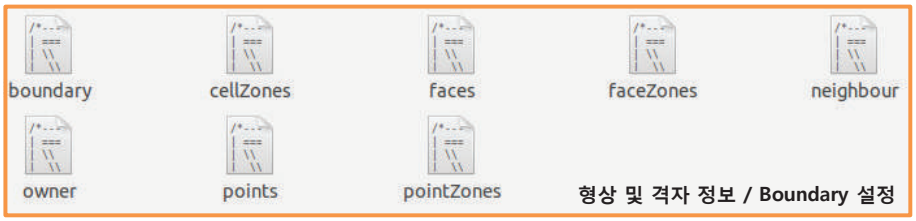
/tutorials/oilSpray-singleBurner/



EdmParcelFoam

PolyMesh

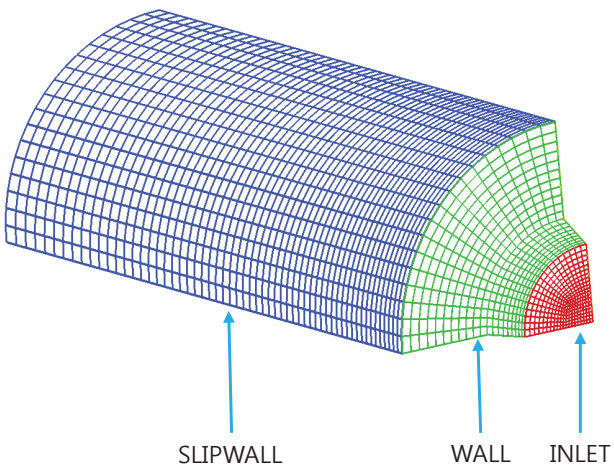
/tutorials/oilSpray-singleBurner/constant/polyMesh




```
INLET
{
    type            patch;
    nFaces          279;
    startFace       88864;
}
WALL
{
    type            wall;
    nFaces          360;
    startFace       89143;
}
SLIPWALL
{
    type            wall;
    nFaces          1062;
    startFace       89503;
}
OUTLET
{
    type            patch;
    nFaces          477;
    startFace       90565;
}
```

```
CYCLIC_half0
{
    type            cyclic;
    inGroups        1(cyclic);
    nFaces          2009;
    startFace       91042;
    matchTolerance  0.0001;
    transform       rotational;
    neighbourPatch  CYCLIC_half1;
    rotationAxis    (1 0 0);
    rotationCentre  (0 0 0);
}
CYCLIC_half1
{
    type            cyclic;
    inGroups        1(cyclic);
    nFaces          2009;
    startFace       93051;
    matchTolerance  0.0001;
    transform       rotational;
    neighbourPatch  CYCLIC_half0;
    rotationAxis    (1 0 0);
    rotationCentre  (0 0 0);
}
```

boundary



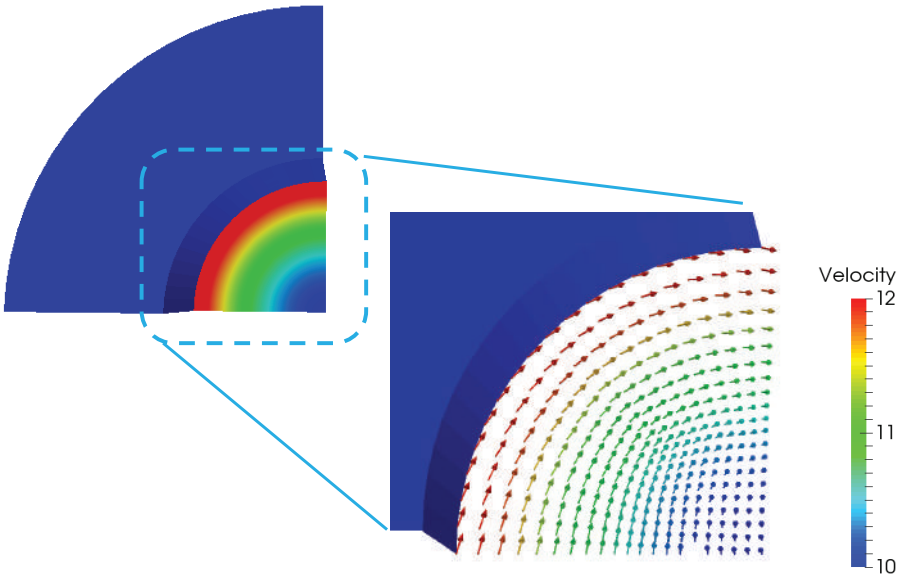
EdmParcelFoam

‘0’ Folder

- U(velocity)

```
dimensions      [0 1 -1 0 0 0 0];

internalField   uniform (10 0 0);

boundaryField
{
    WALL
    {
        type            fixedValue;
        value           uniform (0 0 0);
    }
    SLIPWALL
    {
        type            slip;
    }
    OUTLET
    {
        type            zeroGradient;
    }
    INLET
    {
        type            cylindricalInletVelocity;
        centre          (0 0 0);
        axis            (1 0 0);
        axialVelocity   constant 10;
        radialVelocity  constant 0;
        rpm             constant 350;
        value           uniform (10 0 0);

    }
    CYCLIC_half0
    {
        type            cyclic;
    }
    CYCLIC_half1
    {
        type            cyclic;
    }

}
```

U



EdmParcelFoam

‘constant’ Folder

- Combustion & Turbulence model

```
chemistryType
{
    chemistrySolver   noChemistrySolver;
    chemistryThermo   rho;
}

chemistry       on;

initialChemicalTimeStep 1e-7;

odeCoeffs
{
    solver          seulex;
}
```

chemistryProperties

```
active      true;

combustionModel EDM<rhoChemistryCombustion>;

EDMCoeffs
{
    A 4.0;
    B 0.5;
    finiteRate false;
}
```

combustionProperties

```
FoamFile
{
    version     2.0;
    format      ascii;
    class       dictionary;
    location    "constant";
    object      turbulenceProperties;
}
// * * * * * * * * * * * * * * * * * *

simulationType  RASModel;
```

turbulenceProperties

```
FoamFile
{
    version     2.0;
    format      ascii;
    class       dictionary;
    location    "constant";
    object      RASProperties;
}
// * * * * * * * * * * * * * * * * * *

RASModel        kEpsilon;

turbulence      on;

printCoeffs     yes
```

RASProperties

EdmParcelFoam

‘constant’ Folder

- CloudProperties

```
constantProperties
{
    rho0            960;
    T0              403;
    Cp0             1880;

    constantVolume  false;
}
```

Particle Properties

```
cloudFunctions
{
    patchPostProcessing1
    {
        type            patchPostProcessing;
        maxStoredParcels 100;
        patches         ( outlet );
    }

    particleTracks1
    {
        type            particleTracks;
        trackInterval   5;
        maxSamples      1000000;
        resetOnWrite    yes;
    }
}
```

Cloud Function definition

```
solution
{
    active          yes;

    transient       no;
    calcFrequency   50;
    maxTrackTime    5.0;
    maxCo           0.2;

    coupled         true;
    cellValueSourceCorrection off;

    sourceTerms
    {
        resetOnStartup  false;
        schemes
        {
            rho             semiImplicit 0.1;
            U               semiImplicit 0.1;
            Yi              semiImplicit 0.1;
            h               semiImplicit 0.1;
            radiation       semiImplicit 0.1;
        }
    }

    interpolationSchemes
    {
        rho             cell;
        U               cellPoint;
        thermo:mu       cell;
        T               cell;
        Cp              cell;
        kappa           cell;
        p               cell;
    }

    integrationSchemes
    {
        U               Euler;
        T               analytical;
    }
}
```

Solution definition

EdmParcelFoam

‘constant’ Folder

- CloudProperties

Particle submodel

```
particleForces
{
    sphereDrag;

}
```

```
dispersionModel none;

patchInteractionModel standardWallInteraction;

heatTransferModel RanzMarshall;

compositionModel singleMixtureFraction;

phaseChangeModel liquidEvaporation;

devolatilisationModel none;

surfaceReactionModel none;

stochasticCollisionModel none;

surfaceFilmModel none;

radiation       off;
```

```
standardWallInteractionCoeffs
{
    type            rebound;
}

RanzMarshallCoeffs
{
    BirdCorrection  off;
}

singleMixtureFractionCoeffs
{
    phases
    (
        gas
        {
        }
        liquid
        {
            C7H16 1;
        }
        solid
        {
        }
    );
    YGasTot0        0;
    YLiquidTot0     1;
    YSolidTot0      0;


quidEvaporationCoeffs

    enthalpyTransfer enthalpyDifference;
    activeLiquids   ( C7H16 );
}
```

```
injectionModels
{
    model1
    {
        type            coneNozzleInjection;
        SOI             0;
        massFlowRate      0.0025;
        parcelBasisType mass;
        injectionMethod disc;
        flowType        constantVelocity;
        UMag            30.0;
        outerDiameter   0.005;
        innerDiameter   0.0;
        duration        1;
        position        ( 0.01 0.0231 0.0096 );
        direction       ( 1 0.9239 0.3827 );
        parcelsPerSecond 1000;
        flowRateProfile table
        (
            (0              1.0)
            (1              1.0)
        );
        Cd              constant 0.9;

        thetaInner      constant 0.0;
        thetaOuter      constant 10.0;

        sizeDistribution
        {
            type        RosinRammler;

            RosinRammlerDistribution
            {
                minValue        1e-06;
                maxValue        0.00004;
                d               0.00002;
                n               3.5;
            }
        }
    }
```

Combustion Laboratory POHANG UNIVERSITY OF SCIENCE AND TECHNOLOGY POSTECH

EdmParcelFoam

‘constant’ Folder

- ParticleTracking & Radiation

```
cloudName       reactingCloud1Tracks;

fields
(
    d
    U
    T
);
```

particleTrackDict

```
radiation       on;

radiationModel  P1;

absorptionEmissionModel binaryAbsorptionEmission;

scatterModel    cloudScatter;
```

radiationProperties

```
P1Coeffs
{
}

binaryAbsorptionEmissionCoeffs
{
    model1
    {
        absorptionEmissionModel constantAbsorptionEmission;

        constantAbsorptionEmissionCoeffs
        {
            absorptivity    absorptivity    [ 0 -1 0 0 0 0 0 ] 0.05;
            emissivity      emissivity      [ 0 -1 0 0 0 0 0 ] 0.05;
            E               E   [ 1 -1 -3 0 0 0 0 ] 0;
        }

    }
    model2
    {
        absorptionEmissionModel cloudAbsorptionEmission;
        cloudAbsorptionEmissionCoeffs
        {
            cloudNames
            (
                reactingCloud1
            );
        }
    }
}

cloudScatterCoeffs
{
    cloudNames
    (
        reactingCloud1
    );
}
```


EdmParcelFoam

Solver

- Chemical Reactions

```
thermoType
{
    type            heRhoThermo;
    mixture         reactingMixture;
    transport       sutherland;
    thermo          janaf;
    energy          sensibleEnthalpy;
    equationOfState perfectGas;
    specie          specie;
}
chemistryReader foamChemistryReader;

foamChemistryFile "$FOAM_CASE/constant/chem_FOAM";

foamChemistryThermoFile "$FOAM_CASE/constant/therm_FOAM";
```

thermophysicalProperties

```
reactions
{
    un-named-reaction-0
    {
        type            irreversibleArrheniusReaction;
        reaction        "C7H16^0.25 + 1102^1.5 = 7CO2 + 8H2O";
        A               2.81171e+06;
        beta            0;
        Ta              7940.36;
    }
}
```

foam.inp

```
C7H16
{
    specie
    {
        nMoles          1;
        molWeight       100.206;
    }
    thermodynamics
    {
        Tlow            200;
        Thigh           6000;
        Tcommon         1000;
        highCpCoeffs    ( 20.4565 0.0348575 -1.09227e-05 1.67202e-09
-9.81025e-14 -32555.6 -80.4405 );
        lowCpCoeffs     ( 11.1533 -0.0094942 0.000195572 -2.49754e-07
9.84878e-11 -26768.9 -15.9097 );
    }
    transport
    {
        As              1.67212e-06;
        Ts              170.672;
    }
}
H2O
{
    specie
    {
        nMoles          1;
        molWeight       18.0153;
    }
    thermodynamics
    {
        Tlow            200;
        Thigh           6000;
        Tcommon         1000;
        highCpCoeffs    ( 2.67704 0.00297318 -7.73769e-07 9.44335e-11 -4.269e-15
-29885.9 6.88255 );
        lowCpCoeffs     ( 4.19864 -0.0020364 6.52034e-06 -5.48793e-09
1.77197e-12 -30293.7 -0.849009 );
    }
    transport
    {
        As              1.67212e-06;
        Ts              170.672;
    }
}
```

foam.dat

Combustion Laboratory POHANG UNIVERSITY OF SCIENCE AND TECHNOLOGY POSTECH

EdmParcelFoam

Tutorial

1. **Open ‘Terminal’ : click** 

2. **~$ cd tutorials/oilSpray-singleBurner**

```
cd tutorials/oilSpray singleBurner/
```

3. **~$ decomposePar**

```
~/tutorials/oilSpray singleBurner$ decomposePar
```

```
Time = 0

Processor 0: field transfer
Processor 1: field transfer
Processor 2: field transfer
Processor 3: field transfer

End.
```

4. **~$ mpirun –np 4 EdmParcelFoam -parallel**

```
~/tutorials/oilSpray_singleBurner$ mpirun -np 4 EdmParcelFoam -parallel
```


EdmParcelFoam

Tutorial

- Calculating

Radiation part

```
Selecting radiationModel P1
Selecting absorptionEmissionModel binaryAbsorptionEmission
Selecting absorptionEmissionModel constantAbsorptionEmission
Selecting absorptionEmissionModel cloudAbsorptionEmission
Selecting scatterModel cloudScatter
```

①

Particle part

```
Constructing reacting cloud
Constructing particle forces
    Selecting particle force sphereDrag
Constructing cloud functions
    Selecting cloud function patchPostProcessing1 of type patchPostProcessing
    Selecting cloud function particleTracks1 of type particleTracks
Constructing particle injection models
Creating injector: model1
Selecting injection model coneNozzleInjection
    Constructing 3-D injection
Selecting distribution model RosinRammler
Creating injector: model2
Selecting injection model coneNozzleInjection
    Constructing 3-D injection
Selecting distribution model RosinRammler
Selecting dispersion model none
Selecting patch interaction model standardWallInteraction
Selecting stochastic collision model none
Selecting surface film model none
Selecting U integration scheme Euler
Selecting heat transfer model RanzMarshall
Selecting T integration scheme analytical
Selecting composition model singleMixtureFraction
--> FOAM Warning :
    From function phaseProperties::initialiseGlobalIds(...)
    in file phaseProperties/phaseProperties/phaseProperties.C at line 231
    Assuming no mapping between solid and carrier species
Selecting phase change model liquidEvaporation
Participating liquid species:
    C7H16
Selecting devolatilisation model none
Selecting surface reaction model none
```

②

EdmParcelFoam

Tutorial

- Calculating

```
Time = 50


Solving 3-D cloud reactingCloud1

--> Cloud: reactingCloud1 injector: model1
    Added 1000 new parcels


--> Cloud: reactingCloud1 injector: model2
    Added 1000 new parcels

Cloud: reactingCloud1
    Current number of parcels       = 0
    Current mass in system          = 0
    Linear momentum                 = (0 0 0)
    |Linear momentum|               = 0
    Linear kinetic energy           = 0
    model1:
        number of parcels added     = 1000
        mass introduced             = 0.0025
    model2:
        number of parcels added     = 1000
        mass introduced             = 0.0025
    Parcels absorbed into film      = 0
    New film detached parcels       = 0
    Parcel fate (number, mass)
      - escape                      = 0, 0
      - stick                       = 0, 0
    Temperature min/max             = 0, 0
    Mass transfer phase change      = 0.005
    Mass transfer devolatilisation  = 0
    Mass transfer surface reaction  = 0
```

```
relaxationFactors
{
    fields
    {
        p               0.1;
        rho             0.5;
    }
    equations
    {
        U               0.3;
        h               0.3;
        ".*"            0.3;
    }
}
```

fvSolution

```
sourceTerms
{
    resetOnStartup  false;
    schemes
    {
        rho             semiImplicit 0.1;
        U               semiImplicit 0.1;
        Yi              semiImplicit 0.1;
        h               semiImplicit 0.1;
        radiation       semiImplicit 0.1;
    }
}
```

reactingCloud1Properties

③

EdmParcelFoam

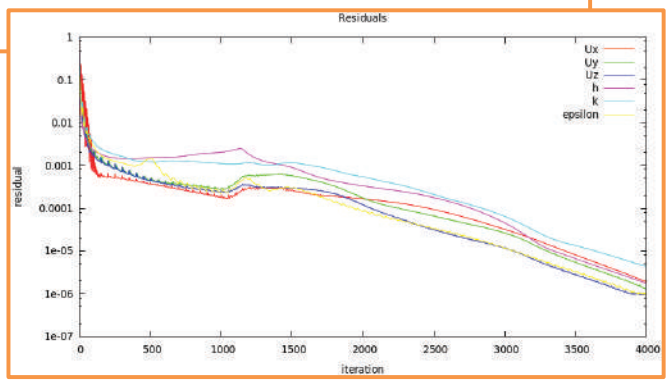
Tutorial

- Calculating

```
DILUPBiCG:  Solving for Ux, Initial residual = 1.878590845e-06, Final residual = 4.523418728e-08, No Iterations 1
DILUPBiCG:  Solving for Uy, Initial residual = 1.298588821e-06, Final residual = 2.056457301e-08, No Iterations 1
DILUPBiCG:  Solving for Uz, Initial residual = 9.391297852e-07, Final residual = 9.391297852e-07, No Iterations 0
DILUPBiCG:  Solving for C7H16, Initial residual = 2.827608117e-06, Final residual = 7.779010922e-08, No Iterations 1
DILUPBiCG:  Solving for O2, Initial residual = 2.716931007e-06, Final residual = 5.095022165e-08, No Iterations 1
DILUPBiCG:  Solving for CO2, Initial residual = 3.046806829e-06, Final residual = 5.251779861e-08, No Iterations 1
DILUPBiCG:  Solving for H2O, Initial residual = 2.841900211e-06, Final residual = 4.890468115e-08, No Iterations 1
DILUPBiCG:  Solving for h, Initial residual = 1.745188062e-06, Final residual = 5.280296428e-08, No Iterations 1
DICPCG:  Solving for G, Initial residual = 3.013691702e-06, Final residual = 2.655402533e-07, No Iterations 95
T gas min/max   = 424.0071719, 2499.615864
GAMG:  Solving for p, Initial residual = 1.373624123e-05, Final residual = 7.502991878e-08, No Iterations 6
GAMG:  Solving for p, Initial residual = 7.504060786e-08, Final residual = 8.126580965e-09, No Iterations 3
GAMG:  Solving for p, Initial residual = 8.126582604e-09, Final residual = 8.126582604e-09, No Iterations 0
time step continuity errors : sum local = 4.997876631e-06, global = 4.832265538e-06, cumulative = 1.115286599
p min/max = 99730.19215, 100019.3388
DILUPBiCG:  Solving for epsilon, Initial residual = 1.079586686e-06, Final residual = 2.734256391e-08, No Iterations 1
DILUPBiCG:  Solving for k, Initial residual = 4.415556082e-06, Final residual = 8.590897219e-08, No Iterations 1
ExecutionTime = 576 s  ClockTime = 577 s

End
```

④

Radiation



Combustion Laboratory POHANG UNIVERSITY OF SCIENCE AND TECHNOLOGY POSTECH

EdmParcelFoam

Post Processing

1. ~$ reconstructPar

```
~/tutorials/oilSpray_singleBurner$ reconstructPar -latestTime
```

2. ~$ cd tutorials/EDM_reacting_flow/paraFoam

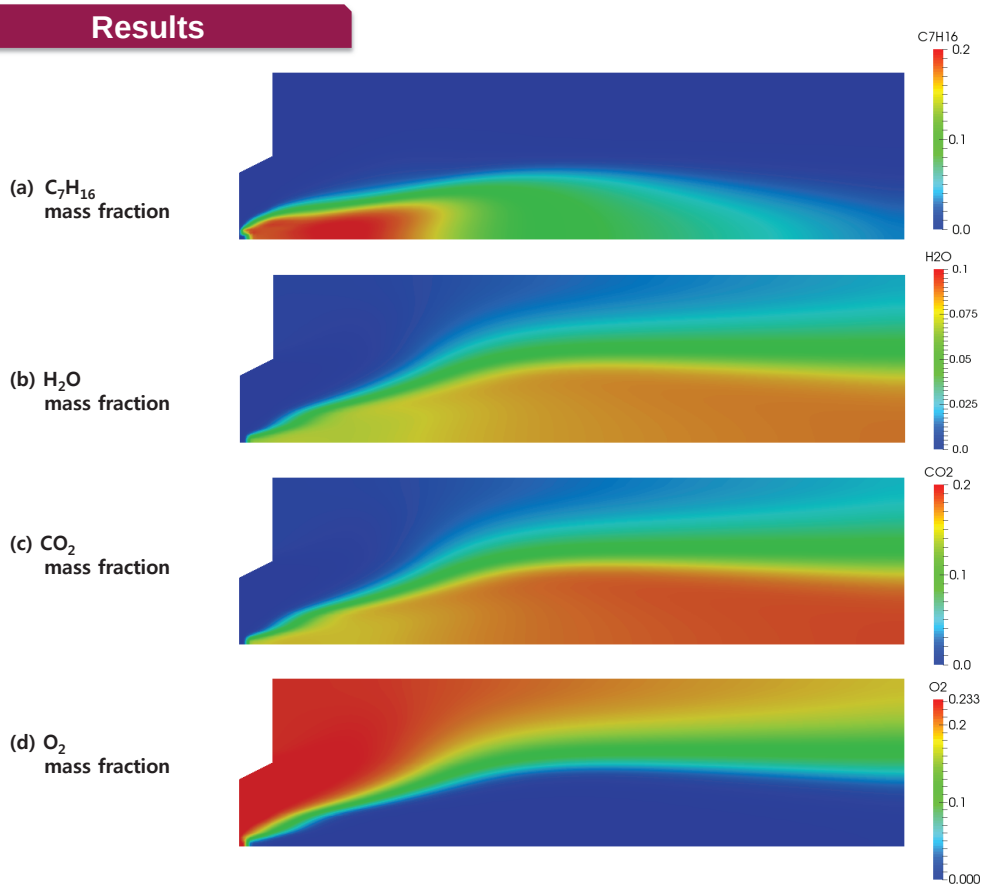
```
:~/tutorials/oilSpray_singleBurner$ paraFoam
```

3. ~$ steadyParticleTracks

```
:~/tutorials/oilSpray_singleBurner$ steadyParticleTracks
```

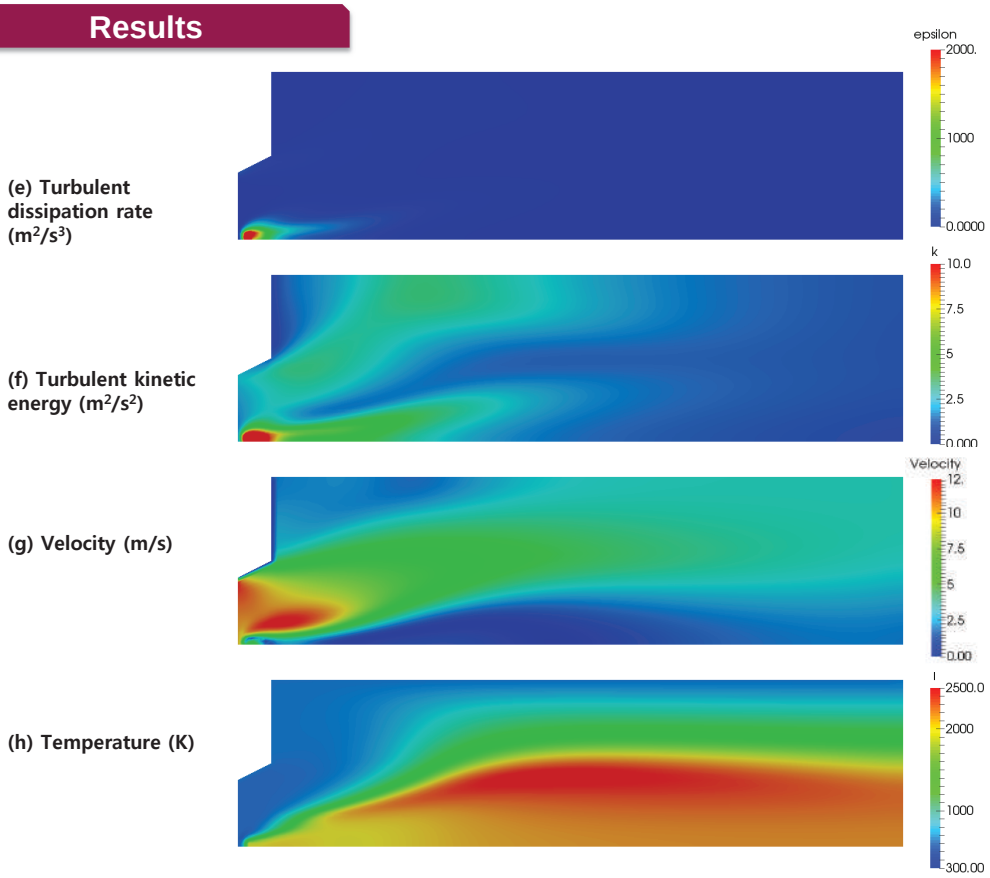

EdmParcelFoam

Results



EdmParcelFoam

Results



(e) Turbulent dissipation rate (m^2/s^3)

(f) Turbulent kinetic energy (m^2/s^2)

(g) Velocity (m/s)

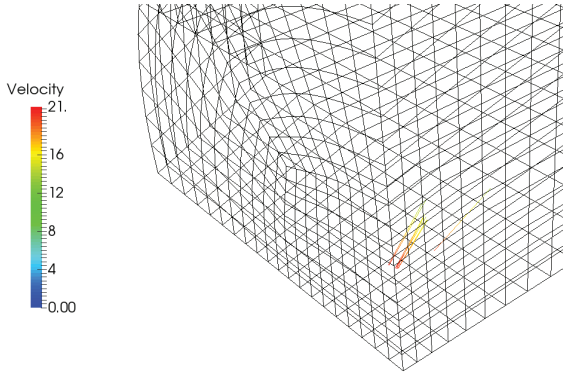
(h) Temperature (K)

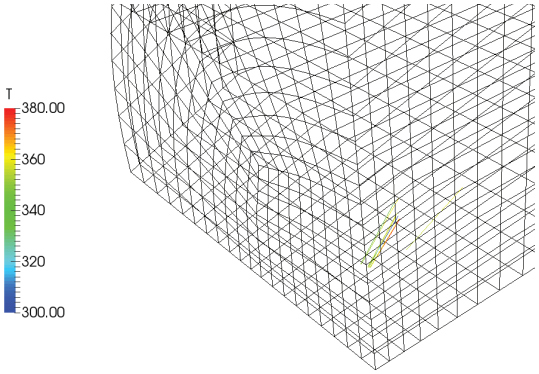
EdmParcelFoam

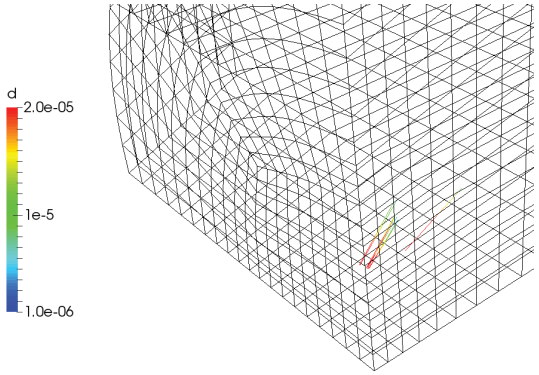
Results

```
cloudName       reactingCloud1Tracks;

fields
(
    d
    U
    T
);
```

particleTrackDict







Combustion Laboratory POHANG UNIVERSITY OF SCIENCE AND TECHNOLOGY POSTECH