

# OpenFOAM: Introduction and Basic Class Layout

Hrvoje Jasak

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

University of Zagreb, Croatia and

Wikki Ltd, United Kingdom



OpenFOAM in Industrial Combustion Simulations, Pohang University Feb/2015

OpenFOAM: Introduction and Basic Class Layout – p. 1

## Background

### Objective

- Provide basic information about OpenFOAM software from User's viewpoint
- Give overview of basic object oriented programming tools
- Give a top-level overview of FOAM data structures
- Provide basic information on Finite Volume Discretisation

### Topics

- OpenFOAM: Brief overview
- Points of interest: Why consider OpenFOAM
- Introduction to object-oriented programming and C++
- Five basic classes in OpenFOAM
- Mesh handling
- Finite volume discretisation and boundary conditions
- Summary

## What is OpenFOAM?

- **OpenFOAM** is a free-to-use Open Source numerical simulation software with extensive CFD and multi-physics capabilities
- Free-to-use means using the software without paying for license and support, including **massively parallel computers**: free 1000-CPU CFD license!
- Software under active development, capabilities mirror those of commercial CFD
- Substantial installed user base in industry, academia and research labs
- Possibility of extension to non-traditional, complex or coupled physics: Fluid-Structure Interaction, complex heat/mass transfer, internal combustion engines, nuclear

## Main Components

- Discretisation: Polyhedral Finite Volume Method, second order in space and time
- Lagrangian particle tracking, Finite Area Method (2-D FVM on curved surface)
- Massive parallelism in domain decomposition mode
- Automatic mesh motion (FEM), support for topological changes
- All components implemented in library form for easy re-use
- Physics model implementation through **equation mimicking**



# OpenFOAM: Brief Overview

## Implementing Continuum Models by Equation Mimicking

- Natural language of continuum mechanics: partial differential equations
- Example: turbulence kinetic energy equation

$$\frac{\partial k}{\partial t} + \nabla \cdot (\mathbf{u}k) - \nabla \cdot [(\nu + \nu_t) \nabla k] = \nu_t \left[ \frac{1}{2} (\nabla \mathbf{u} + \nabla \mathbf{u}^T) \right]^2 - \frac{\epsilon_o}{k_o} k$$

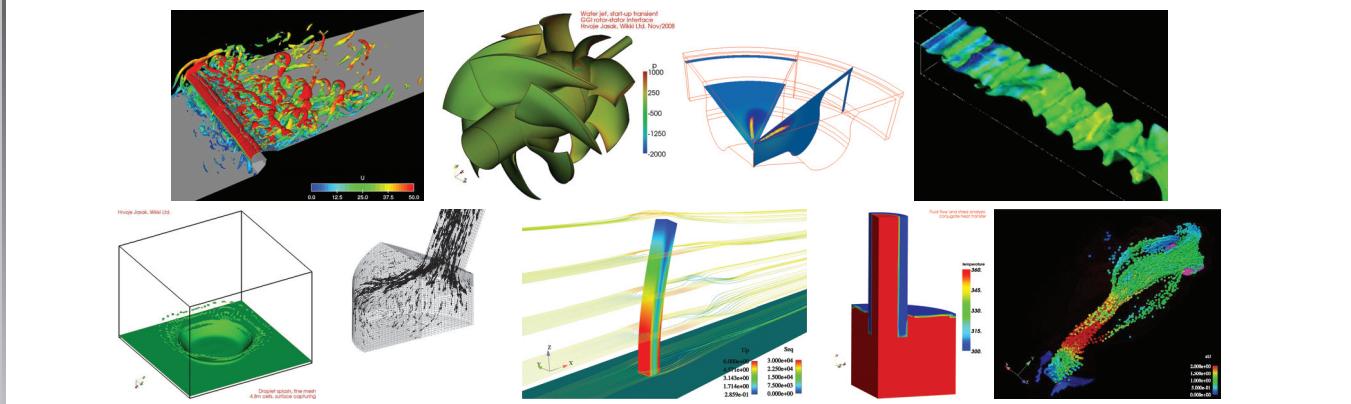
- Objective: **represent differential equations in their natural language**

```
solve
(
    fvm::ddt(k)
    + fvm::div(phi, k)
    - fvm::laplacian(nu() + nut, k)
    == nut*magSqr(symm(fvc::grad(U)))
    - fvm::Sp(epsilon/k, k)
);
```

- Correspondence between the implementation and the original equation is clear

## Physical Modelling Capability Highlights

- Basic: Laplace, potential flow, passive scalar/vector/tensor transport
- Incompressible and compressible flow: segregated pressure-based algorithms
- Heat transfer: buoyancy-driven flows, conjugate heat transfer
- Multiphase: Euler-Euler, VOF free surface capturing and surface tracking
- RANS for turbulent flows: 2-equation, RSTM; full LES capability
- Pre-mixed and Diesel combustion, spray and in-cylinder flows
- Stress analysis, fluid-structure interaction, electromagnetics, MHD, etc.



OpenFOAM in Industrial Combustion Simulations, Pohang University Feb/2015

OpenFOAM: Introduction and Basic Class Layout – p. 5

## Running OpenFOAM

### Structure of OpenFOAM

- OpenFOAM is assembled from components
  - Foundation libraries, containing discretisation, mesh handling etc. in re-usable form. Functionality shared across many application
  - Physical modelling libraries: thermo-physical models (liquids and gasses), viscosity models, turbulence models, chemical reactions interface
  - Utilities: mesh import and manipulation, parallel processing, post processor hook-up (reader module) and data manipulation
  - Customised, purpose-written and optimised executables for each physics segment. All are founded on common components
  - Linkage for user extensions and on-the-fly data analysis

### OpenFOAM Executable

- Custom executable for specific physics segment: few 100s of lines of code
- Easy to read, understand, modify or add further capability
- Existing code is used as a basis for own development: simulation environment
- Low-level functions, eg. mesh handling, parallelisation, data I/O handled transparently: no need for special coding at top level



OpenFOAM in Industrial Combustion Simulations, Pohang University Feb/2015

OpenFOAM: Introduction and Basic Class Layout – p. 6

## Why Consider OpenFOAM?

- Open architecture
  - Access to complete source: no secret modelling tricks, no cutting corners
  - Both community-based and professional support available
  - Common platform for new R&D projects: shipping results of research into the hands of a customer with no delay
- Low-cost CFD
  - No license cost, portable to any computing platform (IBM Blue Gene)
  - Efficient on massively parallel computers, portable to new comms protocols
- Problem-independent numerics and discretisation
  - Tackle non-standard continuum mechanics problem: looking beyond the capabilities of commercial CFD
- Efficient environment for complex physics problems
  - Tackling difficult physics is made easier through equation mimicking
  - Utility-level tools readily available: parallelism, moving mesh
  - Track record in non-linear and strongly coupled problems
  - Excellent piece of C++ and software engineering! Decent piece of CFD

# Object Orientation

## Object-Oriented Software: Create a Language Suitable for the Problem

- Analysis of numerical simulation software through object orientation:  
“Recognise main objects from the numerical modelling viewpoint”
- Objects consist of **data** they encapsulate and **functions** which operate on the data

### Example: Sparse Matrix Class

- **Data members:** protected and managed
  - Sparse addressing pattern (CR format, arrow format)
  - Diagonal coefficients, off-diagonal coefficients
- Operations on matrices or data members: **Public interface**
  - Matrix algebra operations:  $+$ ,  $-$ ,  $*$ ,  $/$ ,
  - Matrix-vector product, transpose, triple product, under-relaxation
- Actual data layout and functionality is important only internally: efficiency

### Example: Linear Equation Solver

- Operate on a system of linear equations  $[A][x] = [b]$  to obtain  $[x]$
- It is irrelevant how the matrix was assembled or what shall be done with solution
- Ultimately, even the solver algorithm is not of interest: all we want is new  $[x]$ !
- Gauss-Seidel, AMG, direct solver: all answer to the same interface

```
class vector
{
    // Private data

    // - Components
    double V[3];

public:

    // Component labeling enumeration
    enum components { X, Y, Z };

    // Constructors

    // - Construct null
    vector() {}

    // - Construct given three scalars
    vector(const double& Vx, const double& Vy, const double& Vz)
    {
        V[X] = Vx; V[Y] = Vy; V[Z] = Vz;
    }
}
```



# Classes: Protecting Your Data

```
// Destructor

~vector();

// Member Functions

const word& name() const;
static const dimension& dimensionOfSpace();

const double& x() const { return V[X]; }
const double& y() const { return V[Y]; }
const double& z() const { return V[Z]; }
double& x() { return V[X]; }
double& y() { return V[Y]; }
double& z() { return V[Z]; }

// Member Operators

void operator=(const vector& v);

inline void operator+=(const vector&);
inline void operator-=(const vector&);
inline void operator*=(const scalar);
```



```
// Friend Functions

    friend vector operator+(const vector& v1, const vector& v2)
    {
        return vector(v1.V[X]+v2.V[X],v1.V[Y]+v2.V[Y],v1.V[Z]+v2.V[Z]);
    }

    friend double operator*(const vector& v1, const vector& v2)
    {
        return v1.V[X]*v2.V[X] + v1.V[Y]*v2.V[Y] + v1.V[Z]*v2.V[Z];
    }

    friend vector operator^(const vector& v1, const vector& v2)
    {
        return vector
        (
            (v1.V[Y]*v2.V[Z] - v1.V[Z]*v2.V[Y]),
            (v1.V[Z]*v2.V[X] - v1.V[X]*v2.V[Z]),
            (v1.V[X]*v2.V[Y] - v1.V[Y]*v2.V[X])
        );
    }
};
```



# Classes: Protecting Your Data

## Vector Class: Summary

- Class is responsible for managing its own data: (x, y, z)
- Class provides interface for data manipulation; private data is accessible only from within the class: **data protection**
- Vector class (code component) can be developed and tested in isolation

## Manipulating Vectors

```
vector a, b, c;
vector area = 0.5*((b - a)^ (c - a));
```

## Constant and Non-Constant Access

- Accessing a vector from another class

```
class cell
{
public:
    const vector& centre() const;
};
```

- Cell class allows me to look at the vector **but not to change it!**
- First const promises not to change the vector
- Second promises not to change class object



## Types of Data Access

- **Pass by value:** make a copy of local data. Changing a copy does not influence the original, since it is a different instance
- **Pass by const reference:** Give read-only access to local data
- **Pass by reference:** Give read-write access to local data. This type of access allows local value to be changed, usually changing the class
- **Pointer handling:** Instead of dealing with data, operative with memory locations (addresses) where the data is stored

## Constant and Non-Constant Access

- Pass-by-value and pass-by-reference: is the data being changed?

```
class cell
{
public:
    vector centre() const;
    vector* centrePtr();
    vector& centre();
    const vector& centre() const;
};
```



## Return a Value, a Pointer or a Reference

- In actual compiler implementation, pointers and references are handled with the same mechanism: memory address of object storage
- Dealing with pointers and protecting pointer data is cumbersome and ugly

```
class cell
{
public:
    vector* centrePtr();
    const vector const* centre() const;
};

cell c(...);

if (c.centrePtr() != NULL)
{
    vector& ctr = *(c.centrePtr());
    ctr += vector(3.3, 1, 1);
}
```

- Is the pointer set? Is it valid: check for NULL. Should I delete it?
- A **reference** is an “always on” pointer with object syntax. No checking required



## Implementing the Same Operation on Different Types

- User-defined types (classes) should behave exactly like the “built-in” types
- Some operations are generic: e.g. magnitude: same name, different argument

```
label m = mag(-3);
scalar n = mag(3.0/m);

vector r(1, 3.5, 8);
scalar magR = mag(r);
```

- Warning: **implicit type conversion** is a part of this game! This allows C++ to convert from some types to others according to specific rules

- Function or operator syntax

```
vector a, b;
vector c = 3.7*(a + b);
```

is identical to

```
vector c(operator*(3.7, operator+(a, b)));
```

- Operator syntax is regularly used because it looks nicer, but for the compiler both are “normal” function calls



# Class Derivation

## Particle Class: Position and Location

- Position in space: vector = point
- Cell index, boundary face index, is on a boundary?

```
class particle
{
    public vector
    {
        // Private data

        //- Index of the cell it is
        label cellIndex_;

        //- Index of the face it is
        label faceIndex_;

        //- Is particle on boundary/outside domain
        bool onBoundary_;
    };
}
```

- *is-a* relationship: class is derived from another class
- *has-a* relationship: class contains member data



## Implementing Boundary Conditions

- Boundary conditions represent a class of related objects, all doing the same job
  - Hold boundary values and rules on how to update them
  - Specify the boundary condition effect on the matrix
- ...but each boundary condition does this job in its own specific way!
- Examples: fixed value (Dirichlet), zero gradient (Neumann), mixed, symmetry plane, periodic and cyclic etc.
- However, the code operates on all boundary conditions in a consistent manner

```
forAll (boundaryConditions, i)
{
    if (boundaryConditions[i].type() == fixedValue)
    {
        // Evaluate fixed value b.c.
    }
    else if (boundaryConditions[i].type() == zeroGradient)
    {
        // Evaluate zero gradient b.c.
    }
    else if (etc...)
}
```



## Implementing Boundary Conditions

- The above pattern is repeated throughout the code. Introducing a new boundary condition type is complex and error-prone: many distributed changes, no checking
- In functional code, the for-loop would contain an if-statement. When a new boundary condition is added, the if-statement needs to be changed (for all operations where boundaries are involved)
- We wish to consolidate a boundary condition into a class. Also, the actual code remains independent on **how** the b.c. does its work!
- Codify a generic boundary condition interface: **virtual base class**

```
class fvPatchField
{
public:
    virtual void evaluate() = 0;
};

List<fvPatchField*> boundaryField;
forAll (boundaryField, patchI)
{
    boundaryField[patchI] ->evaluate();
}
```



## Implementing Boundary Conditions

- Working with **virtual functions**
  1. Define what a “generic boundary condition” is through functions defined on the base class: *virtual functions*
  2. Implement the specific (e.g. fixed value) boundary conditions to answer to generic functionality in its own specific way
  3. The rest of the code operates only with the generic conditions
  4. When a virtual function is called (generic; on the base class), the actual type is recognised and the specific (on the derived class) is called at run-time
- Note that the “generic boundary condition” does not really exist: it only defines the behaviour for all derived (concrete) classes
- Consequences
  - Code will not be changed when new condition is introduced: no if-statement to change: new functionality does not disturb working code
  - New derived class automatically hooks up to all places
  - Shared functions can be implemented in base class

```
template<class Type>
class fvPatchField
:
public Field<Type>
{
public:

    // Construct from patch, internal field and dictionary
    fvPatchField
    (
        const fvPatch&,
        const Field<Type>&,
        const dictionary&
    );

    // Destructor
    virtual ~fvPatchField();

    virtual bool fixesValue() const { return false; }

    virtual void evaluate() = 0;
};
```

```
template<class Type>
class fixedValueFvPatchField
:
public fvPatchField<Type>
{
public:

    // Construct from patch, internal field and dictionary
    fixedValueFvPatchField
    (
        const fvPatch&,
        const Field<Type>&,
        const dictionary&
    );

    // Destructor
    virtual ~fixedValueFvPatchField();

    virtual bool fixesValue() const { return true; }

    virtual void evaluate() {}

};
```



## Generic Programming: Templates

### What are Templates?

- Some operations are independent of the type on which they are being performed.  
Examples: container (list, linked list, hash table), sorting algorithm
- C++ is a **strongly type language**: checks for types of all data and functions and gives compiler errors if they do not fit
- Ideally, we want to implement algorithms generically and produce custom-written code before optimisation
- Templating mechanism in C++
  - Write algorithms with a generic type holder

```
template<class Type>
```
  - Use generic algorithm on specific type

```
List<cell> cellList(3);
```
  - Compiler to expand the code and perform optimisation after expansion
- Generic programming techniques massively increase power of software: less software to do more work
- Debugging is easier: if it works for one type, it will work for all
- ... but writing templates is trickier: need to master new techniques
- Many “CFD” operations are generic: lots of templating in OpenFOAM



```
template<class T>
class List
{
public:
    // Construct with given size
    explicit List(const label);

    // Copy constructor
    List(const List<T>&);

    // Destructor
    ~List();

    // Reset size of List
    void setSize(const label);

    // Return subscript-checked element of List
    inline T& operator[](const label);

    // Return subscript-checked element of constant LList
    inline const T& operator[](const label) const;
};
```



# Generic Programming: Templates

## Bubble sort algorithm

```
template<class Type>
void Foam::bubbleSort(List<Type>& a)
{
    Type tmp;
    for (label i = 0; i < n - 1; i++)
    {
        for (label j = 0; j < n - 1 - i; j++)
        {
            // Compare the two neighbors
            if (a[j+1] < a[j])
            {
                tmp = a[j]; // swap a[j] and a[j+1]
                a[j] = a[j+1];
                a[j+1] = tmp;
            }
        }
    }

    List<cell> cellList(55); // Fill in the list here
    bubbleSort(cellList);
```



## Main Objects

- Computational domain

Object	Software representation	C++ Class
Tensor	(List of) numbers + algebra	vector, tensor
Mesh primitives	Point, face, cell	point, face, cell
Space	Computational mesh	polyMesh
Time	Time steps (database)	time

- Field algebra

Object	Software representation	C++ Class
Field	List of values	Field
Boundary condition	Values + condition	patchField
Dimensions	Dimension set	dimensionSet
Geometric field	Field + mesh + boundary conditions	geometricField
Field algebra	+ - * / $tr()$ , $\sin()$ , $\exp()$ ...	field operators

# Object Orientation

## Main Objects

- Linear equation systems and linear solvers

Object	Software representation	C++ Class
Linear equation matrix	Matrix coefficients	lduMatrix
Solvers	Iterative solvers	lduMatrix::solver

- Numerics: discretisation methods

Object	Software representation	C++ Class
Interpolation	Differencing schemes	interpolation
Differentiation	ddt, div, grad, curl	fvc, fec
Discretisation	ddt, d2dt2, div, laplacian	fvm, fem, fam

- Top-level code organisation

Object	Software representation	C++ Class
Model library	Library	eg. turbulenceModel
Application	main()	–

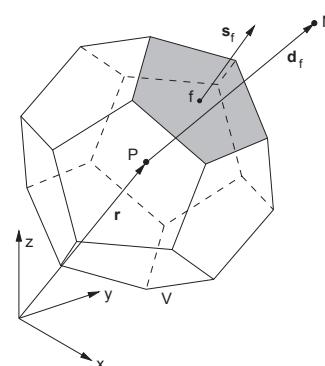
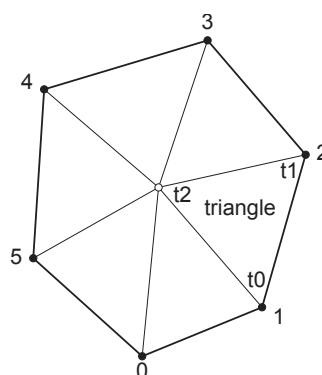
## Polyhedral Mesh Support

- To my knowledge, first CFD/CCM implementation capable of handling polyhedral cells throughout: working in 1995-1996
- Robust fully automatic mesh generator is a requirement in modern CFD
- Rationale
  - A polyhedron is a generic form covering all cell types: consistency in discretisation across the board
  - Finite Volume Method (FVM) naturally applies to polyhedral cells: shape (tet, hex, pyramid, prism) is irrelevant (unlike FEM)
  - Mesh generation is still a bottleneck: polyhedral support simplifies the problem
- Consequences
  - All algorithms must be fully unstructured. Structured mesh implementation possible where desired (e.g. aeroacoustics) but implies separate mesh classes and work on discretisation code re-use
  - In 1990s, fully unstructured support was a challenge: now resolved
  - No problems with imported mesh formats: polyhedral cell covers it all
  - Issues with “old-fashioned software” compatibility with no polyhedral support, e.g. post-processors. On-the-fly decomposition

# Polyhedral Cell Definition

## Nomenclature

- A **polygonal face** is defined by a list of vertex labels. The ordering of vertex labels defines the face normal (orientation) using the right-hand rule
- A **polyhedral cell** is defined by a list of face labels that bound it
- Cell centre is marked by  $P$ , face centre and face interpolates by  $f$ , face normal  $s_f$  and neighbouring cell centre by  $N$
- Face centre and cell volume are calculated using a decomposition into triangles or pyramids



## Strong Ordering Requirement

- Polyhedral mesh definition
  - List of vertices. Vertex position in the list determines its label
  - List of faces, defined in terms of vertex labels
  - List of cells, defined in terms of face labels
  - List of boundary patches
- **All indices start from zero:** C-style numbering (no discussion please)
- OpenFOAM uniquely orders mesh faces for easier manipulation
  - All internal faces are first in the list, ordered by the cell index they belong to. Lower index defines **owner cell** ( $P$ ); face normal points out of the owner cell
  - Faces of a single cell are ordered with increasing neighbour label, *i.e.* face between cell 5 and 7 comes before face between 5 and 12
  - Boundary faces are ordered in patch order. All face normals point outwards of the domain
- With the above ordering, patches are defined by their type, start face in the face list and number of faces
- Above ordering allows use of List slices for geometrical information

# Mesh Conversion

## Converting Meshes for Use with OpenFOAM

- Native mesh generation in OpenFOAM is still work-in-progress. Focus on polyhedral methods and parallelism for large-scale mesh generation
- The source of meshes are external mesh generators and meshes prepared for other packages: **mesh conversion**
- Polyhedral mesh format makes conversion simple: no limitations on OpenFOAM side. Existing classes used to enforce strong mesh ordering, check mesh quality, manipulate boundary patches and perform simple handling operations
- Some packages provide native OpenFOAM mesh format, eg. Harpoon (<http://www.sharc.co.uk/>)

## Current Mesh Converters

- fluentMeshToFoam, gambitToFoam
- ansysToFoam, cfxToFoam
- star4ToFoam, ccm26ToFoam
- ideasUnvToFoam
- kivaToFoam
- netgenToFoam, tetgenToFoam

## Mesh Manipulation Tools

- transformPoints
- autoPatch
- deformedGeom
- mergeMeshes
- mergeOrSplitBaffles
- renumberMesh etc.

## Checking Polyhedral Meshes

- Polyhedral definition introduces new kinds of errors
- Consistency checks
  - All points are used in at least one face
  - Boundary faces are used exactly one cell
  - Internal faces are used in at exactly two cells
- Topology checks
  - For each cell, every edge is used in exactly two faces
  - For complete boundary, every edge is used in exactly two faces
- Geometry checks
  - All volumes and all face areas magnitudes are positive
  - For each cell, sum of face area vectors is zero to machine tolerance. This takes into account the flipping of owner/neighbour face vectors are required
  - For complete boundary, sum of face area vectors is zero to machine tolerance
- Convexness condition: weak definition
  - In face decomposition, normals of all triangles point in the same direction
  - In cell decomposition, volumes of all tetrahedra is positive

# Discretisation Support

## Mesh Classes

- Base mesh classes: topology and geometry analysis engine
  - **polyMesh**: collection of all points, faces, cells and boundary patches in the simulation
  - **primitiveMesh**: active singly connected mesh as described above
- Discretisation types require specific support, re-using basic mesh info
  - **fvMesh**: supports the FVM discretisation
  - **tetFemMesh**: supports the FEM discretisation on tetrahedral decomposition
- Patch information separated along the same lines
  - **PrimitivePatch**: topology/geometry analysis
  - **polyPatch**: patch on the polyhedral mesh
  - **fvPatch**: support for the FVM
  - **tetFemPatch**: support for the FEM
- Patch mapping and access to base topology/geometry available for special cases

## Operating on Subsets

- Zones and sets allow sub-setting of mesh elements
- Note: discretisation and matrix will always be associated with the complete mesh!
- Zones: points, faces and cells
  - Define partition of mesh elements. Each point/face/cell may belong to a maximum of one zone.
  - Fast two-directional query: what zone does this point belong to?
  - Used extensively in topological mesh changes
- Sets
  - Arbitrary grouping of points/faces/cells for manipulation
  - Single cell may belong to multiple sets
  - Sets used to create other sets: data manipulation with `setSet` tool
  - Examples

```
faceSet f0 new patchToFace movingWall
faceSet f0 add labelToFace (0 1 2)
pointSet p0 new faceToPoint f0 all
cellSet c0 new faceToCell f0 any
cellSet c0 add pointToCell p0 any
```

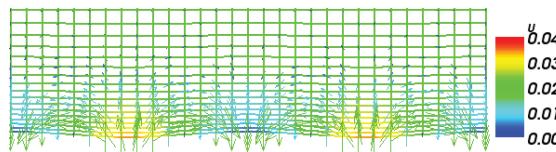
# Moving Mesh Simulations

## Moving Mesh Simulations

- Definition of a moving mesh problem: the number of points, faces and cells in the mesh and their connectivity remains the same but the point position changes
- Sufficient for most cases where shape of domain changes in time
- FVM naturally extends to moving meshes: need to re-calculate cell volume and area swept by a face in motion
- Moving mesh support built into mesh classes and discretisation operators
- In some places, algorithmic changes required in top-level solver code
- Problem: how to specify point motion for every point in every time-step?

## Automatic Mesh Motion Solver

- Input for moving mesh cases is the point position field for every time step. This can be a pre-determined function of time (e.g. engine simulations) or a part of the solution (e.g. large deformation stress analysis, free surface tracking solver)
- Typically, only changes in boundary shape are of interest; internal points are moved to accommodate boundary motion and preserve mesh validity
- Automatic mesh motion solver
  - Internal point motion obtained by solving a “motion equation”
  - Prescribed boundary motion provides boundary conditions
  - Point-based (FEM) solver: no interpolation
  - Mesh validity criteria accounted for during discretisation
  - Easily handles solution-dependent mesh motion: solving one additional equation to obtain point positions



# Topological Changes

## Topological Changes: Mesh Morphing

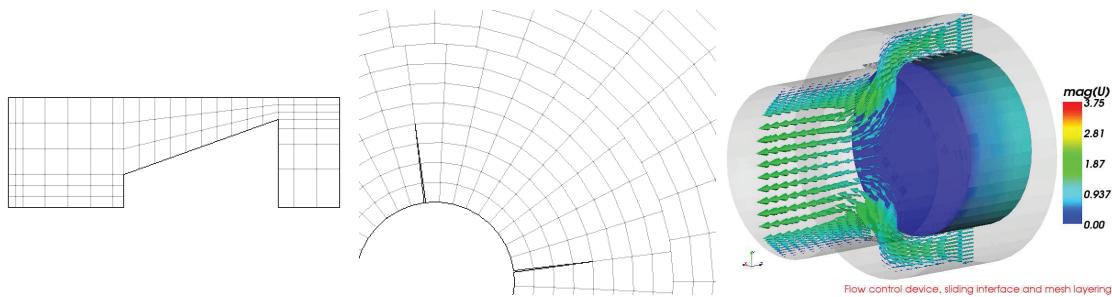
- Typically used with mesh motion; covers cases where the number of points/faces/cells and/or mesh connectivity changes during the simulation
- More complex and demanding than simple mesh motion
- Examples: attach-detach boundaries, cell layer addition/removal, sliding interfaces

## Mesh Morphing Engine

1. Define primitive mesh operations: **add/modify/remove a point/face/cell**
2. Support primitive operations in polyMesh
3. Describe **mesh modifiers** in terms of primitive mesh operations
4. Create the topology instruction by setting up a combination of mesh modifiers on the original mesh
5. Collect changes from mesh modifiers into a **topology change request**
6. Execute topological change, including field mapping!

## Mesh Morphing Engine

- Each mesh modifier is self-contained, including the triggering criterion
- Complex cases contain combinations of modifiers working together, mesh motion and multi-step topological changes
- Polyhedral mesh support makes topological changes easier to handle: solver is always presented with a valid mesh



- This is the current point of development
  - Intersecting mesh modifiers, parallelisation and bugs (sliding interface cutting algorithm), unified solver support (currently requires top-level code changes)
  - Issues with conservation of global properties after topological change. In some cases, this is algorithmic

# What is Discretisation?

## Numerical Discretisation Method

- Generic transport equation can very rarely be solved analytically: this is why we resort to numerical methods
- **Discretisation** is a process of representing the differential equation we wish to solve by a set of algebraic expressions of equivalent properties (typically a matrix)
- Two forms of discretisation operators. We shall use a divergence operator as an example.
  - **Calculus.** Given a vector field  $\mathbf{u}$ , produce a scalar field of  $\nabla \cdot \mathbf{u}$
  - **Method.** For a given divergence operator  $\nabla \cdot$ , create a set of matrix coefficients that represent  $\nabla \cdot \mathbf{u}$  for any given  $\mathbf{u}$
- The Calculus form can be easily obtained from the Method (by evaluating the expression), but this is not computationally efficient

## Discretisation Methodology: Polyhedral Finite Volume Method

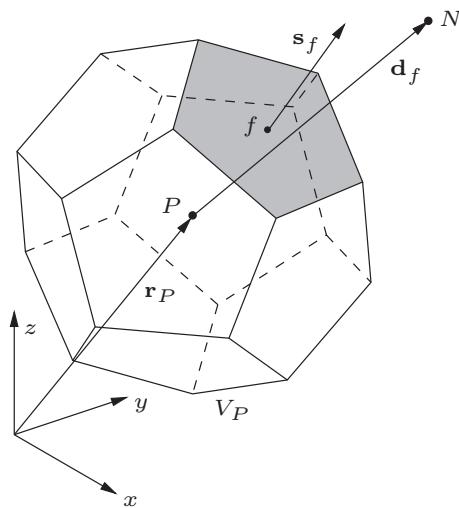
1. We shall assemble the discretisation on a **per-operator** basis: visit each operator in turn and describe a strategy for evaluating the term explicitly and discretising it
2. Describe space and time: a **computational mesh** for the spatial domain and **time-steps** covering the time interval
3. Postulate spatial and temporal variation of  $\phi$  required for a discrete representation of field data
4. Integrate the operator over a cell
5. Use the spatial and temporal variation to interpret the operator in discrete terms

# Discretised Transport Equation

## Representation of a Field Variable

- Equations we operate on work on fields: before we start, we need a discrete representation of the field
- Main solution variable will be stored in cell centroid: collocated cell-centred finite volume method. Boundary data will be stored on face centres of boundary faces
- For some purposes, e.g. face flux, different data is required – in this case it will be a field over all faces in the mesh
- Spatial variation can be used for interpolation in general: post-processing tools typically use point-based data.

## Computational Cell



- This is a convex polyhedral cell boundary be a set of convex polygons
- Point  $P$  is the computational point located at cell centroid  $\mathbf{x}_P$ . The definition of the centroid reads:

$$\int_{V_P} (\mathbf{x} - \mathbf{x}_P) dV = \mathbf{0}$$

# Nomenclature

## Computational Cell

- Cell volume is denoted by  $V_P$
- For the cell, there is one neighbouring cell across each face. Neighbour cell and cell centre will be marked with  $N$ .
- The face centre  $f$  is defined in the equivalent manner, using the centroid rule:

$$\int_{S_f} (\mathbf{x} - \mathbf{x}_f) dS = \mathbf{0}$$

- Delta vector for the face  $f$  is defined as

$$\mathbf{d}_f = \overline{PN}$$

- Face area vector  $s_f$  is a surface normal vector whose magnitude is equal to the area of the face. The face is numerically never flat, so the face centroid and area are calculated from the integrals.

$$\mathbf{s}_f = \int_{S_f} \mathbf{n} dS$$

## Computational Cell

- The fact that the face centroid does not necessarily lay on the plane of the face is not worrying: we are dealing with surface-integrated quantities. However, we shall require the the cell centroid lays within the cell
- In practice, cell volume and face area calculated by decompositions into triangles and pyramids
- Types of faces in a mesh
  - **Internal face**, between two cells
  - **Boundary face**, adjacent to one cell only and pointing outwards of the computational domain
- When operating on a single cell, assume that all face area vectors  $s_f$  point outwards of cell  $P$
- Discretisation is based on the integral form of the transport equation over each cell

$$\int_V \frac{\partial \phi}{\partial t} dV + \oint_S \phi (\mathbf{n} \cdot \mathbf{u}) dS - \oint_S \gamma (\mathbf{n} \cdot \nabla \phi) dS = \int_V Q_v dV$$



# Spatial and Temporal Variation

## Spatial Variation

- Postulating spatial variation of  $\phi$ : second order discretisation in space

$$\phi(\mathbf{x}) = \phi_P + (\mathbf{x} - \mathbf{x}_P) \bullet (\nabla \phi)_P$$

This expression is given for each individual cell. Here,  $\phi_P = \phi(\mathbf{x}_P)$ .

## Temporal Variation

- Postulating linear variation in time: second order in time

$$\phi(t + \Delta t) = \phi^t + \Delta t \left( \frac{\partial \phi}{\partial t} \right)^t$$

where  $\phi^t = \phi(t)$

## Polyhedral Mesh Support

- In FVM, we have specified the “shape function” without reference to the actual cell shape (tetrahedron, prism, brick, wedge). The variation is always linear. Doing polyhedral Finite Volume should be straightforward!

## Evaluating Volume Integrals

$$\begin{aligned}\int_V \phi dV &= \int_V [\phi_P + (\mathbf{x} - \mathbf{x}_P) \bullet (\nabla \phi)_P] dV \\ &= \phi_P \int_V dV + (\nabla \phi)_P \bullet \int_V (\mathbf{x} - \mathbf{x}_P) dV \\ &= \phi_P V_P\end{aligned}$$

## Evaluating Surface Integrals

- Surface integral splits into a sum over faces and evaluates in the same manner

$$\begin{aligned}\oint_S \mathbf{n} \phi dS &= \sum_f \int_{S_f} \mathbf{n} \phi_f dS_f = \sum_f \int_{S_f} \mathbf{n} [\phi_f + (\mathbf{x} - \mathbf{x}_f) \bullet (\nabla \phi)_f] dS_f \\ &= \sum_f \mathbf{s}_f \phi_f\end{aligned}$$

- Assumption of linear variation of  $\phi$  and selection of  $P$  and  $f$  in the centroid creates second-order discretisation

# Software Organisation

## Polyhedral Mesh: Face Addressing

- Thinking about the above, all I want to do is to visit all cell faces and then all boundary faces. For internal face, do the operation and put the result into two cells around the face
- Orient face from  $P$  to  $N$ : add to  $P$  and subtract from  $N$  (because the face area vector points the wrong way)
- Addressing slightly different: for each internal face, record the left and right (owner and neighbour) cell index. Owner will be the first one in the cell list
- Much cleaner, compact addressing, fast and efficient (some cache hit issues are hidden but we can work on that)
- Most importantly, it no longer matters how many faces there is in the cell: nothing special is required for polyhedral cells

## Gauss' Theorem in Finite Volume Discretisation

- Gauss' theorem is a tool we will use for handing the volume integrals of divergence and gradient operators
- Divergence form

$$\int_{V_P} \nabla \cdot \mathbf{a} dV = \oint_{\partial V_P} d\mathbf{s} \cdot \mathbf{a}$$

- Gradient form

$$\int_{V_P} \nabla \phi dV = \oint_{\partial V_P} d\mathbf{s} \phi$$

- Note how the face area vector operates from the same side as the gradient operator: fits with our definition of te gradient of for a vector field
- In the rest of the analysis, we shall look at the problem face by face. A diagram of a face is given below for 2-D. Working with vectors will ensure no changes are required when we need to switch from 2-D to 3-D.

# Matrix Assembly

## From Discretisation to Linear System of Equations

- Assembling the terms from the discretisation method: Thus, the value of the solution in a point depends on the values around it: this is always the case. For each computational point, we will create an equation

$$a_P x_P + \sum_N a_N x_N = b$$

where N denotes the neighbourhood of a computational point

- Every time  $x_P$  depends on itself, add contribution into  $a_P$
- Every time  $x_N$  depends on itself, add contribution into  $a_N$
- Other contributions into  $b$

## Nomenclature

- Equations form a **linear system** or a matrix

$$[A][x] = [b]$$

where  $[A]$  contain matrix coefficients,  $[x]$  is the value of  $x_P$  in all cells and  $[b]$  is the right-hand-side

- $[A]$  is potentially very big: N cells  $\times$  N cells
- This is a **square matrix**: the number of equations equals the number of unknowns
- ... but very few coefficients are non-zero. The matrix connectivity is always local, potentially leading to storage savings if a good format can be found

# Implicit and Explicit Methods

## Solution Advancement Method

- **Explicit method:**  $x_P^n$  depends on the **old** neighbour values  $x_N^o$ 
  - Visit each cell, and using available  $x^o$  calculate
$$x_P^n = \frac{b - \sum_N a_N x_N^o}{a_P}$$
  - No additional information needed
  - Fast and efficient; however, poses the **Courant number limitation**: the information about boundary conditions is propagated very slowly and poses a limitation on the time-step size
- **Implicit method:**  $x_P^n$  depends on the **new** neighbour values  $x_N^n$

$$x_P^n = \frac{b - \sum_N a_N x_N^n}{a_P}$$

- Each cell value of  $x$  for the “new” level depends on others: all equations need to be solved simultaneously

## Discretising Operators

- **The Story So Far...**
  - Split the space into cells and time into time steps
  - Assembled a discrete description of a continuous field variable
  - Postulated spatial and temporal variation of the solution for second-order discretisation
  - Generated expressions for evaluation of volume and surface integrals
- We shall now use this to assemble the discretisation of the differential operators
  1. Rate of change term
  2. Gradient operator
  3. Convection operator
  4. Diffusion operators
  5. Source and sink terms



# Temporal Derivative

## First Derivative in Time

- Time derivative captures the rate-of-change of  $\phi$ . We only need to handle the volume integral.
- Defining time-step size  $\Delta t$
- $t_{new} = t_{old} + \Delta t$ , defining time levels  $\phi^n$  and  $\phi^o$

$$\phi^o = \phi(t = t_{old})$$

$$\phi^n = \phi(t = t_{new})$$

- Temporal derivative, first and second order approximation

$$\frac{\partial \phi}{\partial t} = \frac{\phi^n - \phi^o}{\Delta t}$$
$$\frac{\partial \phi}{\partial t} = \frac{\frac{3}{2}\phi^n - 2\phi^o + \frac{1}{2}\phi^{oo}}{\Delta t}$$

## First Derivative in Time

- Thus, with the volume integral:

$$\int_V \frac{\partial \phi}{\partial t} dV = \frac{\phi^n - \phi^o}{\Delta t} V_P$$

$$\int_V \frac{\partial \phi}{\partial t} dV = \frac{\frac{3}{2}\phi^n - 2\phi^o + \frac{1}{2}\phi^{oo}}{\Delta t} V_P$$

## Temporal Derivative

- Calculus: given  $\phi^n$ ,  $\phi^o$  and  $\Delta t$  create a field of the time derivative of  $\phi$
- Method: matrix representation. Since  $\frac{\partial \phi}{\partial t}$  in cell  $P$  depends on  $\phi_P$ , the matrix will only have a diagonal contribution and a source
  - Diagonal coefficient:  $a_P = \frac{V_P}{\Delta t}$
  - Source contribution:  $r_P = \frac{V_P \phi^o}{\Delta t}$

# Gradient Calculation

## Evaluating the Gradient

- How to evaluate a gradient of a given field: Gauss Theorem

$$\int_{V_P} \nabla \phi dV = \oint_{\partial V_P} dS \phi$$

- Discretised form splits into a sum of face integrals

$$\oint_S \mathbf{n} \phi dS = \sum_f \mathbf{s}_f \phi_f$$

- It still remains to evaluate the face value of  $\phi$ . Consistently with second-order discretisation, we shall assume linear variation between P and N

$$\phi_f = f_x \phi_P + (1 - f_x) \phi_N$$

where  $f_x = \overline{fN}/\overline{PN}$

- Gradient evaluation almost exclusively used as a calculus operation

## Other Forms of Gradient: Least Square

- Consider cell centre  $P$  and a cluster of points around it  $N$ . Fit a plane:

$$e_N = \phi_N - (\phi_P + \mathbf{d}_N \cdot (\nabla \phi)_P)$$

- Minimising the weighted error

$$e_P^2 = \sum_N (w_N e_N)^2 \text{ where } w_N = \frac{1}{|\mathbf{d}_N|}$$

yields a second-order **least-square form of gradient**:

$$(\nabla \phi)_P = \sum_N w_N^2 \mathbf{G}^{-1} \cdot \mathbf{d}_N (\phi_N - \phi_P)$$

- $\mathbf{G}$  is a  $3 \times 3$  symmetric matrix:

$$\mathbf{G} = \sum_N w_N^2 \mathbf{d}_N \mathbf{d}_N$$

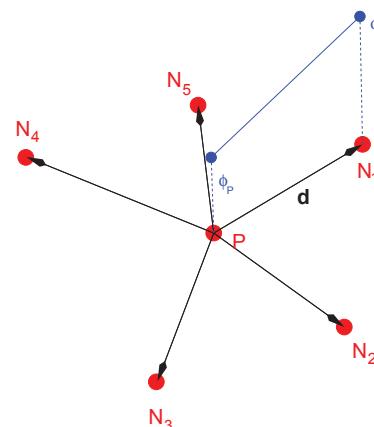


## Other Forms of Gradient: Cell- and Face-Limited Gradient

- Gradient reconstruction may lead to local over- or under-shoots in reconstructed field:

$$\min_N(\phi_N) \leq \phi_P + \mathbf{d}_N \cdot (\nabla \phi)_P \leq \max_N(\phi_N)$$

- This is important for bounded variables, especially when gradients are used in further discretisation or coupling terms
- Solution: based on the gradient, calculate min and max neighbourhood value and apply gradient limiter to preserve bounds in cell centres



## Convection Operator

- Convection term captures the transport by convective velocity
- Convection operator splits into a sum of face integrals (integral and differential form)

$$\oint_S \phi(\mathbf{n} \cdot \mathbf{u}) dS = \int_V \nabla \cdot (\phi \mathbf{u}) dV$$

- Integration follows the same path as before

$$\oint_S \phi(\mathbf{n} \cdot \mathbf{u}) dS = \sum_f \phi_f (\mathbf{s}_f \cdot \mathbf{u}_f) = \sum_f \phi_f F$$

where  $\phi_f$  is the face value of  $\phi$  and

$$F = \mathbf{s}_f \cdot \mathbf{u}_f$$

is the **face flux**: measure of the flow through the face

- In order to close the system, we need a way of evaluating  $\phi_f$  from the cell values  $\phi_P$  and  $\phi_N$ : **face interpolation**



# Face Interpolation

## Face Interpolation Scheme for Convection

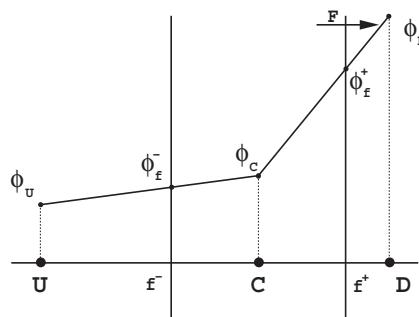
- Simplest face interpolation: **central differencing**. Second-order accurate, but causes oscillations

$$\phi_f = f_x \phi_P + (1 - f_x) \phi_N$$

- Upwind differencing: taking into account the **transportive property** of the term: information comes from upstream. No oscillations, but smears the solution

$$\phi_f = \max(F, 0) \phi_P + \min(F, 0) \phi_N$$

- There exists a large number of schemes, trying to achieve good accuracy without causing oscillations: e.g. TVD, and NVD families:  $\phi_f = f(\phi_P, \phi_N, F, \dots)$



## Convection Discretisation

- In the convection term,  $\phi_f$  depends on the values of  $\phi$  in two computational points:  $P$  and  $N$ .
- Therefore, the solution in  $P$  will depend on the solution in  $N$  and vice versa, which means we've got an **off-diagonal coefficient** in the matrix. In the case of central differencing on a uniform mesh, a contribution for a face  $f$  is
  - Diagonal coefficient:  $a_P = \frac{1}{2}F$
  - Off-diagonal coefficient:  $a_N = \frac{1}{2}F$
  - Source contribution: in our case, nothing. However, some other schemes may have additional (gradient-based) correction terms
  - Note that, in general the  $P$ -to- $N$  coefficient will be different from the  $N$ -to- $P$  coefficient: the matrix is asymmetric
- Upwind differencing
  - Diagonal coefficient:  $a_P = \max(F, 0)$
  - Off-diagonal coefficient:  $a_N = \min(F, 0)$

# Diffusion Operator

## Diffusion Operator

- Diffusion term captures the gradient transport
- Integration same as before

$$\oint_S \gamma(\mathbf{n} \cdot \nabla \phi) dS = \sum_f \int_{S_f} \gamma(\mathbf{n} \cdot \nabla \phi) dS$$

$$= \sum_f \gamma_f \mathbf{s}_f \cdot (\nabla \phi)_f$$

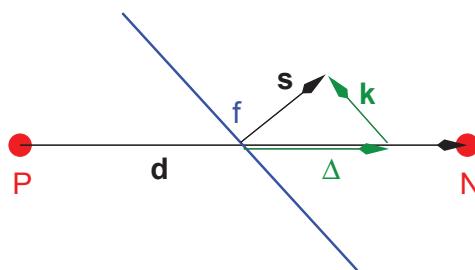
- $\gamma_f$  evaluated from cell values using central differencing
- Evaluation of the face-normal gradient. If  $\mathbf{s}$  and  $\mathbf{d}_f = \overline{PN}$  are aligned, use difference across the face

$$\mathbf{s}_f \cdot (\nabla \phi)_f = |\mathbf{s}_f| \frac{\phi_N - \phi_P}{|\mathbf{d}_f|}$$

- This is the component of the gradient in the direction of the  $\mathbf{d}_f$  vector
- For non-orthogonal meshes, a correction term may be necessary

## Matrix Coefficients

- For an orthogonal mesh, a contribution for a face  $f$  is
  - Diagonal value:  $a_P = -\gamma_f \frac{|\mathbf{s}_f|}{|\mathbf{d}_f|}$
  - Off-diagonal value:  $a_N = \gamma_f \frac{|\mathbf{s}_f|}{|\mathbf{d}_f|}$
  - Source contribution: for orthogonal meshes, nothing. Non-orthogonal correction will produce a source
- The  $P$ -to- $N$  and  $N$ -to- $P$  coefficients are identical: **symmetric matrix**. This is an important characteristic of the diffusion operator
- For **non-orthogonal meshes**, a correction is added to compensate for the angle between the face area and  $\overline{PN}$  vectors



# Diffusion Operator

## Limiting Non-Orthogonal Correction in a Laplacian

- Decomposition of face gradient into “orthogonal component” and “non-orthogonal correction” depends on mesh quality: mesh non-orthogonality is measured from  $\overline{PN}$  and  $\mathbf{s}_f$
- Mathematically, a Laplacian is a perfect operator: smooth, bounded, self-adjoint. Its discretisation yields a symmetric matrix
- In contrast, non-orthogonal correction is explicit, unbounded and unsigned
- Limited non-orthogonal correction: explicit part clipped to be smaller than its implicit counterpart, base on the current solution

$$\lambda \frac{|\mathbf{s}_f|}{|\mathbf{d}_f|} (\phi_N - \phi_P) > \mathbf{k}_f \cdot \nabla(\phi)_f$$

where  $\lambda$  is the limiter value

## Source and Sinks

- Source and sink terms are local in nature

$$\int_V q_v \, dV = q_v V_P$$

- In general,  $q_v$  may be a function of space and time, the solution itself, other variables and can be quite complex. In complex physics cases, the source term can carry the main interaction in the system. Example: complex chemistry mechanisms. We shall for the moment consider only a simple case.
- Typically, linearisation with respect to  $\phi$  is performed to promote stability and boundedness

$$q_v(\phi) = q_u + q_d\phi$$

where  $q_d = \frac{\partial q_v(\phi)}{\partial \phi}$  and for cases where  $q_d < 0$  (sink), treated separately

## Matrix Coefficients

- Source and sink terms do not depend on the neighbourhood
  - Diagonal value created for  $q_d < 0$ : “boosting diagonal dominance”
  - Explicit source contribution:  $q_u$



# Numerical Boundary Conditions

## Implementation of Boundary Conditions

- Boundary conditions will contribute to the discretisation through the prescribed boundary behaviour
- Boundary condition is specified for the whole equation
- ... but we will study them term by term to make the problem simpler



## Dirichlet Condition: Fixed Boundary Value

- Boundary condition specifies  $\phi_f = \phi_b$
- Convection term: fixed contribution  $F \phi_b$ . Source contribution only
- Diffusion term: need to evaluate the near-boundary gradient

$$\mathbf{n} \cdot (\nabla \phi)_b = \frac{\phi_b - \phi_P}{|\mathbf{d}_b|}$$

This produces a source and a diagonal contribution

- What about source, sink, rate of change?

# Neumann Condition

## Neumann and Gradient Condition

- Boundary condition specifies the near-wall gradient  $\mathbf{n} \cdot (\nabla \phi)_b = g_b$
- Convection term: evaluate the boundary value of  $\phi$  from the internal value and the known gradient

$$\phi_b = \phi_P + \mathbf{d}_b \cdot (\nabla \phi)_b = \phi_P + |\mathbf{d}_b| g_b$$

Use the evaluated boundary value as the face value. This creates a source and a diagonal contribution

- Diffusion term: boundary-normal  $g_b$  gradient can be used directly. Source contribution only

## Mixed Condition

- Combination of the above
- Very easy:  $\alpha$  times Dirichlet plus  $(1 - \alpha)$  times Neumann

## Geometric and Coupled Conditions

- Symmetry plane condition enforces using the mirror-image of internal solution
- Cyclic and periodic boundary conditions couple near-boundary cells to cells on another boundary

## Advancing the Solution in Time

- Two basic types of time advancement: Implicit and explicit schemes. Properties of the algorithm critically depend on this choice, but both are useful under given circumstances
- There is a number of methods, with slightly different properties, e.g. fractional step methods,
- Temporal accuracy depends on the choice of scheme and time step size
- Steady-state simulations
  - If equations are linear, this can be solved in one go! (provided the discretisation is linear as well!)
  - For non-linear equations or special discretisation practices, **relaxation methods** are used, which show characteristics of time integration (we are free to re-define the meaning of time)

# Time Advancement

## Explicit Schemes

- The algorithm uses the calculus approach, sometimes said to operate on residuals
- In other words, the expressions are evaluated using the currently available  $\phi$  and the new  $\phi$  is obtained from the time term
- **Courant number limit** is the major limitation of explicit methods: information can only propagate at the order of cell size; otherwise the algorithm is unstable
- Quick and efficient, no additional storage
- Very bad for elliptic behaviour

## Implicit Schemes

- The algorithm is based on the method: each term is expressed in matrix form and the resulting linear system is solved
- A new solution takes into account the new values in the complete domain: ideal for elliptic problems
- Implicitness removed the Courant number limitation: we can take larger time-steps
- Substantial additional storage: matrix coefficients!

## Assembling Equations

- The equation we are trying to solve is simply a collection of terms: therefore, assemble the contribution from
- **Initial condition.** Specifies the initial distribution of  $\phi$
- ... and we are ready to look at examples!

## Examples: Convection Differencing Schemes

- Testing differencing schemes on standard profiles
- Simple second-order discretisation: upwind differencing, central differencing, blended differencing, NVD schemes
- First-order scheme: Upwind differencing. Take into account the transport direction
- Review: Best Practice discretisation guidelines for industrial simulations

## Examples: Stability and Boundedness

- Positive and negative diffusion coefficient
- Temporal discretisation: first and second-order, implicit or explicit discretisation

# Summary

- OpenFOAM is an Open Source Computational Continuum Mechanics library with substantial capabilities. Users are provided with full source code access and execution of simulations is license-free
- The code is written in a legible and consistent way, using object-oriented programming in C++
- Basic discretisation machinery is a second-order accurate Finite Volume Method (in space and time), with support for polyhedral meshes, mesh deformation and topological changes
- Parallel execution is achieved in a domain decomposition mode, using message passingg



# OpenFOAM: Linear System and Linear Solver

Hrvoje Jasak

[hrvoje.jasak@fsb.hr](mailto:hrvoje.jasak@fsb.hr), [h.jasak@wikki.co.uk](mailto:h.jasak@wikki.co.uk)

University of Zagreb, Croatia and  
Wikki Ltd, United Kingdom



OpenFOAM in Industrial Combustion Simulations, Pohang University Feb/2015

OpenFOAM: Linear System and Linear Solver – p. 1

## Matrix Assembly

From Discretisation to Linear System

- Assembling the terms from the discretisation method
  - Time derivative:  $x$  depends on old value
  - Convection:  $\mathbf{u}$  provided;  $x_f$  depends on  $x_P$  and  $x_N$
  - Diffusion:  $\mathbf{s}_f \cdot (\nabla x)_f$  depends on  $x_P$  and  $x_N$
- Thus, the value of the solution in a point depends on the values around it: this is always the case. For each computational point, we will create an equation

$$a_P x_P + \sum_N a_N x_N = R$$

where N denotes the neighbourhood of a computational point

- Every time  $x_P$  depends on itself, add contribution into  $a_P$
- Every time  $x_N$  depends on itself, add contribution into  $a_N$
- Other contributions into  $R$

## Solution Advancement Method

- **Explicit method:**  $x_P^n$  depends on the **old** neighbour values  $x_N^o$ 
  - Visit each cell, and using available  $x^o$  calculate
$$x_P^n = \frac{R - \sum_N a_N x_N^o}{a_P}$$
  - No additional information needed
  - Fast and efficient; however, poses the **Courant number limitation**: the information about boundary conditions is propagated very slowly and poses a limitation on the time-step size
- **Implicit method:**  $x_P^n$  depends on the **new** neighbour values  $x_N^n$

$$x_P^n = \frac{R - \sum_N a_N x_N^n}{a_P}$$

- Each cell value of  $x$  for the “new” level depends on others: all equations need to be solved simultaneously

# Linear System of Equations

## Nomenclature

- Equations form a **linear system** or a matrix

$$[A][x] = [b]$$

where  $[A]$  contain matrix coefficients,  $[x]$  is the value of  $x_P$  in all cells and  $[b]$  is the right-hand-side

- $[A]$  is potentially very big: N cells  $\times$  N cells
- This is a **square matrix**: the number of equations equals the number of unknowns
- ... but very few coefficients are non-zero. The matrix connectivity is always local, potentially leading to storage savings if a good format can be found
- What about non-linearity? Typically, we wish to avoid handling non-linearity at this level due to high cost of non-linear matrix solvers

## Matrix Format

- **Dense matrix format.** All matrix coefficients have are stored, typically in a two-dimensional array
  - Diagonal coefficients:  $a_{ii}$ , off-diagonal coefficients:  $a_{ij}$
  - Convenient for small matrices and direct solver use
  - Matrix coefficients represent a large chunk of memory: efficient operations imply memory management optimisation
  - It is impossible to say if the matrix is symmetric or not without floating point comparisons
- **Sparse matrix format.** Only non-zero coefficients will be stored
  - Considerable savings in memory
  - Need a mechanism to indicate the position of non-zero coefficients
  - This is **static format**, which imposes limitations on the operations: if a coefficient is originally zero, it is very expensive to set its value: recalculating the format. This is usually termed a **zero fill-in** condition
  - Searching for coefficients is out of the question: need to formulate sparse matrix algorithms

# Matrix Storage Formats

## Sparse Matrix: Compressed Row Format

- Operate on a row-by-row basis. Diagonal coefficients may be separate
- Coefficients stored in a single 1-D array. Coefficients ordered row-by-row
- Addressing in two arrays: “row start” and “column”
- The column array records the column index for each coefficients. Size of column array equal to the number of off-diagonal coefficients
- The row array records the start and end of each row in the column array. Thus, row  $i$  has got coefficients from  $\text{row}[i]$  to  $\text{row}[i + 1]$ . Size of row arrays equal to number of rows + 1

```
vectorProduct(b, x) // [b] = [A] [x]
{
    for (int n = 0; n < count; n++)
    {
        for (int ip = row[n]; ip < row[n+1]; ip++)
        {
            b[n] = coeffs[ip]*x[col[ip]];
        }
    }
}
```

## Sparse Matrix: Arrow Format

- Arbitrary sparse format. Diagonal coefficients typically stored separately
  - Coefficients in 2-3 arrays: diagonal, upper and lower triangle
  - Diagonal addressing implied
  - Off-diagonal addressing in 2 arrays: “owner” (row index) “neighbour” (column index) array. Size of addressing equal to the number of coefficients
  - The matrix structure (fill-in) is assumed to be symmetric: presence of  $a_{ij}$  implies the presence of  $a_{ji}$ . Symmetric matrix easily recognised
  - If the matrix coefficients are symmetric, only the upper triangle is stored – a symmetric matrix is easily recognised and stored only half of coefficients
- ```
vectorProduct(b, x) // [b] = [A] [x]
{
    for (int n = 0; n < coeffs.size(); n++)
    {
        int c0 = owner(n);
        int c1 = neighbour(n);
        b[c0] = upperCoeffs[n]*x[c1];
        b[c1] = lowerCoeffs[n]*x[c0];
    }
}
```



# Matrix Format and Discretisation

## FVM and Matrix Structure

- Relationship between the FV mesh and a matrix:
  - A cell value depends on other cell values only if the two cells share a face. Therefore, a correspondence exists between the off-diagonal matrix coefficients and the mesh structure
  - In practice, the matrix is assembled by looping through the mesh
- Finite Element matrix assembly
  - Connectivity depends on the shape function and point-to-cell connectivity in the mesh
  - In assembly, a local matrix is assembled and then inserted into the global matrix
  - Clever FEM implementations talk about the kinds of assembly without the need for searching: a critical part of the algorithm



## Matrix Characterisation

- We shall introduce a set of terms to describe a matrix in general terms
- A matrix is **sparse** if it contains only a few non-zero elements
- A sparse matrix is **banded** if its non-zero coefficients are grouped in a stripe around the diagonal
- A sparse matrix has a **multi-diagonal structure** if its non-zero off-diagonal coefficients form a regular diagonal pattern
- A **symmetric** matrix is equal to its transpose

$$[A] = [A]^T$$

- A matrix is **positive definite** if for every  $[x] \neq [0]$

$$[x]^T [A] [x] > 0$$

## Matrix Characterisation

- A matrix is **diagonally dominant** if in each row the sum of off-diagonal coefficient magnitudes is equal or smaller than the diagonal coefficient

$$a_{ii} \geq \sum_{j=1}^N |a_{ij}| ; j \neq i$$

and for at least one  $i$

$$a_{ii} > \sum_{j=1}^N |a_{ij}| ; j \neq i$$

## Definition of a Residual

- Matrix form of the system we are trying to solve is

$$[A][x] = [b]$$

- The exact solution can be obtained by inverting the matrix  $[A]$ :

$$[x] = [A]^{-1} [b]$$

This is how direct solvers operate: number of operations required for the inversion of  $[A]$  is fixed and until the inverse is constructed we cannot get  $[x]$

- Iterative solvers start from an approximate solution  $[x]_0$  and generates a set of solution estimates  $[x]_k$ , where  $k$  is the iteration counter
- Quality of the solution estimate is measured through a **residual**:

$$[r] = [b] - [A][x]_k$$

Residual is a vector showing how far is the current estimate  $[x]_k$  from the exact solution  $[x]$ . Note that for  $[x]$ ,  $[r]$  will be zero



# Matrix Properties

## Definition of a Residual

- $[r]$  defines a value for every equation (row) in  $[A]$ : we need a better way to measure it. A residual norm  $\|r\|$  can be assembled in many ways, but usually

$$\|r\| = \sum_{j=1}^N |r_j|$$

In CFD software, the residual norm is normalised further for easier comparison between the equations etc.

- Convergence of the iterative solver is usually measured in terms of residual reduction. When

$$\frac{\|r_k\|}{\|r_0\|} < \epsilon$$

the linear system of equations is considered to be solved

## The Role of a Linear Solver

- Good (implicit) numerical simulation software will spend 50 – 90% percent of CPU time inverting matrices: performance of linear solvers is absolutely critical for the performance of the solver
- Like in the case of mesh generation, we will couple the characteristics of a discretisation method and the solution algorithm with the linear solver
- Only a combination of a discretisation method and a linear solver will result in a useful solver. Typically, properties of discretisation will be set up in a way that allows the choice of an efficient solver

## Solution Approach

- **Direct solver.** The solver algorithm will perform a given number of operations, after which a solution will be obtained
- **Iterative solver.** The algorithm will start from an initial solution and perform a number of operations which will result in an improved solution. Iterative solvers may be variants of the direct solution algorithm with special characteristics
- **Explicit method.** New solution depends on currently available values of the variables. The matrix itself is not required or assembled; in reality, the algorithm reduces to point-Jacobi or Gauss-Seidel sweeps

# Choice of Linear Solver

## Direct or Iterative Solver

- Direct solvers: expensive in storage and CPU time but can handle any sort of matrix – no need to worry about matrix properties during discretisation
- Iterative solvers: work by starting from an initial guess and improving the solution. However, require matrices with “special” properties
- For large problems, iterative solvers are the only option
- Fortunately, the FVM matrices are ideally suited (read: carefully constructed) for use with iterative solvers
- Direct solver is typically used for cases where it is difficult to control matrix properties through discretisation: high-order FEM methods, Hermitian elements, Discontinuous Galerkin etc.

## Partial Convergence

- When we are working on linear problems with linear discretisation in steady-state, the solution algorithm will only use a single solver call. This is very quick and very rare: linear systems are easy to simulate
- Example: linear stress analysis. In some FEM implementations, for matrices under a certain size the direct solver will be used exclusively for matrices under a given size
- In cases of coupled or non-linear partial differential equations, the solution algorithm will iterate over the non-linearity. Therefore, intermediate solution will only be used to update the non-linear parameters.
- With this in mind, we can choose to use partial convergence, update the non-linearity and solve again: capability of obtaining an intermediate solution at a fraction of the cost becomes beneficial
- Moreover, in iterative procedures or time-marching simulations, it is quite easy to provide a good initial guess for the new solution: solution from the previous iteration or time-step. This further improves the efficiency of the algorithm
- Historically, in partial convergence cases, FEM solvers use tighter tolerances than FVM: 6 orders of magnitude for FEM vs. 1-2 orders of magnitude for the FVM

# Direct Solver

## Properties of Direct Solvers

- The most important property from the numerical point of view is that the number of operations required for the solution is known and intermediate solutions are of no interest
- **Matrix fill-in.** When operating on a large sparse matrix like the one from discretisation methods, the direct solver will create entries for coefficients that were not previously present. As a consequence, formal matrix storage requirement for a direct solver is a full matrix for a complete system: huge! This is something that needs to be handled in a special way
- Advantage of direct solvers is that they can handle any sort of well-posed linear system
- In reality, we additionally have to worry about pollution by the round-off error. This is partially taken into account through the details of the solution algorithm, but for really bad matrices this cannot be helped

## Gaussian Elimination

- Gaussian elimination is the easiest direct solver: standard mathematics. Elimination is performed by combining row coefficients until a matrix becomes triangular. The elimination step is followed by backwards substitution to obtain the solution.
- **Pivoting:** in order to control the discretisation error, equations are chosen for elimination based on the central coefficient
- Combination of matrix rows leads to fill in
- Gaussian elimination is one of the cases of I-L-U decomposition solvers and is rarely used in practices
- The number of operations in direct solvers scales with the number of equations cubed: very expensive!

## Multi-Frontal Solver

- When handling very sparse systems, the fill-in is very problematic: leads to a large increase in storage size and accounts for the bulk of operations
- **Window approach:** modern implementation of direct solvers
  - Looking at the structure of the sparse system, it can be established that equation for  $x_P$  depends only on a small subset of other nodes: in principle, it should be possible to eliminate the equation for P just by looking at a small subset of the complete matrix
  - If all equations under elimination have overlapping regions of zero off-diagonal coefficients, there will be no fill-in in the shared regions of zeros!
  - Idea: Instead of operating on the complete matrix, create an active window for elimination. The window will sweep over the matrix, adding equations one by one and performing elimination immediately
  - The window matrix will be dense, but much smaller than the complete matrix. The triangular matrix (needed for back-substitution) can be stored in a sparse format
- The window approach may reduce the cost of direct solvers by several orders of magnitude: acceptable for medium-sized systems. The number of operations scales roughly with  $N M^2$ , where N is the number of equations and M is the maximum size of the solution window

## Implementing Direct Solvers

- The first step in the implementation is control of the window size: the window changes its width dynamically and in the worst case may be the size of the complete matrix
- Maximum size of the window depends on the matrix connectivity and ordering of equation. Special optimisation software is used to control the window size: matrix renumbering and ordering heuristics
- Example: ordering of a Cartesian matrix for minimisation of the band
- Most expensive operation in the multi-frontal solver is the calculation of the Schur's complement: the difference between the trivial and optimised operation can be a factor of 10000! In practice, you will not attempt this (cache hit rate and processor-specific pre-fetch operations)
- **Basic Linear Algebra (BLAs)** library: special assembly code implementation for matrix manipulation. Code is optimised by hand and sometimes written specially for processor architecture. It is unlikely that a hand-written code for the same operation achieves more than 10 % efficiency of BLAs. A good implementation can now be measured in how much the code spends on operations outside of BLAs.

# Simple Iterative Solvers

## Simple Iterative Solvers: Fixed-Point Methods

- Performance of iterative solvers depends on the **matrix characteristics**. The solver operates by incrementally improving the solution, which leads to the concept of error propagation: if the error is augmented in the iterative process, the solver diverges
- The easiest way of analysing the error is in terms of **eigen-spectrum** of the matrix
- The general idea of iterative solvers is to replace  $[A]$  with a matrix that is easy to invert and approximates  $[A]$  and use this to obtain the new solution
  - **Point-Jacobi** solution
  - **Gauss-Seidel** solver
  - Tri-diagonal system and generalisation to 5- or 7-diagonal matrices
- **Propagation of information** in simple iterative solvers. Point Jacobi propagates the “data” one equation at a time: very slow. For Gauss-Seidel, the information propagation depends on the matrix ordering and sweep direction. In practice **forward** and **reverse** sweeps are alternated

## Mathematical Formalism for Fixed-Point Methods

- Consider again linear problem  $[A][x] = [b]$
- A **stationary iterative method** is obtained by splitting  $[A] = [M] - [N]$ :

$$[x]^{(\nu+1)} = [R][x]^{(\nu)} + [M]^{-1}[b]$$

- Here  $[R]$  is the iteration matrix

$$[R] = [M]^{-1}[N]$$

- Define solution error  $[e]$  and error propagation equation:

$$\begin{aligned}[e]^{(\nu)} &= [x]^{(\nu)} - [x]^* \\ [e]^{(\nu+1)} &= [R][e]^{(\nu)}\end{aligned}$$

- Iteration matrix possesses the recursive property:

$$[e]^{(\nu+1)} = [R]^\nu [e]^{(0)} \quad (1)$$



# Krylov Space Solvers

## Krylov Subspace Methods

- Looking at the direct solver, we can imagine that it operates in N-dimensional space, where N is the number of equations and searches for a point which minimises the residual
- In Gaussian elimination, we will be visiting each direction of the N-dimensional space and eliminating it from further consideration
- The idea of Krylov space solvers is that an approximate solution can be found more efficiently if we look for search directions more intelligently. A residual vector  $[r]$  at each point contains the “direction” we should search in; additionally, we would like to always search in a direction orthogonal to all previous search directions
- On their own, Krylov space solvers are poor; however, when **matrix preconditioning** is used, we can assemble efficient methods. This is an example of an iterative roughener
- In terms of performance, the number of operations in Krylov space solvers scales with  $N \log(N)$ , where N is the number of unknowns

## Mathematical Formalism for Krylov Space Solvers

- The Conjugate Gradient (CG) solver is an orthogonal projection technique onto the Krylov space  $\mathcal{K}([A], [r]^{(0)})$ :

$$\mathcal{K}([A], [r]^{(0)}) = \text{span}([r]^{(0)}, [A][r]^{(0)}, \dots, [A]^\nu[r]^{(0)})$$

- Conjugate Gradient algorithm

```

CG([A], [x], [b]):
[r]^(0) = [b] - [A][x]^(0), [p]^(0) = [r]^(0)
for j = 0, 1, ...
    αj = ([r](j), [r](j)) / ([A][p](j), [p](j))
    [x](j+1) = [x](j) + αj[p](j)
    [r](j+1) = [r](j) - αj[A][p](j)
    βj = ([r](j+1), [r](j+1)) / ([r](j), [r](j))
    [p](j+1) = [r](j+1) + βj[p](j)
end

```

## Mathematical Formalism for Krylov Space Solvers

- CG solver seeks the solution in the Krylov space in the following form:

$$[x]^{(\nu+1)} = [x]^{(0)} + \alpha_\nu[p]^{(\nu)}$$

- Auxiliary vectors  $[p]^{(\nu)}$  are chosen to have the following property:

$$([A][p]^{(\nu)}, [p]^{(\nu)}) = 0$$

- Here, symbol  $(\cdot, \cdot)$  is the scalar product of two vectors

Preconditioner  $[M]$  is a matrix which approximates  $[A]$  such that equation

$$[M][x] = [b]$$

may be inexpensive to solve. Then, we solve the following:

$$\begin{aligned} [M]^{-1}[A][x] &= [M]^{-1}[b] \\ [A][M]^{-1}[u] &= [b], \quad [x] = [M]^{-1}[u] \end{aligned}$$

```

PCG([A], [x], [b]):
[r]^(0) = [b] - [A][x]^(0), [z]^(0) = [M]^{-1}[r]^(0), [p]^(0) = [z]^(0)
for j = 0, 1, ...
    α_j = ([r]^(j), [z]^(j)) / ([A][p]^(j), [p]^(j))
    [x]^(j+1) = [x]^(j) + α_j[p]^(j)
    [r]^(j+1) = [r]^(j) - α_j[A][p]^(j)
    z^(j+1) = [M]^{-1}[r]^(j+1)
    β_j = ([r]^(j+1), [z]^(j+1)) / ([r]^(j), [z]^(j))
    [p]^(j+1) = [z]^(j+1) + β_j[p]^(j)
end

```



## Algebraic Multigrid

### Basic Idea of Multigrid

- Mathematical analysis of discretisation shown it makes sense to use coarse-mesh solutions to accelerate the solution process on the fine mesh, through initialisation and coarse correction: Multigrid
- In terms of matrices and linear solvers, the same principle should apply: our matrices come from discretisation! However, it would be impractical to build a series of coarse meshes just to solve a system of linear equations
- We can readily recognise that all the information about the coarse mesh (and therefore the coarse matrix) already exists in the fine mesh

Can we do the same with a linear equation solver?: Algebraic Multigrid (AMG)

- Operation of a multigrid solver relies on the fact that a **high-frequency error** is easy to eliminate: consider the operation of the Gauss-Seidel algorithm
- Once the high-frequency error is removed, iterative convergence slows down. At the same time, the error that looks smooth on the current mesh will behave as high-frequency on a coarser mesh
- If the mesh is coarser, the error is both eliminated faster and in fewer iterations.
- Thus, in multigrid the solution is mapped through a series of coarse levels, each of the levels being responsible for a “band” of error

## Algebraic Multigrid Solver: Main Ingredients

- Restriction operator:  $[R]_n^{n+1}$
- Prolongation operator:  $[P]_{n+1}^n$
- Single-level smoother
- Type of multigrid cycle

## Construction of a Coarse Matrix

- Coarse matrix  $[A]^{n+1}$  is constructed through projection:

$$[A]^{n+1} = [R]_n^{n+1} [A]^n [P]_{n+1}^n$$

## Algebraic Multigrid Solver: $\mu$ -Cycle

- Number of pre-sweeps ( $\nu_1$ )
- Number of post-sweeps ( $\nu_2$ )
- Smoother type
- Prolongation ( $[P]_{n+1}^n$ ) and restriction operators ( $[R]_n^{n+1}$ )
- Type of the cycle ( $\mu$ )



# Algebraic Multigrid Solver

## Algebraic Multigrid Solver: $\mu$ -Cycle

$\mu$ -Cycle( $[x]^n, [r]^n$ ):  
Create multigrid levels:  
 $[A]^n, [R]_n^{n+1}, [P]_{n+1}^n, n = 0, 1, 2, \dots, N - 1$   
for  $n = 0$  to  $N - 1$   
     $\nu_1$  pre-smoothing sweeps:  
        solve  $[A]^n[x]^n = [b]^n, [r]^n = [b]^n - [A]^n[x]^n$   
        if  $n \neq N - 1$   
             $[b]^{n+1} = [R]_n^{n+1}[r]^n$   
             $[x]^{n+1} = 0$   
             $[x]^{n+1} = \mu$ -Cycle( $[x]^{n+1}, [r]^{n+1}$ )  $\mu$  times  
            Correct  $[x]_{new}^n = [x]^n + [P]_{n+1}^n[x]^{n+1}$   
        end  
     $\nu_2$  post-smoothing sweeps:  
        solve  $[A]^n[x]_{new}^n = [b]^n$   
end



## Algebraic Multigrid Operations

- **Matrix coarsening.** This is roughly equivalent to creation of coarse mesh cells. Two main approaches are:
  - **Aggregative multigrid (AAMG).** Equations are grouped into clusters in a manner similar to grouping fine cells to form a coarse cell. The grouping pattern is based on the strength of off-diagonal coefficients
  - **Selective multigrid (SAMG).** In selective multigrid, the equations are separated into two groups: the **coarse** and **fine** equations. Selection rules specifies that no two coarse points should be connected to each other, creating a maximum possible set. Fine equations form a fine-to-coarse interpolation method (restriction matrix),  $[R]$ , which is used to form the coarse system.
- **Restriction of residual** handles the transfer of information from fine to coarse levels. A fine residual, containing the smooth error component, is restricted and used as the r.h.s. (right-hand-side) of the coarse system.
- **Prolongation of correction.** Once the coarse system is solved, coarse correction is prolongated to the fine level and added to the solution. Interpolation introduces **aliasing errors**, which can be efficiently removed by smoothing on the fine level.

# Algebraic Multigrid

## Algebraic Multigrid Operations

- **Multigrid smoothers.** The bulk of multigrid work is performed by transferring the error and correction through the multigrid levels. Smoothers only act to remove high-frequency error: simple and quick. Smoothing can be applied on each level:
  - Before the restriction of the residual, called **pre-smoothing**
  - After the coarse correction has been added, called **post-smoothing**
- Algorithmically, post-smoothing is more efficient
- **Cycle types.** Based on the above, AMG can be considered a two-level solver. In practice, the “coarse level” solution is also assembled using multigrid, leading to multi-level systems.
- The most important multigrid cycle types are
  - **V-cycle:** residual reduction is performed all the way to the coarsest level, followed by prolongation and post-smoothing. Mathematically, it is possible to show that the V-cycle is optimal and leads to the solution algorithm where the number of operations scales linearly with the number of unknowns
  - **Flex cycle.** Here, the creation of coarse levels is done on demand, when the smoother stops converging efficiently
- Other cycles, e.g. W-cycle or F-cycle are a variation on the V-cycle theme



# Finite Volume Discretisation in OpenFOAM

## Best Practice Guidelines

Hrvoje Jasak

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

University of Zagreb, Croatia and

Wikki Ltd, United Kingdom



OpenFOAM in Industrial Combustion Simulations, Pohang University Feb/2015

Finite Volume Discretisation in OpenFOAM – p. 1

## Outline

### Objective

- Review the best practice guidelines for the Finite Volume (FV) discretisation in OpenFOAM and compare it with commercial CFD solvers
  - Background on discretisation
  - Default settings on dominantly hex and dominantly tet meshes

### Topics

- Background
- Discretisation requirements: gradient scheme; convection; diffusion
- Proposed default settings: hexahedral meshes
- Proposed default settings: tetrahedral meshes
- Summary

## Best Practice Guidelines in CFD

- Commercial CFD codes offer robust set of default settings for the FVM: make the code run on a bad mesh and by inexpert users
- **Priority is in producing a result:** substantial improvements in solution quality and accuracy is possible
- ... but only for an expert user!
- Default settings are extremely important and change only after large validation and robustness testing campaigns

## Default Settings in OpenFOAM

- ...are practically non-existent: the code is written by experts and defaults are changed on a whim
- Some tutorials have settings appropriate for the case, but not recommended in general
- To remedy this, we need automatic test loops with 5000+ validation cases
- Improvements are in the pipeline: community effort and validation harness

# Discretisation

## Finite Volume Discretisation

- Main concerns of FVM accuracy are mesh structure and quality and choice of discretisation schemes
- Mesh structure determines the choice of appropriate gradient calculation algorithm
- For transport of bounded scalars, it is essential to use bounded differencing schemes: both for convection and diffusion

## Gauss Gradient Scheme

- Gradient calculated using integrals over faces

$$\int_{V_P} \nabla \phi \, dV = \oint_{\partial V_P} d\mathbf{s} \, \phi = \sum_f \mathbf{s}_f \phi_f$$

- Evaluate the face value of  $\phi$  from cell centre values

$$\phi_f = f_x \phi_P + (1 - f_x) \phi_N$$

where  $f_x = \overline{fN}/\overline{PN}$

- Expression is second-order accurate only if  $\phi_f$  is the face centre value
- Accurate on hexahedral meshes, but loses accuracy on tetrahedra: large skewness error



# Gradient Scheme

## Least Squares Gradient: Second Order Accuracy On All Meshes

- Consider cell centre  $P$  and a cluster of points around it  $N$ . Fit a plane:

$$e_N = \phi_N - (\phi_P + \mathbf{d}_N \cdot (\nabla \phi)_P)$$

- Minimising the weighted error: second-order accuracy on all meshes

$$e_P^2 = \sum_N (w_N e_N)^2 \text{ where } w_N = \frac{1}{|\mathbf{d}_N|}$$

yields a second-order **least-square form of gradient**:

$$(\nabla \phi)_P = \sum_N w_N^2 \mathbf{G}^{-1} \cdot \mathbf{d}_N (\phi_N - \phi_P)$$

- $\mathbf{G}$  is a  $3 \times 3$  symmetric matrix:

$$\mathbf{G} = \sum_N w_N^2 \mathbf{d}_N \mathbf{d}_N$$

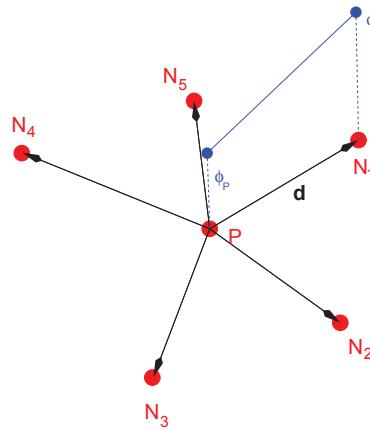


## Cell- and Face-Limited Gradient

- Gradient reconstruction may lead to local over- or under-shoots in reconstructed field:

$$\min_N(\phi_N) \leq \phi_P + \mathbf{d}_N \cdot (\nabla \phi)_P \leq \max_N(\phi_N)$$

- This is important for bounded variables, especially when gradients are used in further discretisation or coupling terms
- Solution: based on the gradient, calculate min and max neighbourhood value and apply gradient limiter to preserve bounds in cell centres



# Convection Discretisation

## Convection Operator

- Convection operator splits into a sum of face integrals (integral and differential form)

$$\oint_S \phi(\mathbf{n} \cdot \mathbf{u}) dS = \int_V \nabla \cdot (\phi \mathbf{u}) dV = \sum_f \phi_f (\mathbf{s}_f \cdot \mathbf{u}_f) = \sum_f \phi_f F$$

where  $\phi_f$  is the face value of  $\phi$  and

$$F = \mathbf{s}_f \cdot \mathbf{u}_f$$

is the **face flux**: measure of the flow through the face

- Simplest face interpolation: **central differencing**. Second-order accurate, but causes oscillations

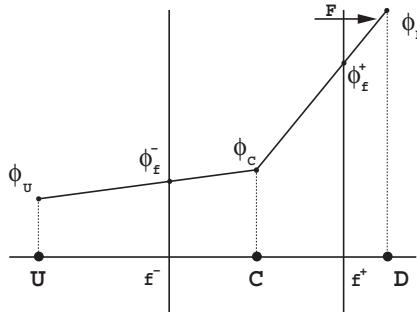
$$\phi_f = f_x \phi_P + (1 - f_x) \phi_N$$

- **Upwind differencing**: taking into account the transportive property of the term: information comes from upstream. No oscillations, but smears the solution

$$\phi_f = \max(F, 0) \phi_P + \min(F, 0) \phi_N$$

## Face Interpolation Scheme for Convection

- In order to close the system, we need a way of evaluating  $\phi_f$  from the cell values  $\phi_P$  and  $\phi_N$ : **face interpolation**
- In order to preserve the iteration sequence, the convection operator for bounded (scalar) properties must preserve boundedness
- There exists a large number of schemes, trying to achieve good accuracy while preserving boundedness: e.g. TVD, and NVD families:  $\phi_f = f(\phi_P, \phi_N, F, \dots)$



- Special differencing schemes for strictly bounded scalars: switching to UD when a variable violates the bound. Example: Gamma01

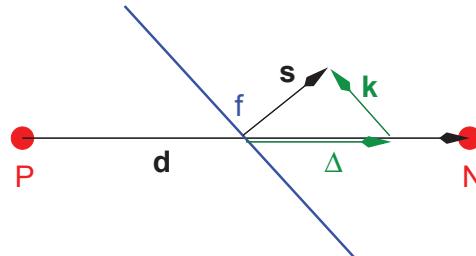
# Diffusion Discretisation

## Diffusion Operator and Mesh Non-Orthogonality

- Diffusion term is discretised using the Gauss Theorem

$$\oint_S \gamma(\mathbf{n} \cdot \nabla \phi) dS = \sum_f \int_{S_f} \gamma(\mathbf{n} \cdot \nabla \phi) dS = \sum_f \gamma_f \mathbf{s}_f \cdot (\nabla \phi)_f$$

- Evaluation of the face-normal gradient. If  $\mathbf{s}$  and  $\mathbf{d}_f = \overline{PN}$  are aligned, use difference across the face. For non-orthogonal meshes, a correction term may be necessary



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

## Limiting Non-Orthogonal Correction in a Laplacian

- Decomposition of face gradient into “orthogonal component” and “non-orthogonal correction” depends on mesh quality: mesh non-orthogonality is measured from  $\overline{PN}$  and  $s_f$
- Mathematically, a Laplacian is a perfect operator: smooth, bounded, self-adjoint. Its discretisation yields a symmetric matrix
- In contrast, non-orthogonal correction is explicit, unbounded and unsigned
- Limited non-orthogonal correction: explicit part clipped to be smaller than its implicit counterpart, base on the current solution

$$\lambda \frac{|s_f|}{|\mathbf{d}_f|} (\phi_N - \phi_P) > \mathbf{k}_f \cdot \nabla(\phi)_f$$

where  $\lambda$  is the limiter value

- Treatment of mesh non-orthogonality over  $90^\circ$ : mesh is formally invalid
  - This corresponds to a Laplacian operator with negative diffusion
  - Stabilise the calculation and remove non-orthogonal correction term
  - Note: This is a “rescue procedure”: reconsider mesh and results!



# Discretisation Settings

## Proposed Settings for Hexahedral Meshes

- Gradient scheme: Gauss or Gauss with limiters
- Convection scheme
  - In initial settings or unknown mesh quality, always start with Upwind. If this fails, there are problems elsewhere in case setup
  - Momentum equation: for second order, recommend linear upwind. with optional gradient limiters
  - TVD/NVD schemes for bounded scalars (eg. turbulence); optionally, use deferred correction formulation
- Diffusion scheme: settings depend on max non-orthogonality
  - Below 60 deg, no special practice: Gauss linear corrected
  - Above 70 deg, non-orthogonality limiter: Gauss linear limited 0.5
- In all cases, monitor boundedness of scalars and adjust convection and diffusion schemes to remove bounding messages



## Proposed Settings for Tetrahedral Meshes

- On tetrahedral meshes, cell neighbourhood is minimal: a tet has only 4 neighbours
- Skewness and non-orthogonality errors are larger and with substantial effect on the solution: it is essential to re-adjust the discretisation
- Gradient scheme: least squares; in most cases without limiters
- Convection scheme
  - On simple cases, use upwinding; nature of discretisation error changes due to lack of mesh-to-flow alignment
  - For highly accurate simulations, special (reconstructed) schemes are used
- Diffusion scheme: always with non-orthogonality limiters. Control limiter based on boundedness messages on scalars

# Summary

## Summary

- Discretisation settings in tutorials are a good starting point
- Variation in mesh structure (tetrahedral, hexahedral and polyhedral) means that no single choice will work for all meshes
- In complex physics, consider physical properties of variables: boundedness and conservation
- OpenFOAM is regularly set up for high accuracy rather than convergence to steady-state: The fact that a solver converges does not necessarily mean the results are correct (or physical!)
- “Special applications” like LES require additional care: energy conserving numerics, low diffusion and dispersion errors
- Guidance provided for main mesh types: hex and tet. Polyhedral meshes use hex settings
- Further complications may be introduced by moving mesh and topological changes



# Dynamic mesh capabilities and algorithms for simulation of IC engines and volumetric machines

Dr. Gianluca Montenegro

Department of energy – Politecnico di Milano



## Introduction

Simulations with Time-Varying Geometry in CFD

- In many simulations, shape of computational domain changes during the solution, either in a prescribed manner or as a part of the solution
- In cases of **prescribed motion**, it is possible to pre-define a sequence of meshes or mesh changes which accommodate the motion

Mesh generation now becomes substantially more complex

**Solution-dependent motion** implies the shape of computational domain is a part of the solution itself: depends on solution parameters

Examples of solution-dependent motion:

- Contact stress analysis: shape and location of contact region is unknown
- Free surface tracking simulation: unknown shape of free surface
- Fluid-structure interaction



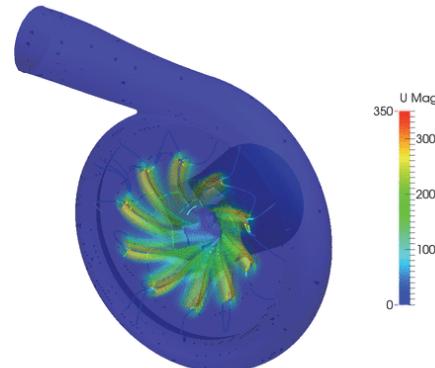
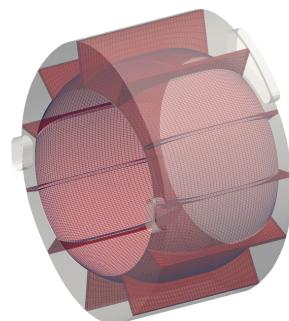
# Prescribed motion

Example: Prescribed Mesh Motion

- Domain shape is changing during the simulation in a prescribed manner
- Motion is known and independent of the solution, usually only prescribed at boundaries
- Definition of moving mesh involves point position and mesh connectivity for every point and cell for every time-step of the simulation. This is typically defined with reference to a pre-processor or parametrically in terms of motion parameters (crank angle, valve lift curve, etc.)
- Solution-dependent mesh changes can be performed without affecting the motion: eg. mesh refinement



Time: 386.00

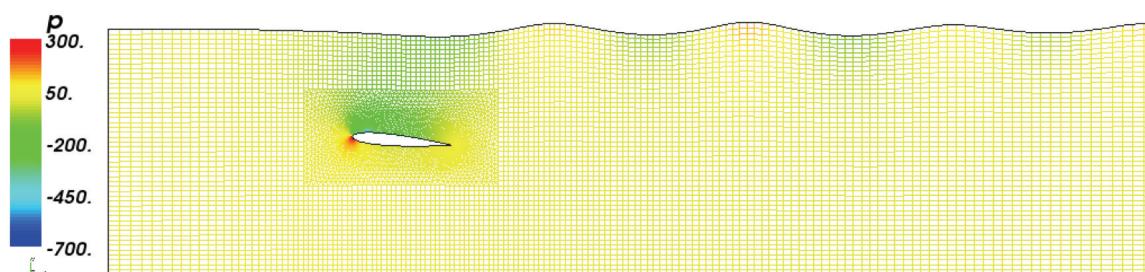
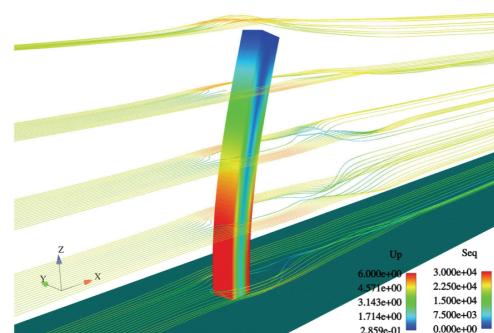


**ICE GROUP**  
POLITECNICO DI MILANO

# Solution – Dependent motion

Example: Solution-Dependent Motion

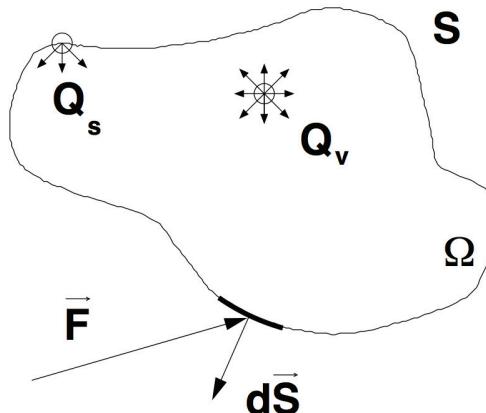
- External shape of the domain is unknown and a part of the solution
- By definition, it is impossible to pre-define mesh motion a-priori
- In all cases, it is the **motion of the boundary** that is known or calculated
- Automatic mesh motion determines the position of internal points based on boundary motion



**ICE GROUP**  
POLITECNICO DI MILANO

# Conservation equation for moving grids

- First of all we need to work out the formulation of the conservation equation when the moving frame is considered
- We shall start considering the continuity equation referred to an arbitrary control volume



$$\int_V \frac{\partial \rho}{\partial t} d\Omega + \int_V \nabla \cdot (\rho \vec{U}) d\Omega = 0$$

# Conservation equation for moving grids

- For simplicity sake we shall consider the 1D continuity equation

$$\int_L \frac{\partial \rho}{\partial t} dx + \int_L \frac{\partial (\rho U_x)}{\partial x} dx = 0$$

- ... and refer to a case with moving boundaries

$$\int_{x_1(t)}^{x_2(t)} \frac{\partial \rho}{\partial t} dx + \int_{x_1(t)}^{x_2(t)} \frac{\partial (\rho U_x)}{\partial x} dx = 0$$

- Recalling the Leibniz's rule, where the integration boundaries are function of an independent variable

$$\frac{\partial}{\partial t} \int_{x_1(t)}^{x_2(t)} f(x, t) dx = \int_{x_1(t)}^{x_2(t)} \frac{\partial f}{\partial t} dx + f(x_2(t), t) \frac{\partial x_2}{\partial t} - f(x_1(t), t) \frac{\partial x_1}{\partial t}$$

- or ...

$$\int_{x_1(t)}^{x_2(t)} \frac{\partial f}{\partial t} dx = \frac{\partial}{\partial t} \int_{x_1(t)}^{x_2(t)} f(x, t) dx - f(x_2(t), t) \frac{\partial x_2}{\partial t} + f(x_1(t), t) \frac{\partial x_1}{\partial t}$$

# Conservation equation for moving grids

- Referring to the continuity equation

$$\int_{x_1}^{x_2} \frac{\partial \rho}{\partial t} dx \implies \frac{\partial}{\partial t} \int_{x_1}^{x_2} \rho dx - \left[ \rho_2 \frac{\partial x_2}{\partial t} - \rho_1 \frac{\partial x_1}{\partial t} \right]$$

$$\int_{x_1}^{x_2} \frac{\partial (\rho U_x)}{\partial x} dx \implies \rho_2 U_{x_2} - \rho_1 U_{x_1}$$

- Gathering all the terms in the same equation, the continuity will become

$$\frac{d}{dt} \int_{x_1}^{x_2} \rho dx - \left[ \rho_2 \frac{\partial x_2}{\partial t} - \rho_1 \frac{\partial x_1}{\partial t} \right] + \rho_2 U_{x_2} - \rho_1 U_{x_1}$$

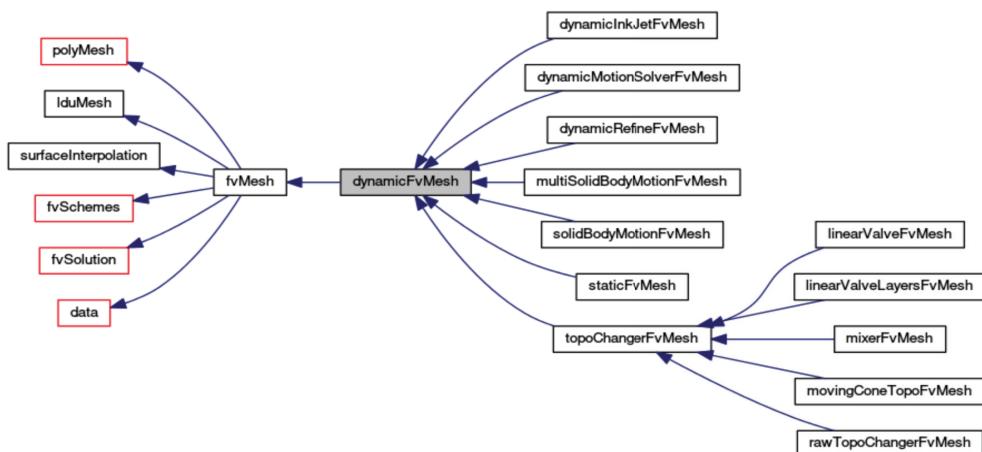
- The terms  $\frac{\partial x_i}{\partial t}$  express the velocity which the integration boundaries are moving with

$$\frac{d}{dt} \int_{x_1}^{x_2} \rho dx - \int_{x_1}^{x_2} \frac{\partial}{\partial x} [\rho (U_x - U_b)] dx \implies \frac{d}{dt} \int_V \rho d\Omega - \int_V \nabla \cdot \rho (U - U_b) d\Omega$$

- This reads that the divergence term is integrated subtracting the mesh flux to the variable flux

## How is this implemented

- In OpenFOAM mesh motion algorithm and solvers are separated the structure of the solver is independent from the mesh motion algorithm
- Any type of solver can be adapted to account for mesh motion with limited effort.
- The solver account for the relative volumetric flux or for the absolute one (depending on the term of the equation), while The mesh object accounts for the volumetric flux due to the mesh motion.



# How is this implemented: rhoPimpleFoam

```
#include "fvCFD.H"
#include "dynamicFvMesh.H"
#include "psiThermo.H"
#include "turbulenceModel.H"
#include "bound.H"
#include "pimpleControl.H"
#include "fvIOoptionList.H"

// * /


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

    pimpleControl pimple(mesh);

    #include "readControls.H"
    #include "createFields.H"
    #include "createFvOptions.H"
    #include "createPcorrTypes.H"
    #include "createRhoUf.H"
    #include "CourantNo.H"
    #include "setInitialDeltaT.H"
}
```

Inclusion of classes to handle the dynamic mesh

Creates the mesh object as a **dynamicFvMesh** and not as an **fvMesh**

```
surfaceVectorField rhoUf
(
    IOobject
    (
        "rhoUf",
        runTime.timeName(),
        mesh,
        IOobject::READ_IF_PRESENT,
        IOobject::AUTO_WRITE
    ),
    fvc::interpolate(rho*U)
);
```



# How is this implemented: rhoPimpleFoam

```
while (runTime.run())
{
    {
        volScalarField divrhou
        (
            "divrhou",
            fvc::div(fvc::absolute(phi, rho, U))
        );
    }
    runTime++;

    volVectorField rhoU("rhoU", rho*U);

    // Do any mesh changes
    mesh.update();

    if (mesh.changing() && correctPhi)
    {
        phi = mesh.Sf() & rhoUf;

        #include "correctPhi.H"

        // Make the fluxes relative to the mesh-motion
        fvc::makeRelative(phi, rho, U);
    }
}
```

Absolute convective flux due to the flow motion

Triggers the mesh motion updating the new point positions and calculating the mesh flux

```
void Foam::fvc::makeRelative
(
    surfaceScalarField& phi,
    const volScalarField& rho,
    const volVectorField& U
)
{
    if (phi.mesh().moving())
    {
        phi -=
        fvc::interpolate(rho)*fvc::mesh
        Phi(rho, U);
    }
}
```



# How is this implemented: correctPhi.H

```

volScalarField pcorr
(
    IOobject
    (
        "pcorr",
        runTime.timeName(),
        mesh,
        IOobject::NO_READ,
        IOobject::NO_WRITE
    ),
    mesh,
    dimensionedScalar("pcorr", p.dimensions(), 0.0),
    pcorrTypes
);

dimensionedScalar rAUf("rAUf", dimTime, 1.0);

while (pimple.correctNonOrthogonal())
{
    fvScalarMatrix pcorrEqn
    (
        fvm::ddt(psi, pcorr)
        + fvc::div(phi)
        - fvm::laplacian(rAUf, pcorr)
        ==
        divrhoU
    );
    pcorrEqn.solve();
    if (pimple.finalNonOrthogonalIter())
    {
        phi += pcorrEqn.flux();
    }
}

```

Creates a pcorr field initialized as zero and a rAUf scalar set as 1

Set the equation matrix for the corrected pressure

Solves the pcorr equation and updates the flux



# How is this implemented: rhoPimpleFoam

```

if (mesh.changing() && checkMeshCourantNo)
{
    #include "meshCourantNo.H"
}

#include "rhoEqn.H"
Info<< "rhoEqn max/min : " << max(rho).value()
     << " " << min(rho).value() << endl;

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

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

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

runTime.write();

```

Checks the mesh for stability

Pimple loop: outer loop is PISO, inner loop is SIMPLE



# How is this implemented: meshCoNum.H

```

scalar meshCoNum = 0.0;
scalar meanMeshCoNum = 0.0;

if (mesh.nInternalFaces())
{
    scalarField sumPhi
    (
        fvc::surfaceSum(mag(mesh.phi()))().internalField()
    );

    meshCoNum = 0.5*gMax(sumPhi/mesh.V().field())*runTime.deltaTValue();

    meanMeshCoNum =
        0.5*(gSum(sumPhi)/gSum(mesh.V().field()))*runTime.deltaTValue();
}

Info<< "Mesh Courant Number mean: " << meanMeshCoNum
      << " max: " << meshCoNum << endl;

```

Computes the total volumetric flux through the faces of each cell

The volume swept during a time step must be smaller than the cell volume



# How is this implemented: UEqn.H

```

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

UEqn().relax();

fvOptions.constrain(UEqn());

if (pimple.momentumPredictor())
{
    solve(UEqn() == -fvc::grad(p));
    fvOptions.correct(U);
    K = 0.5*magSqr(U);
}

```

Advection of momentum is relative as well

fvOption operates as a constraint to limit the velocity between certain bounds



# How is this implemented: EEqn.H

```

volScalarField& he = thermo.he();

fvScalarMatrix EEqn
(
    fvm::ddt(rho, he) + fvm::div(phi, he)
    + fvc::ddt(rho, K) + fvc::div(phi, K)
    +
        he.name() == "e"
    ? fvc::div
        (
            fvc::absolute(phi/fvc::interpolate(rho), U),
            p,
            "div(phiv,p)"
        )
    : -dpdt
)
- fvm::laplacian(turbulence->alphaEff(), he)
==
    fvOptions(rho, he)
);

EEqn.relax();

fvOptions.constrain(EEqn);

EEqn.solve();

fvOptions.correct(he);

thermo.correct();

```

The convective contribution is given by the relative volumetric flux

The absolute flux is exploited for the work of the pressure forces



# How is this implemented: pEqn.H

```

surfaceScalarField phiHbyA
(
    "phiHbyA",
    (fvc::interpolate(rho*HbyA) & mesh.Sf())
    + rhorAUf*fvc::ddtCorr(rho, U, rhoUf)
);
{
    fvc::makeRelative(phiHbyA, rho, U);
    fvOptions.makeRelative(fvc::interpolate(rho), phiHbyA);

while (pimple.correctNonOrthogonal())
{
    // Pressure corrector
    fvScalarMatrix pEqn
    (
        fvm::ddt(psi, p)
        + fvc::div(phiHbyA)
        - fvm::laplacian(rhorAUf, p)
        ==
            fvOptions(psi, p, rho.name())
    );
    pEqn.solve(mesh.solver(p.select(pimple.finalInnerIter())));
    if (pimple.finalNonOrthogonalIter())
    {
        phi = phiHbyA + pEqn.flux();
    }
}

```

Flux from the corrected velocity of the SIMPLE algorithm

The flux is made relative to derive the pressure equation from the substitution of the corrected velocity into the continuity equation

# How to move the mesh

One can distinguish between boundary motion and internal point motion:

- **Boundary motion** can be considered as given: either prescribed by external factors, e.g. piston and valve motion for in-cylinder flow simulations in internal combustion engines, or a part of the solution as in free surface tracking simulations
- The role of internal point motion is to accommodate changes in the domain shape (boundary motion) and preserve the validity and quality of the mesh.

The objective of automatic mesh motion is to determine internal point motion (not involving topological changes) to conform with the given boundary motion while preserving mesh validity and quality:

- The investigation of mesh validity can be separated into topological and geometrical tests



## How to move the mesh

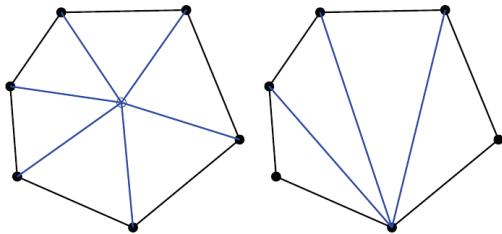
**Topological validity** tests consist of the following criteria:

- A point can appear in a face only once;
- A face can appear in a cell only once
- A face cannot belong to more than two cells.
- A boundary face can belong to only one patch;
- Two cells can share no more than one face;
- Collecting all faces from one cell and decomposing faces into edges, every edge must appear in exactly two cell faces;
- Collecting all faces from the boundary and decomposing faces into edges, every edge must appear in exactly two boundary faces.

# How to move the mesh

- **Geometrical tests** deal with the positivity of face areas and cell volumes, as well as convexity and orientation requirements.

- ✓ The geometrical measures (face area and normal vector, face and cell centroid, volume swept by the face in motion etc.) for a polygonal face are calculated by decomposing the face into triangles.
- ✓ Two possible triangular decompositions of a polygon (with different severity of the convexity criteria)



- A face is considered convex if all triangle normals point in the same direction
- The tetrahedra are constructed using the (approximate) cell centroid and the triangles of the face decomposition.

## Cell based motion equation

- The simplest suggestion for automatic mesh motion in the FV framework would be to use the available numerics and solve a Laplace (or a linear elastic solid) equation to provide vertex motion.
- As the FVM provides the solution in cell centres and motion is required on the points, this necessarily leads to interpolation.
- It is extremely difficult to construct an interpolation practice which stops the cells from flipping and degenerating even if the cell-centred motion field is bounded
- Finally, while the FVM is unconditionally bounded for the convection operator, on badly distorted meshes one needs to sacrifice either the second-order accuracy or boundedness in the Laplacian, due to the explicit (and unbounded) nature of the non-orthogonal correction

# Cell based motion equation

- The Laplace operator with constant or variable diffusion field is chosen to govern mesh motion:

$$\nabla(\gamma \nabla \mathbf{U}_b) = 0$$

- Modify point positions using the point velocity field  $\mathbf{U}_b$ :

$$x_{new} = x_{old} + \mathbf{U}_b \Delta t$$

- Where  $x_{old}$  and  $x_{new}$  are the point positions before and after mesh motion and  $\Delta t$  is the time-step.
- Boundary conditions for the motion equation are enforced from the known boundary motion; this may include free boundaries, symmetry planes, prescribed motion boundary etc.
- The matrix is solved using an iterative linear equation solver; here the choice falls on the Incomplete Cholesky preconditioned Conjugate Gradient (ICCG) solver.
- Point motion allows to estimate the face swept volume or the mesh flux



## Automatic mesh motion solver

### Definition of Automatic Mesh Motion

- Automatic mesh motion will determine the position of mesh points based on the prescribed boundary motion
- Motion will be obtained by solving a **mesh motion equation**, where boundary motion acts as a boundary condition
- The “correct” space-preserving equation is a large deformation formulation of linear elasticity . . . but it is too expensive to solve
- Choices for a simplified mesh motion equation:
  - ✓ Spring analogy: insufficiently robust
  - ✓ Linear + torsional spring analogy: complex, expensive and non-linear
  - ✓ Laplace equation with constant and variable diffusivity
  - ✓  $\nabla \cdot (\mathbf{k} \nabla \mathbf{u}) = 0$ 
    - Linear pseudo-solid equation for small deformations  $\nabla \cdot [\mu(\nabla \mathbf{u} + (\nabla \mathbf{u})^T) + \lambda I \nabla \cdot \mathbf{u}] = 0$

# Automatic mesh motion solver

Control of Mesh Spacing and Discretisation Error

- Mesh spacing and quality control by **variable diffusivity**
- Changing diffusivity implies redistribution of the boundary motion through the volume of the mesh: it is not necessarily good to absorb deformation next to the moving boundary
  - ✓ Distance-based methods:  $1/L$ , quadratic or exponential distance from a moving boundary
  - ✓ Quality- and distortion-based methods

Preserving Mesh Quality

- Definition of valid motion from an initially valid mesh implies that no faces or cells are inverted during motion
- Face area and cell volume on polyhedral meshes is calculated using triangular/tetrahedral decomposition of the cell or face. Motion validity is guaranteed if no triangles or tetrahedra in the cell and face decomposition are inverted during motion
- The rest reduces to controlling the discretisation error in the motion equation

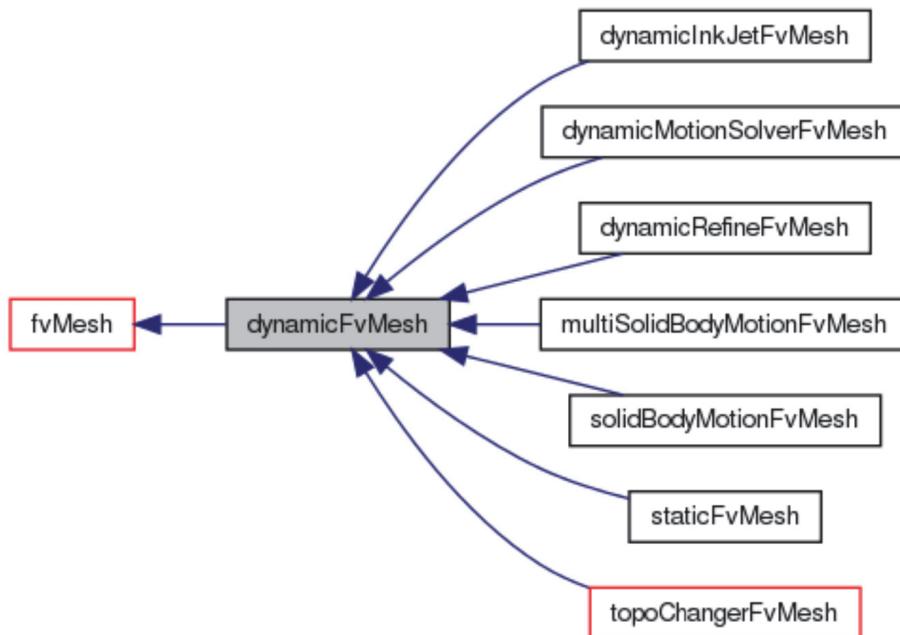


## Choice of the motion solver

Motion Solver Equations in OpenFOAM

- Solving motion equation on cell centers, interpolating motion into points
  - ✓ Special corrections and extrapolation on corners and patch intersections
  - ✓ Smaller equation set, but problems in cell-to-point interpolation
- Second-order Finite Element method with polyhedral support
  - ✓ Vertex-based method: no interpolation required
  - ✓ FE shape function does not allow tetrahedral or triangular flip: perfect for large boundary deformation
  - ✓ Mini-element technique involves enriching the point set: more equations but lower discretisation error
  - ✓ Choice of element decomposition
    - **Cell decomposition:** additional point in every cell centroid
    - **Cell-and-face decomposition:** additional point in cell and face centroid
  - ✓ Validated and efficient for large deformation, eg. internal combustion engines
  - ✓ Cell-based methods

# The dynamicFvMesh class



- `dynamicFvMesh` inherits functionalities from the `fvMesh` class
- `fvMesh` has `movePoints()` function for the calculation of the swept volumes but not functionalities on how to move points

# The dynamicFvMesh class

```
class dynamicFvMesh
:
    public fvMesh // Inherited from the fvMesh class
{
    // Private Member Functions

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

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

public:

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

    // Declare run-time constructor selection table
    declareRunTimeSelectionTable
    (
        autoPtr,
        dynamicFvMesh,
        IOobject,
        (const IOobject& io),
        (io)
    );
}
```

# The dynamicFvMesh class

```

// Constructors

    // Construct from objectRegistry, and read/write options
    explicit dynamicFvMesh(const IOobject& io);

    // Construct from components without boundary.
    // Boundary is added using addFvPatches() member function
    dynamicFvMesh
    (
        const IOobject& io,
        const Xfer<pointField>& points,
        const Xfer<faceList>& faces,
        const Xfer<labelList>& allOwner,
        const Xfer<labelList>& allNeighbour,
        const bool syncPar = true
    );
// Selectors

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

// Destructor
virtual ~dynamicFvMesh();

// Member Functions

    // Update the mesh for both mesh motion and topology change
    virtual bool update() = 0;

```

Static autoPtr for run-time selectable dynamicFvMesh

Pure virtual function adds functionalities for triggering the mesh motion: motionSolvers or topoChangers



# The dynamicFvMesh class

```

Foam::autoPtr<Foam::dynamicFvMesh> Foam::dynamicFvMesh::New(const
IOobject& io)
{
    IODictionary dict
    (
        IOobject
        (
            "dynamicMeshDict",
            io.time().constant(),
            (io.name() == polyMesh::defaultRegion ? "" : io.name()),
            io.db(),
            IOobject::MUST_READ_IF_MODIFIED,
            IOobject::NO_WRITE,
            false
        )
    );

    const word dynamicFvMeshTypeName(dict.lookup("dynamicFvMesh"));

    const_cast<Time&>(io.time()).libs().open
    (
        dict,
        "dynamicFvMeshLibs",
        IOobjectConstructorTablePtr_
    );

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

    return autoPtr<dynamicFvMesh>(cstrIter()(io));
}

```

Access to the dictionary which controls the dynamicMesh selection

Selection of the dynamic library which handles the motion solver



# The staticFvMesh class

```
class staticFvMesh
:
{   public dynamicFvMesh
    // Private Member Functions

    // Disallow default bitwise copy construct
    staticFvMesh(const staticFvMesh&);

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

public:

    // Runtime type information
    TypeName("staticFvMesh");

    // Constructors

    // Construct from IOobject
    staticFvMesh(const IOobject& io);

    // Destructor
    ~staticFvMesh();

    // Member Functions

    // Dummy update function which does not change the mesh
    virtual bool update();
};
```

Inherits the functionalities of dynamicFvMesh

The virtual update() function is defined. However, it is just a dummy function that does not perform any operation



# The staticFvMesh class

```
#include "staticFvMesh.H"
#include "addToRunTimeSelectionTable.H"

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

namespace Foam
{
    defineTypeNameAndDebug(staticFvMesh, 0);
    addToRunTimeSelectionTable(dynamicFvMesh, staticFvMesh,
    IOobject);
}

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

Foam::staticFvMesh::staticFvMesh(const IOobject& io)
:
    dynamicFvMesh(io)
{}

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

Foam::staticFvMesh::~staticFvMesh()
{}

// * * * * * Member Functions * * * * *
bool Foam::staticFvMesh::update()
{
    return false;
}
```

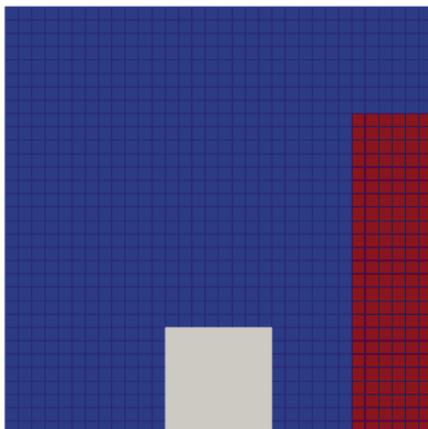
No operations are performed. This is a class used to convert dynamic cases into static ones.

The mesh is not updated since it is not changing



# The dynamicRefineFvMesh class

- This class is a fvMesh with built-in refinement. It determines which cells to refine/unrefine and does all in the update() function
- The mesh does not deform or moves, it just changes dynamically on the basis of certain criteria.
- The refinement can be operated on the basis of the values of certain fields in the cells.



```
dynamicFvMesh dynamicRefineFvMesh;  
  
dynamicRefineFvMeshCoeffs  
{  
    refineInterval 1;  
    field alpha.water;  
    lowerRefineLevel 0.001;  
    upperRefineLevel 0.999;  
    unrefineLevel 10;  
    nBufferLayers 1;  
    maxRefinement 2;  
    maxCells 200000;  
    correctFluxes  
    {  
        (phi none)  
        (nHatf none)  
        (rhoPhi none)  
        (ghf none)  
    };  
    // Write the refinement level as a volScalarField  
    dumpLevel true;
```

# The solidBodyMotionFvMesh class

```
class solidBodyMotionFvMesh  
:  
public dynamicFvMesh  
{  
    //-- Dictionary of motion control parameters  
    const dictionary dynamicMeshCoeffs_;  
  
    //-- The motion control function  
    autoPtr<solidBodyMotionFunction> SBMFPtr_;  
  
    //-- The reference points which are transformed  
    pointIOField undisplacedPoints_;  
  
    //-- Points to move when cell zone is supplied  
    labelList pointIDs_;  
  
    //-- Flag to indicate whether all cells should move  
    bool moveAllCells_;  
  
    //-- Name of velocity field  
    word UName_;  
  
    //-- Runtime type information  
    TypeName("solidBodyMotionFvMesh");  
  
    //-- Construct from IOobject  
    solidBodyMotionFvMesh(const IOobject& io);  
  
    //-- Update the mesh for both mesh motion and topology change  
    virtual bool update();
```

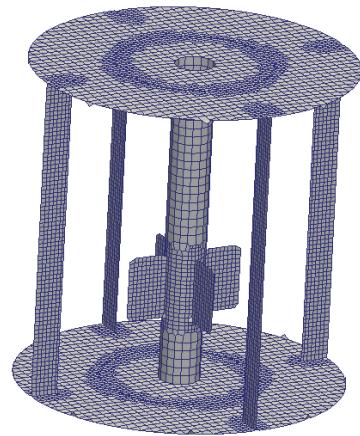
Specifies which function must be used for the solid body motion

Motion can be applied to reference points of to points defined via a cell zone

# The solidBodyMotionFvMesh class

- To realize the motion the class relies to specific functions operating with quaternions and septernions in order to realize rotations and translations:
  - SDA**: specifically developed for ship design analysis. It is a set of 3DoF motion functions comprising sinusoidal roll (rotation about x), heave (z-translation) and sway (y-translation) motions with changing amplitude and phase.
  - axisRotationMotion/rotatingMotion**: move the points around a center along three axes specified in different ways
  - linearMotion/oscillatingLinearMotion**: constant or oscillating velocity translation
  - oscillatingRotatingMotion**: sinusoidal oscillating rotation
  - rotoTranslation**: translation and rotation combined
  - multiMotion**: combination of multiple solidBodyMotionFunctions (based on PtrList<SBMF>)

```
dynamicFvMesh solidBodyMotionFvMesh;  
  
solidBodyMotionFvMeshCoeffs  
{  
    cellZone      rotatingZone;  
  
    solidBodyMotionFunction rotatingMotion;  
    rotatingMotionCoeffs  
    {  
        origin      (0 0 0);  
        axis        (0 0 1);  
        omega       $:meshMotionProperties.omega;  
    }  
}
```



**ICE GROUP**  
POLITECNICO DI MILANO

# The dynamicMotionSolverFvMesh class

```
class dynamicMotionSolverFvMesh  
:  
    public dynamicFvMesh  
{  
    // Private data  
    autoPtr<motionSolver> motionPtr_;  
  
    // Private Member Functions  
    //-- Disallow default bitwise copy construct  
    dynamicMotionSolverFvMesh(const dynamicMotionSolverFvMesh&);  
  
    //-- Disallow default bitwise assignment  
    void operator=(const dynamicMotionSolverFvMesh&);  
public:  
  
    //-- Runtime type information  
    TypeName("dynamicMotionSolverFvMesh");  
  
    // Constructors  
  
    //-- Construct from IOobject  
    dynamicMotionSolverFvMesh(const IOobject& io);  
  
    //-- Destructor  
    ~dynamicMotionSolverFvMesh();  
  
    // Member Functions  
    //-- Update the mesh for both mesh motion and topology change  
    virtual bool update();
```

Reference to the motion solver is added at this level. Motion solver will be selected run time

Update function is no more pure virtual, Functionalities are added

**ICE GROUP**  
POLITECNICO DI MILANO

# The dynamicMotionSolverFvMesh class

```
Foam::dynamicMotionSolverFvMesh::dynamicMotionSolverFvMesh(const  
IObject& io)  
:  
    dynamicFvMesh(io),  
    motionPtr_(motionSolver::New(*this))  
{}  
  
// * * * * * * * * * * * * * * * * * Destructor * * * * * * * //  
  
Foam::dynamicMotionSolverFvMesh::~dynamicMotionSolverFvMesh()  
{}  
  
// * * * * * * * * * Member Functions * * * * * * * * * //  
  
bool Foam::dynamicMotionSolverFvMesh::update()  
{  
    fvMesh::movePoints(motionPtr_->newPoints());  
  
    if (foundObject<volVectorField>("U"))  
    {  
        volVectorField& U =  
  
const_cast<volVectorField&>(lookupObject<volVectorField>("U"));  
        U.correctBoundaryConditions();  
    }  
  
    return true;  
}
```

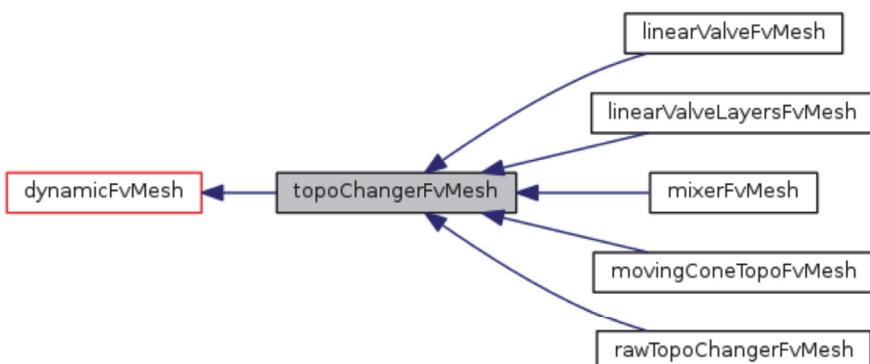
The update function uses the movePoints member function provided by the fvMesh class providing the new point positions returned by the motion solver



## Topology change

### Topological Changes on Polyhedral Meshes

- For extreme cases of mesh motion, changing point positions is not sufficient to accommodate boundary motion and preserve mesh quality
- Definition of a **topological change**: number or connectivity of points, faces or cells in the mesh changes during the simulation
- Motion can be handled by the FVM with no error (moving volume), while a topological change requires additional algorithmic steps



# The topoChangerFvMesh class

```

class topoChangerFvMesh
{
    public dynamicFvMesh
    {
        // Private Member Functions

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

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

protected:
    polyTopoChanger topoChanger_;

public:
    //-- Runtime type information
    TypeName("topoChangerFvMesh");

    //-- Construct from objectRegistry, and read/write options
    explicit topoChangerFvMesh(const IOobject& io);

    //-- Destructor
    virtual ~topoChangerFvMesh();

    //-- Update the mesh for both mesh motion and topology change
    virtual bool update() = 0;
};

```

Still the base class is the dynamicFvMesh

Added the capability of handling topological changes

The update() function triggers the topology change as well as the mesh motion



## The mixerFvMesh class: mixerFvMesh.H

```

class mixerFvMesh
{
    public topoChangerFvMesh
    {
        // Private data

        //-- Motion dictionary
        dictionary motionDict_;

        //-- Coordinate system
        autoPtr<coordinateSystem> csPtr_;

        // - Rotational speed in rotations per minute (rpm)
        scalar rpm_;

        //-- Markup field for points. Moving points marked with 1
        mutable scalarField* movingPointsMaskPtr_;

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

        //-- Add mixer zones and modifiers
        void addZonesAndModifiers();

        //-- Calculate moving masks
        void calcMovingMasks() const;

        //-- Return moving points mask
        const scalarField& movingPointsMask() const;
    };
}

```

Extends the topoChangerFvMesh class

Dictionaries and definition to handle the particular motion

Add the modifiers, namely the objects dedicated to change the topology



# The mixerFvMesh class: mixerFvMesh.H

```
void Foam::mixerFvMesh::addZonesAndModifiers()
{
    // Add zones and modifiers for motion action
    List<pointZone*> pz(1);

    pz[0] = new pointZone
    (
        "cutPointZone",
        labelList(0),
        0,
        pointZones()
    );

    List<faceZone*> fz(3);

    // Inner slider
    const word innerSliderName(motionDict_.subDict("slider").lookup("inside"));
    const polyPatch& innerSlider = boundaryMesh()[innerSliderName];

    fz[0] = new faceZone
    (
        "insideSliderZone",
        isf,
        boolList(innerSlider.size(), false),
        0,
        faceZones()
    );
}
```

Definition of a pointZone for point projection

Definition of faceZones for patch relative motion

# The mixerFvMesh class: mixerFvMesh.H

```
cz[0] = new cellZone
(
    "movingCells",
    movingCells,
    0,
    cellZones()
);
addZones(pz, fz, cz);

topoChanger_.setSize(1);
topoChanger_.set
(
    0,
    new slidingInterface
    (
        "mixerSlider",
        0,
        topoChanger_,
        outerSliderName + "Zone",
        innerSliderName + "Zone",
        "cutPointZone",
        "cutFaceZone",
        outerSliderName,
        innerSliderName,
        slidingInterface::INTEGRAL
    )
);
topoChanger_.writeOpt() = IOobject::AUTO_WRITE;
```

cellZone to define the moving cells

Zones are added to the mesh

The defined zones are used to build the topology changer

# The mixerFvMesh class: mixerFvMesh.C

```

bool Foam::mixerFvMesh::update()
{
    movePoints
    (
        csPtr_->globalPosition
        (
            csPtr_->localPosition(points())
            + vector(0, rpm_*360.0*time().deltaTValue()/60.0, 0)
            *movingPointsMask()
        )
    );
    autoPtr<mapPolyMesh> topoChangeMap = topoChanger_.changeMesh(true);

    if (topoChangeMap.valid())
    {
        deleteDemandDrivenData(movingPointsMaskPtr_);
    }

    movePoints
    (
        csPtr_->globalPosition
        (
            csPtr_->localPosition(oldPoints())
            + vector(0, rpm_*360.0*time().deltaTValue()/60.0, 0)
            *movingPointsMask()
        )
    );
    return true;
}

```

Motion is handled with the movePoints() function of the dynamicFvMesh class

The topology changer is triggered

## Topology change

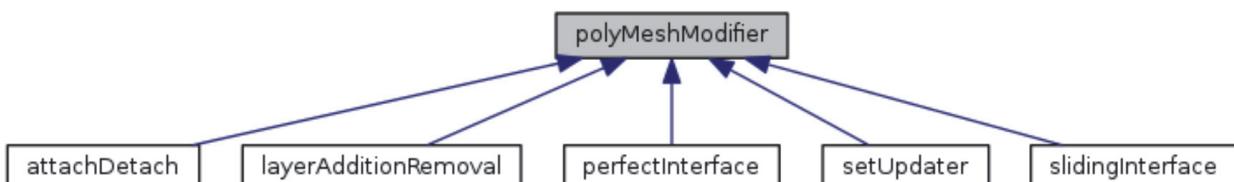
Implementation of Topological Changes in OpenFOAM

- **Primitive mesh operations**

- ✓ Add/modify/remove a point, a face or a cell
- ✓ This is sufficient to describe all cases, even to build a mesh from scratch
- ✓ . . . but using it directly is very inconvenient

- **Topology modifiers**

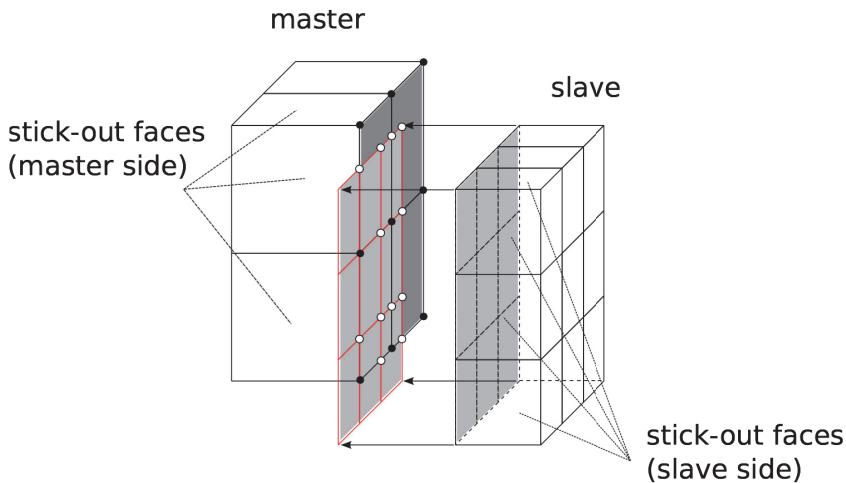
- ✓ Typical dynamic mesh operations can be described in terms of primitive operations. Adding a user-friendly definition and triggering logic creates a “topology modifier” class for typical operations
- ✓ Some implemented topology modifiers
  - Attach-detach boundary
  - Cell layer additional-removal interface
  - Sliding interface
  - Error-driven adaptive mesh refinement



# Topology change: sliding interface

## Rotating Components

- Mesh sliding techniques for stator-rotor interaction
- Mesh separated into two disconnected parts, moving relative to each other
- For every time-step, patches defining the interface are converted into internal faces using a face cutting technique
- Topological changers handle mesh and data mapping

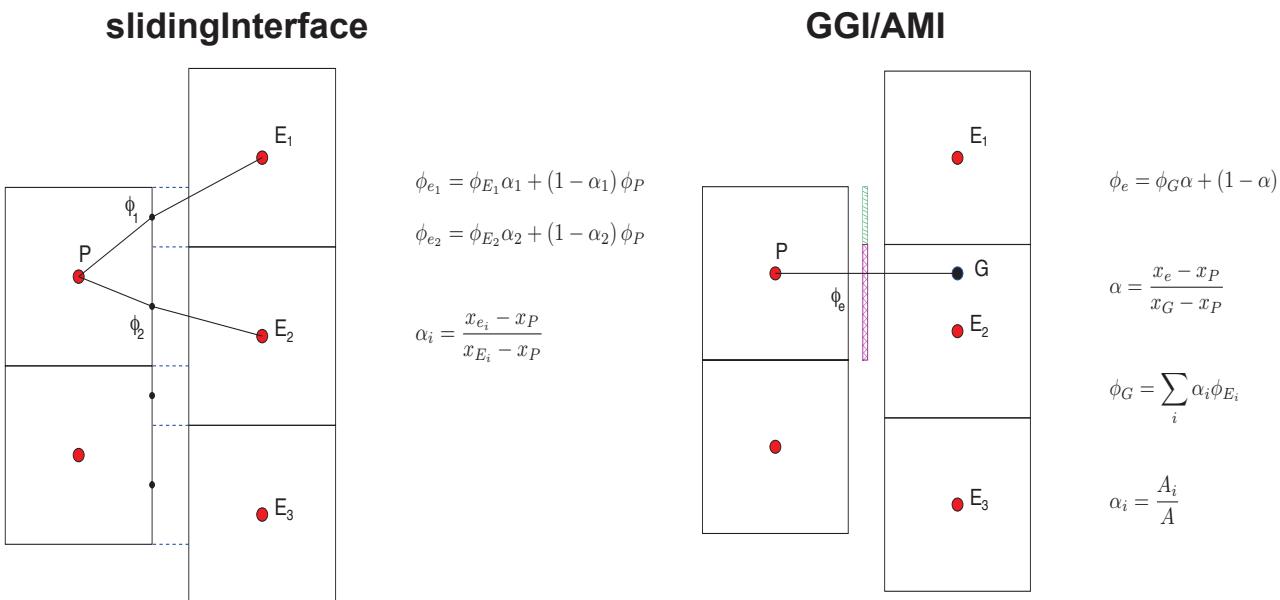


# Topology change: sliding interface

## General Grid Interface

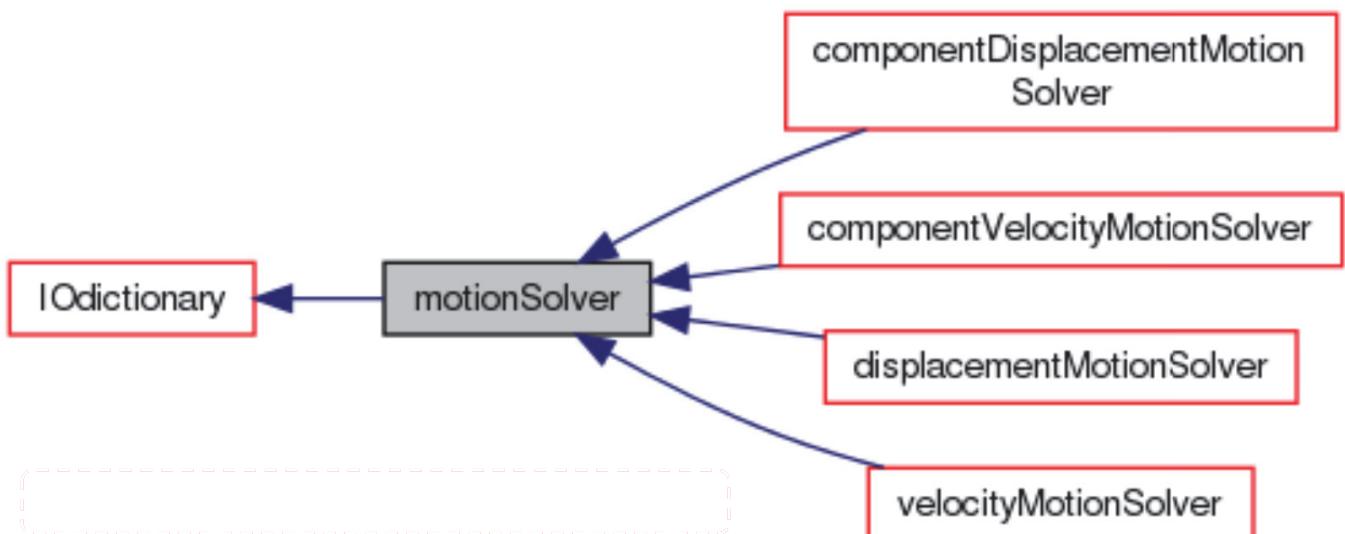
- Sliding interface mesh modifier couples two mesh components in relative motion to create a continuous mesh: topological change
- Treatment of coupled boundaries without topology change: General Grid Interface (GGI)
- Face cutting separates surface integral across multiple neighbours. The same can be done with appropriate interpolation coefficients
- Coupled patch treatment implicit in discretisation and solvers
- Special handling for patch evaluation and operator discretisation
- Special turbo-specific steady-state discretisation can be implemented as a complex coupled boundary condition with averaging: mixing plane

# Sliding interface vs GGI



- The GGI operation mode operates at a matrix level suitably weighting the cells sharing part of their surface with the same face on the coupled patch

## The motionSolver class



- `motionSolver` class defines the functionalities of the motion solver and provides access to point position.
- It is then specialized on the basis of the particular motion solver: displacement or velocity
- The member function `solve()` recalls the solution of the point motion

# The motionSolver class

```

class motionSolver
:
    public IOdictionary
{

private:

    // Reference to mesh
    const polyMesh& mesh_;

    // Model coefficients dictionary
    dictionary coeffDict_;

// Private Member Functions

    // De-register object if registered and assign to current
    static IOobject stealRegistration(const IOdictionary& dict);

// Selectors

    // Select constructed from polyMesh
    static autoPtr<motionSolver> New(const polyMesh&);

// Solve for motion
    virtual void solve() = 0;

    // Update local data for geometry changes
    virtual void movePoints(const pointField&) = 0;

    // Update local data for topology changes
    virtual void updateMesh(const mapPolyMesh&) = 0;

```

It is run time selectable and read its own dictionary

Solve function is pure virtual. It will be implemented in the specialization of the class

It moves the points according to the strategy selected: displacement or velocity



# The motionSolver class

```

Foam::autoPtr<Foam::motionSolver> Foam::motionSolver::New
(
    const polyMesh& mesh,
    const IOdictionary& solverDict
)
{
    const word solverTypeName(solverDict.lookup("solver"));

    Info<< "Selecting motion solver: " << solverTypeName << endl;

    const_cast<Time&>(mesh.time()).libs().open
    (
        solverDict,
        "motionSolverLibs",
        dictionaryConstructorTablePtr_
    );
    dictionaryConstructorTable::iterator cstrIter =
        dictionaryConstructorTablePtr_->find(solverTypeName);

    return autoPtr<motionSolver>(cstrIter()(mesh, solverDict));
}

Foam::tmp<Foam::pointField> Foam::motionSolver::newPoints()
{
    solve();
    return curPoints();
}

```

Libraries of the fvMotionSolver library are recalled

The motion is solved and then the new point positions are returned



# The displacementMotionSolver class

```
class displacementMotionSolver
:
    public motionSolver
{
protected:
    // Protected data
    //-- Point motion field
    mutable pointVectorField pointDisplacement_;

private:
    //-- Starting points
    pointField points0_;

// Private Member Functions

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

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

public:
    //-- Runtime type information
    TypeName("displacementMotionSolver");
}
```

pointDisplacement are stored as mutable to adapt to topology changes

Old points position are stored for the definition of the new point position:  
 $X_{\text{new}} = X_{\text{old}} + \Delta X$



# The displacementMotionSolver class

```
Foam::displacementMotionSolver::displacementMotionSolver
(
    const polyMesh& mesh,
    const IOdictionary& dict,
    const word& type
) :
    motionSolver(mesh, dict, type),
    pointDisplacement_
    (
        IOobject
        (
            "pointDisplacement",
            mesh.time().timeName(),
            mesh,
            IOobject::MUST_READ,
            IOobject::AUTO_WRITE
        ),
        pointMesh::New(mesh)
    );
}
```

A pointField is required .  
For velocityMotionSolver  
the motionU file will be  
required

NB: it does not implement  
the solve() member  
function. This is something  
the fvMotionSolver will take  
care of.



# displacementLaplacianFvMotionSolver

```
class displacementLaplacianFvMotionSolver
{
    public displacementMotionSolver,
    public fvMotionSolverCore
}

// Private data

// Cell-centre motion field
mutable volVectorField cellDisplacement_;

// Optionally read point-position field. Used
// boundary conditions.
mutable autoPtr<pointVectorField> pointLocation_;

// Diffusivity used to control the motion
autoPtr<motionDiffusivity> diffusivityPtr_;

// Frozen points
label frozenPointsZone_;
```

// Private Member Functions

```
// Disallow default bitwise copy construct
displacementLaplacianFvMotionSolver
(
    const displacementLaplacianFvMotionSolver&
);

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

Derived by the displacementMotionSolver. It implements the solution of the point displacement equation.

Diffusivity controls how the boundary motion is distributed onto the internal points of the mesh.

Frozen point are points whose position is not affected by the motion solver



# displacementLaplacianFvMotionSolver

```
// Return reference to the cell motion displacement field
volVectorField& cellDisplacement()
{
    return cellDisplacement_;
}

// Return const reference to the cell motion displacement
field
const volVectorField& cellDisplacement() const
{
    return cellDisplacement_;
}

// Return reference to the diffusivity field
motionDiffusivity& diffusivity();

// Return point location obtained from the current motion
field
virtual tmp<pointField> curPoints() const;

// Solve for motion
virtual void solve();

// Update topology
virtual void updateMesh(const mapPolyMesh&);
```

Implements the equation for the solution of the point displacement.

This function handles the changes occurred after a change of topology



# displacementLaplacianFvMotionSolver

```

void Foam::displacementLaplacianFvMotionSolver::solve()
{
    // The points have moved so before interpolation update
    // the motionSolver accordingly
    movePoints(fvMesh_.points());

    diffusivity_.correct();
    pointDisplacement_.boundaryField().updateCoeffs();

    Foam::solve
    (
        fvm::laplacian
        (
            diffusivity_.operator(),
            cellDisplacement_,
            "laplacian(diffusivity,cellDisplacement)"
        )
    );
}

void Foam::displacementLaplacianFvMotionSolver::updateMesh
(
    const mapPolyMesh& mpm
)
{
    displacementMotionSolver::updateMesh(mpm);

    // Update diffusivity. Note two stage to make sure old one
    // is de-registered before creating/registering new one.
    diffusivityPtr_.clear();
}

```

Solves the diffusion equation of point motion



# displacementLaplacianFvMotionSolver

```

Foam::tmp<Foam::pointField>
Foam::displacementLaplacianFvMotionSolver::curPoints() const
{
    volPointInterpolation::New(fvMesh_).interpolate
    (
        cellDisplacement_,
        pointDisplacement_
    );

    if (pointLocation_.valid())
    {
        pointLocation_().internalField() =
            points0_()
            + pointDisplacement_.internalField();
        pointLocation_().correctBoundaryConditions();

        tmp<pointField> tcurPoints
        (
            points0() + pointDisplacement_.internalField()
        );

        if (frozenPointsZone_ != -1)
        {
            const pointZone& pz = fvMesh_.pointZones()[frozenPointsZone_];

            forAll(pz, i)
            {
                tcurPoints()[pz[i]] = points0()[pz[i]];
            }
        }
    }
}

```

Position of points after the displacement

Frozen points are assigned the point0\_ position



# displacementLaplacianFvMotionSolver

```
Foam::tmp<Foam::pointField>
Foam::displacementLaplacianFvMotionSolver::curPoints() const
{
    volPointInterpolation::New(fvMesh_).interpolate
    (
        cellDisplacement_,
        pointDisplacement_
    );

    if (pointLocation_.valid())
    {
        pointLocation_().internalField() =
            points0()
            + pointDisplacement_.internalField();
        pointLocation_().correctBoundaryConditions();

        tmp<pointField> tcurPoints
        (
            points0() + pointDisplacement_.internalField()
        );

        if (frozenPointsZone_ != -1)
        {
            const pointZone& pz = fvMesh_.pointZones()[frozenPointsZone_];

            forAll(pz, i)
            {
                tcurPoints()[pz[i]] = points0()[pz[i]];
            }
        }
    }
}
```

Position of points after the displacement

Frozen points are assigned the point0\_ position

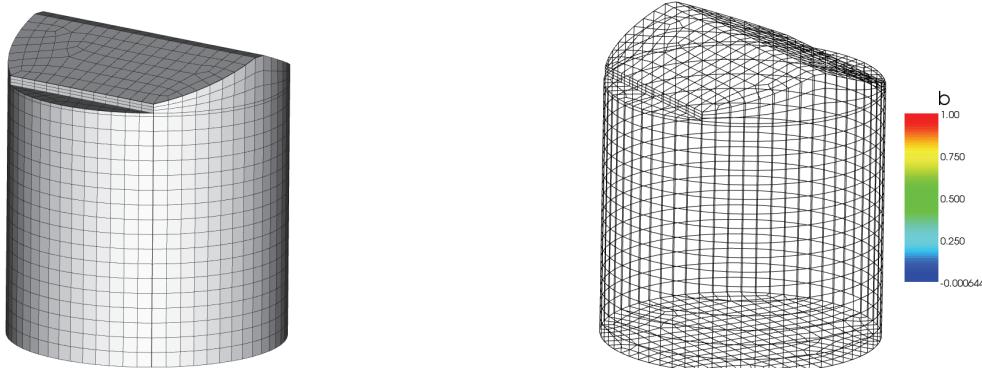


## Dynamic mesh for ICE

- Mesh deformation can be an interesting technique, it is fast and straightforward.
- The main limitation is the quality level of the mesh that is reached when the piston is in the proximity of the top dead center.
- One can think of starting with a very coarse mesh in order to have an acceptable level of refinement in proximity of the upper position of the piston.
- This is bad for modeling the charge motion inside the cylinder: turbulence is not well captured.... Useless
- There are two way of getting rid of this inconvenience:
  - ✓ Maintain a constant level of refinement adding or removing a set of layers when performing the compression/expansion stroke
  - ✓ Deform the mesh until the quality is acceptable, then map the solution filed onto a new mesh complying with the quality requirements: aspect ratio, skewness and non orthogonality.

# Topological changes: layering

- The topology of the mesh is changed, layers of cells are removed during compression and added during expansion phase.
- Need of having a region with structured mesh (layered mesh) were to perform the topology change
- This strategy can be performed in a fully automatic way once the user has defined the boundary motion and where to add/remove layers and

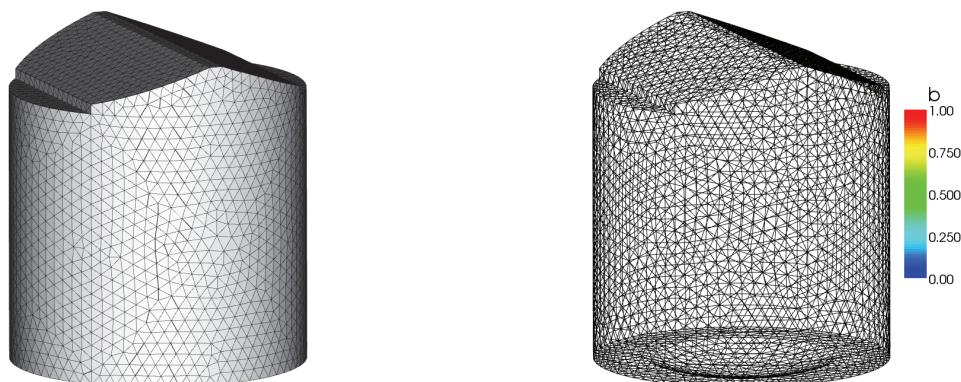


- Not all the engines can be modeled this way. Think about a GDI engine with a shaped piston top.



# Mesh deformation and remapping

- The topology of the mesh is not changed, connectivity and matrix shape remain the same.
- The switch from one mesh to the new one can be done in a discrete manner once a set of grid is provided
- This approach does not have limitations on the mesh complexity



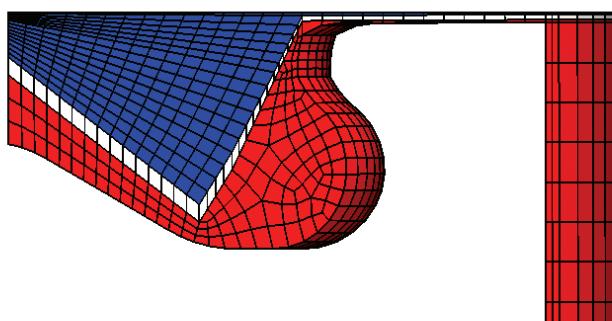
# Some considerations

- Layering techniques allows to preserve mesh quality and to keep the number of cells limited.
- Layering can be used in conjunction with sliding interface and mesh deformation in regions where layered mesh cannot be created.
- Internal combustion engine cylinders are complex devices: they have multiple components moving simultaneously and with different motion.
- Mesh to mesh interpolation needs the usage of high order interpolation schemes, at least of the same order of the schemes adopted for the discretization of operators.
- For LES simulation high order is mandatory, in order not to loose energy components.
- Rarely only one strategy is used to model, rather frequently a combination of approaches is adopted.
- Some example ...



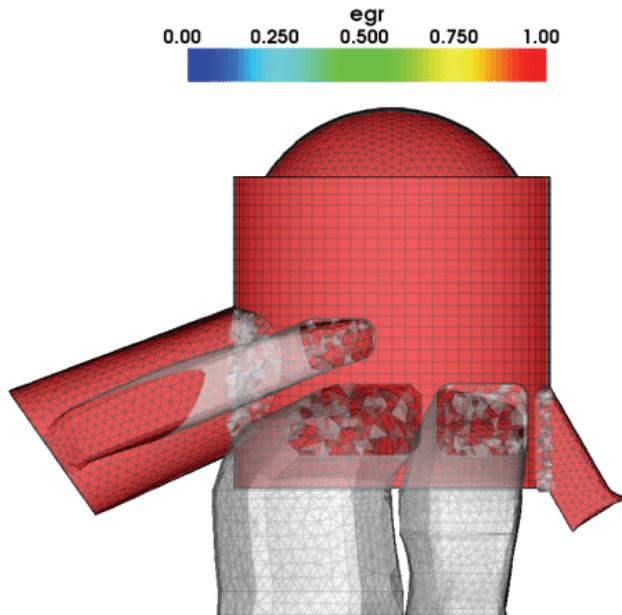
## Examples

- Layering can be operated in order to guarantee also mesh orientation (spray modeling, high velocity jets)



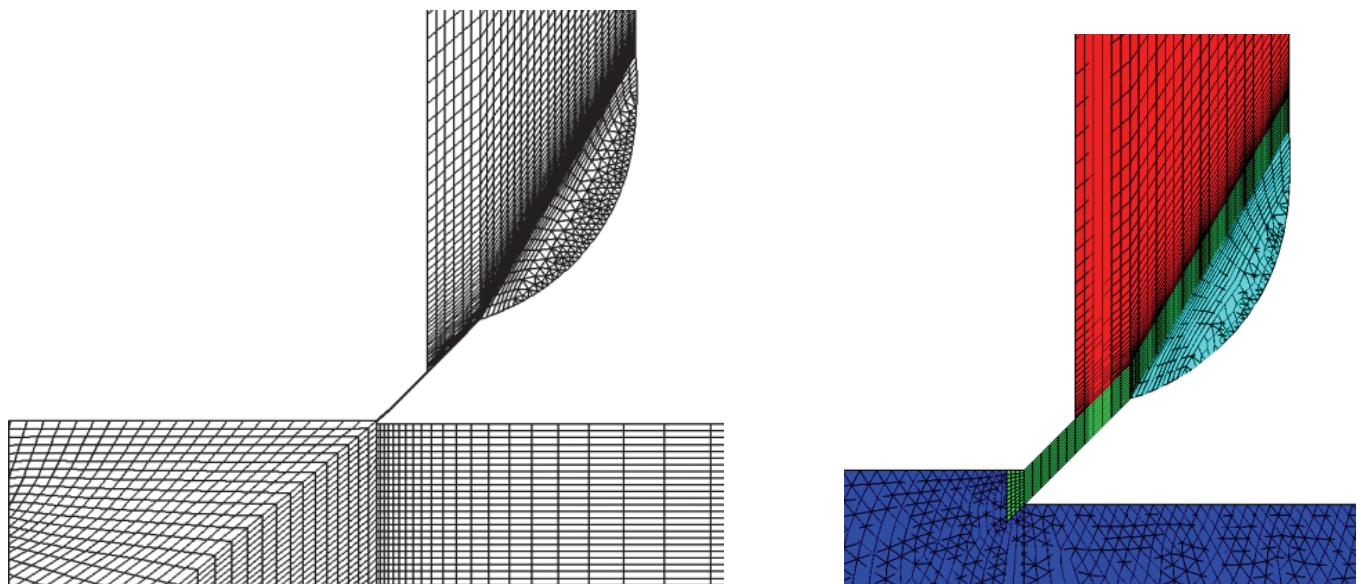
## Examples: layering plus sliding interface

- Sliding interface with partial overlap can be used for two stroke engines



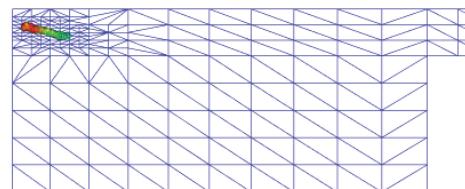
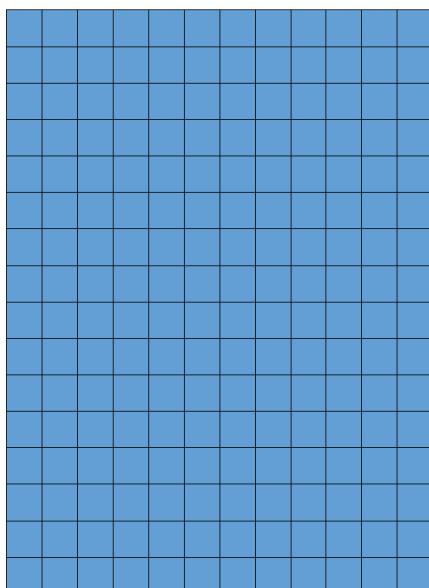
## Examples: layering plus sliding interface

- Layering performed on the top of the poppet valve.
- Sliding interface allows the motion of the valve



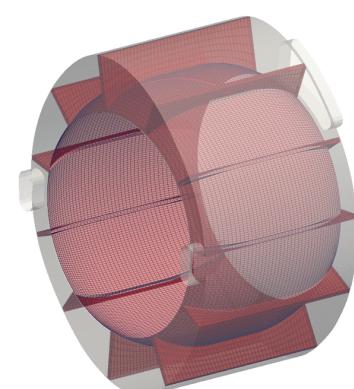
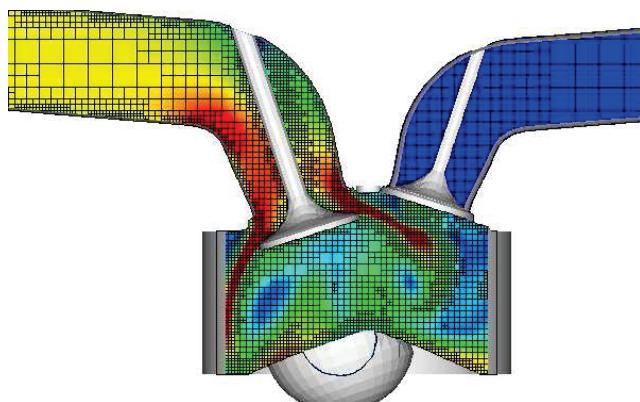
## Examples: mesh refinement

- In some regions it may be needed to refine the mesh to capture correctly steep gradients of certain quantities
- Typical example is the gradient in concentration of chemical species which may control diffusion controlled mixing: evaporation of a fuel



## Examples: automatic remeshing

- Eventually it is possible to generate a new mesh at every time step
- This needs to be done automatically, for example exploiting a mesh template that is adapted quickly to the new time step configuration
- Cartesian grid generator with cut cell features are usually applied
- Especially with the cut cell adjustment the treatment of turbulence becomes a relevant issue at boundaries where the cut cell operates



# Mesh handling & generation

- ★ Mesh quality must be kept as high as possible during the whole cycle simulation
- ★ **Topological changes** may be needed to model the vane motion and the compression:
  - ★ Sliding interfaces (top, bottom, lateral)
  - ★ Layer addition and removal strategy
  - ★ Mesh to mesh interpolation
- ★ **Accurate definition and creation of the mesh, computational burden**
- ★ **Immersed boundary method (IBM)** may be not adequate to the resolution of the boundary layer in tiny gaps
- ★ **Mesh deformation and mesh to mesh interpolation**, along with an automatic and parametric mesh generation, have been implemented

Minimum number of information

Quick and automatic mesh generation

High quality mesh



## Methodology

- ★ Each mesh is deformed for a rotation of  $\Delta\theta$ .
- ★ Solution fields are mapped from one mesh to the following one.

Mesh 1

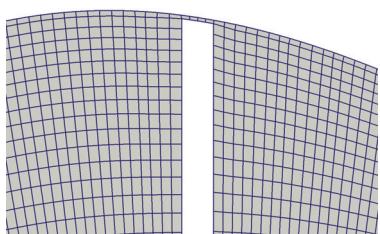


Mesh 1\*

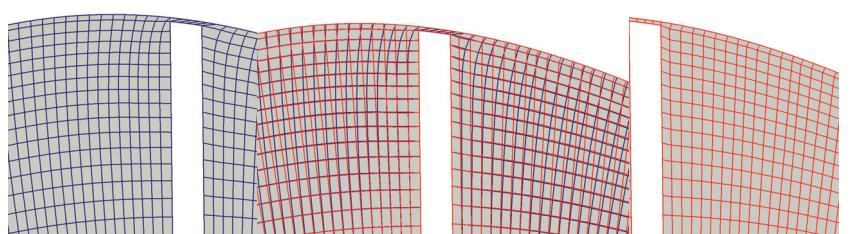


Mesh 2

**Mesh deformation**



**Generation of new mesh**

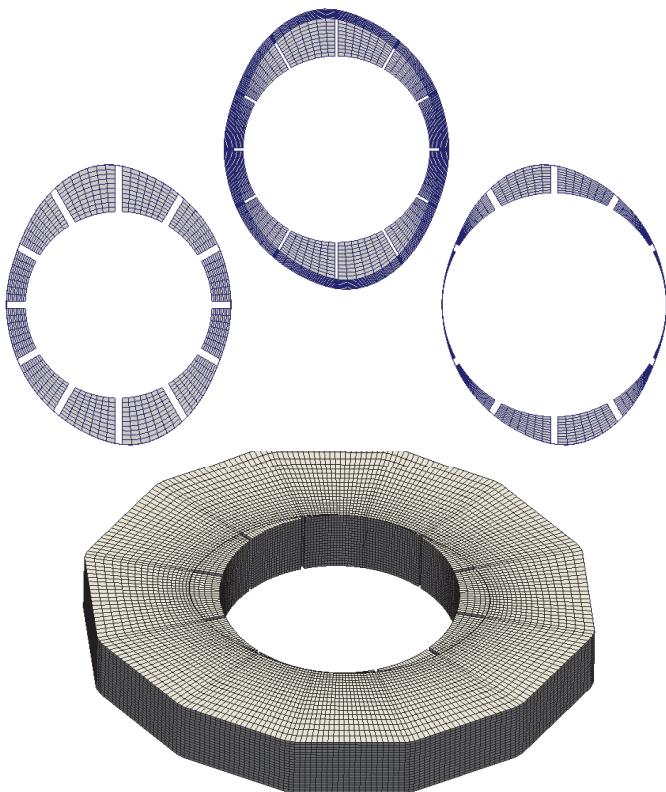


- ★ A set of mesh is dynamically created at specific crank angles on the basis of **runtime check** of mesh quality.

# Mesh generation: base mesh

## ROTOR

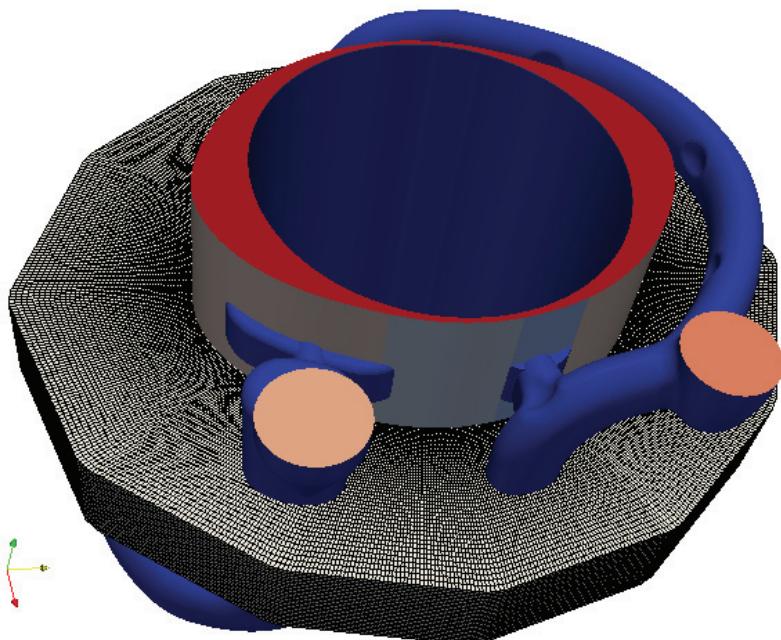
- ✧ Minimum number of parametric data:
  - ✧ Number of vanes
  - ✧ Vane height
  - ✧ Vane thickness
  - ✧ Mesh spacing (x,y,z)
- ✧ Mesh is completely hexahedral with controlled aspect ratio and non orthogonality
- ✧ Stator profile defined by a function in cylindrical coordinates
- ✧ Every base mesh is created setting a rotation equal to the shaft rotation



**ICE GROUP**  
POLITECNICO DI MILANO

# Mesh generation

- ★ Static part of the mesh is generated from a surface representation of the machine (triangulated surface, for instance stl) resorting to a built-in mesh generator (snappyHexMesh).

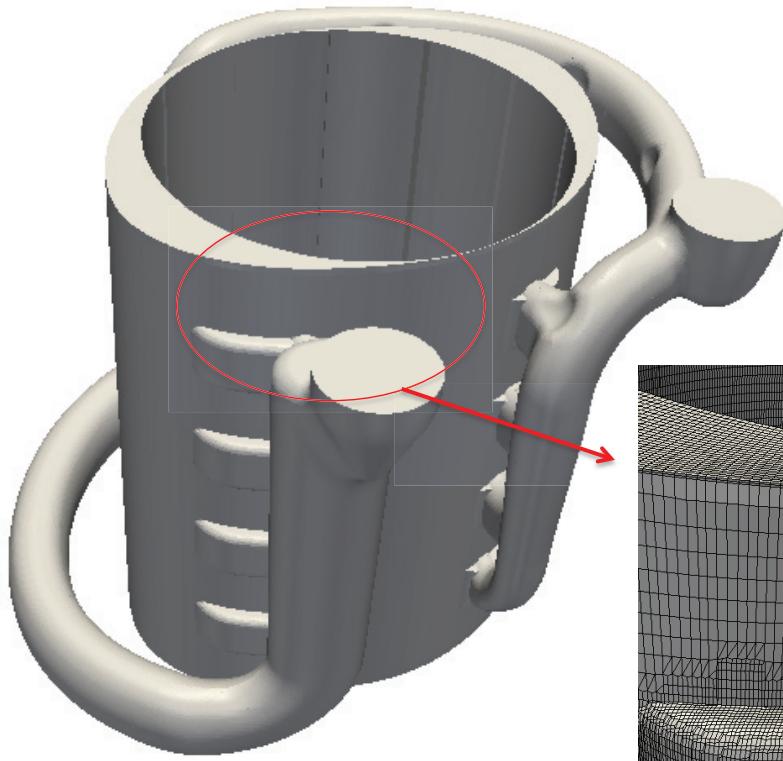


## STATOR

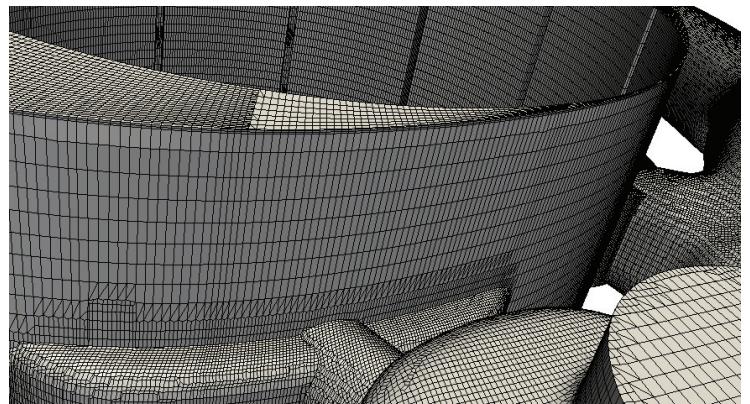
- ✧ Outbound cells are eliminated
- ✧ Controlled refinement is performed for edge capturing (octree)
- ✧ Morphing algorithm to adapt vertexes to the real surface curvature

**ICE GROUP**  
POLITECNICO DI MILANO

## Final mesh



- ★ Mesh accounts for vane tip-stator and vane top/blade bottom-stator leakages



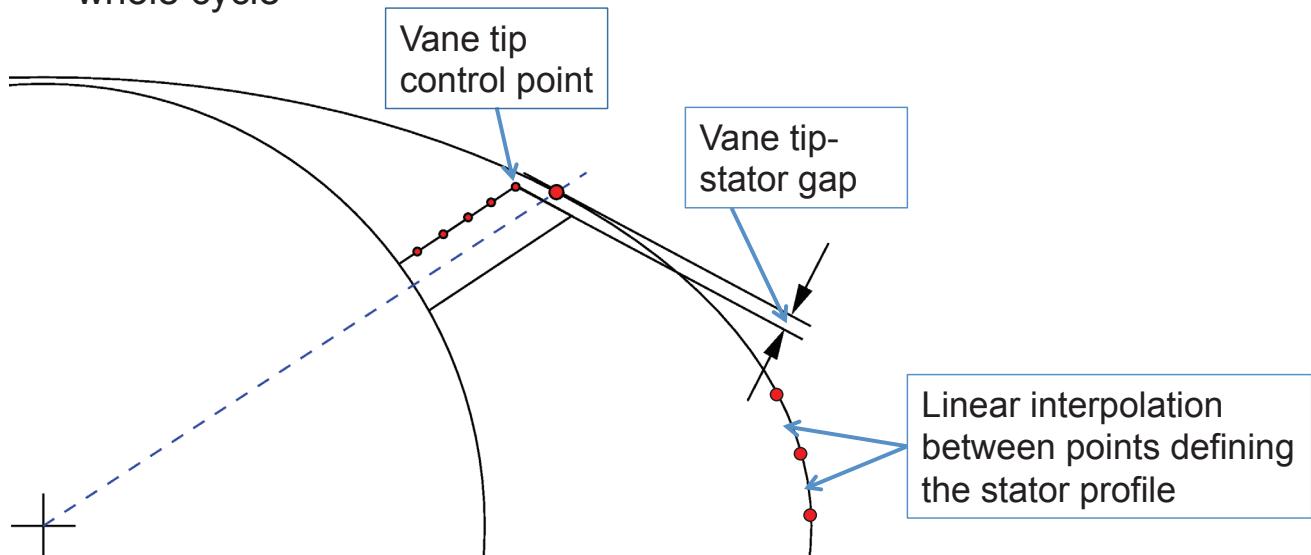
## Mesh motion

- ★ Boundary motion is prescribed for the following parts:
  - ★ **Shaft** or inner region of the vane: displacement depends on the revolution speed and time
  - ★ **Vane**: vanes are rotated and translated radially to follow the desired stator profile
- ★ Inner mesh points are moved solving the point displacement Laplace equation:
$$\nabla \cdot (\gamma \nabla \mathbf{x}) = 0$$
- ★ The new point position depends on the point diffusion algorithm adopted:

$$x_{new} = x_{old} + \Delta x$$

# Blade motion

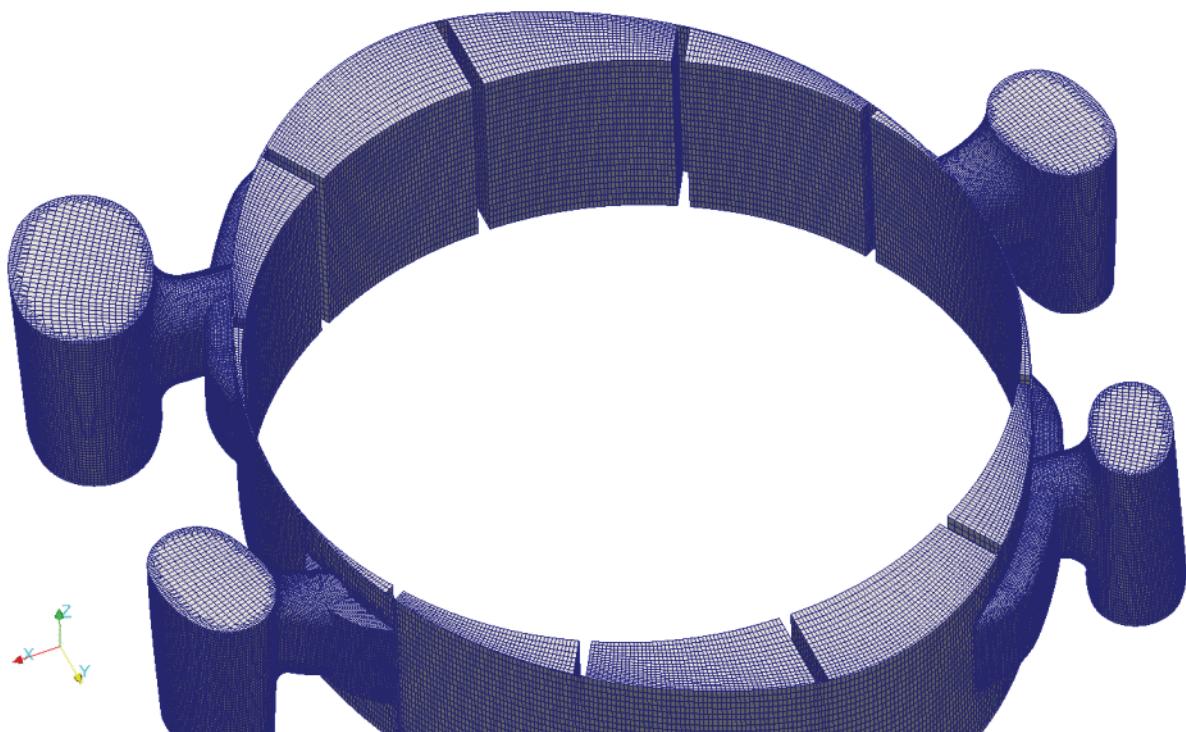
- ★ The gap between stator and vane tip is assumed constant along the whole cycle



- ★ The displacement of points belonging to vane sides is proportional to the displacement in the radial direction of the vane tip control point

**ICE GROUP**  
POLITECNICO DI MILANO

## Mesh motion: example



**ICE GROUP**  
POLITECNICO DI MILANO

# How it is implemented

```

class prescribedRotaryMachineMesh
{
    public rotaryMachineMesh
    {
        // Private data

        displacementLaplacianFvMotionSolver motionSolver_;

        //-- The series being used to describe the stator profile
        interpolationTable<scalar> profileLaw_;

        //-- Reference to initial point field
        pointField points0_;

        //-- Reference to initial angular position of points
        scalarField pointToAngle0_;

        //-- Number of vanes
        scalar numberOfVanels;

        //-- Diameter of the rotor
        scalar innerRotorDiameter_;

        //-- Inner rotor index
        label innerRotorIndex_;

        //-- List of patchIndex for tips
        labelList tipIndex_;
    }
}

```

Definition of a custom fvMesh type, similar to engineMesh to handle specific geometric paramenters

Static selection of the motionSolver type

Definition of the starto profile as points to be interpolated



# How it is implemented

```

void Foam::prescribedRotaryMachineMesh::move()
{
    pointField newPoints(points());

    forAll(tipPatch.meshPoints(), pointI)
    {
        if(!isMoved[tipPatch.meshPoints()[pointI]])
        {
            isMoved[tipPatch.meshPoints()[pointI]] = true;
            point p = newPoints[tipPatch.meshPoints()[pointI]];

            vector displacement = direction[tipI]*tipDisplacement;

            newPoints[tipPatch.meshPoints()[pointI]] = p + displacement;
            tipPatchDisplacement[pointI] = newPoints[tipPatch.meshPoints()[pointI]] - points0()
[tipPatch.meshPoints()[pointI]];
        }
    }

    motionSolver_.pointDisplacement().boundaryField()[tipIndex()][tipI] == tipPatchDisplacement;

    motionSolver_.solve();
}

```

Customize calculation of the displacement of a certain patch (the blade tip)

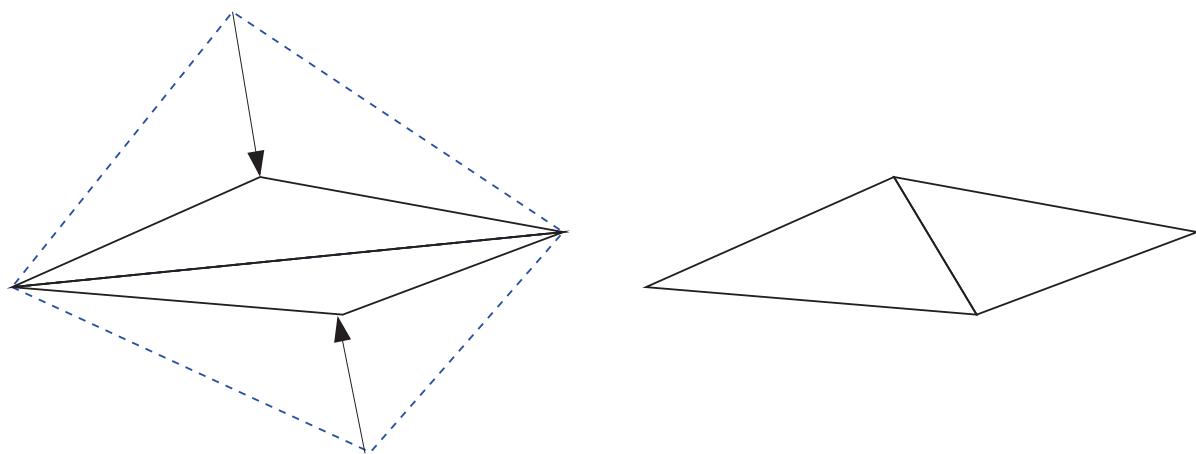
The motionSolver diffuses the displacement of the boundary to the internal points

This motion is enforced on the pointDisplacement boundary field



## Other techniques: tetrahedral reconnection

- Tetrahedral meshes are attractive for their simplicity and easiness in generating them.
- They are frequently used coupled to mesh deformation with mesh to mes remapping
- A strategy which allow to preserve good mesh quality during the deformation of a mesh is the tetrahedral reconnection



## Other techniques: tetrahedral reconnection

- Applicability of the tetrahedral reconnection algorithm for Internal Combustion Engine simulations



# Immersed boundary method

## Considerations

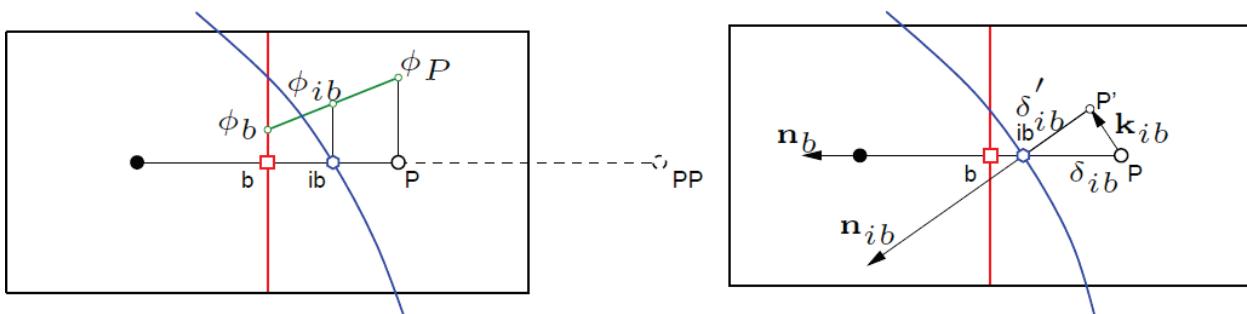
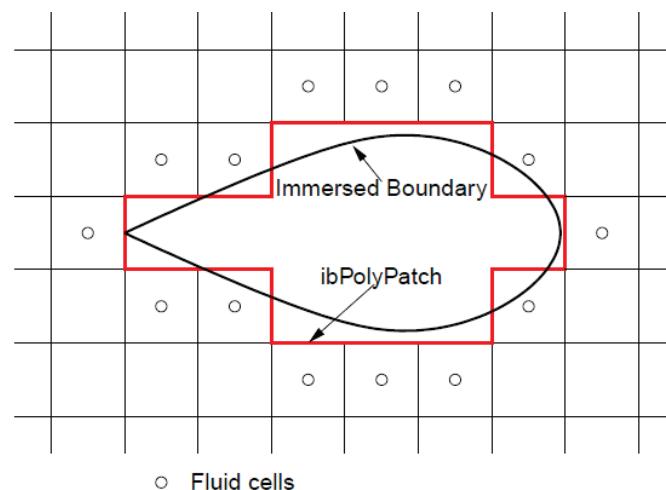
- One of the advantages of using an immersed boundary method is that grid generation is much easier, because a body does not necessarily have to conform a Cartesian grid.
- An immersed boundary method can handle moving boundaries, due to the stationary non-deforming Cartesian grid.

## The concept

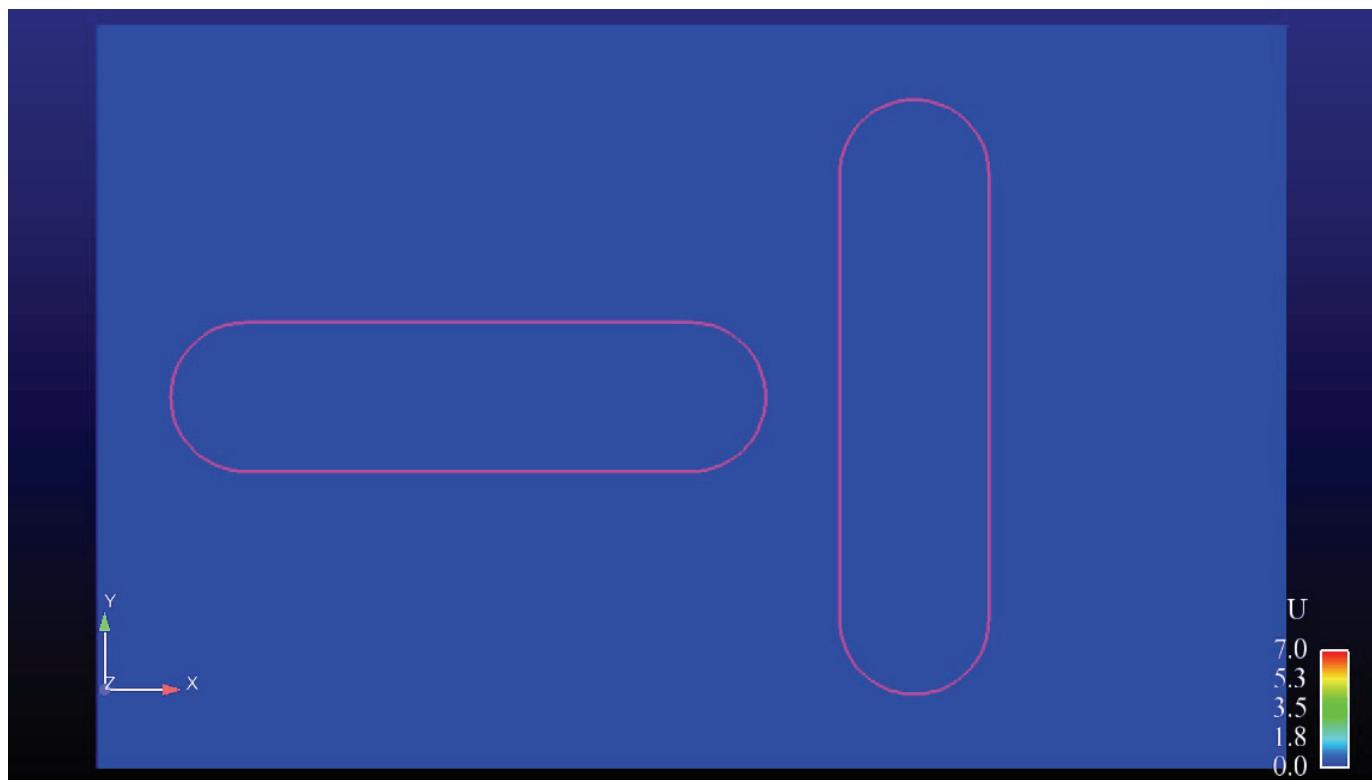
- The Navier Stokes equations will be discretized on a non-boundary conforming cartesian grid and the boundary condition will be imposed indirectly through modifications of equation.
- The modification takes the form of a forcing function in the governing equations that reproduces the effect of the boundary.
  - **continuous forcing** : the forcing function is included into the momentum equation
  - **discrete forcing** : the governing equations are discretized neglecting the immersed boundary then the discretization in the cells near the IB is adjusted to account for their presence using interpolation schemes



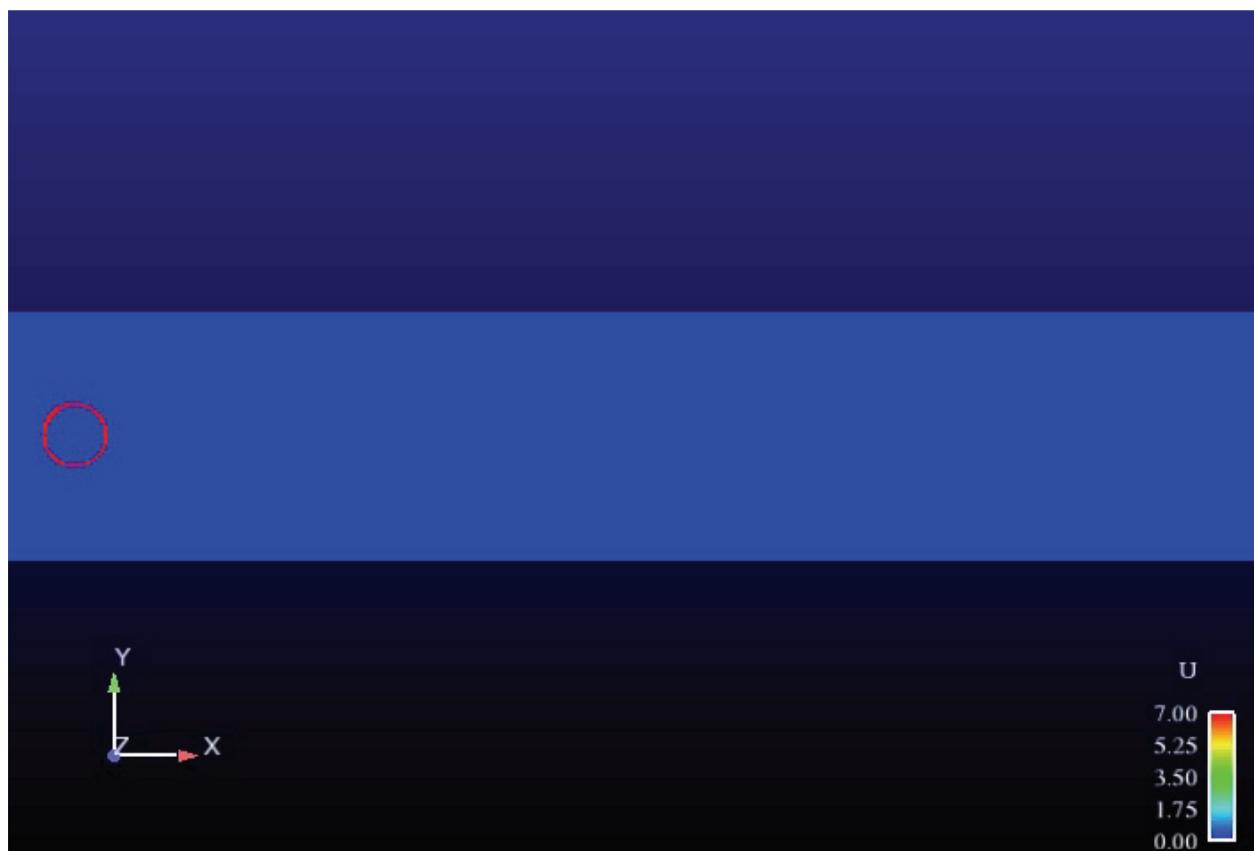
# Immersed boundary method



# IB method: discrete forcing

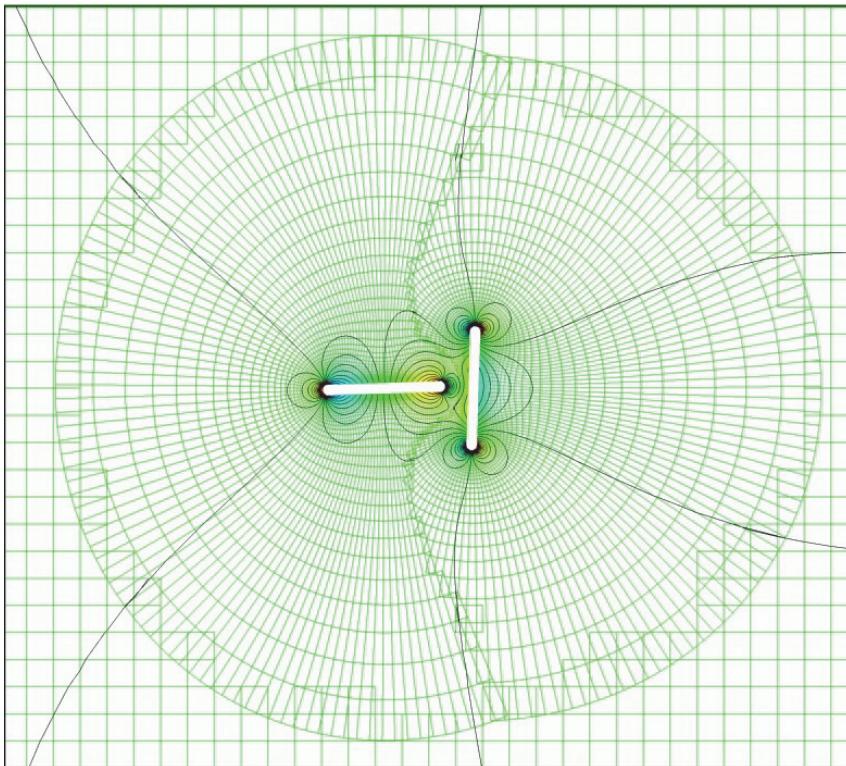


# IB method: discrete forcing



# Overset meshes

- Body-fitted component meshes: preserving quality and near-wall resolution



## Acknowledgments

- Prof. Tommaso Lucchini
- Dr. Augusto Della Torre



# Development of OpenFOAM® libraries for combustion and spray modeling

**Tommaso Lucchini**

Department of Energy, Politecnico di Milano

---

## Topics



### Spray and combustion modeling using OpenFOAM technology

- 1) Overview of models and examples available in the standard OpenFOAM distribution
  - lagrangian library
  - combustionModels library
  - Premixed combustion in OpenFOAM
  - Solvers: reactingFoam, engineFoam, fireFoam, reactingParcelFilmFoam, sprayFoam, ....
  - Examples (tutorials)
- 2) Development of libraries for wall-film, spray and combustion models in OpenFOAM at Politecnico di Milano
  - Library dieselSprayPolimi for Diesel and GDI spray modeling
  - flameletCombustionModels library for non-premixed combustion with detailed chemistry
  - sparkIgnition library to model flame propagation in SI engines
  - **Experimental validation**

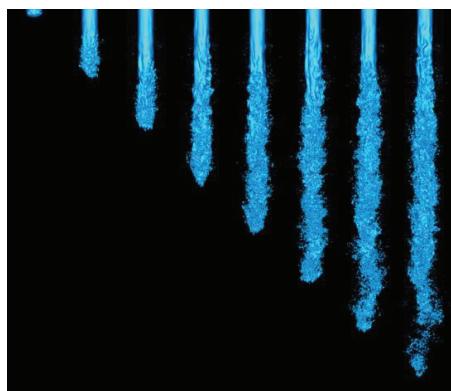
# Introduction

## Spray and combustion modeling....

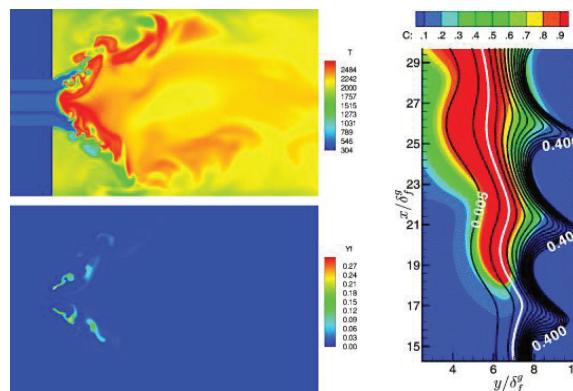
- Very complex topics:

- **Spray:** multiphase flow, strong interaction between liquid and gas phases (breakup, evaporation, air-entrainment,...), scales
- **Combustion:** detailed kinetics, turbulence-chemistry interaction, regimes, flame structure, scales

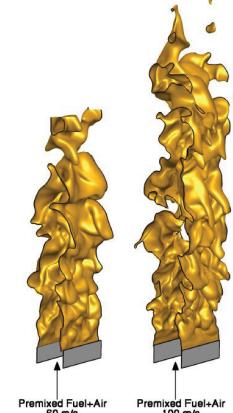
DNS of spray primary breakup



DNS of spray combustion



DNS of premixed flames



T. Lucchini: Development of OpenFOAM® libraries for combustion and spray modeling

3

# Introduction

## Spray and combustion modeling in CFD codes

### 1) General purpose codes: reliable and fast approaches

- **Spray:** robust particle tracking supporting polyhedral grids. Reliable sub-models to be applied to a wide range of problems.
- **Combustion:** fast models to achieve results in an acceptable amount of computational time. Models based on one global reaction with assumptions about flame structure.

### 2) Dedicated codes: detailed approaches

- **Spray:** robustness but also availability of a wide range of sub-models to accurately describe the evolution of liquid jets (breakup, evaporation, wall-impingement...) in a wide range of operating conditions.
- **Combustion:** models based on detailed kinetics, including turbulence-chemistry interaction. Prediction of pollutant emissions (soot, NOx). Detailed modeling of the spark-ignition process.

# Introduction

## Spray and combustion modeling in CFD codes

### 1) General purpose codes: reliable and fast approaches

- Spray and combustion models available in general CFD codes
- **OpenFOAM® official distribution**

### 2) Dedicated codes: detailed approaches

- Spray and combustion models available in specific codes
- **Lib-ICE: libraries and solvers for IC engine simulations based on OpenFOAM**

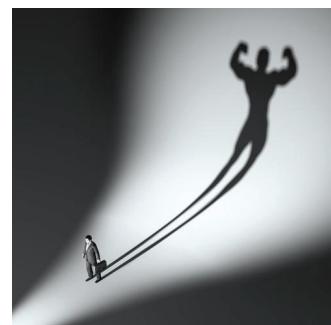
T. Lucchini: Development of OpenFOAM® libraries for combustion and spray modeling

5

## Spray combustion in OpenFOAM



*“I want to simulate  
combustion with  
OpenFOAM”*



- 1) Take a look at the [user guide](#) to see which solvers are available
- 2) Look inside the code:

```
> sol
> ls
basic      compressible      DNS          financial      incompressible    multiphase
combustion  discreteMethods  electromagnetics  heatTransfer  lagrangian       stressAnalysis
> cd combustion
> ls
chemFoam   coldEngineFoam   engineFoam   fireFoam    PDRFoam   reactingFoam  XiFoam
> sol
> cd lagrangian
coalChemistryFoam  icoUncoupledKinematicParcelFoam  reactingParcelFoam  uncoupledKinematicParcelFoam
DPMFoam           reactingParcelFilmFoam           sprayFoam
```

# Spray combustion in OpenFOAM



## Combustion solvers

|                              |                                                                                                                                                                                                                                       |
|------------------------------|---------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------|
| chemFoam                     | Solver for chemistry problems (auto-ignition, ...)                                                                                                                                                                                    |
| coldEngineFoam<br>engineFoam | Solvers for in internal combustion engines.                                                                                                                                                                                           |
| fireFoam                     | Transient PIMPLE solver for Fires and turbulent diffusion flames with reacting Lagrangian parcels, surface film and pyrolysis modelling.                                                                                              |
| XiFoam<br>PDRFoam            | Solver for compressible premixed/partially-premixed combustion with turbulence modelling.<br>PDR (porosity/distributed resistance) modelling is included to handle regions containing blockages which cannot be resolved by the mesh. |
| reactingFoam                 | Solver for combustion with chemical reactions.                                                                                                                                                                                        |

T. Lucchini: Development of OpenFOAM® libraries for combustion and spray modeling

7

# Spray combustion in OpenFOAM



## Solvers including particle tracking (lagrangian)

|                        |                                                                                                                                                                                                 |
|------------------------|-------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------|
| coalChemistryFoam      | Transient solver for compressible, turbulent flow, With coal and limestone parcel injections, energy source, and combustion.                                                                    |
| reactingParcelFilmFoam | Transient PIMPLE solver for compressible, laminar or turbulent flow with reacting Lagrangian parcels, and surface film modelling.                                                               |
| reactingParcelFoam     | Transient PIMPLE solver for compressible, laminar or turbulent flow with reacting multiphase Lagrangian parcels, including run-time selectable finite volume options, e.g. sources, constraints |
| sprayFoam              | Transient PIMPLE solver for compressible, laminar or turbulent flow with spray parcels.                                                                                                         |

T. Lucchini: Development of OpenFOAM® libraries for combustion and spray modeling

8

# Spray combustion in OpenFOAM



Code structure for combustion modeling in OpenFOAM: solvers, library components

## Solvers (\$FOAM\_SOL)

- combustion
  - chemFoam
  - coldEngineFoam
  - engineFoam
  - fireFoam
  - PDRFoam
  - reactingFoam
  - XiFoam
- lagrangian
  - coalChemistryFoam
  - reactingParcelFoam
  - reactingParcelFilmFoam
  - sprayEngineFoam
  - sprayFoam

## Library (\$FOAM\_SRC)

- combustionModels
  - combustionModel
  - diffusion
  - FSD
  - infinitelyFastChemistry
  - laminar
  - noCombustion
  - PaSR
  - psiCombustionModel
  - rhoCombustionModel
  - singleStepCombustion
- combustionModels
  - basic
  - coalCombustion
  - distributionModels
  - intermediate
  - solidParticle
  - spray

T. Lucchini: Development of OpenFOAM® libraries for combustion and spray modeling

9

# chemFoam



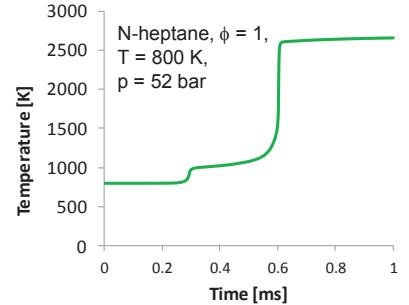
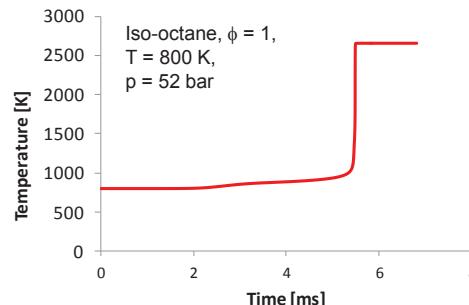
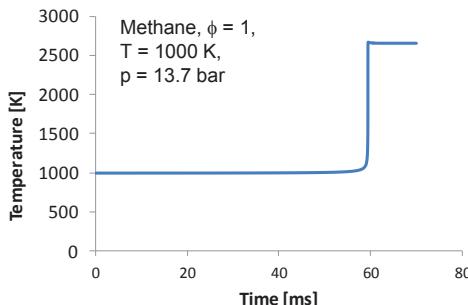
## Solver for chemistry problems

- Designed for use on single cell cases to provide comparison against other chemistry solvers
- Single cell mesh created on-the-fly
- Fields created on the fly from the initial conditions

## Tutorials available in \$FOAM\_TUTORIALS/combustion/chemFoam

- Auto-ignition of fuel-air mixture (hydrogen, methane, iso-octane, n-heptane)
- **Case setup:** see constant directory and set properly the dictionaries initialConditions, thermophysicalProperties, chemistryProperties. A kinetic mechanism is required to be provided in chemkin or Foam format.
- **Output data** written in the chemFoam.out file

## Some results....



T. Lucchini: Development of OpenFOAM® libraries for combustion and spray modeling

10

# engineFoam / coldEngineFoam



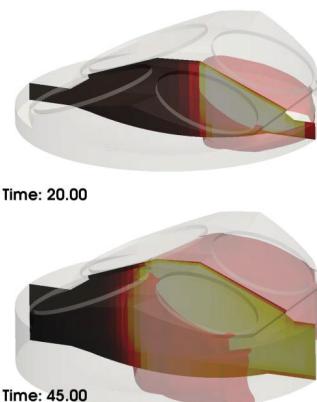
## Solver for premixed combustion in IC engines

- It operates only on deforming grids (`$FOAM_SRC/engine/engineMesh`) and is only suitable for compression/combustion simulations (no valve motion).
- The Weller 2-equation model is used to predict flame propagation. `coldEngineFoam` does not include combustion.
- Simplified chemistry (air, fuel, burned gases), deposition model for ignition, no pollutant prediction, no spray.

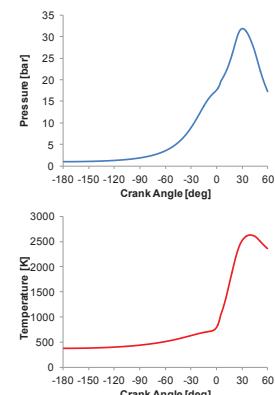
Tutorial available in `$FOAM_TUTORIALS/combustion/engineFoam`

- Compression and combustion in an engine combustion chamber (`kivaTest`)
- **Case setup:** `combustionProperties` (combustion model parameters), `thermophysicalProperties` (fuel and burned gas properties), `engineGeometry` (speed, bore, stroke, ...). Mesh boundary definition is important (piston, liner, cylinder head)
- **Post-processing:** the `logSummary` file includes average in-cylinder data
- To **initialize** the in-cylinder flow field use the `engineSwirl` utility

### Flame propagation



### Outputs from logSummary



T. Lucchini: Development of OpenFOAM® libraries for combustion and spray modeling

11

# fireFoam

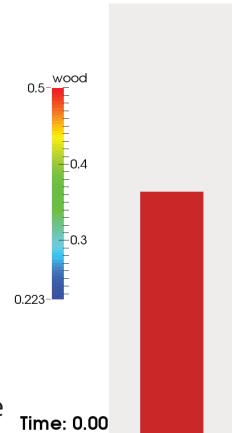


## Transient solver for fires and turbulent diffusion flames

- Comprehensive solver for fire extinction.
- A combustion model is employed for diffusion flame propagation
- Spray modeling: Lagrangian. It can be fuel or the extinguisher
- Surface film modeling (finite volume with region extrusion from mesh boundary)
- Pyrolysis modeling to describe the consumption of the solid domain by the fire.
- Radiation is taken into account by the energy equation

Tutorials available in `$FOAM_TUTORIALS/combustion/fireFoam/les`

- Development of pool fires: `smallPoolFire2D`, `smallPoolFire3D`
- Fire suppression by a liquid film: `flameSpreadWaterSuppressionPanel`
- Burning of a wood panel: `oppositeBurningPanel`
- Case setup:
  - `combustionProperties`: set the combustion model
  - `pyrolysisZones`: definition of zones of the domain and suitable model where pyrolysis should be considered
  - `radiationProperties`: selection of radiation model
  - `reactingCloud1Properties` : spray properties (sub-models)
  - `surfaceFilmProperties` : liquid film model
  - `thermophysicalProperties`: selection of thermodynamic properties for the gas phase.



T. Lucchini: Development of OpenFOAM® libraries for combustion and spray modeling

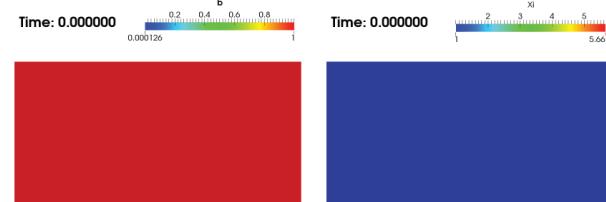
12

## Solver for premixed combustion

- Flame propagation by Weller combustion model (2 equations)
- Simplified chemistry (air, fuel, burned gases), deposition model for ignition, no pollutant prediction
- In PDRFoam, porosity/distributed resistance modeling is included to handle regions containing blockages that cannot be resolved by the mesh. Typical application of PDRFoam: fire simulation.

**XiFoam tutorial available in \$FOAM\_TUTORIALS/combustion/XiFoam/ras**

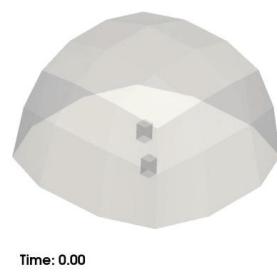
- **Flame propagation in a constant pressure vessel**
- **Case setup:** combustionProperties (combustion model parameters, laminar flame speed correlation, ....), thermophysicalProperties (fuel and burned gas properties)



**PDRFoam tutorial available in \$FOAM\_TUTORIALS/combustion/XiFoam/ras**

- **Flame propagation in an open environment with obstacles**
- **Case setup:** same as XiFoam except for the PDRProperties dictionary where the model to include effects of porosity/distributed resistance on flame wrinkling has to be included.
- Suitable fields needs to be initialized to specify the volume porosity and obstacle properties

Additional parameters are



Time: 0.00

T. Lucchini: Development of OpenFOAM® libraries for combustion and spray modeling

13

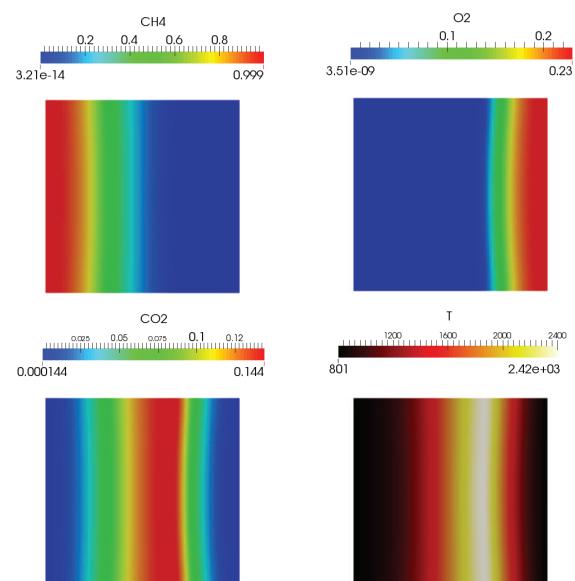
# reactingFoam

## Solver for combustion with detailed chemistry

- It accounts for reactions in the gas phase computed by a kinetic mechanism.
- Two different models available: laminar (no turbulence-chemistry interaction) and PaSR (Partially Stirred Reactor) which a simplified version of EDC.

**Tutorial available in \$FOAM\_TUTORIALS/combustion/reactingFoam/ras**

- **Counterflow methane laminar diffusion flame**
  - Laminar chemistry with one global reaction
- **Case setup:**
  - combustionProperties : selection of combustion model and related tuning coeffs (Cmix for PaSR)
  - chemistryProperties : parameters for integration of stiff chemistry (solver, tolerances, ...)
  - thermophysicalProperties : thermodynamic properties of the mixture, selection of the kinetic mechanism (CHEMKIN® format is supported)



T. Lucchini: Development of OpenFOAM® libraries for combustion and spray modeling

14

# coalChemistryFoam



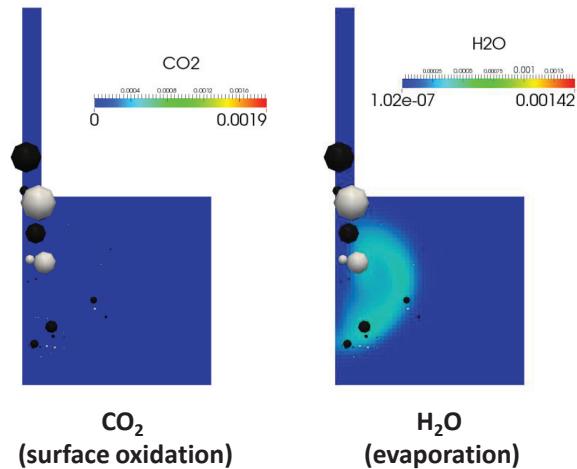
**Transient solver for compressible, turbulent flow, with coal and limestone parcel injections, energy source, and combustion.**

- Coal and limestone are treated as Lagrangian particles
- Sub-models for coal particles (reacting): forces (gravity, drag), injection, dispersion, interaction with walls, composition, phase change, devolatilisation, collision, surface reaction, radiation. Coal is a mixture of solid and liquid.
- Sub-models for limestone particles (inert): forces (gravity, drag), injection, dispersion, interaction with walls, composition, collision, surface reaction, radiation.

**Tutorial available in \$FOAM\_TUTORIALS/lagrangian/coalChemistryFoam**

- **Evolution of coal (black) and limestone (white) particles in a Siwek chamber.**
- **Case setup:**

- combustionProperties,  
chemistryProperties,  
thermophysicalProperties : gas phase  
(same as reactingFoam);
- coalCloud1Properties : selection of  
sub-models for the coal cloud;
- limestoneCloud1Properties : selection  
of sub-models for the limestone cloud;



T. Lucchini: Development of OpenFOAM® libraries for combustion and spray modeling

15

# reactingParcelFoam



**Transient solver for compressible, laminar or turbulent flow with reacting multiphase Lagrangian parcels.**

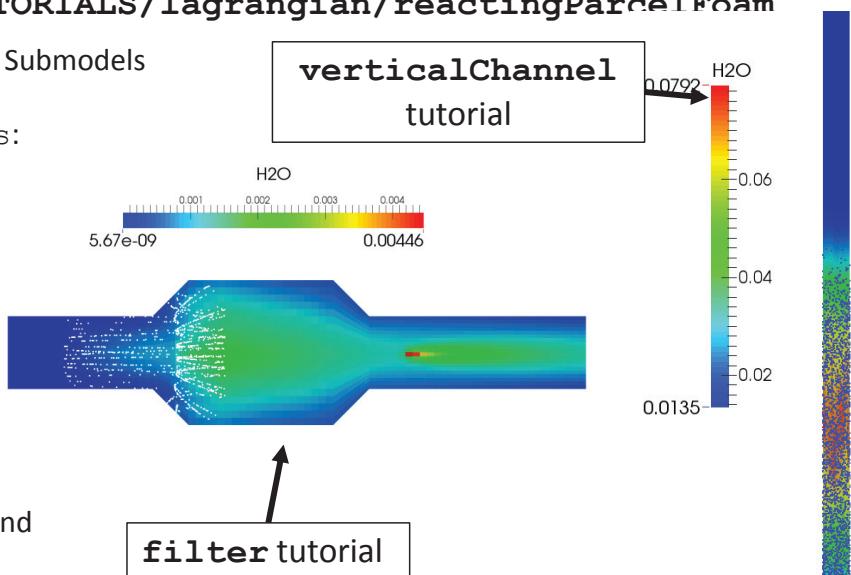
- Examples of use: problems involving liquid droplet evolution in gas phase with evaporation, reactions, ...
- Droplets are lagrangian particles.

**Tutorials available in \$FOAM\_TUTORIALS/lagrangian/reactingParcelFoam**

- **Case setup** same as reactingFoam. Submodels for lagrangian particles to be set in reactingCloud1Properties:

- Forces
- Injection
- Dispersion
- Patch interaction
- Heat transfer
- Phase change
- Devolatilisation
- Surface reaction
- Collision

Possibility to decouple flow and Lagrangian particle solution



T. Lucchini: Development of OpenFOAM® libraries for combustion and spray modeling

16

# reactingParcelFilmFoam

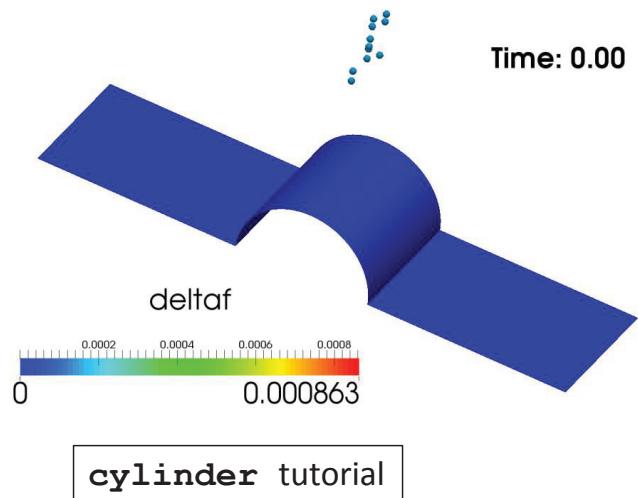


Transient solver solver for compressible, laminar or turbulent flow with reacting multiphase Lagrangian parcels and liquid film.

- Same features of reactingParcelFoam but it includes also the evolution of liquid film.
- Liquid film equations (mass, momentum and energy) are solved on an extruded volume (1 cell thickness, regionMesh) from the wall boundaries and mutual interactions between particles, film and gas phase are taken into account.

Tutorials available in `$FOAM_TUTORIALS/lagrangian/reactingParcelFilmFoam`

- **Case setup** same as reactingParcelFoam, except for an additional dictionary called `surfaceFilmProperties`, where all the sub-models necessary for wall-film modeling need to be specified and properly set.
- Utility `extrudeToRegionMesh` is used to generate the wall-film mesh from specified mesh boundary patches where wall film has to be simulated.
- Tutorials: cylinder, hotBoxes, splashPanel, rivuletPanel



T. Lucchini: Development of OpenFOAM® libraries for combustion and spray modeling

17

# sprayFoam / sprayEngineFoam

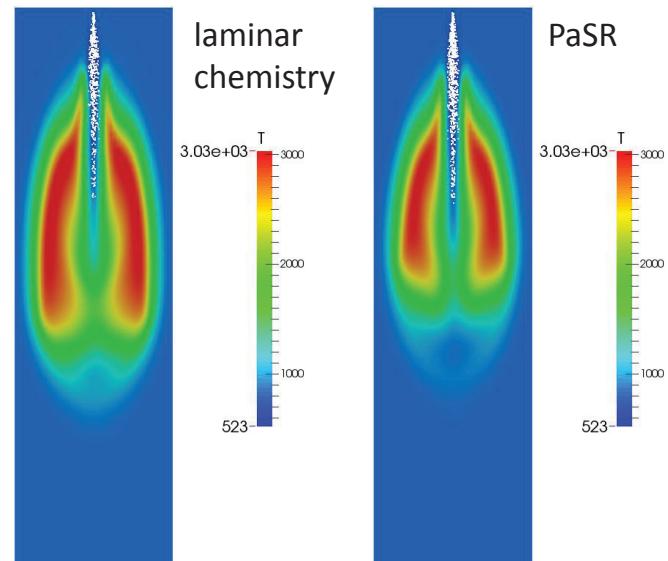


Transient solver solver for spray combustion (in engines)

- A spray is a reacting particle (same as reactingParcelFoam) but with two additional sub-models: breakup and atomization.
- `sprayFoam` is for constant-volume combustion, `sprayEngineFoam` is for engine (same features of `engineFoam` regarding mesh handling)

Tutorials available in `$FOAM_TUTORIALS/lagrangian/sprayFoam`

- **Case setup** same as reactingParcelFoam, but with the `sprayCloudProperties` where additional submodels (`atomizationModel`, `breakupModel`) must be specified.
- Make your own tutorial for `sprayEngineFoam` by crossing `kivaTest` and `aachenBomb`



T. Lucchini: Development of OpenFOAM® libraries for combustion and spray modeling

18

# Spray models in OpenFOAM



|                          |                                                                         |
|--------------------------|-------------------------------------------------------------------------|
| <b>Atomization</b>       | LISA, Blob (liquid sheet)                                               |
| <b>Secondary breakup</b> | e-TAB, Pilch-Erdman, Reitz-Diwakar, KHRT, Schmel , TAB                  |
| <b>Evaporation</b>       | Spalding, Flash boiling                                                 |
| <b>Collision</b>         | O'Rourke, trajectory                                                    |
| <b>Injection</b>         | Cone, cone-nozzle, patch, ....                                          |
| <b>Heat transfer</b>     | Ranz-Marshall                                                           |
| <b>Drag</b>              | Ergun-Wen-Yu, non-spherical, Plessis-Masliyah, spherical, Wen-Yu        |
| <b>Wall interaction</b>  | Local interaction, Multiple interaction, rebound, standard (Bai-Gosman) |

T. Lucchini: Development of OpenFOAM® libraries for combustion and spray modeling

19

# Wall-film models in OpenFOAM



## Two available approaches:

- `kinematicSingleLayer`: solution of mass and momentum equation (no energy).
- `thermoSingleLayer`: solution of mass, momentum and energy equation with wall film evaporation

## Available sub-models:

- Kinematic:
  - Thermodynamic: film properties depending or not on temperature
  - Turbulence: laminar flow (turbulent in the future?)
  - Forces: contact-angle, thermo-capillary
  - Particle injection: curvature separation, dripping injection
- Thermo-physical:
  - Radiation: primary, standard
  - Viscosity: constant, temperature dependent, thixotropic
  - Phase change: solidification, standard

|                    | Premixed                                                                                                         | Non-premixed                                                                                                                                       |
|--------------------|------------------------------------------------------------------------------------------------------------------|----------------------------------------------------------------------------------------------------------------------------------------------------|
| Reduced chemistry  | <ul style="list-style-type: none"> <li>• Weller model</li> <li>• FSD (for LES)</li> <li>• Single-step</li> </ul> | <ul style="list-style-type: none"> <li>• FSD (for LES)</li> <li>• Infinitely fast chemistry</li> <li>• Diffusion</li> <li>• Single-step</li> </ul> |
| Detailed chemistry | <ul style="list-style-type: none"> <li>• Laminar</li> <li>• PaSR</li> </ul>                                      | <ul style="list-style-type: none"> <li>• Laminar</li> <li>• PaSR</li> </ul>                                                                        |

## Spray and combustion in OpenFOAM

### What can be done

- OpenFOAM incorporates suitable libraries to describe unsteady flame propagation (diffusive and premixed), fuel-air mixture formation (Lagrangian spray), wall-film and pyrolysis.
- Most of the developments were dedicated to unsteady approaches with simplified chemistry and sub-models for spray.
  - ⇒ **Simplified is not a synonym of inaccurate.**
- Pollutant prediction: possible by incorporating global reactions for NO<sub>x</sub> and soot.
- Which combustion applications are suitable for the standard OpenFOAM libraries?
  - Fire safety
  - Simulation of very large scale devices and all other applications where CPU time is an issue and simplified models are necessary.
  - Mixing

# Spray and combustion in OpenFOAM



## What CANNOT be done

- Detailed chemistry possible only by direct integration of a kinetic mechanism. CPU time drastically increase when detailed mechanisms (more than 50 species and 100 reactions) have to be used.
- Alternative, validated and widely used approaches (chemical equilibrium, PDF, flamelets) are not available.
- No semi-empirical models for soot.
- Spray sub-models: library needs to be extended to incorporate approaches to simulate fuel-air mixing in IC engines
- ⇒ Detailed combustion simulation of industrial combustors or engine is not possible with standard OpenFOAM distribution.....
- **However, OpenFOAM is an open-source code and on the basis of the available library more detailed models can be implemented and used!**

T. Lucchini: Development of OpenFOAM® libraries for combustion and spray modeling

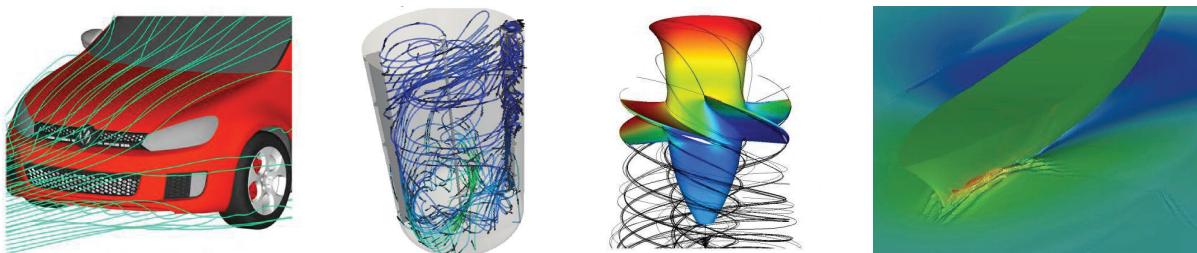
23

# Developing OpenFOAM



OpenFOAM is.....

## 1) A ready to be used CFD code



## 2) An open-source library for advanced CFD modeling

- Add an equation to an existing solver
- Make a new solver
- Implement a new sub-model
- Make a new library

**OpenFOAM**  
=   
**capabilities for  
advanced CFD modeling**

# Developing OpenFOAM for combustion



- a) Add a transport equation to an existing solver for post-processing purpose
- **Mixture fraction equation in sprayFoam**

Modified pimple loop in sprayFoam

```
/* some code here */
#include "createFields.H"

/* some code here */
while (pimple.loop())
{
    #include "UEqn.H"
    #include "YEqn.H"
    #include "EEqn.H"
    #include "ZEqn.H"

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

        if (pimple.turbCorr())
        {
            turbulence->correct();
        }
    }
    /* some code here */
}
```

Mixture fraction field in **createFields.H**

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

Mixture fraction transport equation **ZEqn.H**

```
fvScalarMatrix ZEqn
(
    fvm::ddt(rho, Z)
    + mvConvection->fvmDiv(phi, Z)
    - fvm::laplacian(turbulence->muEff(), Z)
    ==
    parcels.Srho(i, Yi)
);
ZEqn.relax();
ZEqn.solve(mesh.solver("Yi"));
Z.max(0.0);
```

T. Lucchini: Development of OpenFOAM® libraries for combustion and spray modeling

25

# Developing OpenFOAM for combustion



- a) Add a transport equation to an existing solver for post-processing purpose
- **Mixture fraction equation in sprayFoam**
  - **Mixture fraction variance equation in sprayFoam**

Mixture fraction variance field in **createFields.H**

```
volScalarField Z2
(
    IOobject
    (
        "Z2",
        runTime.timeName(),
        mesh,
        IOobject::MUST_READ,
        IOobject::AUTO_WRITE
    ),
    mesh
);

dimensionedScalar Cchi
(
    combustionProperties.lookup("Cchi")
);
```

Mixture fraction variance transport equation

```
volScalarField Chi =
Cchi*turbulence->epsilon()/turbulence->k()*Z2;

fvScalarMatrix Z2Eqn
(
    fvm::ddt(rho, Z2)
    + mvConvection->fvmDiv(phi, Z2)
    - fvm::laplacian(turbulence->muEff(), Z2)
    ==
    + 2.0*turbulence->muEff()*sqr(mag(fvc::grad(Z)))
    - rho*Chi
);
Z2Eqn.relax();
Z2Eqn.solve(mesh.solver("Yi"));
Z2.max(0.0);
```

T. Lucchini: Development of OpenFOAM® libraries for combustion and spray modeling

26

# Developing OpenFOAM for combustion



- b) Make a new solver

- Premixed combustion with CFM model instead of the  $b-\Xi$  model

b.1) Clone XiFoam solver and call it ECFMFoam

b.2) Modify the bEqn.H file: remove Xi and introduce the  $\Sigma$  transport equation:

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

```
fvScalarMatrix bEqn
(
    fvm::ddt(rho, b)
    + mvConvection->fvmDiv(phi, b)
    - fvm::laplacian(turbulence->alphaEff(), b)
===
    - thermo->rhou() * Su * Sigma
    fvOptions(rho, b)
);
/* some code here */
volScalarField P1=alphaSigma*GammaK*turbulence->epsilon()/turbulence->k();
volScalarField P2=2.0/3.0*fvc::div(U);
volScalarField Ptot=P1+P2;
volScalarField Distraction=betaSigma*Su*(Sigma)/(b + SMALL);

fvScalarMatrix SigmaEqn
(
    fvm::ddt(Sigma)
    + fvm::div(phi/fvc::interpolate(rho), Sigma, "div(phi,Sigma)")
    - fvm::laplacian(turbulence->muEff()/rho, Sigma)
===
    fvm::Sp(Ptot, Sigma)
    - fvm::Sp(Distraction, Sigma)
);
SigmaEqn.solve();
Sigma.max(0.0);
```

# Developing OpenFOAM for combustion



- c) Implement a new spray sub-model

- Huh-Gosman atomization model in lagrangian/spray library

c.1) Derive it from the AtomizationModel generic class

c.2) Add suitable coefficients

c.3) Then implement diameter reduction according to model's equations

```
namespace Foam
{
template<class CloudType>
class HuhGosmanAtomization
:
    public AtomizationModel<CloudType>
{
private:
    scalar C1_;
    scalar C2_;
    scalar C3_;
    scalar C4_;
    scalar C5_;
    scalar Ctau_;
    scalar calcStrippedD(scalar Lt);
    /* some code here */
public:
    virtual void update
    (
        /* arguments */
    ) const;
}
```

```
template<class CloudType>
void Foam::LISAAtomization<CloudType>::update
(
    /* arguments */
) const
{
    /* some code here */

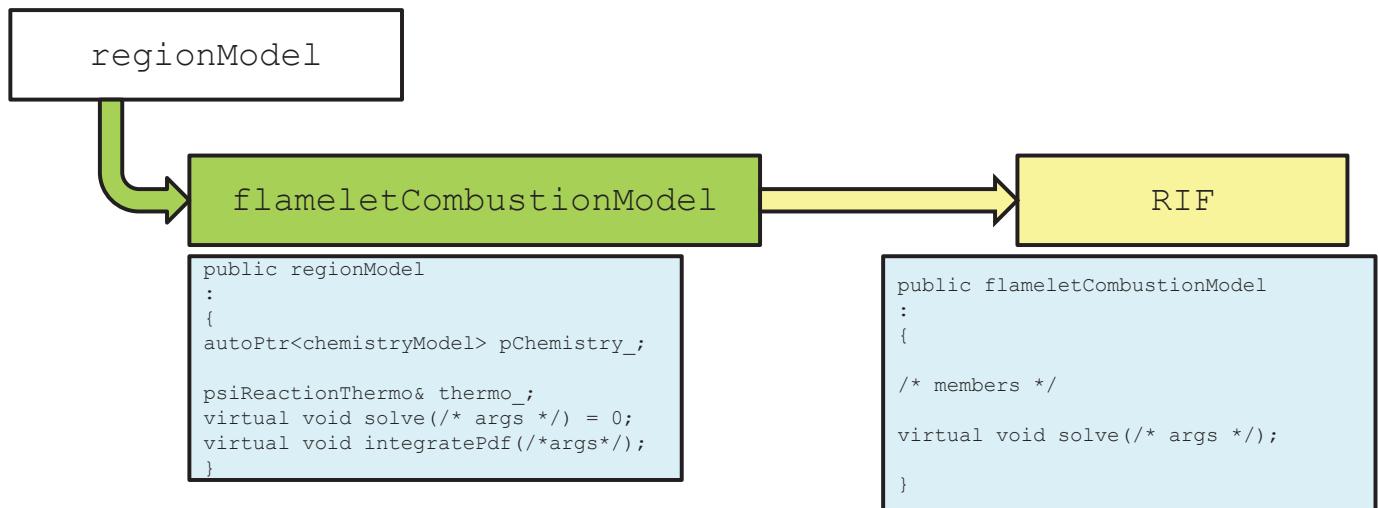
    scalar dD = -C5_ * La/tauA * dt;
    d -= dD;

    /* implement stripping */
}
```

# Developing OpenFOAM for combustion



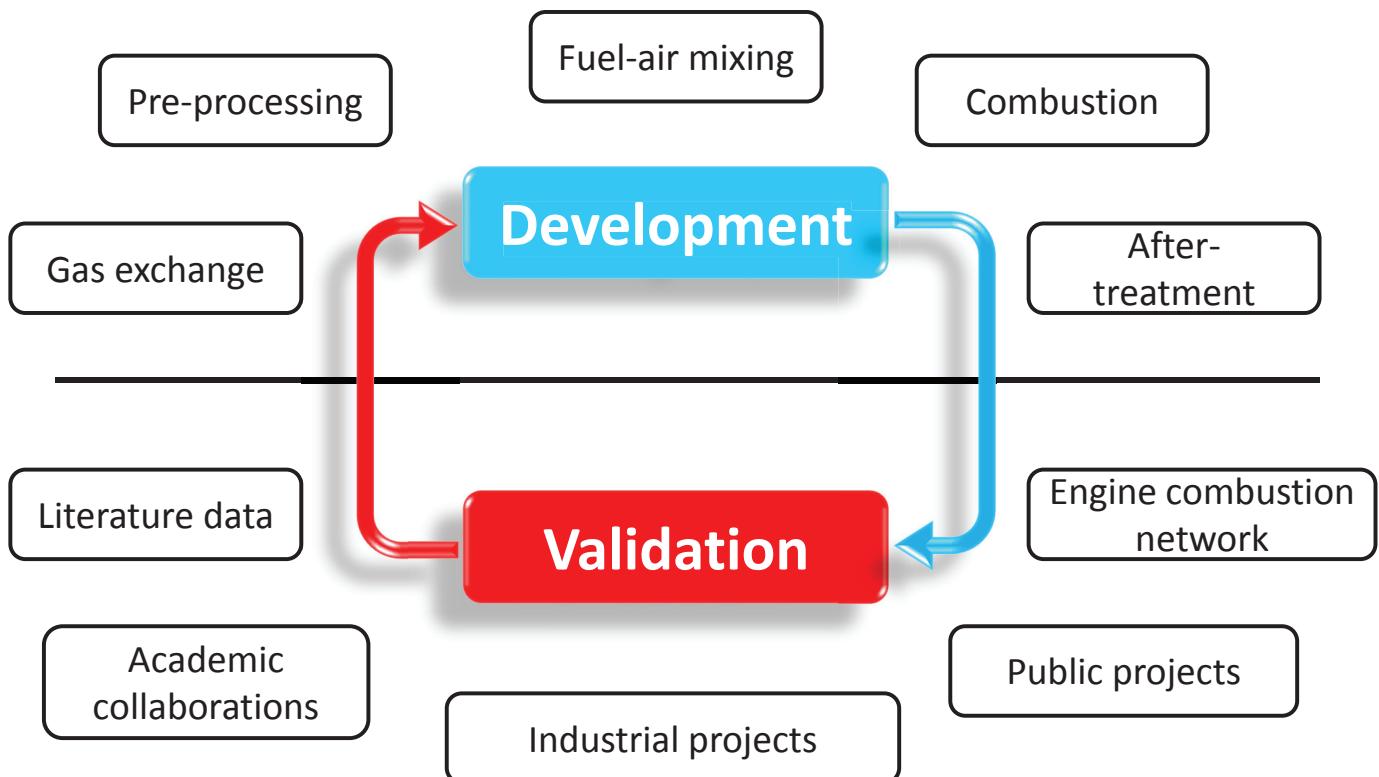
- d) Make a new library: **Flamelet combustion model**
  - d.1) 1d mesh representing the Z-space (regionModel)
  - d.2) Re-use all the thermodynamic objects already implemented (chemistryModel, psiReactionThermo...) to solve flamelet equations
  - d.3) Implement suitable interfaces for information exchange between the CFD and flamelet domain (scalar dissipation rate, beta-pdf integration,...)



T. Lucchini: Development of OpenFOAM® libraries for combustion and spray modeling

29

## OpenFOAM at Polimi





## Spray modeling for engine simulations

- Development and validation of a comprehensive library (`dieselSprayPolimi`) for simulation of fuel air mixing in direct-injection engines (Diesel and GDI).
- Development of a liquid film model (`wallFilmPolimi`) to take fuel impingement effect into account and supporting moving grids

## Diesel combustion

- Provide a complete library of combustion models for Diesel engines:
  - Simplified models with reduced chemistry for engine design
  - Detailed models for advanced combustion modes and diagnostic purposes

## Spark-ignition combustion

- Comprehensive model including ignition (Lagrangian) and flame propagation (Choi and Huh).



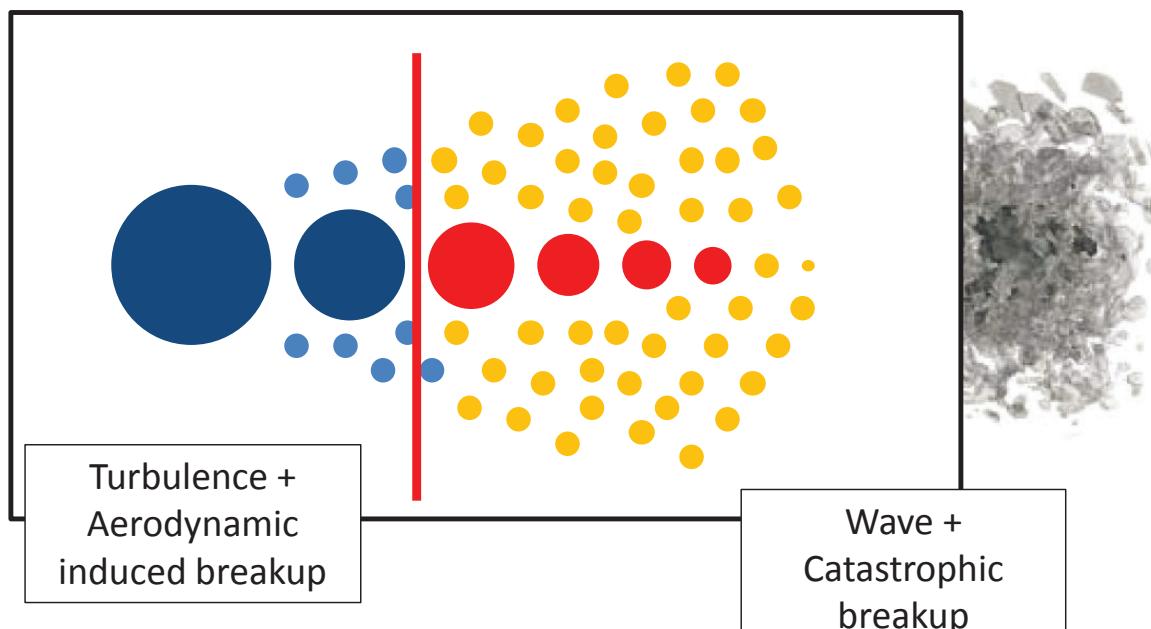
# Spray modeling

## Objectives

- Including new capabilities
  - ✓ Atomization, injection, breakup, evaporation and wall-film model
- Provide full support for engine simulations: field mapping and mesh motion
- Validation and assessment with a large range of experimental data from different sources (literature, industries, research institutes)
  - ⇒ Availability of a comprehensive methodology for the simulation of spray evolution in Direct-Injection engines (diesel and GDI)

# Spray modeling for IC engines

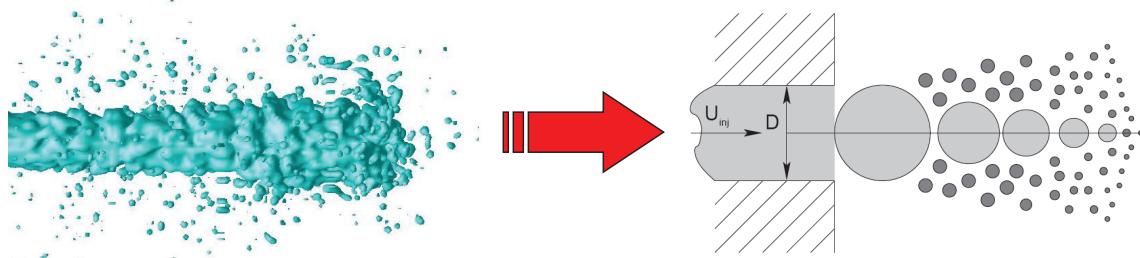
## Comprehensive spray model for Diesel and GDI sprays



- Breakup transition: Weber number, Ohnesorge number, droplet diameter, breakup length.

## Turbulent induced break-up: Huh-Gosman

- **Blob injection:** droplets are injected with the same nozzle diameter and injection velocity.
- Turbulent quantities ( $L_t$ ,  $\tau_t$ ) initialized for each droplet according to the nozzle flow conditions.

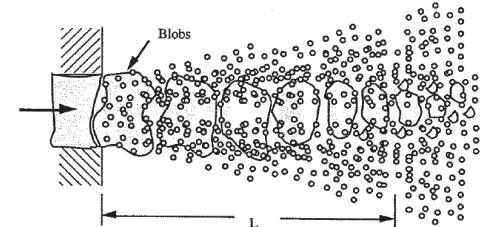
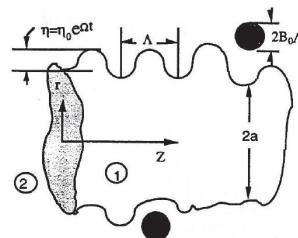


- Liquid jet atomization is modeled as:
  - 1) Diameter reduction of the injected droplets
  - 2) Stripping of secondary droplets from the liquid jet

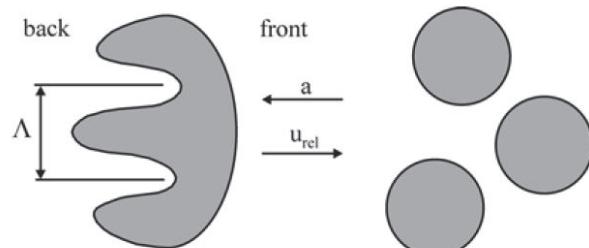
# Spray modeling for IC engines

## Secondary breakup

**Wave (Kelvin-Helmholtz):**  
aerodynamic-induced  
breakup with droplet  
stripping



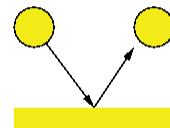
**Rayleigh-Taylor:**  
catastrophic breakup



## Droplet-wall interaction model – Bai and Gosman

$$We = \frac{\rho_l \cdot v_n^2 \cdot d}{\sigma}$$

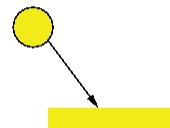
Rebound



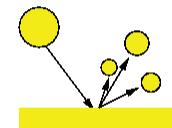
$$La = \frac{\rho_l \cdot \sigma \cdot d}{\mu_l^2}$$

$We < 5.0$

Adhesion



Splash



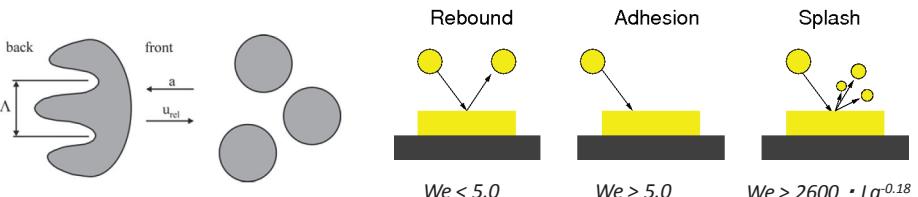
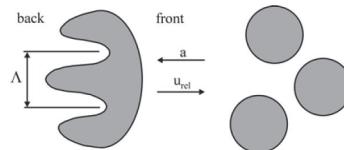
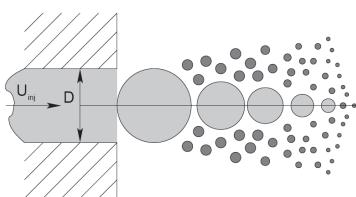
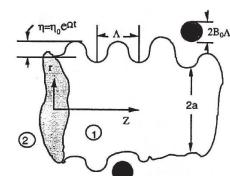
$We > 2600 \cdot La^{-0.18}$

- **Other models:** Ranz-Marshall (heat-transfer), Spalding (evaporation), stochastic turbulent dispersion.
- **Evaporation model:** partially re-written, evaporation rate now computed from mass fraction and not volume fractions.

# Spray modeling for IC engines

## Spray sub-models implemented/revised by Polimi

- Injection models: Huh, Reitz-Bracco, Nurick
- Atomization models: Huh-Gosman, Bianchi
- Breakup model: KHRT
- Wall-interaction: Bai and Gosman, Stanton and Rutland
- Evaporation: based on Spalding mass number



## Spray library improvements

- Lagrangian particle tracking with polyhedral cells
- Mesh-to-mesh mapping of Lagrangian particles
- Full support to moving grids and topological changes

# Spray modeling for IC engines



## dieselSprayPolimi library in Lib-ICE

polimiParticle, polimiCloud

- dieselSprayPolimi
  - injector
    - SOIInjector
  - parcel
  - spray
  - spraySubModels
    - atomizationModel  
**Huh-Gosman, Bianchi**
    - breakupModel  
**KHRT (revised), unified, boilKHRT**
    - collisionModel
    - dispersionModel
    - dragModel
    - **evaporationModel**
    - heatTransferModel
    - injectorModel
      - blobInjector, huhInjector, cavitatingHuhInjector
    - wallModel  
**BaiGosman, NaberReitz**

- Revised implementation of base classes (parcel, spray) and tracking algorithm for mesh motion and mapping
- Revised sub-models (evaporation, breakup)
- New sub-models (injection, atomization, wall)
- New post-processing functions

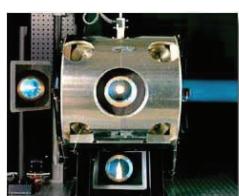
T. Lucchini: Development of OpenFOAM® libraries for combustion and spray modeling

39

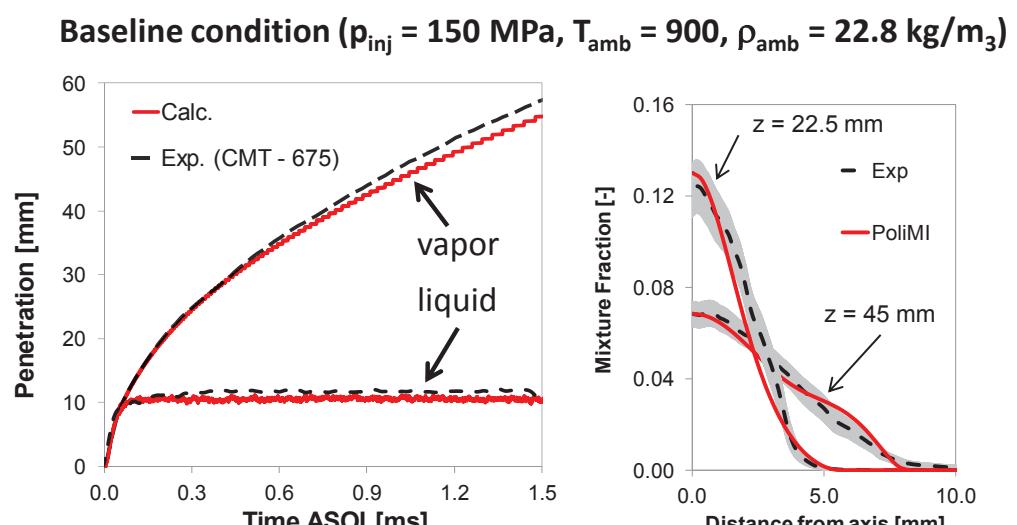
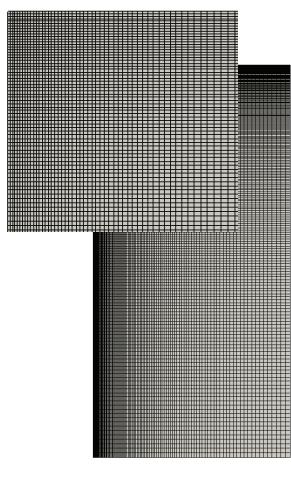
# Spray modeling for IC engines



## Validation: Sandia combustion vessel from ECN



- Fuel: n-dodecane (“Spray-A”), Diesel-like spray
- Effects of ambient conditions and injection pressure on liquid and vapor penetration.
- Spray setup: KHRT for breakup, blob injection
- Turbulence model: standard  $k-\epsilon$  with modified  $C_1$  (1.55)



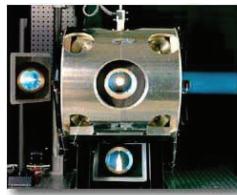
T. Lucchini: Development of OpenFOAM® libraries for combustion and spray modeling

40

# Spray modeling for IC engines

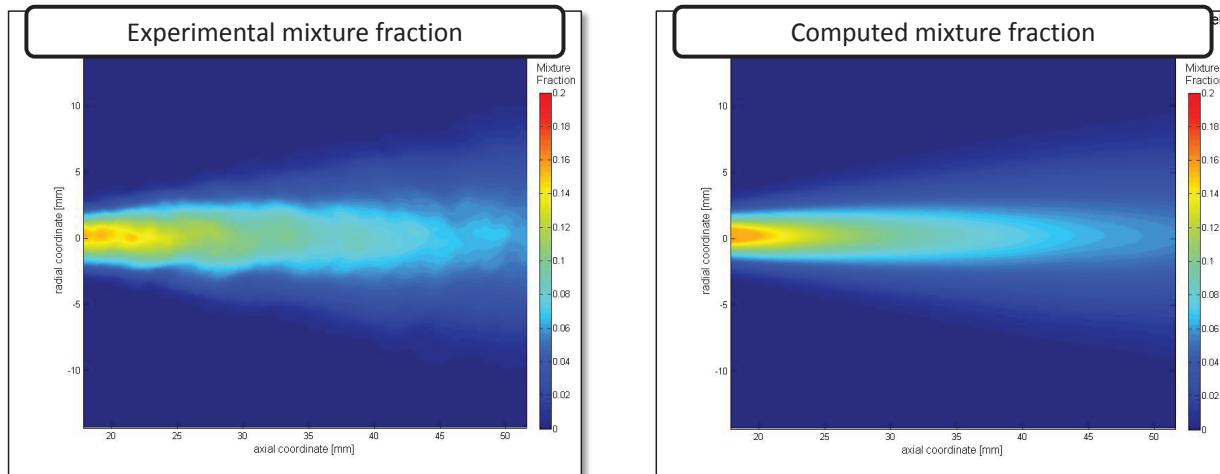


## Validation: Sandia combustion vessel from ECN



- Fuel: n-dodecane (“Spray-A”), Diesel-like spray
- Effects of ambient conditions and injection pressure on liquid and vapor penetration.
- Spray setup: KHRT for breakup, blob injection
- Turbulence model: standard  $k$ - $\epsilon$  with modified  $C_1$  (1.55)

Baseline condition ( $p_{inj} = 150 \text{ MPa}$ ,  $T_{amb} = 900$ ,  $\rho_{amb} = 22.8 \text{ kg/m}^3$ )



T. Lucchini: Development of OpenFOAM® libraries for combustion and spray modeling

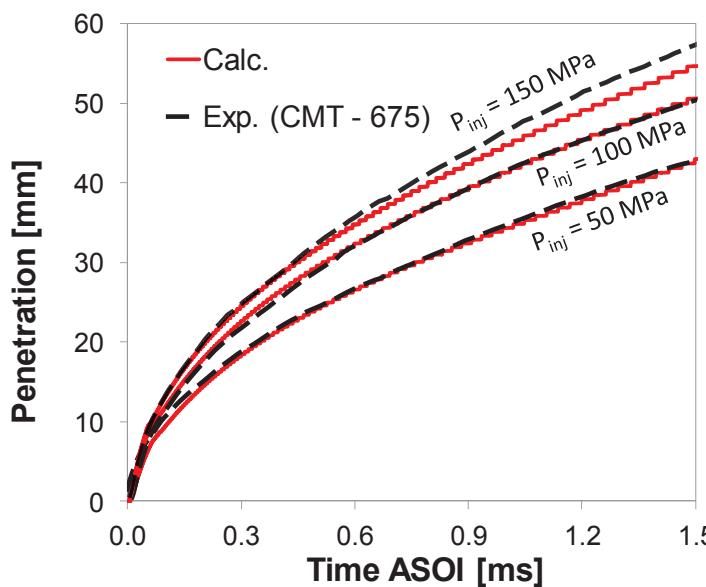
41

# Spray modeling for IC engines

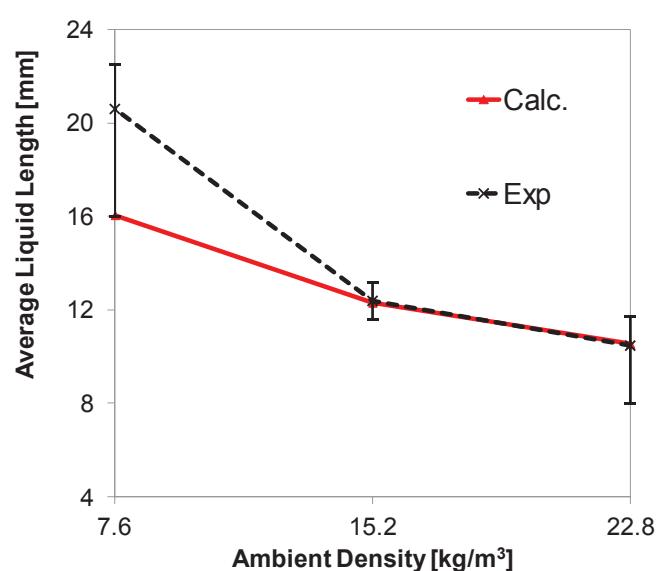


## Validation: Sandia combustion vessel from ECN

Injection pressure effects on vapor penetration



Ambient density effects on liquid penetration



T. Lucchini: Development of OpenFOAM® libraries for combustion and spray modeling

42

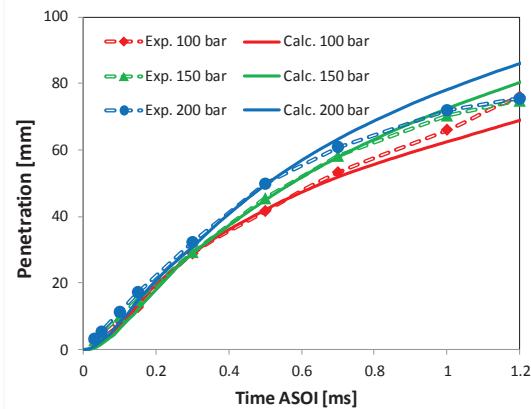
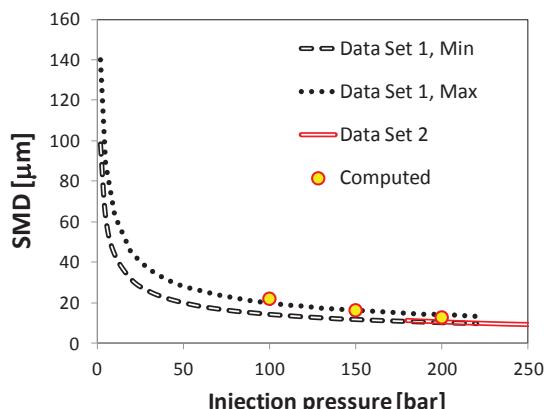
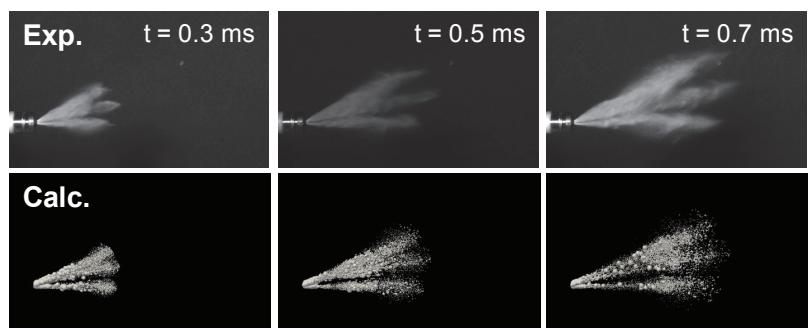
# Spray modeling for IC engines



## Validation: GDI spray, multi-hole

- Atomization model: Huh-Gosman
- Injection model: Huh
- Breakup model: KH-RT
- Non-evaporating conditions

| $p_{inj}$<br>[MPa] | $m_{fuel}$<br>[mg/shot] | $\Delta t_{inj}$<br>[ms] |
|--------------------|-------------------------|--------------------------|
| 10/15/20           | 48/50/50                | 3.7/3/2.6                |



Experimental data are courtesy of Ing. Montanaro and Dr. Allocca (CNR-Istituto Motori, Naples)

T. Lucchini: Development of OpenFOAM® libraries for combustion and spray modeling

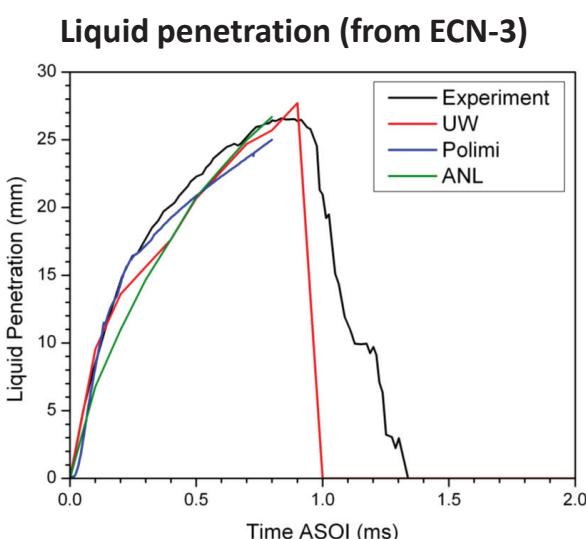
43

# Spray modeling for IC engines

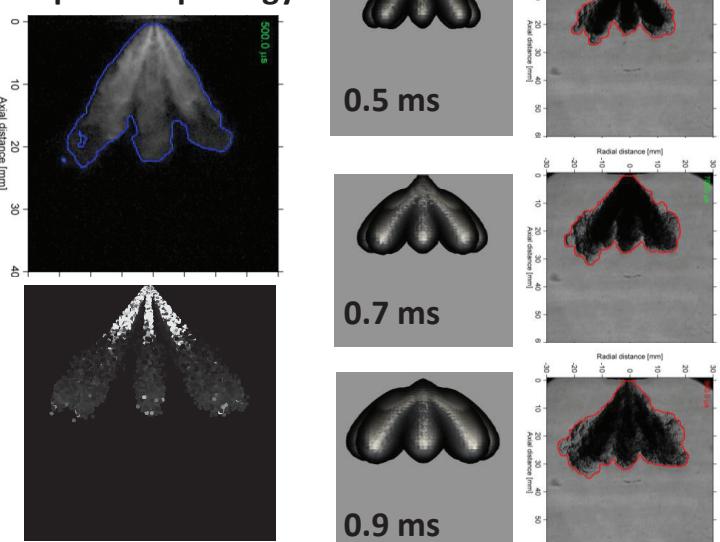


## Validation: ECN Spray G (multi-hole)

- Atomization model: Huh-Gosman
- Injection model: Huh
- Breakup model: KH-RT
- Weakly evaporating conditions:  $T_{amb} = 300 \text{ }^{\circ}\text{C}$ ,  $\rho = 3.5 \text{ kg/m}^3$



Liquid morphology

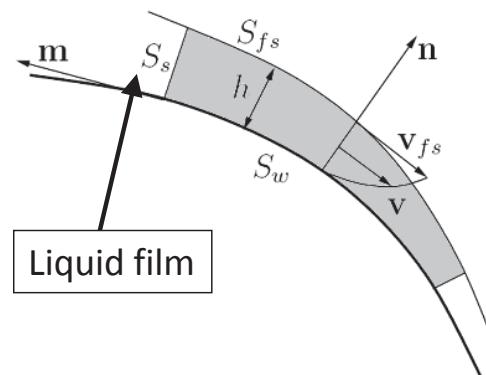


T. Lucchini: Development of OpenFOAM® libraries for combustion and spray modeling

44

## Liquid film model

- Approach originally proposed by Bai and Gosman (SAE-960626). Thin film approximation.
- Developed in collaboration with Dr. Z. Tukovic and Dr. H. Jasak.
- Mass, momentum and energy equation for the liquid film solved on the mesh boundary, using the finite area method. Mesh motion supported, fully parallel.
- Droplet formation from the liquid film taken also into account.
  - Curved wall surface:  $S_w$
  - Free liquid surface:  $S_{fs}$
  - Liquid film thickness:  $h$
  - Velocity profile



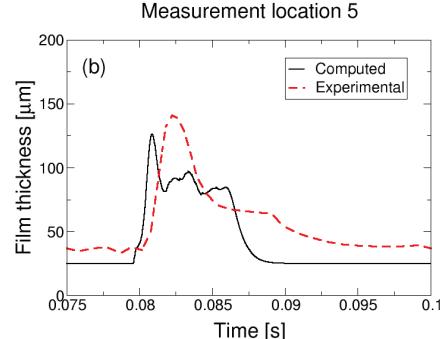
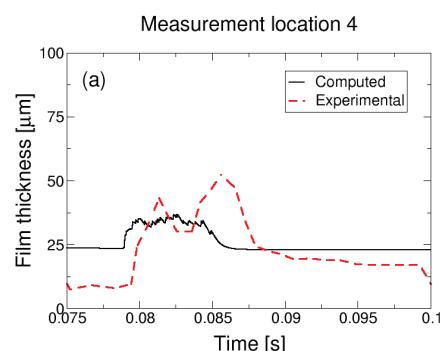
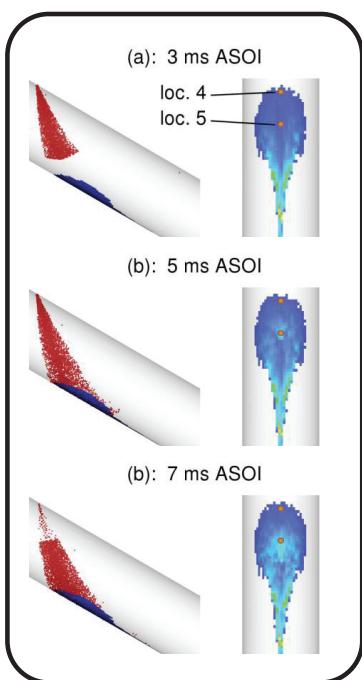
T. Lucchini: Development of OpenFOAM® libraries for combustion and spray modeling

45

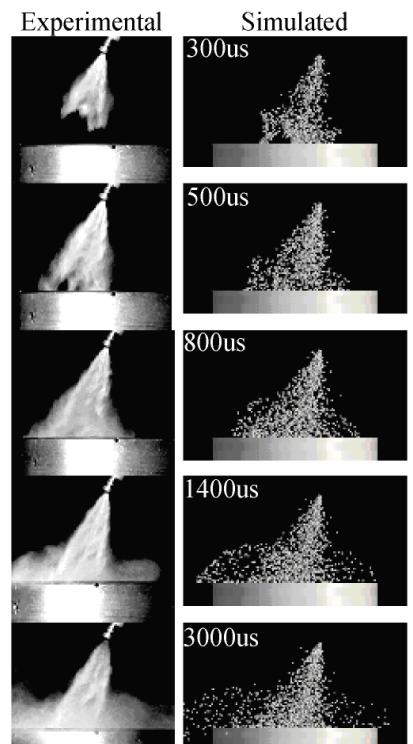
# Spray modeling for IC engines

## Liquid film model: validation

### Le-Coz experiment



### Impinging spray



Experimental data courtesy of Ing. Montanaro and Dr. Allocca (CNR-Istituto Motori)



Stay tuned....

Tomorrow's presentation including simulation of real GDI engines:

- 1) Understanding how spray targeting affects soot emissions
- 2) Correlation between soot formation and fuel-vapor distribution under stratified charge combustion operation.



# Diesel combustion

# Diesel combustion modeling

## Objectives

- Provide a complete library of combustion models for Diesel engines:
  - Simplified models with reduced chemistry for engine design
  - Detailed models for advanced combustion modes and diagnostic purposes

| Reduced chemistry               | Detailed chemistry                                                                                     |
|---------------------------------|--------------------------------------------------------------------------------------------------------|
| Characteristic-Time scale (CTC) | Well stirred reactor model<br>Multiple Representative Interactive Flamelet<br>Eulerian Transported PDF |

# Diesel combustion modeling

## Characteristic time-scale (CTC) combustion model

- 11 chemical species (fuel, O<sub>2</sub>, N<sub>2</sub>, CO, CO<sub>2</sub>, H<sub>2</sub>O, O, OH, NO, H, H<sub>2</sub>)
- Auto-ignition computed by the Shell auto-ignition model; turbulent combustion simulated accounting for both laminar and turbulent time scales:

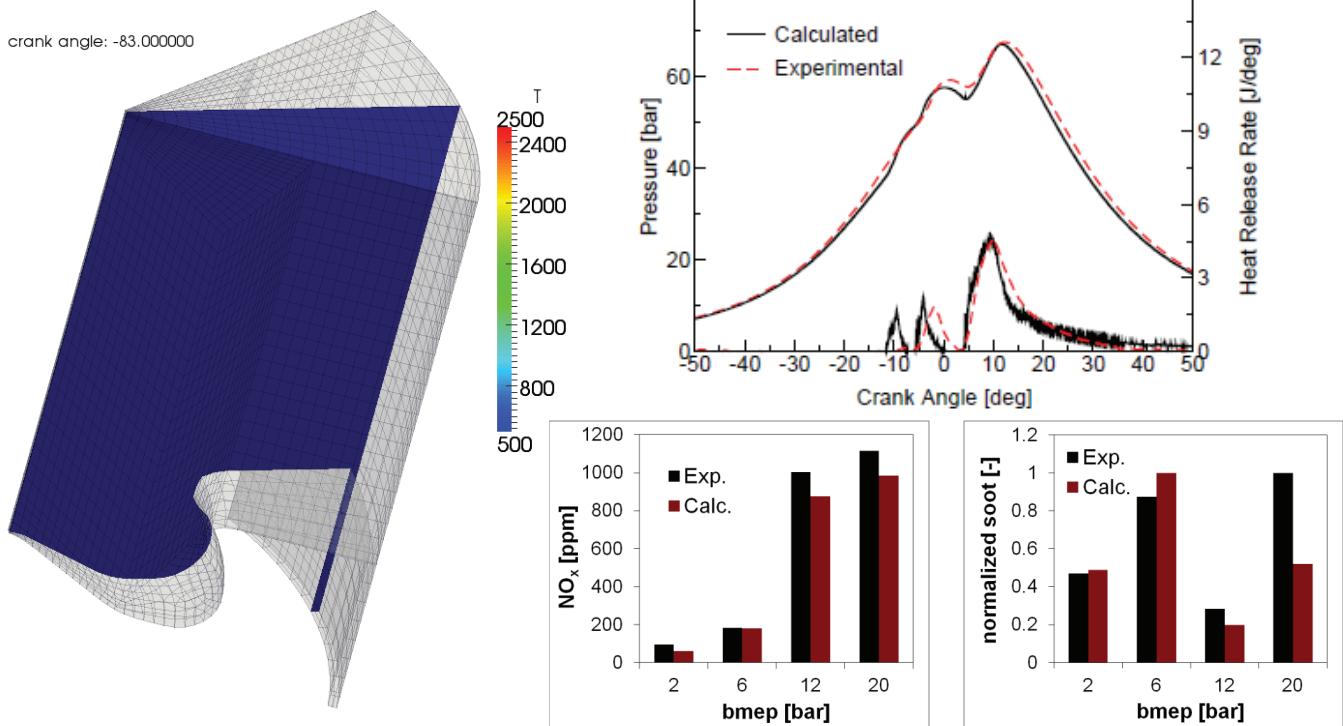
$$\dot{Y}_i = \alpha \cdot \dot{Y}_{i,Shell} + (1 - \alpha) \cdot \dot{Y}_{i,CTC} \quad \dot{Y}_{i,CTC} = -\frac{Y_i - Y_i^*}{\tau_{CTC}}$$

- Temperature threshold to switch between auto-ignition ( $\alpha=1$ ) and turbulent combustion ( $\alpha=0$ ).  $Y^*$ : composition at equilibrium (Rakopoulos approach)
- Pollutant prediction: Fusco model for soot, Zeldovich for NO<sub>x</sub>
- Why CTC? It is fast, quite generic (it can be applicable also to SI combustion with few modifications) and accurate under conventional Diesel combustion.
- Before running complex combustion models, run CTC to verify the validity of the setup (initial conditions, heat transfer model, ...)

# Diesel combustion modeling



## Characteristic time-scale (CTC) combustion model: validation



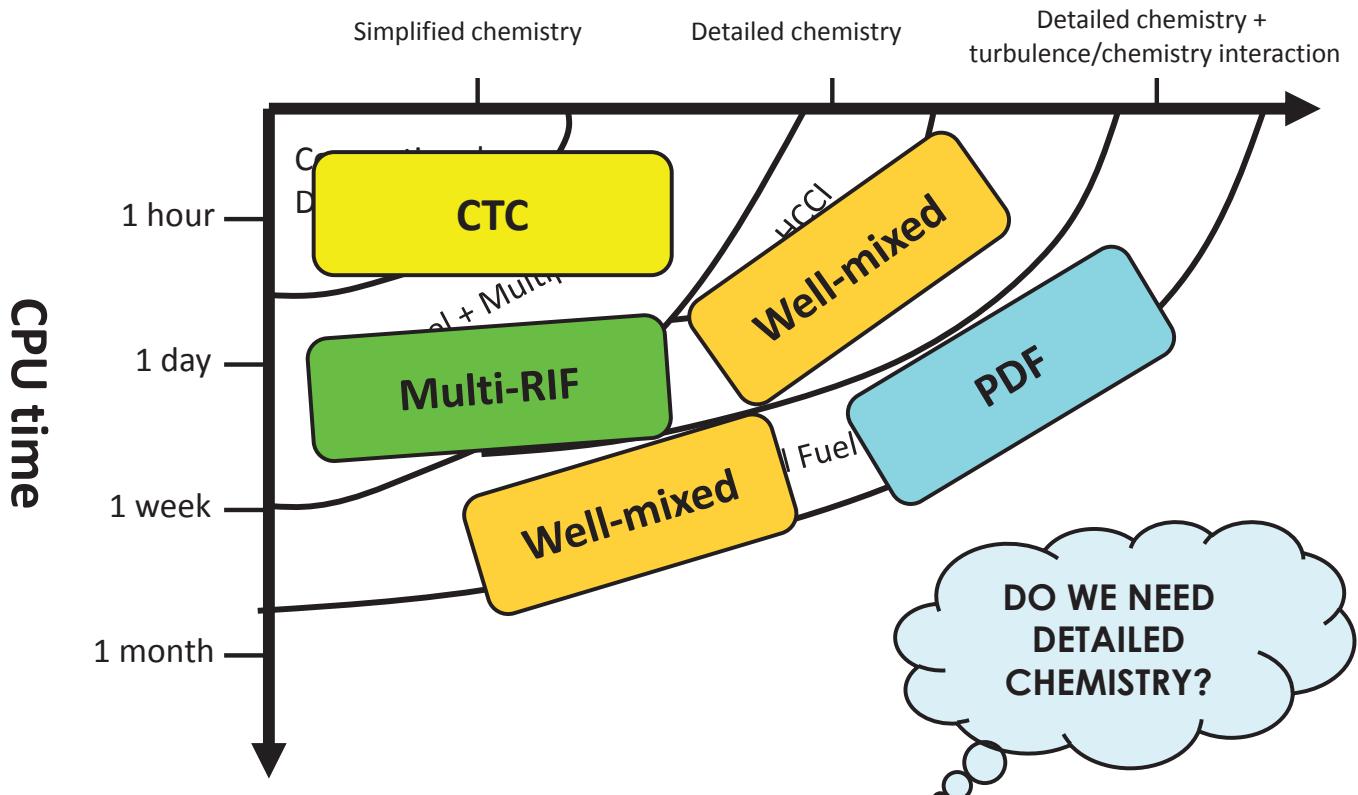
T. Lucchini: Development of OpenFOAM® libraries for combustion and spray modeling

51

# Diesel combustion modeling



Level of detail (chemistry, turbulence-chemistry interaction, ...)



T. Lucchini: Development of OpenFOAM® libraries for combustion and spray modeling

52

# Diesel combustion modeling

- **Detailed chemistry is necessary:**
  - Diesel engines operate on a large range of speed, loads (T and p @ SOI) and EGR rates.
  - Need for a precise estimation of NOX and proper trends in terms of soot emissions.
- **But...**
  - CPU time drastically grows
  - Which mechanism?
  - How to deal with turbulence-chemistry interaction?
  - Which data for validation?

# Diesel combustion modeling

## The Engine Combustion Network (ECN)

The purpose of ECN is to provide an open forum for international collaboration among experimental and computational researchers in engine combustion.

- ### ECN Workshops (every 1 and ½ years)

  - 2011 : ECN 1, Ventura, USA
  - 2012 : ECN 2, Heidelberg, Germany
  - 2014 : ECN 3, Ann Arbor, USA
  - 2015 : ECN 4, Kyoto

Laboratories.



Engine Combustion Network

## POLIMI contributions to Diesel combustion modeling

- Implementation and validation inside the same platform (Lib-ICE) of different combustion models based on detailed chemistry:
  - Well-stirred reactor model
  - Multiple Representative Interactive Flamelet model
  - Stochastic Eulerian PDF

All models validated with the same spray, mesh and turbulence setup.
- **Chemistry acceleration:** multi-zone **CCM** (collaboration with Dr. Jangi) ; **TDAC** algorithm (collaboration with Dr. Contino and prof. Jeanmart)

T. Lucchini: Development of OpenFOAM® libraries for combustion and spray modeling

55

# Diesel combustion modeling



## Well-mixed model

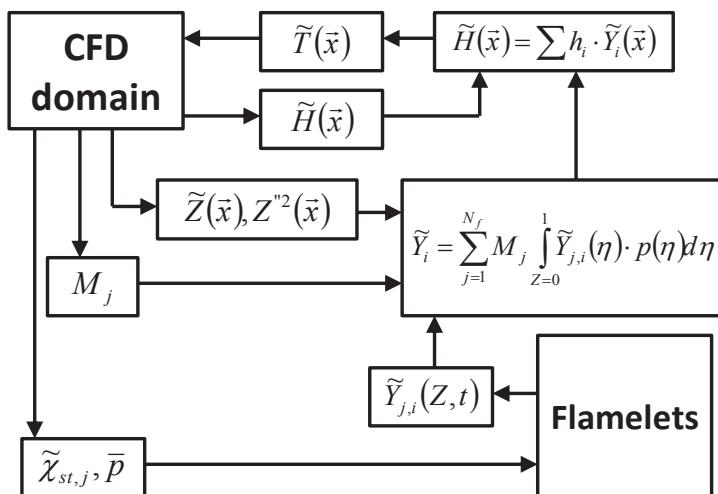
- Direct integration of complex chemistry in each computational cell. Each cell is an homogeneous reactor and species reaction rates are computed by means of an ODE stiff solver
- $$Y_i^*(t + \Delta t) = Y_i(t) + \int_t^{t + \Delta t} \dot{\omega}_i \frac{W_i}{\rho} dt' \quad \rightarrow \quad \dot{Y}_i = \frac{Y_i^*(t + \Delta t) - Y_i(t)}{\Delta t}$$
- Advantages: flexible with respect to fuel type, chemical composition and physical properties of the fuel. Disadvantages: models are strongly affected by assumption of homogeneous composition.
- ~~$\bar{\omega}_F = -A_1 \bar{\rho}^2 \tilde{T}^{\beta_1} \tilde{Y}_F \tilde{Y}_O \exp\left(-\frac{T_A}{\tilde{T}}\right) \left[ 1 + \frac{\tilde{Y}_F}{\tilde{Y}_F \tilde{Y}_O} (P_1 + Q_1) \left( \frac{\tilde{Y}_F'' \tilde{T}''}{\tilde{Y}_F \tilde{T}'} + \frac{\tilde{Y}_O'' \tilde{T}''}{\tilde{Y}_O \tilde{T}'} \right) + (P_2 + Q_2 + P_1 Q_1) \left( \frac{\tilde{T}''}{\tilde{T}^2} + \frac{\tilde{Y}_F'' \tilde{T}''}{\tilde{Y}_F \tilde{T}^2} + \frac{\tilde{Y}_O'' \tilde{T}''^2}{\tilde{Y}_O \tilde{T}^2} \right) + \dots \right]$~~
- drastically affected by assumption of homogeneous composition.

# Diesel combustion modeling



## Multiple Representative Interactive Flamelets (mRIF)

- **Multiple unsteady flamelets:** a set of unsteady diffusion flames represents Diesel combustion.
- First model towards the definition of a generic model for partially-premixed combustion



### Flamelet Equations

$$\frac{\partial Y_i}{\partial t} - \frac{\chi(Z)}{2} \frac{\partial^2 Y_i}{\partial Z^2} = \dot{Y}$$

$$\frac{\partial h_s}{\partial t} - \frac{\chi(Z)}{2} \frac{\partial^2 h_s}{\partial Z^2} = \dot{q}_{chem}$$

T. Lucchini: Development of OpenFOAM® libraries for combustion and spray modeling

57

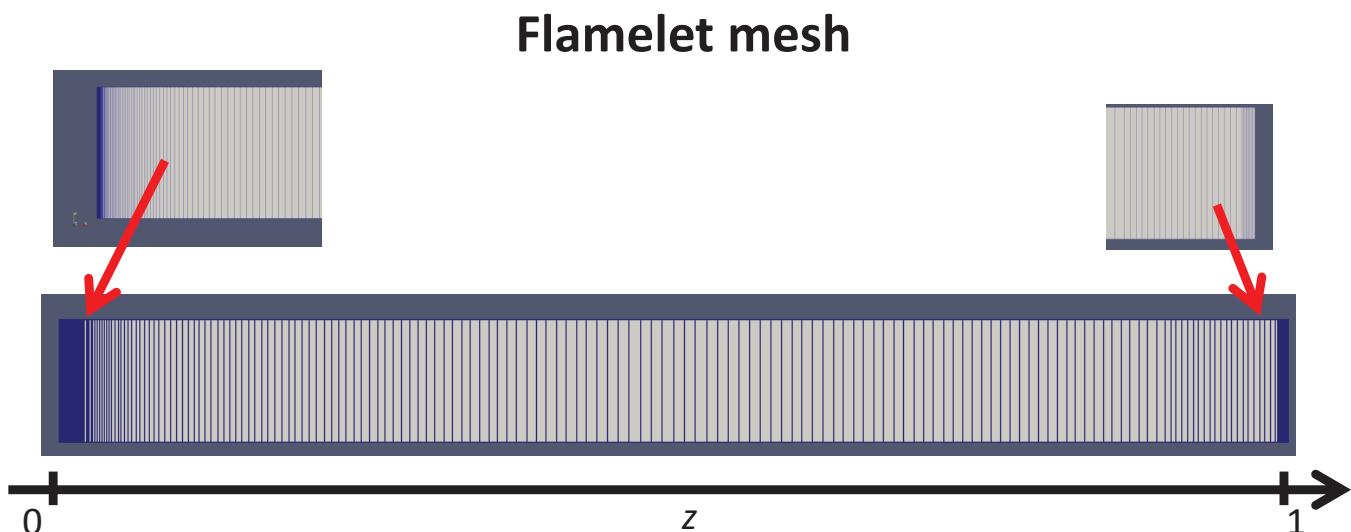
# Diesel combustion modeling



## Multiple Representative Interactive Flamelets (mRIF)

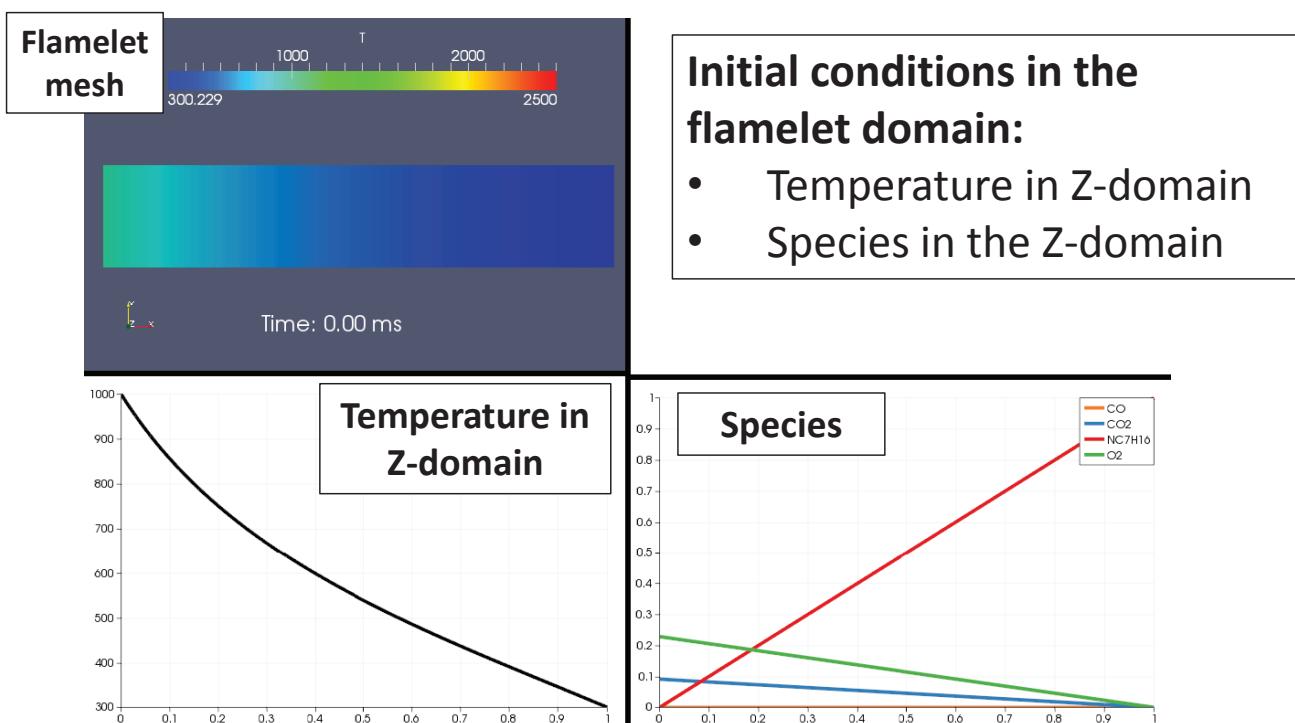
- **Implementation in Lib-ICE**
- regionModel approach for flamelet combustion models:
  - Flamelet mesh, representing the Z space
  - Flamelet equations (species and enthalpy) solved on the flamelet mesh, relying on native OpenFOAM matrix algebra and ODE solvers.
  - PDF averaging:
    - Standard OpenFOAM functions (integration, averaging) applied on the flamelet mesh, that requires accurate grading at the Z = 0 and Z = 1 boundaries.

## Simulation: SANDIA ECN Spray H flame (n-heptane)



The grid is refined at the boundaries ( $Z=0$ ,  $Z=1$ ), to correctly integrate the PDF function in presence of high mixture fraction variances

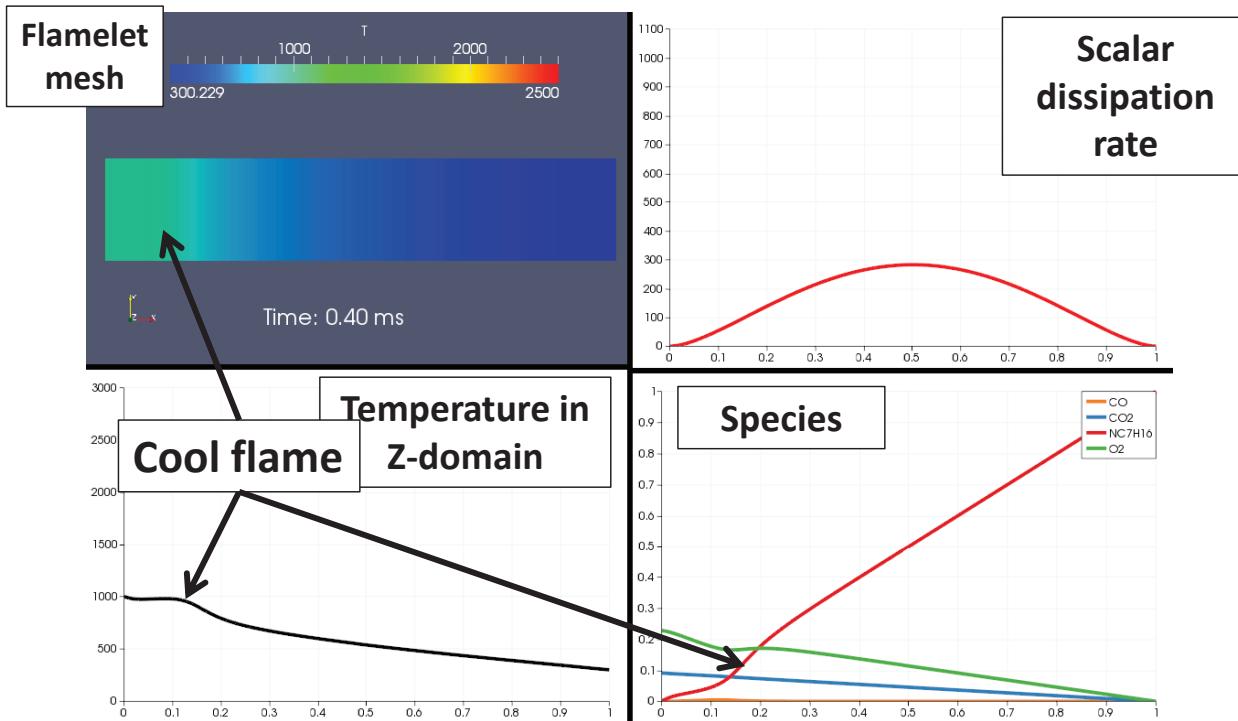
## Simulation: SANDIA ECN Spray H flame (n-heptane)



# Diesel combustion modeling: mRIF



## Simulation: SANDIA ECN Spray H flame (n-heptane)



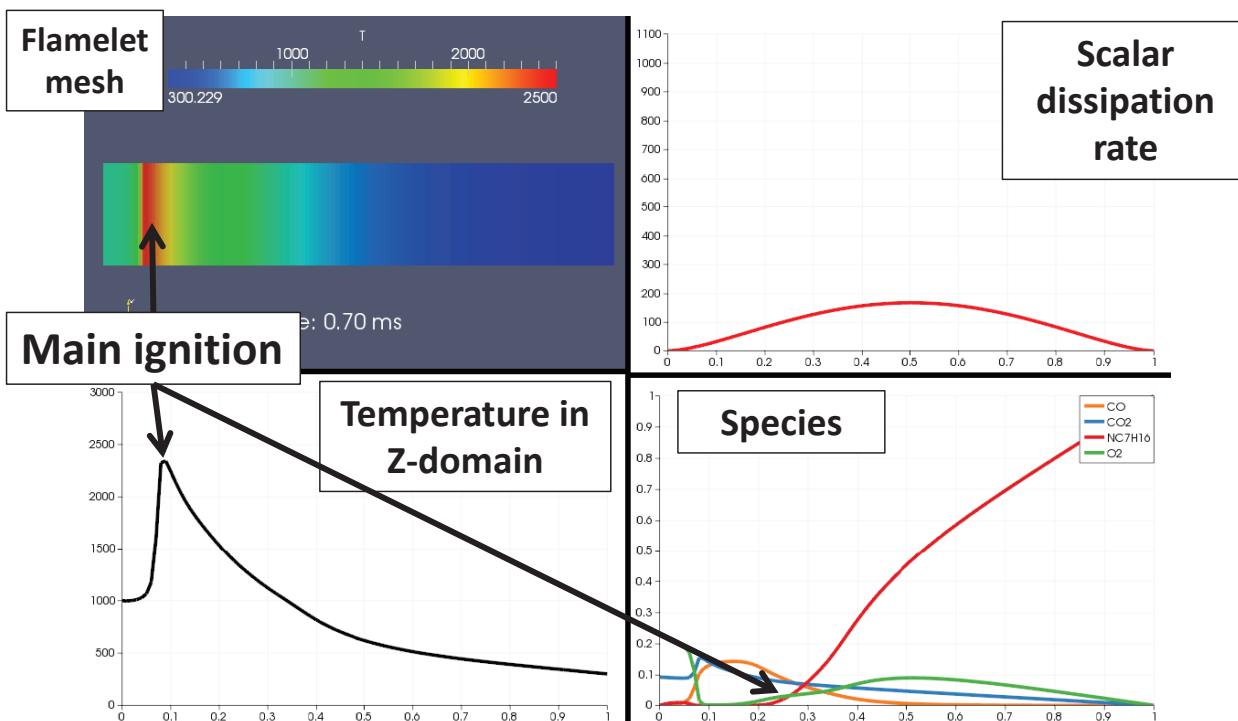
T. Lucchini: Development of OpenFOAM® libraries for combustion and spray modeling

61

# Diesel combustion modeling: mRIF



## Simulation: SANDIA ECN Spray H flame (n-heptane)



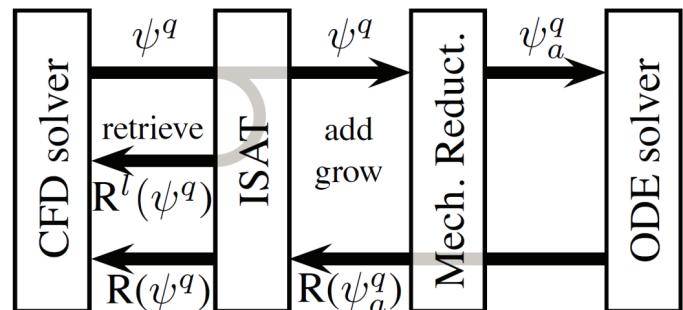
T. Lucchini: Development of OpenFOAM® libraries for combustion and spray modeling

62

# Diesel combustion modeling



- Chemistry acceleration TDAC: Tabulation of Dynamic Adaptive Chemistry
  - Combination of mechanism reduction and tabulation techniques
  - Work developed in collaboration with Dr. F. Contino (Vrije Universiteit Brussel) and Prof. H. Jeanmart (Univ. of Louvain)
- CPU time speed-up compared to direct-integration is of the order of  $n \times m$  ( $n$ : number of species,  $m$ : number of reactions).
- Very general approach, flexible with respect to the mechanism and fuel composition.
- Up to 300 species and 1000 reactions can be used in a reasonable amount of time.



TDAC fully integrated in all combustion models implemented in Lib-ICE

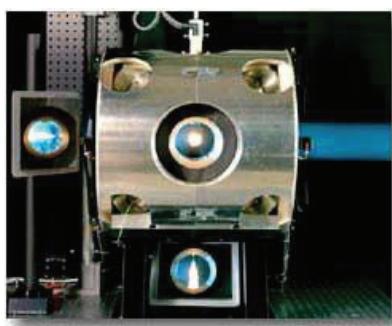
T. Lucchini: Development of OpenFOAM® libraries for combustion and spray modeling

63

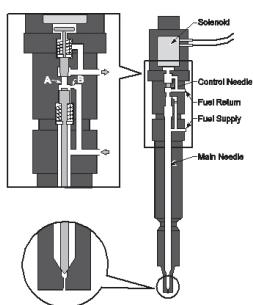
# Diesel combustion modeling



## Validation: SANDIA COMBUSTION VESSEL



Experimental data provided by the Engine Combustion Network (ECN) database.

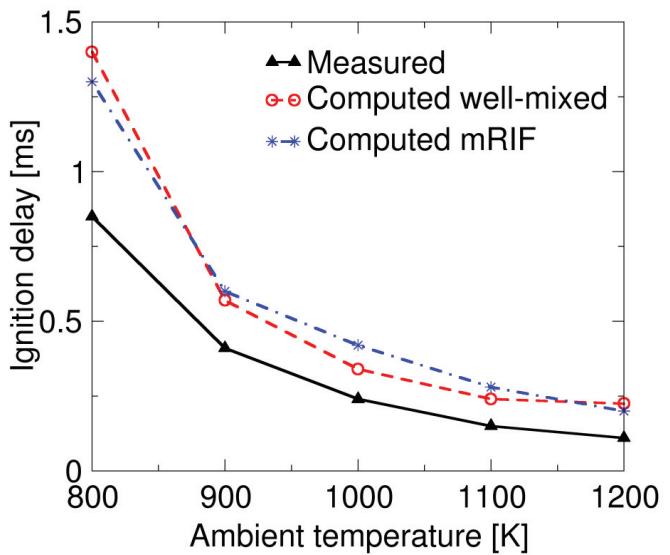
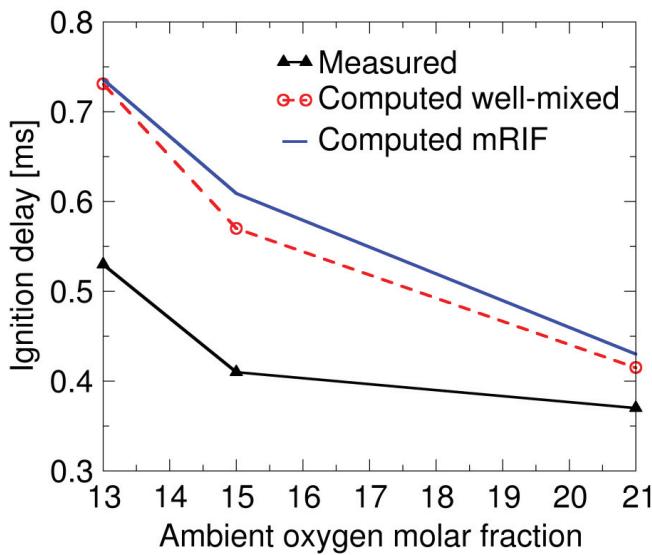


- Fuel: n-dodecane
- Spray model: atomization (Huh-Gosman) + breakup (KH)
- Mechanism: Luo et al.  
103 species, 370 reactions
- Tested operating conditions:
  - 1) Effect of oxygen concentration
  - 2) Effect of ambient density
  - 3) Effect of ambient temperature

# Diesel combustion modeling



- Validation: ignition delay

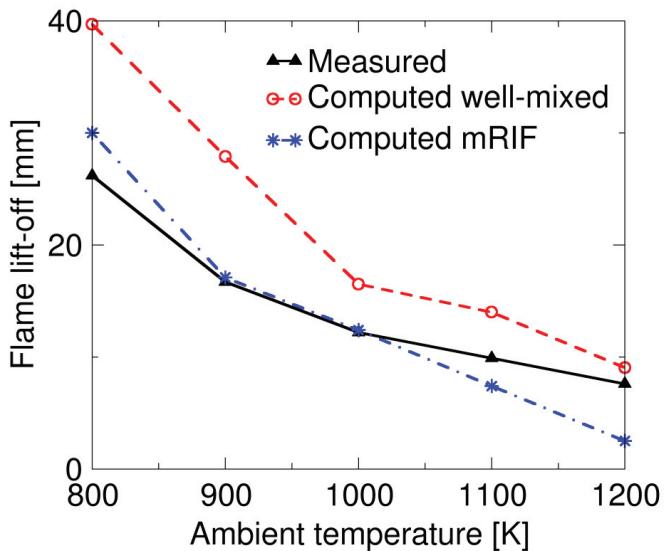
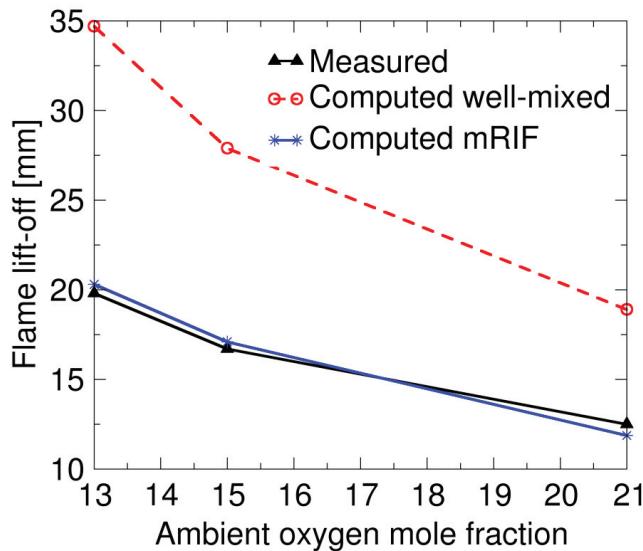


- Very similar values predicted: this is due to the fact that auto-ignition takes place at low scalar dissipation rate values.

# Diesel combustion modeling



- Validation: flame lift-off

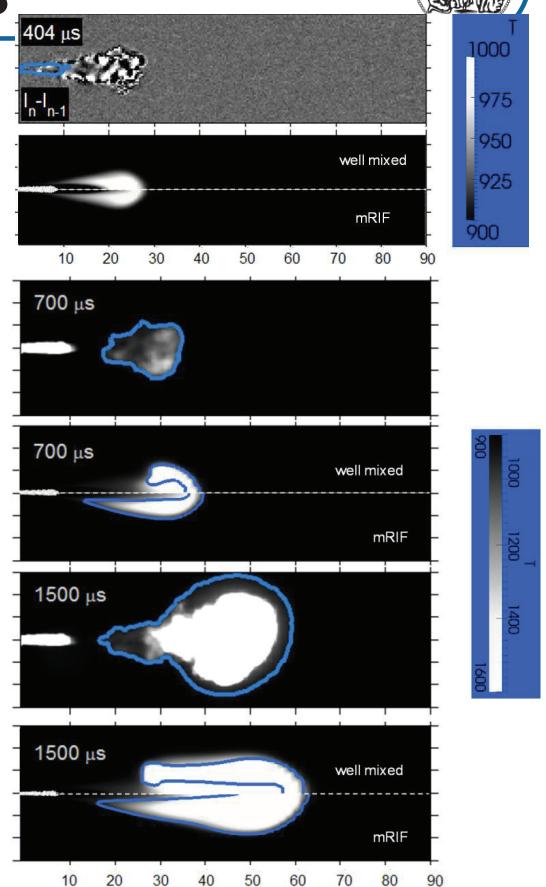
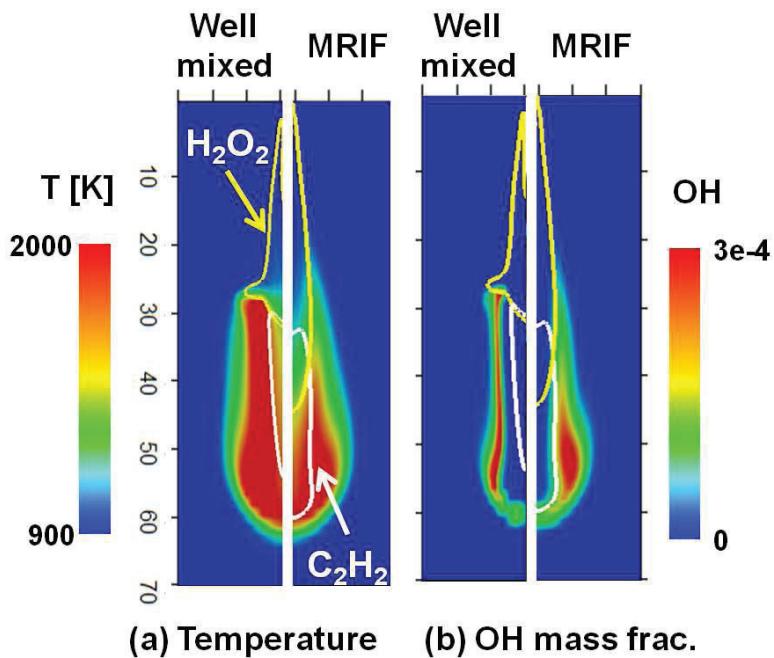


- Different stabilization mechanisms between mRIF and well-mixed model. Well mixed: triple flame; mRIF: auto-ignition of a diffusion flame.

# Diesel combustion modeling



- Validation: flame structure



T. Lucchini: Development of OpenFOAM® libraries for combustion and spray modeling

67

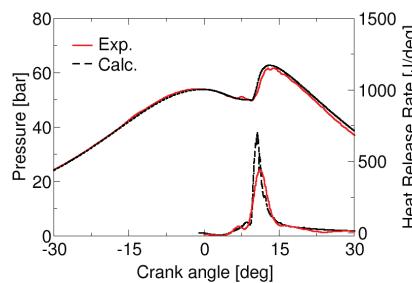
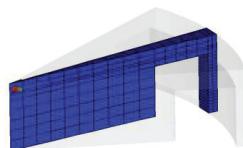
# Diesel combustion modeling



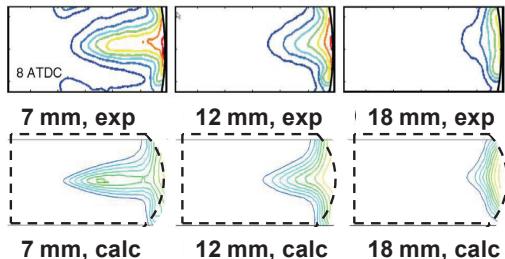
- Well-mixed model: validation

## PCCI combustion

Heavy-duty optical engine (SANDIA) operating with PRF30 fuel.

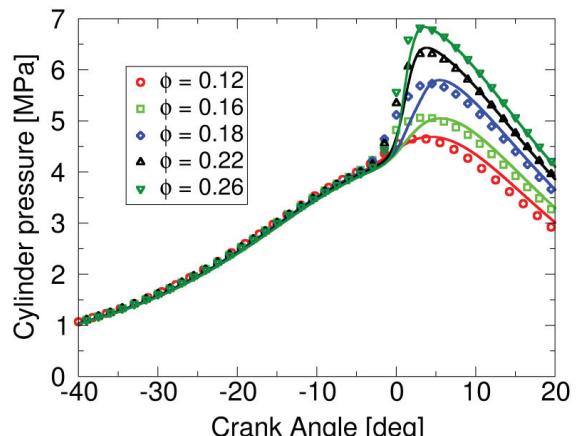
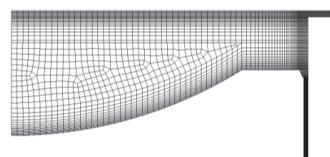


## Air-fuel mixing model validation



## HCCI combustion

Heavy-duty engine operating at low-load with iso-octane fuel.





Stay tuned....

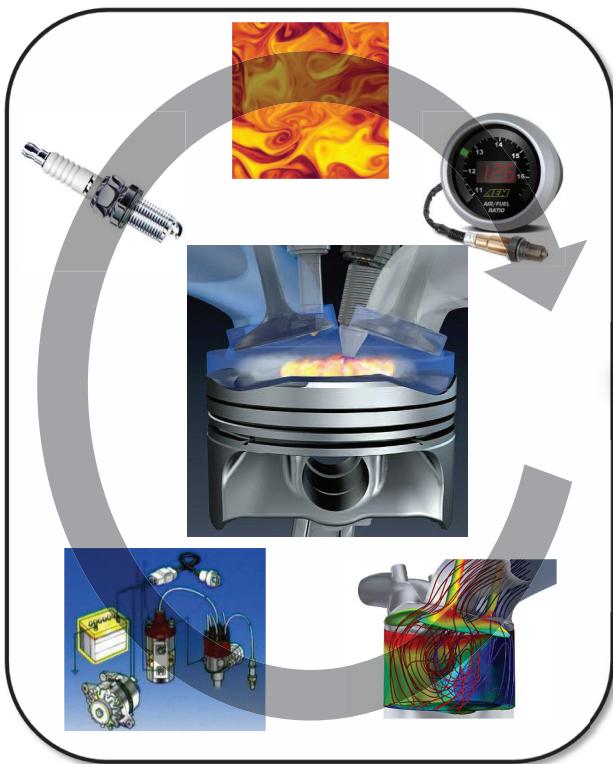
Tomorrow's presentation will include simulation of combustion in real Diesel engine geometries:

- Automotive engine at critical operating points in the EUDC cycle
- Heavy duty diesel engine



# Spark-ignition combustion

# Spark-ignition combustion



Development of a CFD model in an open-source code

Understanding the complex interplay between ignition-system, local flow, turbulence, air/fuel ratio...

Design and development of more efficient combustion systems

T. Lucchini: Development of OpenFOAM® libraries for combustion and spray modeling

# Spark-ignition combustion



## State of the art

- Different models available for a proper description of flame propagation (Weller, G-Equation, ECFM...);
- Simulation results strongly affected by the way flame kernel growth is modeled:
  - Eulerian models - simplified
  - Lagrangian models
    - AKTIM, Bianchi, DPIK, Spark-CIMM
- **Proposed approach:** Detailed Lagrangian ignition model + turbulent combustion model (Eulerian).

T. Lucchini: Development of OpenFOAM® libraries for combustion and spray modeling

# Spark-ignition combustion

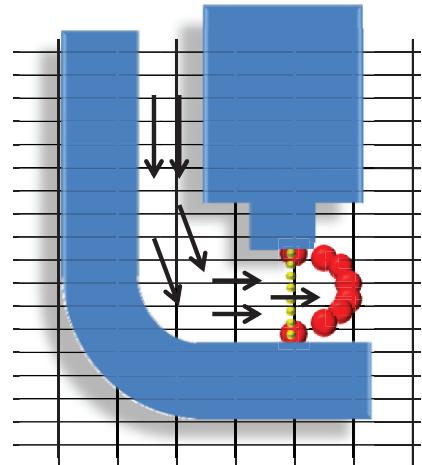


## Lagrangian ignition model

- 1) The spark channel is represented by a set of Lagrangian particles, convected by the mean flow.

$$\frac{d\mathbf{x}_p}{dt} = \mathbf{U}_g(\mathbf{x}_p)$$

- 2) Proper correlations to determine the initial size and temperature of the channel after breakdown



T. Lucchini: Development of OpenFOAM® libraries for combustion and spray modeling

# Spark-ignition combustion



## Lagrangian ignition model

- 3) For each flame kernel, conservation equations of mass, energy and volume are solved:

- **Volume** (flame kernel radius)

$$\frac{dr_k}{dt} = \frac{\rho_u}{\rho_k} \cdot S_t + S_{plasma} + \frac{V_k}{A_k} \left( \frac{1}{T_k} \cdot \frac{dT_k}{dt} - \frac{1}{p} \cdot \frac{dp}{dt} \right)$$

➤  $S_t$  : turbulent flame speed

➤  $S_{plasma}$  : heat conduction  
(relevant for few ns after ignition)

- **Energy** ( $T < 3 T_{ad}$ )

$$\frac{dT_p}{dt} = -\frac{\dot{m}_p}{m_p} (T_p - T_b) + \frac{\dot{Q}_{spk} \cdot \eta_{eff}}{m_p \cdot c_p} + \frac{1}{\rho_b c_p} \cdot \frac{dp}{dt}$$

➤  $\dot{Q}_{spk}$  : energy transferred from the electrical circuit

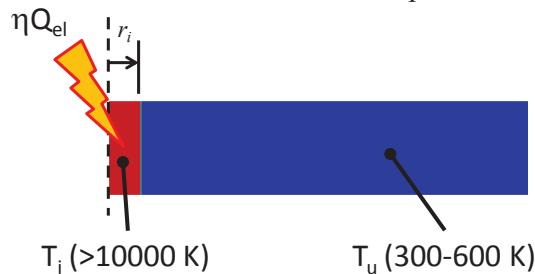
➤  $\eta_{eff}$  : energy transfer efficiency

T. Lucchini: Development of OpenFOAM® libraries for combustion and spray modeling

# Spark-ignition combustion

## Ignition: submodels

- Turbulent flame speed:  $S_t = I_0 \cdot S_u \cdot \Xi$
- Expansion velocity  $S_{plasma}$

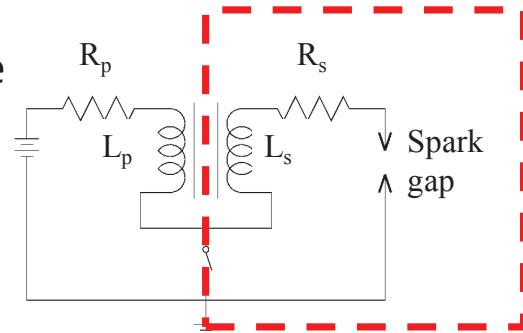


### Heat conduction equation

$$\frac{\partial T_{pl}}{\partial t} = \alpha \cdot \nabla^2 T_{pl} + \frac{\eta \cdot \dot{Q}_{el}}{\rho c_p V_{pl}} = \alpha \cdot \nabla^2 T_{pl} + \frac{\eta \cdot V \cdot i}{\rho c_p V_{pl}}$$

- Secondary circuit to compute the amount of transferred energy to the spark channel:

$$\dot{Q}_{spk} = V(t) \cdot i(t)$$



T. Lucchini: Development of OpenFOAM® libraries for combustion and spray modeling

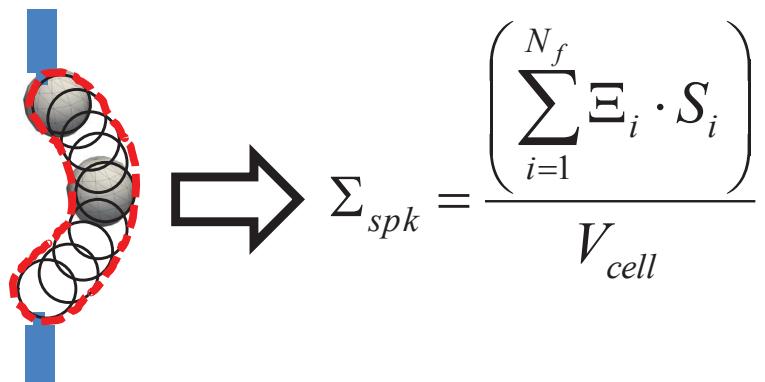
# Spark-ignition combustion

## Restrike

- Multiple ignitions due to channel elongation included. When secondary voltage exceeds a threshold value, a new channel is created.

## Coupling with CFD code

- Flame surface density distribution computed from particle positions and size. Spark channels de-activated when they reach a critical size.



T. Lucchini: Development of OpenFOAM® libraries for combustion and spray modeling

# Spark-ignition combustion

## Eulerian combustion model

- Coherent flamelet model (CFM) proposed by Choi and Huh:

$$\frac{\partial \rho \Sigma}{\partial t} + \frac{\partial \rho u_i \Sigma}{\partial x_i} = \frac{\partial}{\partial x_i} \left[ \left( \frac{\mu}{Sc} + \frac{\mu_t}{Sc_t} \right) \frac{\partial \Sigma}{\partial x_i} \right] + P_1 \cdot \Sigma - D \cdot \Sigma + P_k$$

➤ Production term:  $P_1 = \alpha_\Sigma \cdot \frac{u'}{l_{tc}}$

| $\alpha$ | $\beta$ |
|----------|---------|
| 4        | 0.1     |

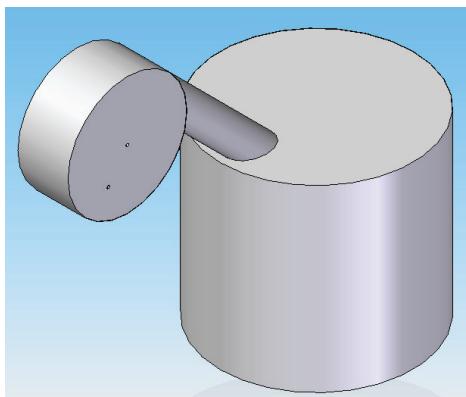
➤ Destruction term:  $D = \beta \cdot S_u \frac{\Sigma}{c \cdot (1 - c)}$

➤  $P_k$  accounts for flame kernel growth

➤ Reaction rate:  $\dot{\omega}_u = \rho_u \cdot S_u \cdot I_0 \cdot \Sigma$

T. Lucchini: Development of OpenFOAM® libraries for combustion and spray modeling

## Spark-ignition combustion



- Pre-chamber with optical access.
- Tested operating conditions:
  - 1) Effect of engine speed
  - 2) Effect of air/fuel ratio
  - 3) Effect of spark-plug position

### Main engine data

|                        |       |
|------------------------|-------|
| Side-chamber diameter  | 45 mm |
| Compression ratio      | 7.3   |
| Side chamber clearance | 19 mm |
| Spark-gap              | 1 mm  |

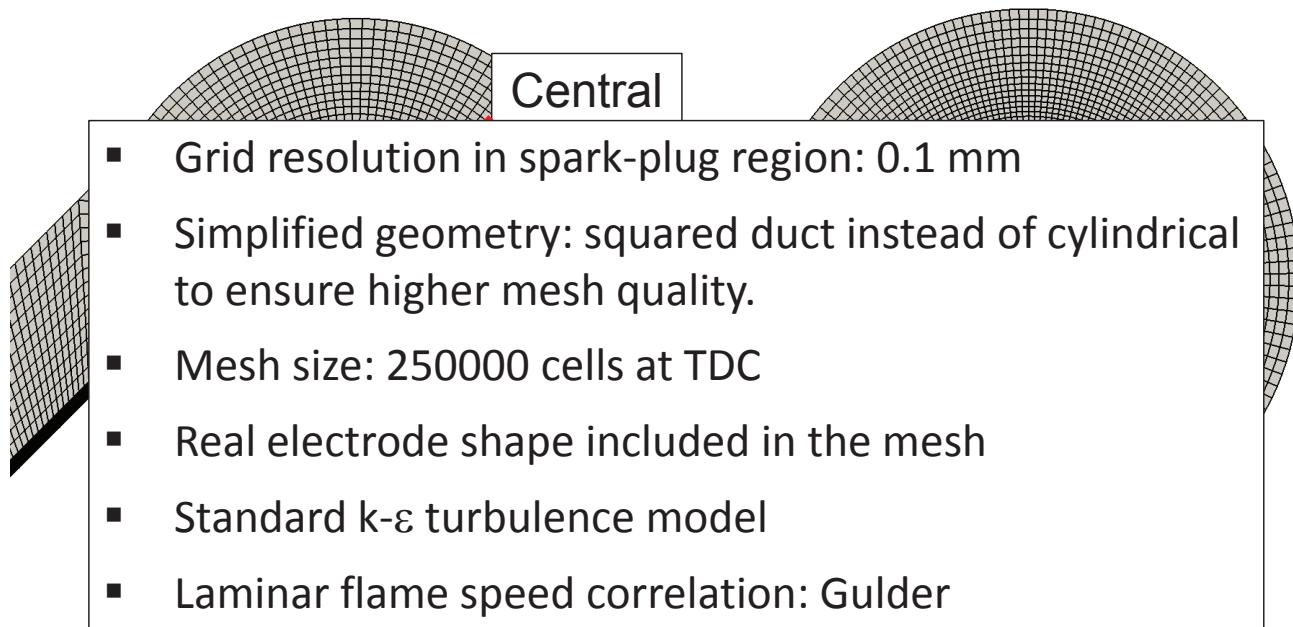
|                   |                    |
|-------------------|--------------------|
| Equivalence ratio | 1, 1.3, 1.5        |
| Ignition timing   | 350 CAD            |
| Fuel              | Propane            |
| Engine speed      | 300, 100, 1250 rpm |

T. Lucchini: Development of OpenFOAM® libraries for combustion and spray modeling

# Spark-ignition combustion



- Simulation setup: mesh details

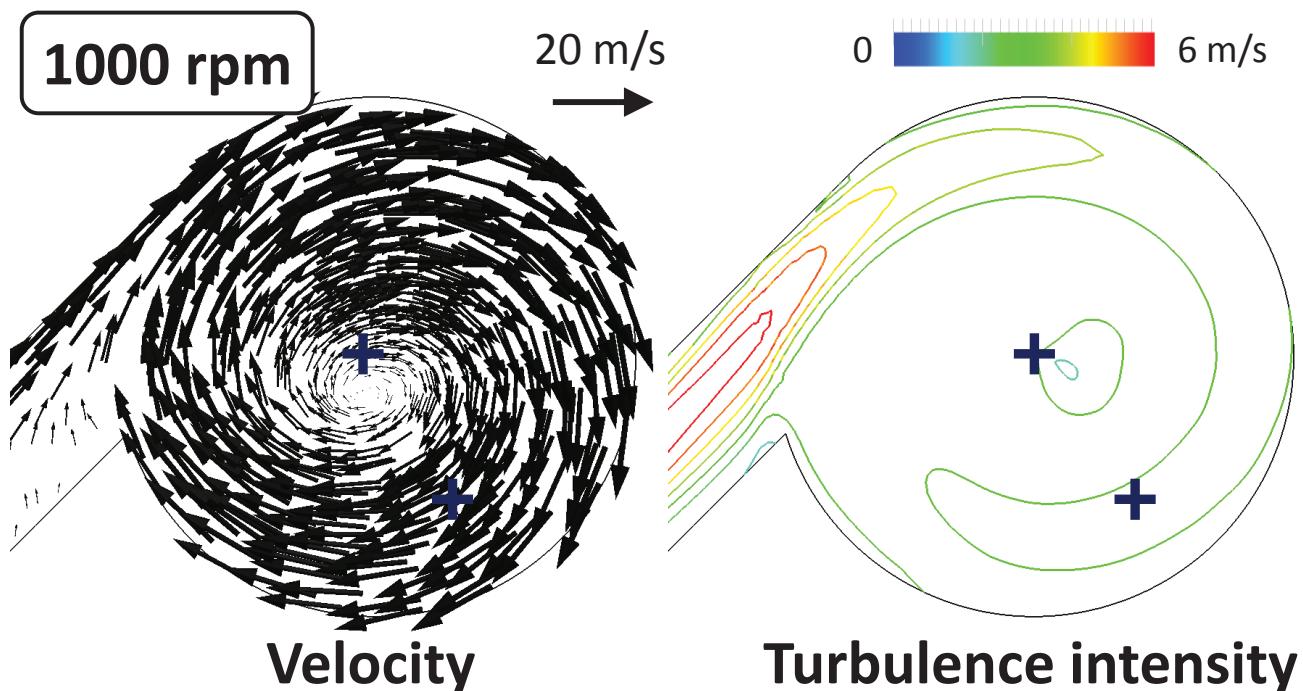


T. Lucchini: Development of OpenFOAM® libraries for combustion and spray modeling

# Spark-ignition combustion



- Non-reacting simulations: flow field at spark-time

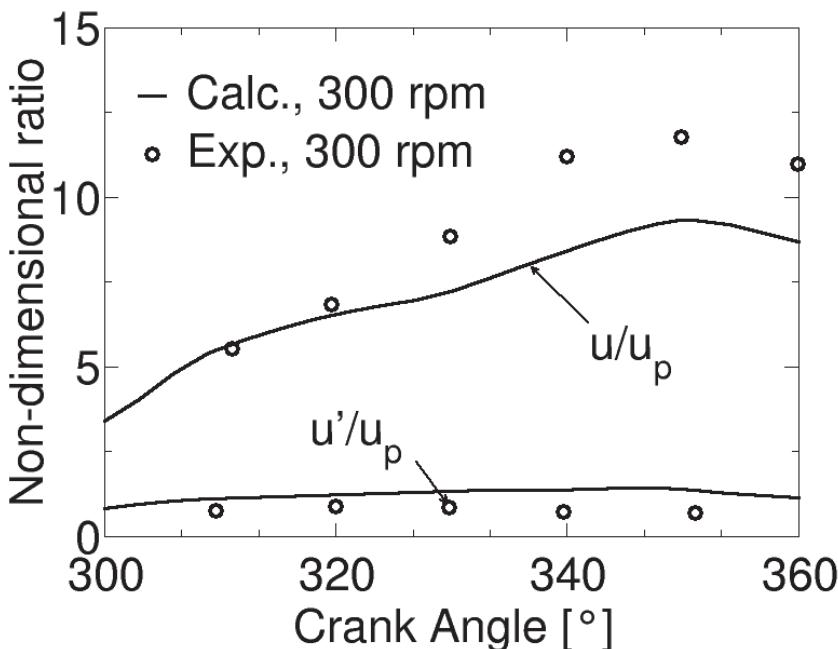


T. Lucchini: Development of OpenFOAM® libraries for combustion and spray modeling

# Spark-ignition combustion



- Non-reacting simulations: validation



## Acceptable results:

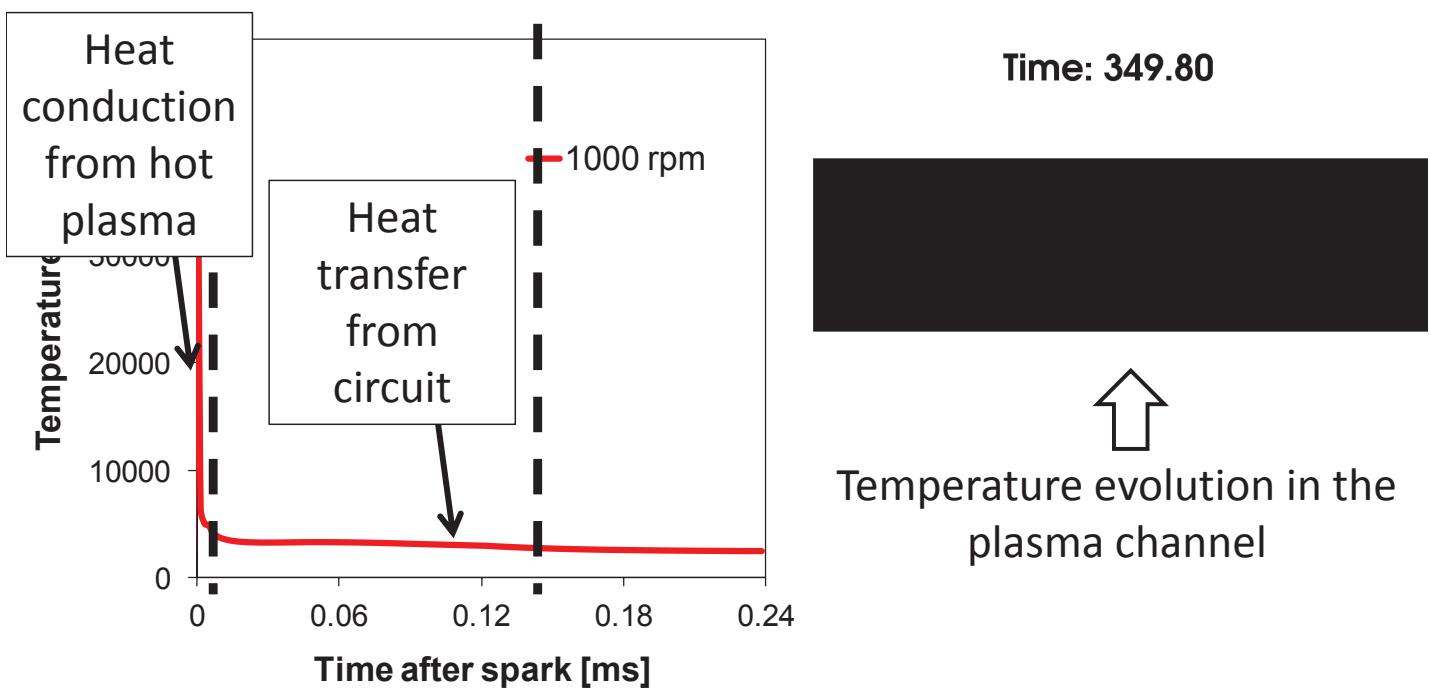
- Geometry was simplified, many details not included:
  - Connecting throat shape
  - Sharp corners due to large optical windows

T. Lucchini: Development of OpenFOAM® libraries for combustion and spray modeling

# Spark-ignition combustion



- Single flame kernel evolution: 1000 rpm, central

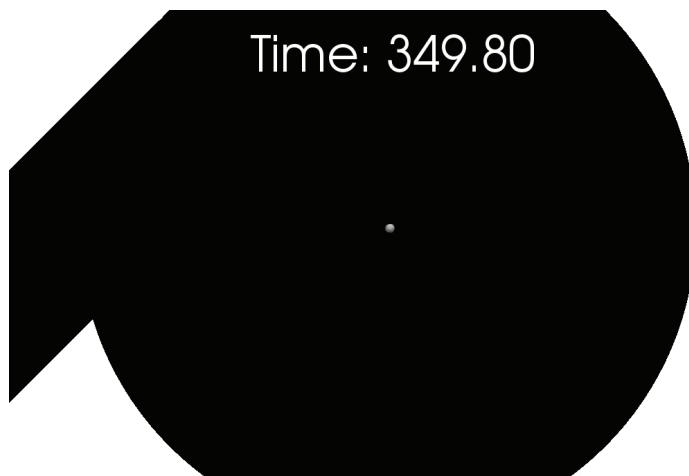


T. Lucchini: Development of OpenFOAM® libraries for combustion and spray modeling

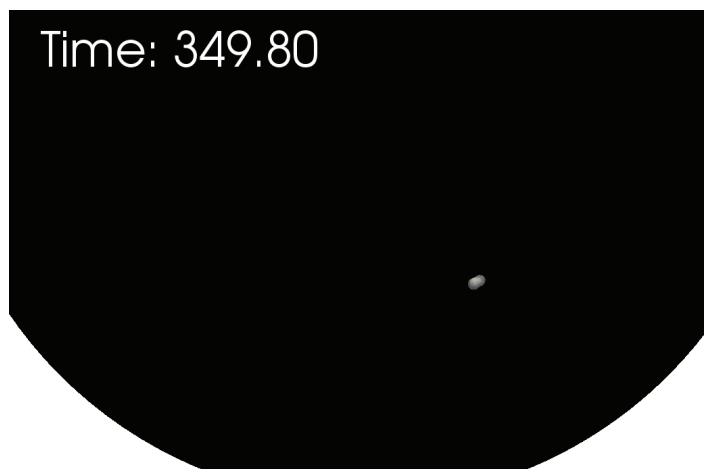
# Spark-ignition combustion

- Local flow effects on flame surface density distribution

**1000 rpm, central**



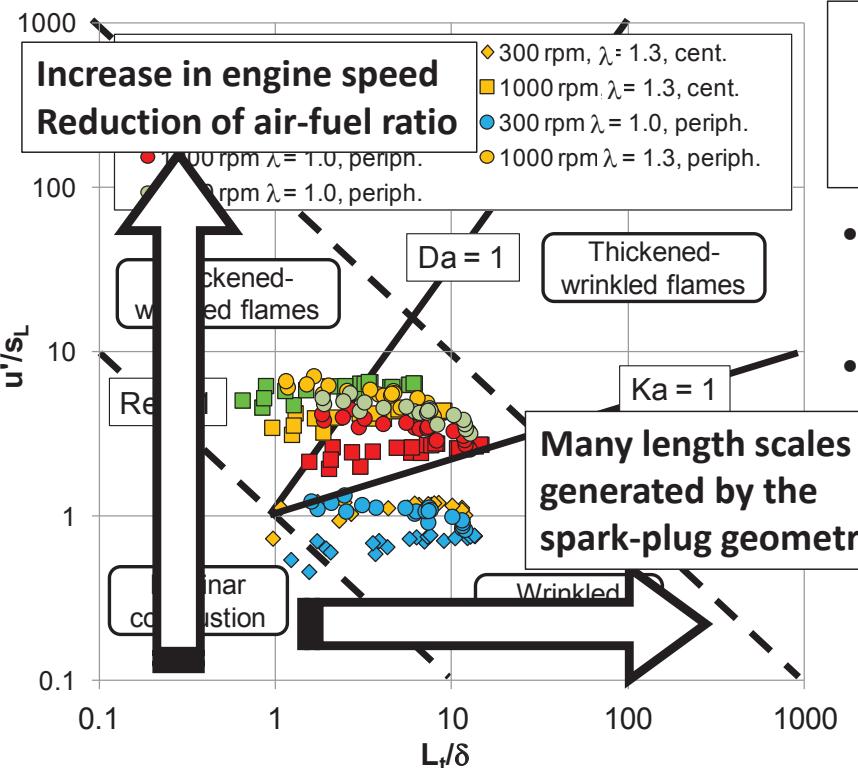
**1000 rpm, peripheral**



- Restrike process affects flame propagation

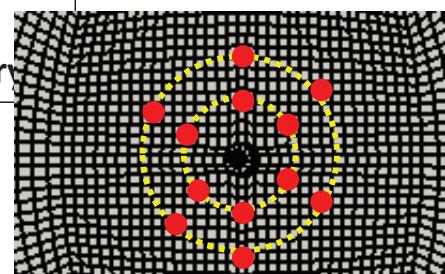
T. Lucchini: Development of OpenFOAM® libraries for combustion and spray modeling

# Spark-ignition combustion



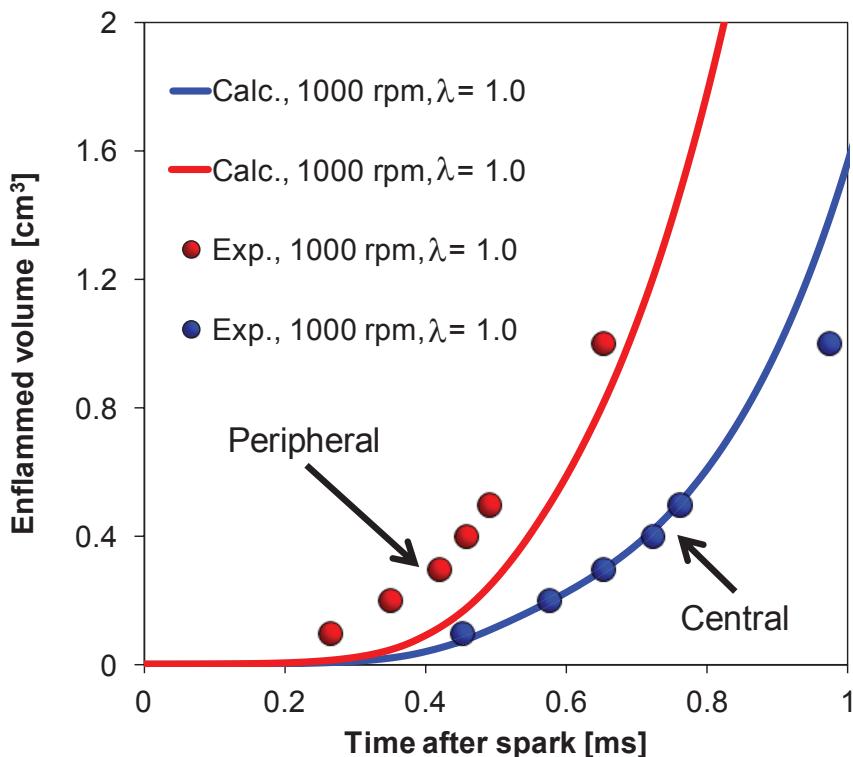
**Expected combustion regimes for the simulated conditions**

- Sampled data of  $u'/s_L$  and  $L_t/\delta$
- Different expected combustion regimes



T. Lucchini: Development of OpenFOAM® libraries for combustion and spray modeling

# Spark-ignition combustion

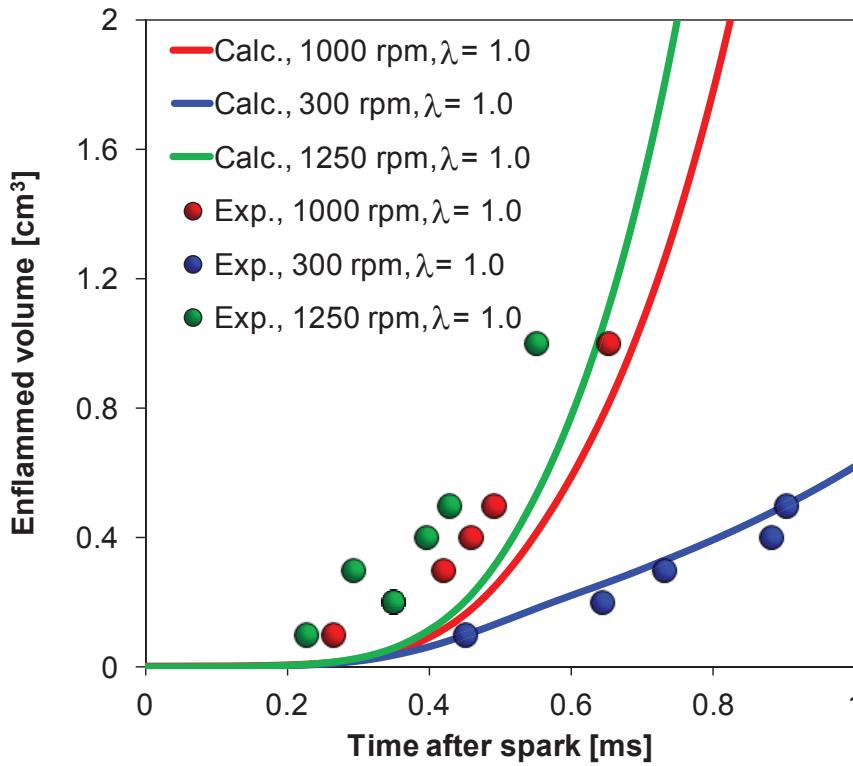


## Central vs peripheral ignition

- Effects of local flow for the same turbulence levels
  - Modifications in flame surface density evolution correctly taken into account:
    - ✓ Flow, flame-holder effect

T. Lucchini: Development of OpenFOAM® libraries for combustion and spray modeling

# Spark-ignition combustion

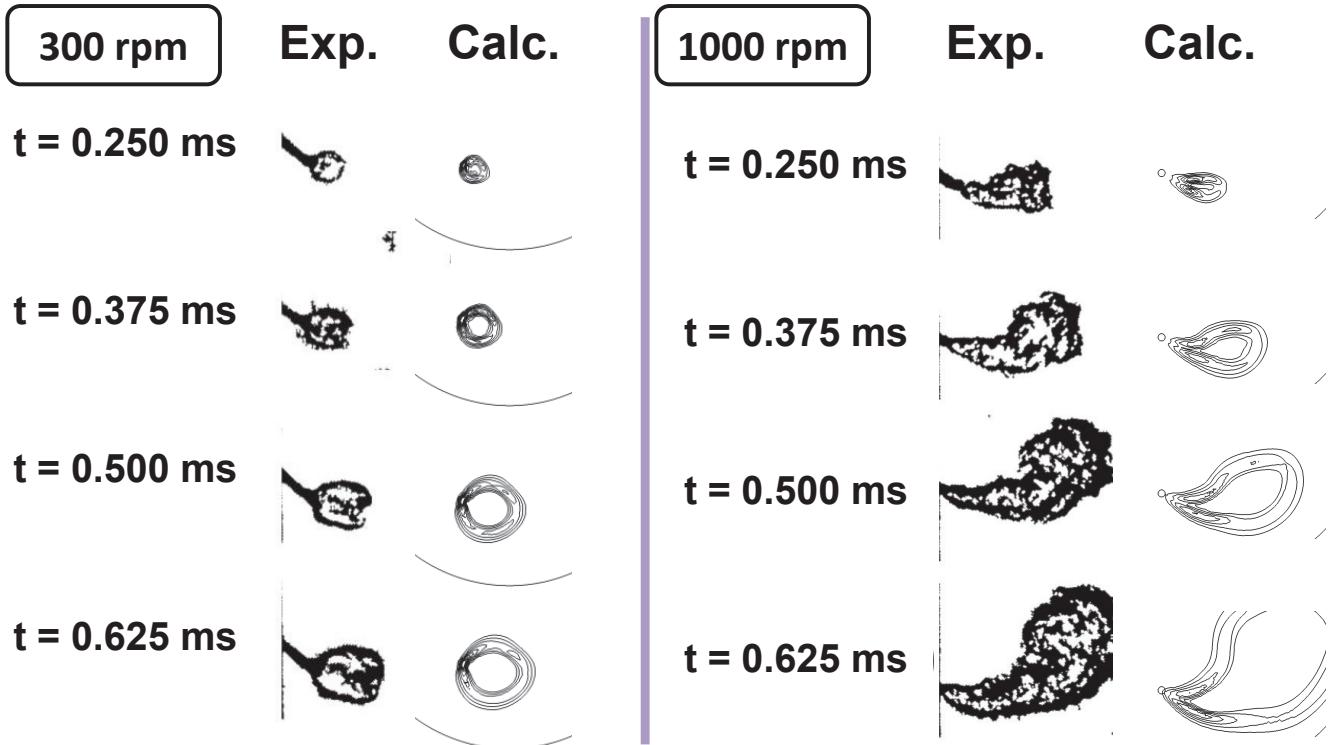


## Peripheral ignition

- Effects of engine speed (turbulence)
  - Effects of  $u'$  correctly reproduced.
  - Proper prediction of increase in flame speed with  $u'$

T. Lucchini: Development of OpenFOAM® libraries for combustion and spray modeling

# Spark-ignition combustion



T. Lucchini: Development of OpenFOAM® libraries for combustion and spray modeling

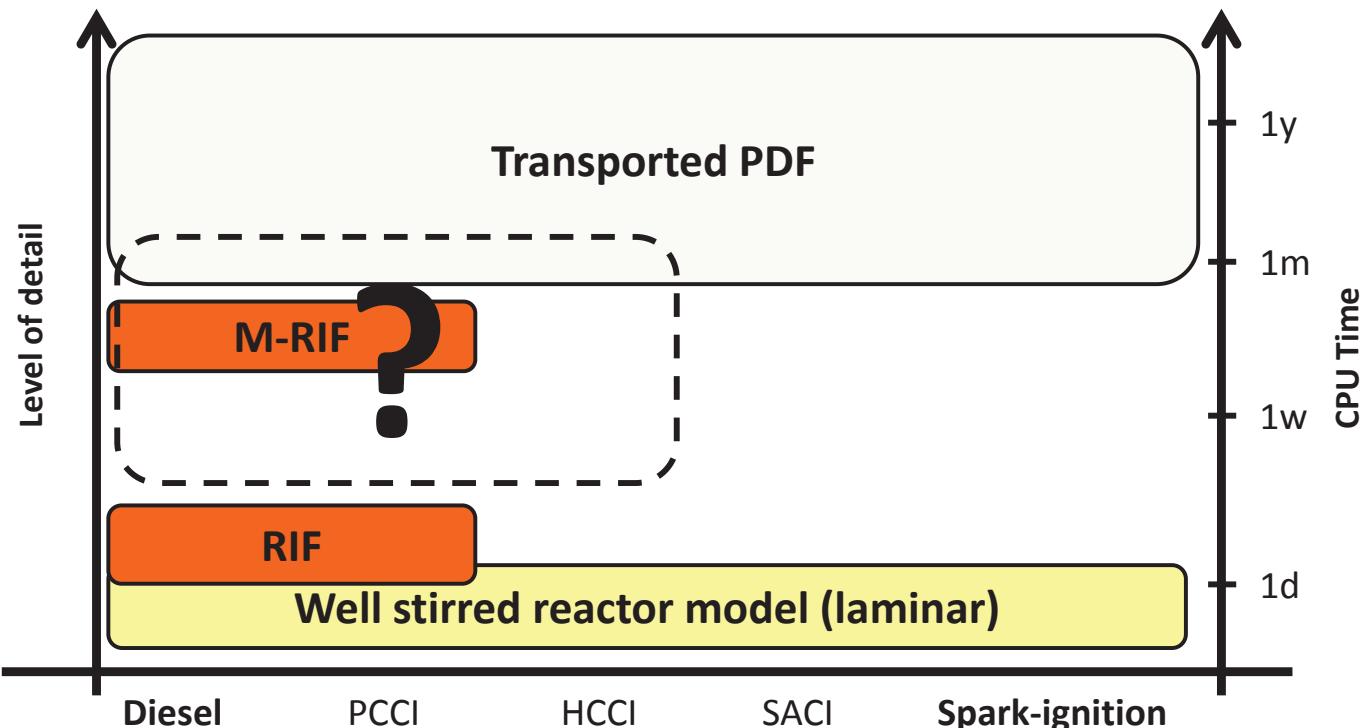
# Spark-ignition combustion



- **Current state**
  - A detailed ignition model must be included
  - Rather well performance at stoichiometric conditions
- **What's next?**
  - Lean mixtures and Lewis number effect
  - More combustion models to be tested (Zimont, G-Equation, ...)
  - Inclusion of detailed chemistry for stratified SI combustion:
    - Tabulated strained laminar premixed flamelets?

T. Lucchini: Development of OpenFOAM® libraries for combustion and spray modeling

## What's next?



T. Lucchini: Development of OpenFOAM® libraries for combustion and spray modeling

91

## Combustion modeling: stochastic PDF

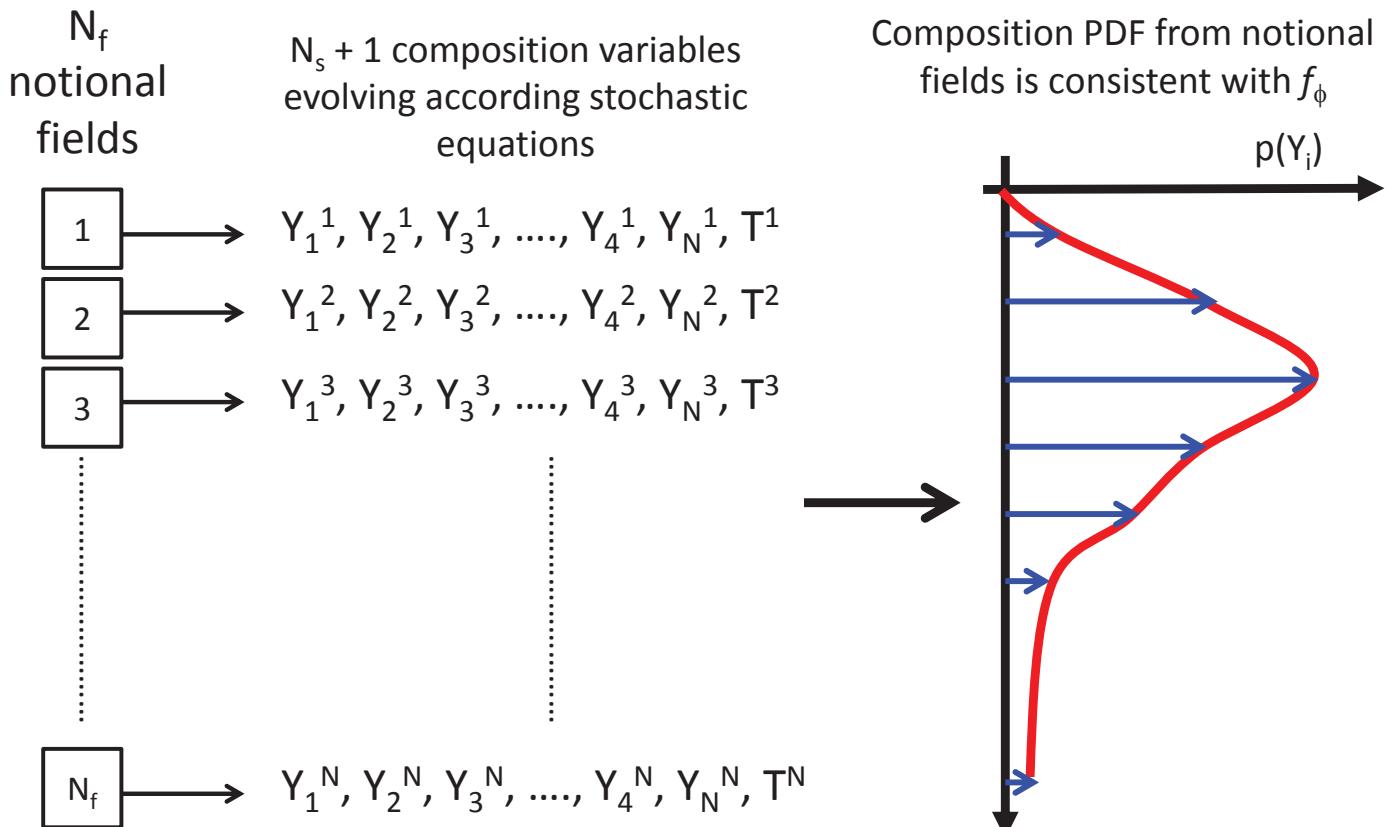
### Composition PDF transport equation ( $f_\phi$ )

$$\frac{\partial \rho f_\phi}{\partial t} + \frac{\partial \rho \tilde{u}_i f_\phi}{\partial x_i} + \frac{\partial \rho \Omega_a f_\phi}{\partial x_i} + \frac{\partial \rho S_a f_\phi}{\partial x_i} = - \frac{\partial}{\partial x_i} \left[ \Gamma_{T\phi} \frac{\partial f_\phi}{\partial x_i} \right] + \frac{\partial}{\partial \psi_\alpha} \left[ \left\langle \frac{\partial J_i^\alpha}{\partial x_i} \right| \psi \right] f_\phi$$

↑      ↑      ↑      ↑      ↑  
 Convection      Chemical reactions      Spray      Turbulent transport      Mixing

- $f_\phi$  is a function, not a scalar or a vector field:  $f_\phi = f_\phi(Y_1, \dots, Y_n, h)$
- Monte Carlo methods to estimate  $f_\phi$ :
  - Lagrangian
  - Eulerian Field

# Combustion modeling: stochastic PDF



T. Lucchini: Development of OpenFOAM® libraries for combustion and spray modeling

# Combustion modeling: stochastic PDF



## Governing equations

- For each composition variable (species or enthalpy) in a **stochastic field**, the following transport equation is solved:

$$d\rho\phi_\alpha^\# = -\frac{\partial\rho\phi_\alpha^\# u_i}{\partial x_i} + \frac{\partial}{\partial x_i} \left[ \Gamma_{T\phi} \frac{\partial\phi_\alpha^\#}{\partial x_i} \right] dt + \Omega_\alpha(\phi^\#)dt + S_\alpha(\phi^\#)dt - \frac{1}{2} \rho C_\phi (\phi_\alpha^\# - \tilde{\phi}_\alpha) \frac{k}{\varepsilon} dt + \sqrt{2\rho\Gamma_{T\phi}} \frac{\partial\phi_\alpha^\#}{\partial x_i} dW_i^\#$$

Convection, turbulent diffusion

Chemistry (closed), spray evaporation

Turbulent mixing

Stochastic term (Wiener process)

## Averaging

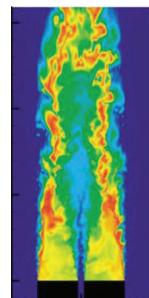
- Composition and enthalpy are computed at each domain point as the average of the corresponding values in each stochastic field:

$$Y_i = \frac{1}{N_f} \sum_{i=1}^{N_f} Y_i^{\#} \quad h = \frac{1}{N_f} \sum_{i=1}^{N_f} h_i^{\#}$$

- ..... so far ... so good... but...

**Is PDF combustion modeling like a roulette game?**

- 1) How many stochastic fields to be used?
- 2) How many chemical species?
- 3) Is the SEF method suitable for engine simulations?



T. Lucchini: Development of OpenFOAM® libraries for combustion and spray modeling

# Combustion modeling: stochastic PDF

## Stochastic error and simulation setup

- When performing the averaging directly on stochastic fields transport equations, an unbounded source terms appears:

$$d\rho\tilde{\phi} = -\frac{\partial \rho\tilde{\phi} u_i}{\partial x_i} + \frac{\partial}{\partial x_i} \left[ \Gamma_{T\phi} \frac{\partial \tilde{\phi}}{\partial x_i} \right] dt + \frac{1}{N_f} \sum_{j=1}^{N_f} \Omega_\alpha(\phi^{\#}) dt + S dt$$

$$\left. + \frac{1}{N_f} \sum_{j=1}^{N_f} \sqrt{2\rho\Gamma_{T\phi}} \frac{\partial \phi^j}{\partial x_i} \eta_i dt^{1/2} \right]$$

**How to reduce the stochastic error?**

- 1) System error: smaller time-steps

$$\Delta t_{SEF} / \Delta t_{CFD, engine} < 0.1$$

- 2) Stochastic error: increase the number of stochastic fields

Flamelets suggests  $N_f > 50$



CPU Time increases

T. Lucchini: Development of OpenFOAM® libraries for combustion and spray modeling

# Combustion modeling: stochastic PDF



## Methodology assessment

- Time-step
- Number of stochastic fields
- Mass and energy conservation
- Capability to predict temperature and species variances

Two test cases:

- 1) ECN, non-reacting Spray-A simulations
- 2) HCCI engine compression

Preliminary results about combustion simulations

---

T. Lucchini: Development of OpenFOAM® libraries for combustion and spray modeling

# Combustion modeling: stochastic PDF



- Comparison with experimental data
- Consistency between computed distributions and integral values:

- 1) From stochastic fields

$$\tilde{Z}_{SEF} = \frac{1}{N_f} \sum_{i=1}^{N_f} Y_{nC_{12}H_{26},i}^{\#} \quad \widetilde{Z}_{SEF}^{\prime 2} = \frac{1}{N_f} \sum_{i=1}^{N_f} \left( Y_{nC_{12}H_{26},i}^{\#} - \tilde{Z}_{SEF} \right)^2$$

- 2) From mixture fraction and variance transport equations

$$\begin{aligned} \frac{\partial \tilde{Z}}{\partial t} + \frac{\partial \tilde{\rho} \tilde{u}_i \tilde{Z}}{\partial x_i} - \frac{\partial}{\partial x_i} \left[ \Gamma_z \frac{\partial \tilde{Z}}{\partial x_i} \right] &= \dot{S} \\ \frac{\partial \widetilde{Z}^{\prime 2}}{\partial t} + \frac{\partial \widetilde{\rho} \widetilde{u}_i \widetilde{Z}^{\prime 2}}{\partial x_i} - \frac{\partial}{\partial x_i} \left[ \Gamma_z \frac{\partial \widetilde{Z}^{\prime 2}}{\partial x_i} \right] &= 2 \Gamma_z \left( \frac{\partial \widetilde{Z}_i}{\partial x_i} \frac{\partial \widetilde{Z}_i}{\partial x_i} \right) - \rho C_x \widetilde{Z}^{\prime 2} \frac{k}{\varepsilon} \quad C_x = 2.0 \end{aligned}$$

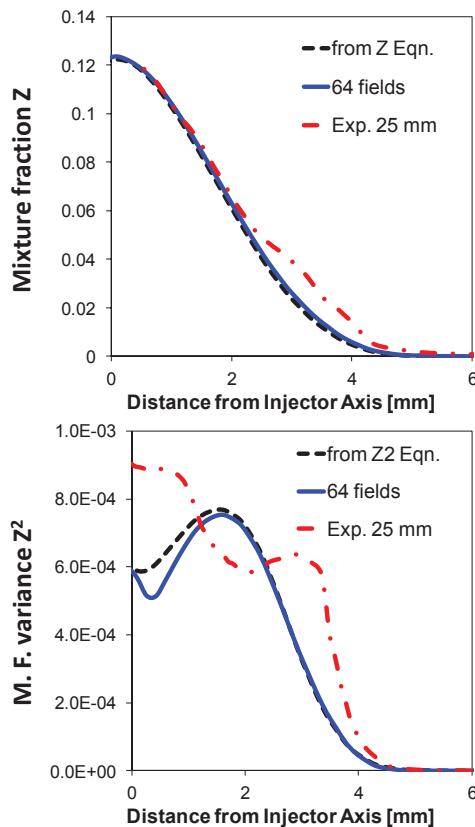
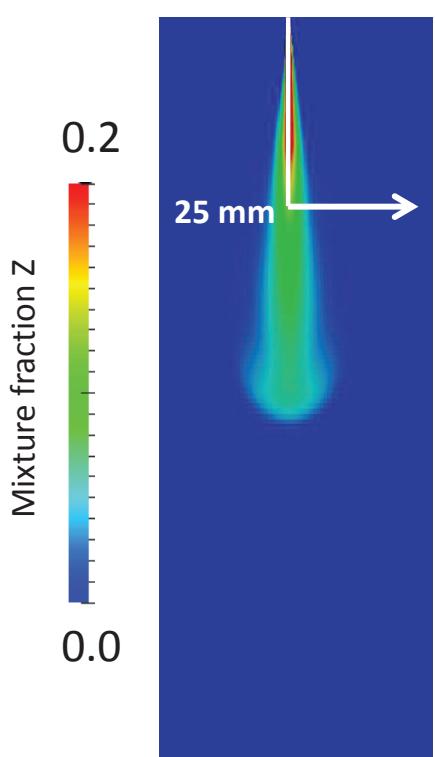
---

T. Lucchini: Development of OpenFOAM® libraries for combustion and spray modeling

# Combustion modeling: stochastic PDF



$t = 1.5 \text{ ms ASOI}$



- Same mixture fraction distribution predicted by SEF and Z equations.

⇒ SEF model is consistent with standard Eulerian models

- Predicted mixture fraction variance distributions are very similar between SEF model and  $Z^2$  equations

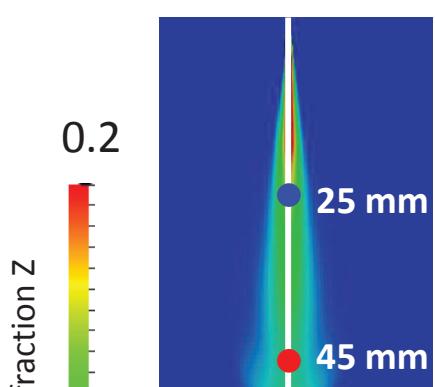
⇒ SEF model can properly predict effects of variances on diffusion combustion

T. Lucchini: Development of OpenFOAM® libraries for combustion and spray modeling

# Combustion modeling: stochastic PDF

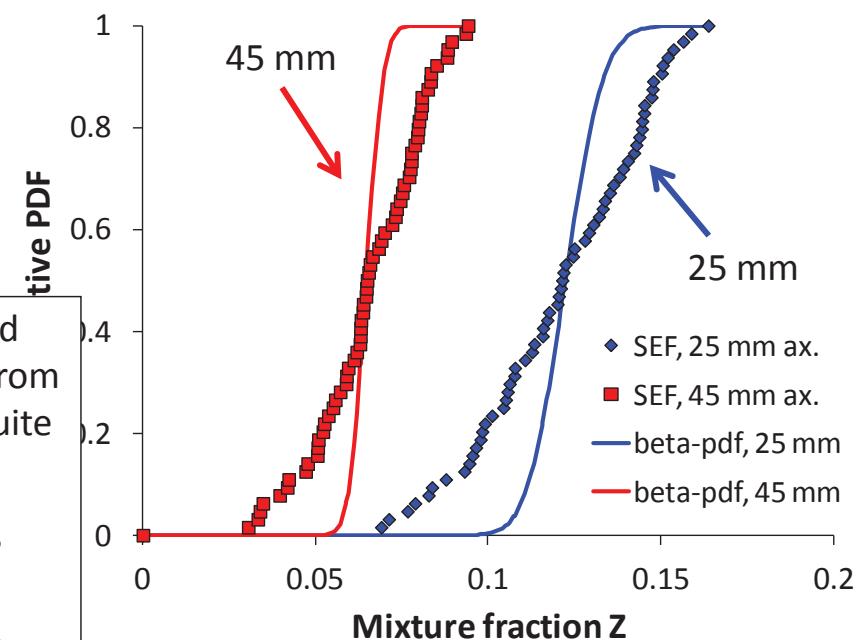


$t = 1.5 \text{ ms ASOI}$



- Despite same mean and variance, distribution from SEF method appears quite different from a  $\beta$ -pdf.
- Larger scatter of points with the SEF method compared to the  $\beta$ -pdf.

- Cumulative PDF of mixture fraction from stochastic fields and comparison with  $\beta$ -pdf (widely used in flamelet combustion models)



T. Lucchini: Development of OpenFOAM® libraries for combustion and spray modeling

# Combustion modeling: stochastic PDF



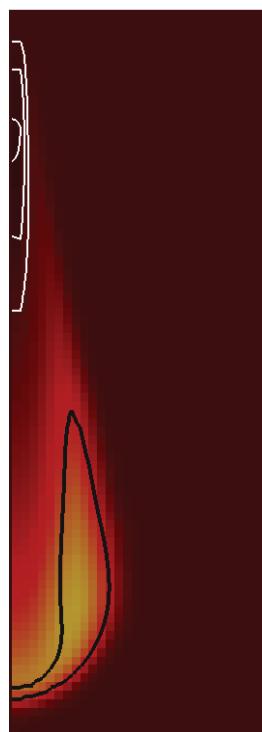
Well-mixed



SEF  
(16 fields)



mRIF  
(15 flamelets)



Results at 0.7 ms

Flame starts to stabilize

Well-mixed model: flame stabilization affected by local flow and turbulence diffusivity. Triple flame structure.

mRIF model: stabilization by auto-ignition of a diffusion flame

SEF model: capability to introduce either premixed and diffusive combustion modes contributing to flame stabilization.

T. Lucchini: Development of OpenFOAM® libraries for combustion and spray modeling

# Combustion modeling: stochastic PDF



## Spray and combustion modeling using OpenFOAM technology

- Implemented capabilities allow to simulate industrial devices with simplified and fast models
- Specific developments are necessary for simulation of IC Engines, gas turbines for a better prediction of fuel-air mixing in engines, flame propagation and pollutant formation.

## Spray and combustion research at Polimi

- Development of Lib-ICE code for spray and combustion modeling: library of validated approaches which can be applied for simulation of fuel-air mixing and combustion in engines:
  - Comprehensive methodology for spray modeling in direct-injection engines
  - Combustion modeling for Diesel engines based on both detailed and simplified chemistry
  - Detailed approach for simulation of SI combustion taking effects of ignition system into account.

T. Lucchini: Development of OpenFOAM® libraries for combustion and spray modeling

# Acknowledgements

- Prof. Gianluca D'Errico
- Prof. Angelo Onorati
- Prof. Gianluca Montenegro
- Dr. Tarcisio Cerri
- Dr. Francesco Contino (Vrije Universiteit Brussel)
- Dr. Mehdi Jangi (Lund University of Technology)
  
- Dr. Lyle Pickett (SANDIA National Laboratories)
- Dr. Scott Parrish (GM)
- Dr. Luigi Allocca, Ing. Alessandro Montanaro (CNR, Istituto Motori)



# References

1. Contino, F., Maserier, J.-B., Foucher, F., Lucchini, T., D'Errico, G., Dagaut, P., "CFD simulations using the TDAC method to model iso-octane combustion for a large range of ozone seeding and temperature conditions in a single cylinder HCCI engine" (2014), Fuel, 137, pp. 179-184.
2. D'Errico, G., Lucchini, T., Contino, F., Jangi, M., Bai, X.-S., "Comparison of well-mixed and multiple representative interactive flamelet approaches for diesel spray combustion modelling" (2014), Combustion Theory and Modelling, 18 (1), pp. 65-88.
3. Lucchini, T., D'Errico, G., Contino, F., Jangi, M., "Towards the Use of Eulerian Field PDF Methods for Combustion Modeling in IC Engines" (2014), SAE International Journal of Engines, 7 (1), pp. 286-296.
4. Cornolti, L., Lucchini, T., Montenegro, G., D'Errico, G., "A comprehensive Lagrangian flame-kernel model to predict ignition in SI engines" (2014), International Journal of Computer Mathematics, 91 (1), pp. 157-174.
5. Lucchini, T., D'Errico, G., Onorati, A., Bonandrini, G., Venturoli, L., Di Gioia, R., "Development and application of a computational fluid dynamics methodology to predict fuel-air mixing and sources of soot formation in gasoline direct injection engines" (2014), International Journal of Engine Research, 15 (5), pp. 581-596.
6. Mohamed Ismail, H., Ng, H.K., Gan, S., Lucchini, T., "Computational study of biodiesel-diesel fuel blends on emission characteristics for a light-duty diesel engine using OpenFOAM" (2013), Applied Energy, 111, pp. 827-841.
7. Lucchini, T., Fiocco, M., Onorati, A., Montanaro, A., Allocca, L., Sementa, P., Vaglieco, B.M., Catapano, F., "Full-Cycle CFD Modeling of Air/Fuel Mixing Process in an Optically Accessible GDI Engine" (2013), SAE International Journal of Engines, 6 (3), pp. 1610-1625.
8. Contino, F., Foucher, F., Dagaut, P., Lucchini, T., D'Errico, G., Mounaim-Rousselle, C., "Experimental and numerical analysis of nitric oxide effect on the ignition of iso-octane in a single cylinder HCCI engine" (2013), Combustion and Flame, 160 (8), pp. 1476-1483.
9. Mohamed Ismail, H., Ng, H.K., Gan, S., Cheng, X., Lucchini, T., "Investigation of biodiesel-diesel fuel blends on combustion characteristics in a light-duty diesel engine using OpenFOAM" (2013), Energy and Fuels, 27 (1), pp. 208-219.



# References

10. Jangi, M., Lucchini, T., D'Errico, G., Bai, X.-S., "Effects of EGR on the structure and emissions of diesel combustion" (2013), Proceedings of the Combustion Institute, 34 (2), pp. 3091-3098.
11. Ismail, H.M., Ng, H.K., Gan, S., Lucchini, T., Onorati, A., "Development of a reduced biodiesel combustion kinetics mechanism for CFD modelling of a light-duty diesel engine" (2013), Fuel, 106, pp. 388-400.
12. Cheng, X., Ismail, H.M., Ng, K.H., Gan, S., Lucchini, T., "Effects of fuel thermo-physical properties on spray characteristics of biodiesel fuels" (2013), Lecture Notes in Electrical Engineering, 191 LNEE (VOL. 3), pp. 117-126.
13. Ismail, H.M., Ng, H.K., Cheng, X., Gan, S., Lucchini, T., D'Errico, G., "Development of thermophysical and transport properties for the CFD simulations of in-cylinder biodiesel spray combustion" (2012), Energy and Fuels, 26 (8), pp. 4857-4870.
14. D'Errico, G., Lucchini, T., Atzler, F., Rotondi, R., "Computational fluid dynamics simulation of diesel engines with sophisticated injection strategies for in-cylinder pollutant controls" (2012), Energy and Fuels, 26 (7), pp. 4212-4223.
15. Contino, F., Lucchini, T., D'Errico, G., Duynslaegher, C., Dias, V., Jeanmart, H., "Simulations of advanced combustion modes using detailed chemistry combined with tabulation and mechanism reduction techniques" (2012), SAE International Journal of Engines, 5(2) pp. 185-196, 2012.
16. Montanaro, A., Allocca, L., Ettorre, D., Lucchini, T., Brusiani, F., Cazzoli, G., "Experimental Characterization of High-Pressure Impinging Sprays for CFD Modeling of GDI Engines" (2011), SAE International Journal of Engines, 4 (1), pp. 747-763.
17. Contino, F., Jeanmart, H., Lucchini, T., D'Errico, G., "Coupling of in situ adaptive tabulation and dynamic adaptive chemistry: An effective method for solving combustion in engine simulations" (2011), Proceedings of the Combustion Institute, 33 (2), pp. 3057-3064.
18. Lucchini, T., D'Errico, G., Ettorre, D., "Numerical investigation of the spray-mesh-turbulence interactions for high-pressure, evaporating sprays at engine conditions" (2011), International Journal of Heat and Fluid Flow, 32 (1), pp. 285-297.
19. Lucchini, T., D'Errico, G., Ettorre, D., Ferrari, G., "Numerical investigation of non-reacting and reacting diesel sprays in constant-volume vessels" (2009), SAE International Journal of Fuels and Lubricants, 2 (1), pp. 966-975.

T. Lucchini: Development of OpenFOAM® libraries for combustion and spray modeling

105



# References

20. D'Errico, G., Ettorre, D., Lucchini, T., "Simplified and detailed chemistry modeling of constant-volume diesel combustion experiments" (2009), SAE International Journal of Fuels and Lubricants, 1 (1), pp. 452-465.
21. D'Errico, G., Lucchini, T., Onorati, A., Hardy, G., "CFD modelling of combustion in Heavy-Duty Diesel Engines" (2014), 8<sup>TH</sup> edition of the International Conference on Thermo-and Fluid Dynamic processes in Direct Injection Engines, Valencia (Spain).
22. Lucchini, T., Fiocco, M., Torelli, R., D'Errico, G., "Automatic mesh generation for full-cycle CFD modeling of IC engines: Application to the TCC test case" (2014), SAE Technical Paper 2014-01-1131, SAE World Congress 2014, Detroit (USA).
23. Brusiani, F., Bianchi, G.M., Falfari, S., Onorati, A., Lucchini, T., Di Gioia, R., "Influence of Cylindrical, k, and ks diesel nozzle shape on the injector internal flow field and on the emerging spray characteristics" (2014), SAE Technical Paper 2014-01-1428, SAE World Congress 2014, Detroit (USA).
24. Pasunurthi, S.S., Hawkes, E.R., Joelsson, T., Rusly, A.M., Kook, S., Lucchini, T., D'Errico, G., "Numerical simulation of sprays and combustion in an automotive-size diesel engine under non-reacting and reacting conditions" (2013), Proceedings of the 9<sup>TH</sup> Asia-Pacific Conference on Combustion, ASPACC 2013.
25. D'Errico, G., Lucchini, T., Stagni, A., Frassoldati, A., Faravelli, T., Ranzi, E., "Reduced kinetic mechanisms for diesel spray combustion simulations" (2013), SAE Technical Paper 2013-24-0014, 11<sup>TH</sup> International Conference on Engines & Vehicles, Capri (Italy).
26. Lucchini, T., Cornolti, L., Montenegro, G., D'Errico, G., Fiocco, M., Teraji, A., Shiraishi, T., "A comprehensive model to predict the initial stage of combustion in SI engines" (2013), SAE Technical Paper 2013-01-1087, SAE World Congress 2013, Detroit (USA).
27. Lucchini, T., D'Errico, G., Contino, F., Jangi, M., Bai, X.-S., "Diesel combustion modeling with detailed chemistry and turbulence-chemistry interaction" (2013), International Multi-Dimensional Engine Modeling Meeting, Detroit (USA).
28. Lucchini, T., D'Errico, G., Onorati, A., Montenegro, G., "Multi-dimensional Modeling of Flame Kernel Growth and Flame Propagation Processes in Spark-Ignition Engines" (2013), SIA International Conference The Spark Ignition Engine of the Future, Strasbourg (France).

T. Lucchini: Development of OpenFOAM® libraries for combustion and spray modeling

106



# References

29. Lucchini, T., D'Errico, G., Onorati, A., Bonandrini, G., Venturoli, L., Di Gioia, R., "Development of a CFD approach to model fuel-air mixing in gasoline direct-injection engines" (2012), SAE Technical Paper 2012-01-0146, SAE World Congress 2012, Detroit (USA).
30. Som, S., D'Errico, G., Longman, D., Lucchini, T., "Comparison and standardization of numerical approaches for the prediction of non-reacting and reacting diesel sprays" (2012), SAE Technical Paper 2012-01-1263, SAE World Congress 2012, Detroit (USA).
31. Lucchini, T., D'Errico, G., Cornolti, L., Montenegro, G., Onorati, A., "Development of a CFD methodology for fuel-air mixing and combustion modeling of GDI Engines" (2012), International Multi-Dimensional Engine Modeling Meeting, Detroit (USA).
32. Briani, M., Fraioli, V., Migliaccio, M., Di Blasio, G., Lucchini, T., Ettorre, D., "Multi-dimensional modeling of combustion in compression ignition engines operating with variable charge premixing levels" (2011), SAE Technical Paper 2011-24-0027, 10<sup>TH</sup> International Conference on Engines & Vehicles, Capri (Italy).
33. Lucchini, T., D'Errico, G., Fiocco, M., "Multi-dimensional modeling of gas exchange and fuel-air mixing processes in a direct-injection, gas fueled engine" (2011), SAE Technical Paper 2011-24-0036, 10<sup>TH</sup> International Conference on Engines & Vehicles, Capri (Italy).
34. Onorati, A., Lucchini, T., D'Errico, G., Di Gioia, R., Bonandrini, G., Venturoli, L., "Development of a Comprehensive CFD Methodology for Spray Targeting Optimization in Gasoline Direct-injection Engines" (2011), SIA International Conference The Spark Ignition Engine of the Future, Strasbourg (France).
35. Lucchini, T., D'Errico, G., Ettorre, D., Brusiani, F., Bianchi, G.M., Montanaro, A., Allocata, L., "Experimental and numerical investigation of high-pressure diesel sprays with multiple injections at engine conditions" (2010), SAE Technical Paper 2010-01-0179, SAE World Congress 2010, Detroit (USA).
36. D'Errico, G., Ettorre, D., Lucchini, T., Contino, F., "Diesel Engine Simulation with Tabulation of Dynamic Adaptive Chemistry and Adaptive Local Mesh Refinement", (2010), International Multi-Dimensional Engine Modeling Meeting, Detroit (USA).
37. Lucchini, T., D'Errico, G., Brusiani, F., Bianchi, G.M., Tukovic, Z., Jasak, H., "Multi-dimensional modeling of the air/fuel mixture formation process in a PFI engine for motorcycle applications" (2009), SAE Technical Paper 2009-24-0015, 9<sup>TH</sup> International Conference on Engines & Vehicles, Capri (Italy).

T. Lucchini: Development of OpenFOAM® libraries for combustion and spray modeling

107

# References

38. Brusiani, F., Bianchi, G.M., Lucchini, T., D'Errico, G., "Implementation of a finite-element based mesh motion technique in an open source CFD code" (2009), Proceedings of the Spring Technical Conference of the ASME Internal Combustion Engine Division, pp. 611-624.
39. Lucchini, T., D'Errico, G., Brusiani, F., Bianchi, G.M., "A finite-element based mesh motion technique for internal combustion engine simulations" (2008), Proceedings of the 7th International Conference on Modeling and Diagnostics for Advanced Engine Systems, COMODIA 2008, pp. 671-678.
40. D'Errico, G., Lucchini, T., Montenegro, G., Onorati, A., Ettorre, D., "Fundamental and Applied Studies of Detailed Chemistry Based Models for Diesel Combustion" (2008), Proceedings of the THIESEL 2008 Conference on Thermo-and Fluid Dynamic Processes in Diesel Engines, Valencia (Spain).
41. Lucchini, T., D'Errico, G., Jasak, H., Tukovic, Z., "Automatic mesh motion with topological changes for engine simulation" (2007), SAE Technical Paper 2007-01-0170, SAE World Congress 2007, Detroit (USA).
42. D'Errico, G., Lucchini, T., Onorati, A., Mehl, M., Faravelli, T., Ranzi, E., Merola, S., Vaglieco, B.M., "Development and experimental validation of a combustion model with detailed chemistry for knock predictions" (2007), SAE Technical Paper 2007-01-0938, SAE World Congress 2007, Detroit (USA).
43. D'Errico, G., Lucchini, T., Onorati, A., "Development and implementation of Diesel combustion models into an opensource CFD platform" (2007), 6th Symposium Towards Clean Diesel Engines, Ischia (Italy).
44. D'Errico, G., Ettorre, D., Lucchini, T., "Comparison of combustion and pollutant emission models for DI diesel engines" (2007), SAE Technical Paper 2007-24-0045, 8<sup>TH</sup> International Conference on Engines & Vehicles, Capri (Italy).
45. D'Errico, G., Lucchini, T., Montenegro, G., Zanardi, M., "Development of OpenFOAM application for Internal Combustion Engine Simulation" (2007), OpenFOAM International Conference, Windsor (Great Britain).

T. Lucchini: Development of OpenFOAM® libraries for combustion and spray modeling

108

# References



46. Lucchini, T., D'Errico, G., "Automatic mesh motion, topological changes and innovative mesh setup for I.C.E. simulations" (2006), High Tech Engines and Car Conference, Modena (Italy).
47. Lucchini, T., D'Errico, G., Nordin, N., "CFD Modelling of Gasoline Sprays" (2005), SAE Technical Paper 2005-24-086, 7<sup>TH</sup> International Conference on Engines & Vehicles, Capri (Italy).

# Turbulent Combustion Modeling

Workshop for OpenFOAM and its Application in Industrial Combustion Devices

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

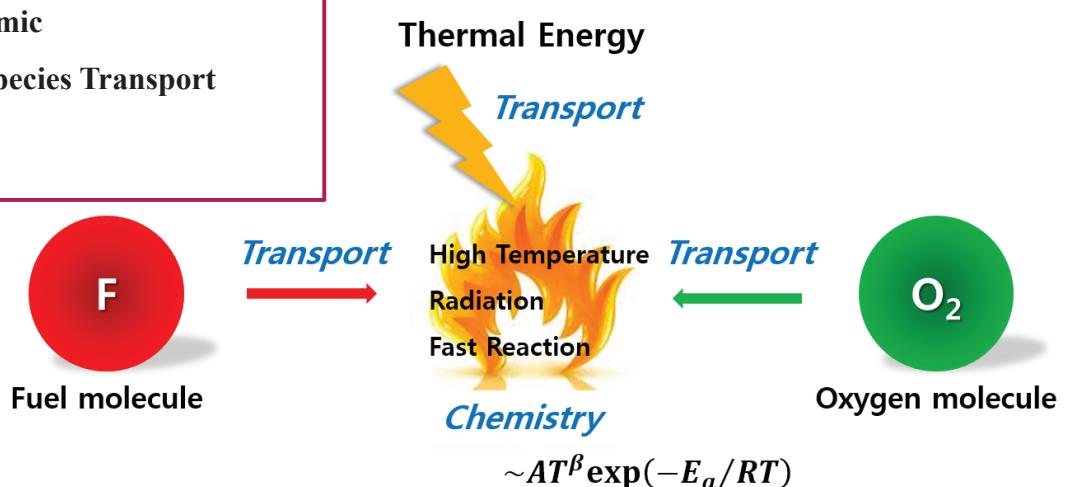
Kang Y. Huh  
Pohang University of Science and Technology



## Combustion

### Basic principle

- Thermodynamics
- Transport
  - Fluid Dynamic
  - Heat and Species Transport
- Chemistry



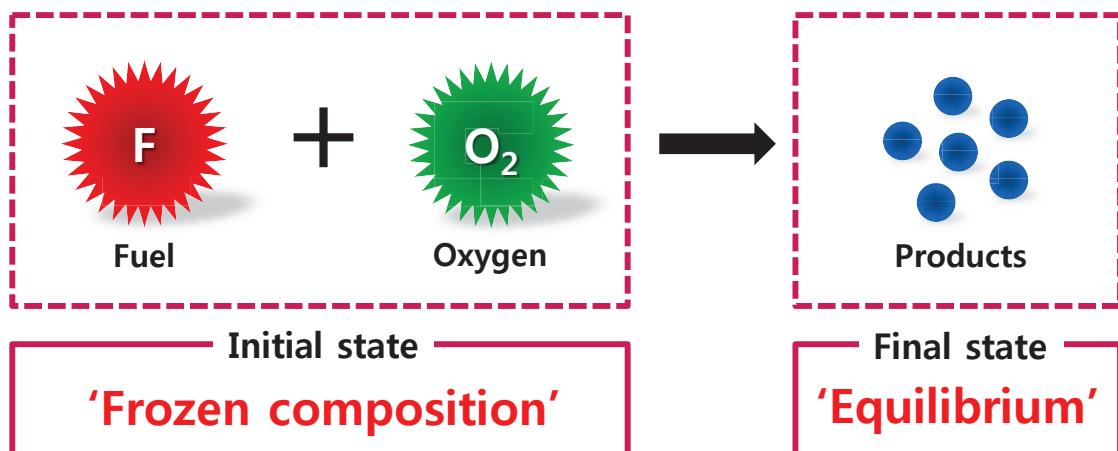
We need both **transport** and **chemistry** for combustion to occur.

$$\tau_{comb} \sim \tau_{trans} + \tau_{chem}$$

Usually  $\tau_{chem} \ll \tau_{trans}$  (fast chemistry or equilibrium assumption)

# Combustion

## Thermodynamics



How to relate initial and final states?

- **1<sup>st</sup> law – Energy Conservation**      Chemical energy  $\rightarrow$  Sensible energy

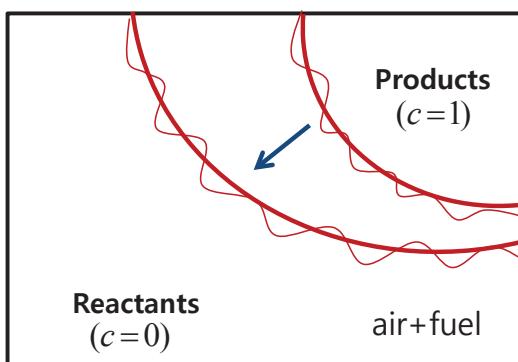
- **2<sup>nd</sup> law – Direction**

$$dS \geq \frac{\delta Q}{T} \quad \text{One way only (IS} \rightarrow \text{FS)}$$

Combustion Laboratory POSTECH

# Combustion

## Premixed Flame



- Reaction Progress Variable

$$c = \frac{\sum_i a_i (Y_i - Y_i^u)}{\sum_i a_i (Y_i^{eq} - Y_i^u)} = \frac{Y_c}{Y_c^{eq}}$$

$^u$  : unburnt reactant

$^{eq}$  : chemical equilibrium

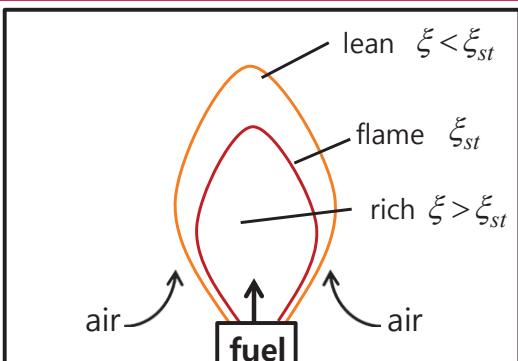
$Y_i$  : i-th species mass fraction

$a_i$  : constant

$c = 0$  where the mixture is unburnt

$c = 1$  where the mixture is burnt

## Nonpremixed Flame



- Mixture Fraction

$$\begin{aligned} \xi &= \frac{\text{mass originating from fuel stream}}{\text{mass of mixture}} \\ &= \frac{m_{fuel}}{m_{fuel} + m_{oxidizer}} \end{aligned}$$

# Turbulent Combustion

## Governing equations

- Favre Averaged Conservation Eqs with Nonlinear Terms

$$\frac{\partial \bar{\rho}}{\partial t} + \frac{\partial}{\partial x_k} \bar{\rho} \tilde{v}_k = 0$$

$$\frac{\partial}{\partial t} \bar{\rho} \tilde{Y}_i + \frac{\partial}{\partial x_k} \bar{\rho} \tilde{v}_k \tilde{Y}_i = - \frac{\partial}{\partial x_k} \bar{J}_{ik} + \dot{\tilde{w}}_i \quad \text{Nonlinear Reaction Term}$$

$$\frac{\partial}{\partial t} \bar{\rho} \tilde{v}_i + \frac{\partial}{\partial x_k} \bar{\rho} \tilde{v}_k \tilde{v}_i = - \frac{\partial \bar{p}}{\partial x_i} + \frac{\partial}{\partial x_k} \bar{\tau}_{ik} + \bar{g}_i$$

$$\frac{\partial}{\partial t} \bar{\rho} \tilde{h} + \frac{\partial}{\partial x_k} \bar{\rho} \tilde{v}_k \tilde{h} = \frac{\partial \bar{p}}{\partial t} + \frac{\partial}{\partial x_k} \left[ \frac{\mu}{\sigma} \frac{\partial \bar{h}}{\partial x_k} + \mu \sum_{i=1}^N \left( \frac{1}{Sc_i} - \frac{1}{\sigma} \right) h_i \frac{\partial \bar{Y}_i}{\partial x_k} \right]$$

$$\bar{p} = \bar{\rho} R \tilde{T}$$

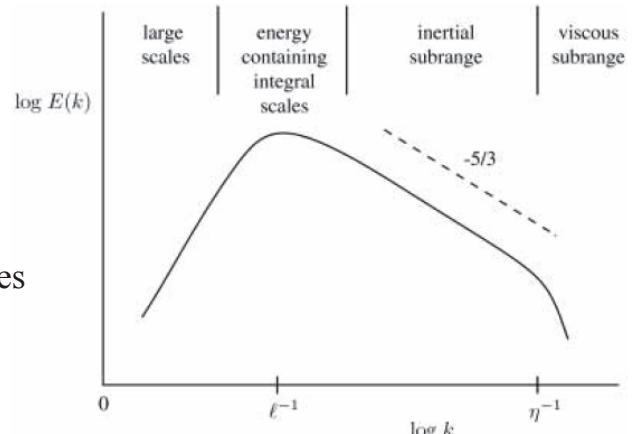
$$\tilde{h} = h(\tilde{Y}_i, \bar{p}, \tilde{T})$$

Nonlinear Convection Term

# Turbulent Combustion

## Turbulence

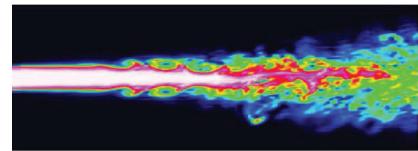
- Irregularity**
  - Deterministic Approach ( X )
  - Statistical / Probabilistic Approach ( O )
- Enhanced Diffusivity**
  - Rapid mixing – Engineering concern
- 3D Vortex Stretching**
  - Energy cascade from large to small eddies
- Dissipation**
  - Kinetic energy → Thermal energy



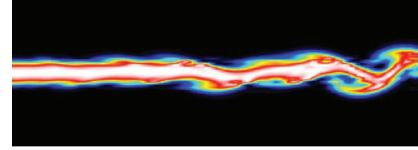
# Turbulent Combustion

## Numerical Simulation

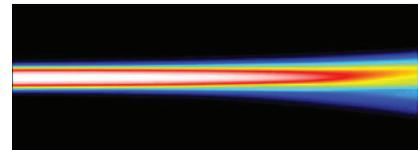
- **DNS:** No Averaging, Fully Resolved  
No turbulent combustion model



- **LES:** Spatial Filtering, Partially Resolved  
Subgrid model



- **RANS:** Averaging, Mean Resolved/Fluctuations Modeled  
Unconditional model  
Conditional model



Norbert Peters. *Turbulent combustion*. Cambridge university press, 2000.

Combustion Laboratory  POSTECH

# Turbulent Combustion

## Reynolds vs. Favre Averaging (Unconditional Averaging)

- Reynolds Averaging

$$u_i(x) = \bar{u}_i(x) + u'_i$$

$$\overline{u_i u_j} = \bar{u}_i \bar{u}_j + \overline{u'_i u'_j}$$

- Favre Averaging (Density-weighted Averaging)

$$\tilde{u}_i(x) = \overline{\rho u_i}(x) / \bar{\rho}(x)$$

$$u_i(x) = \tilde{u}_i(x) + u''_i \quad \tilde{u}'' = 0$$

$$\overline{\rho u_i u_j} = \bar{\rho} \bar{u}_i \bar{u}_j + \bar{\rho} \overline{u'_i u'_j} + \bar{u}_i \overline{\rho' u'_j} + \bar{u}_i \overline{\rho' u'_j} + \bar{\rho}' \overline{u'_i u'_j} = \bar{\rho} \tilde{u}_i \tilde{u}_j + \overline{\rho u''_i u''_j}$$

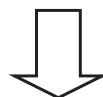
$$\widetilde{u'_i u'_j} = \tilde{u}_i \tilde{u}_j + \widetilde{u''_i u''_j}$$

Same simple formalism as Reynolds averaging and same closure models employed in practice without rigorous validation or measurement

Norbert Peters. *Turbulent combustion*. Cambridge university press, 2000.

Combustion Laboratory  POSTECH

# Turbulent Combustion



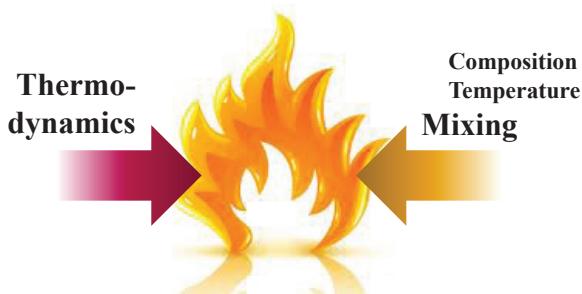
**Large Fluctuations of all Scalar  
and Vector Quantities**



**Problems both in Measurement  
and Computation**

## Combustion

Fast chemistry



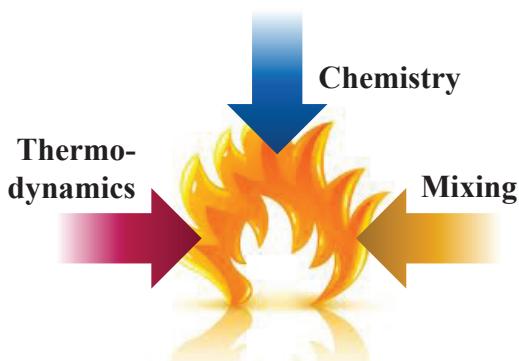
**Localized reaction in flamelets**

**Near equilibrium throughout the domain**



**Simplified treatment possible**

Finite/Slow Chemistry



**Distributed reaction away from  
equilibrium**

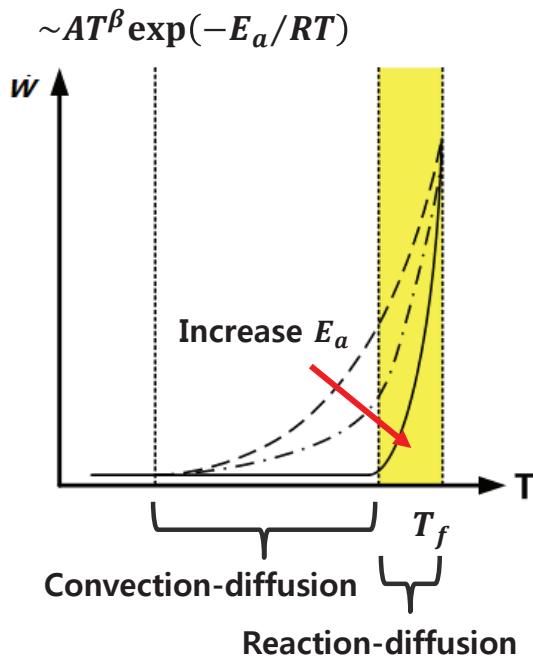


**Tight coupling between turbulence and  
chemistry**

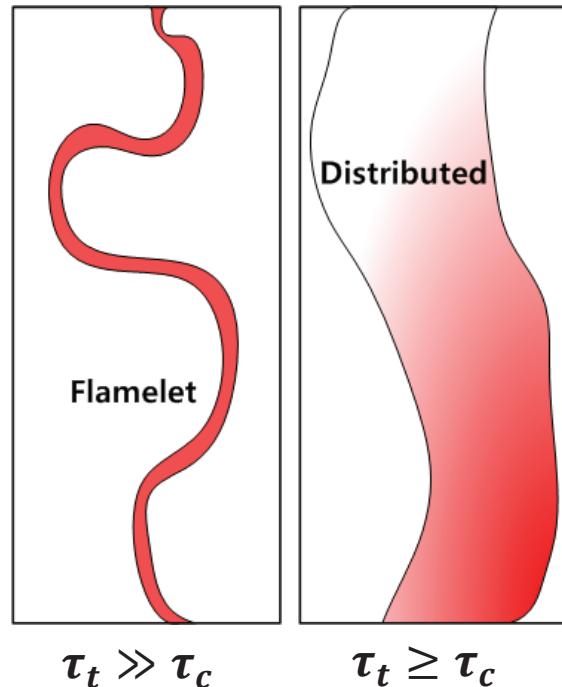
***More Difficult Problem!***

# Why Is Reaction Localized in Space?

## Large Activation Energy



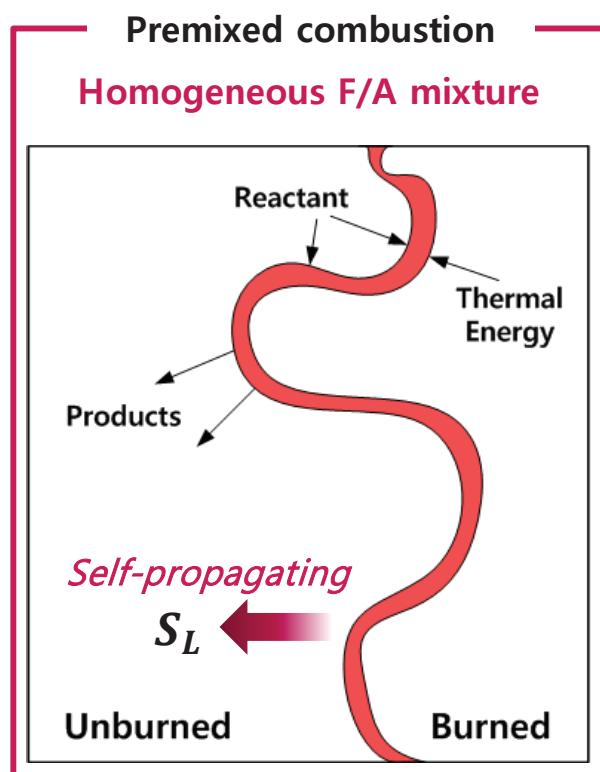
## Chemistry Faster than Mixing



Combustion Laboratory POHANG UNIVERSITY OF SCIENCE AND TECHNOLOGY POSTECH

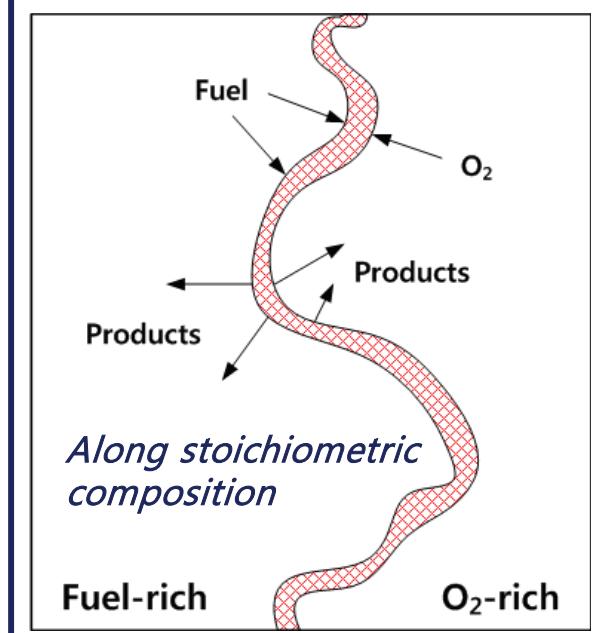
# Combustion

## Laminar Flamelet Regime



## Nonpremixed combustion

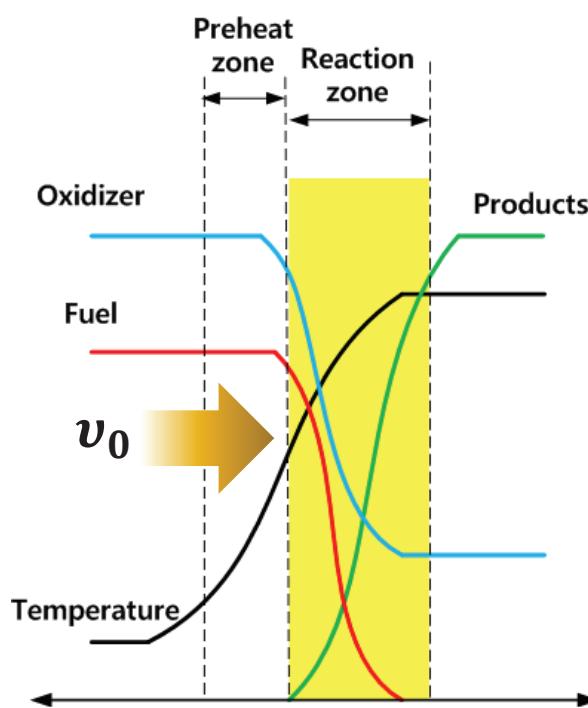
F/A separate at mixing



Combustion Laboratory POHANG UNIVERSITY OF SCIENCE AND TECHNOLOGY POSTECH

# Laminar Premixed Flame

## Fast Chemistry Flamelet Regimes



$$\text{Reaction rate} \sim \exp\left(-\frac{T_a}{T}\right)$$

AEA (Activation Energy Asymptotics)

$$T_a/T_f \gg 1$$

Conservation equation:

$$\alpha \frac{d^2\delta}{dz^2} - \nu \frac{d\delta}{dz} + \delta A \exp\left(-\frac{E}{R(T_b - \delta)}\right) = 0$$

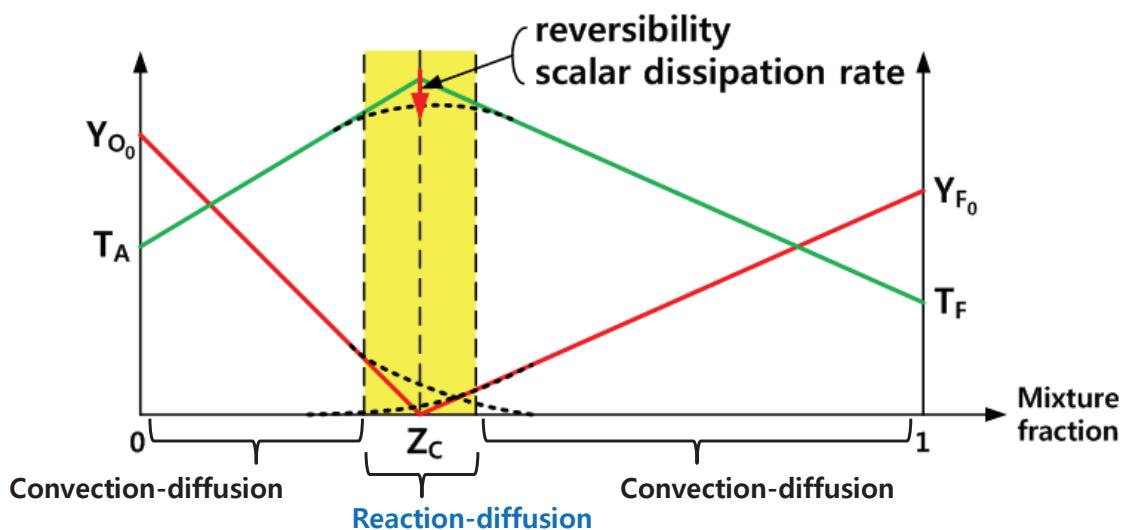
$$\text{Eigenvalue problem} \quad v_0 = \sqrt{\frac{\alpha}{\tau}}$$

or numerical solution

Combustion Laboratory  POSTECH

# Laminar Diffusion Flame

## Structure for Laminar diffusion flame



$$-\rho D |\nabla z|^2 \frac{d^2 Y}{dz^2} = w_i \quad \text{or numerical solution}$$

SDR(Scalar Dissipation Rate) – *leads to Extinction*

Combustion Laboratory  POSTECH

# Turbulent Premixed Flame

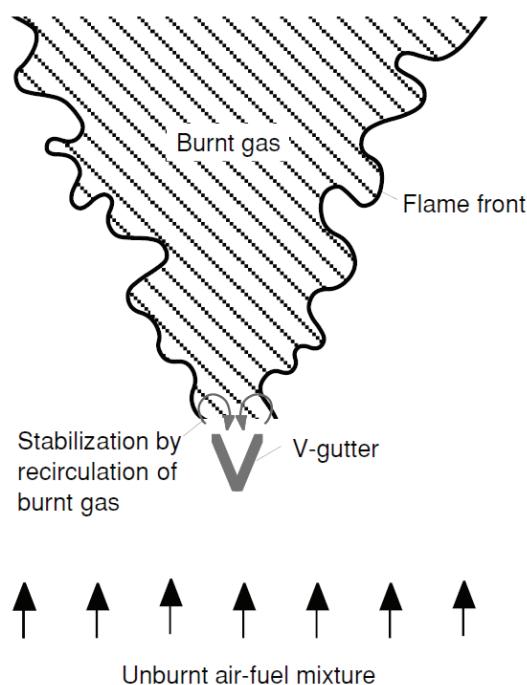


Fig. Schematic illustration of an instantaneous picture of a "V-shaped" turbulent premixed flame stabilized by a bluff-body

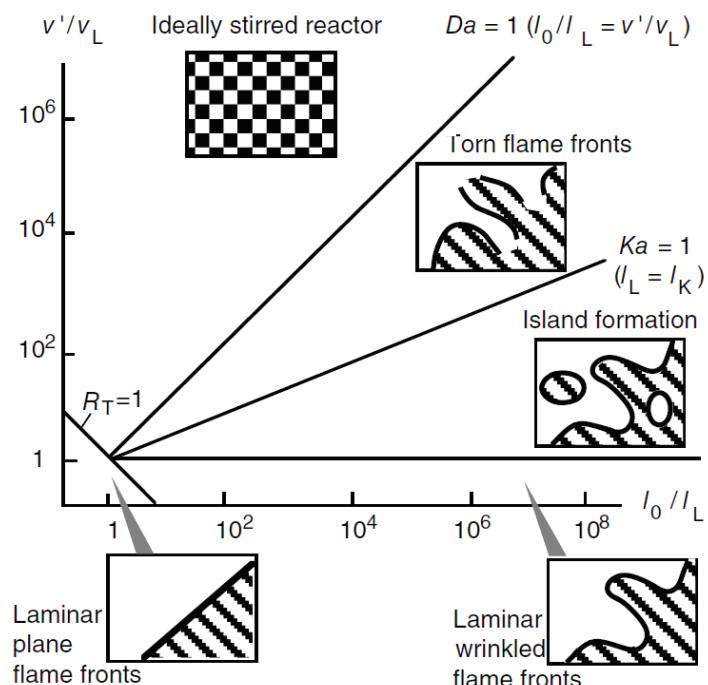
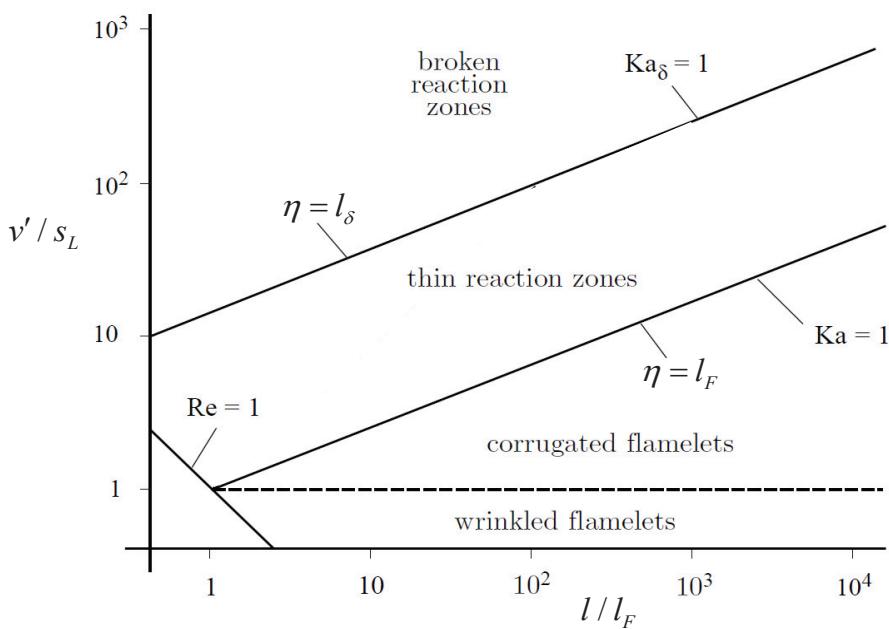


Fig. Borghi Diagram

# Turbulent Premixed Flame



Regime diagram for premixed turbulent combustion

$$l_F = \frac{D}{s_L}, \quad t_F = \frac{D}{s_L^2}.$$

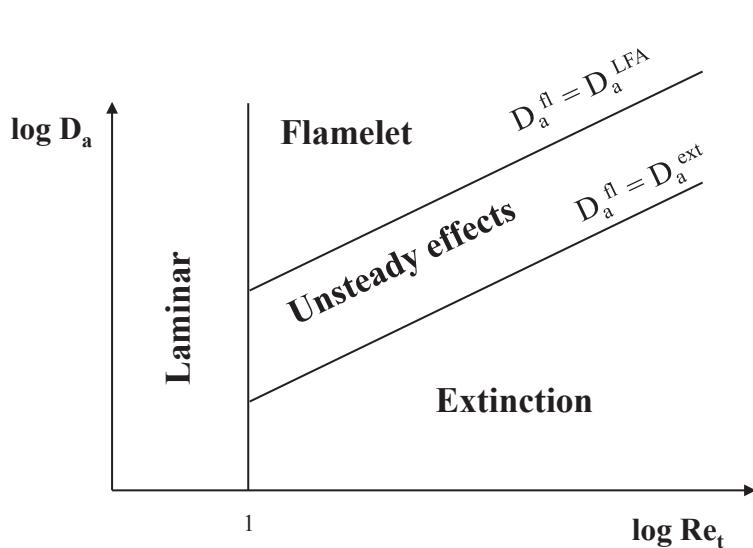
$$\text{Re} = \frac{v'l}{s_L l_F}$$

$$Da = \frac{s_L l}{v' l_F}$$

$$Ka = \frac{t_F}{t_\eta} = \frac{l_F^2}{\eta^2} = \frac{v_\eta^2}{s_L^2}$$

$$\text{Re} = Da^2 Ka^2$$

# Turbulent Diffusion Flame



$$D_a^{fl} = \frac{1}{\tau_c \chi_{st}}$$

$$\chi_{st} = \left\langle 2D \left( \frac{\partial \xi}{\partial x_i} \frac{\partial \xi}{\partial x_i} \right) \middle| \xi = \eta_{st} \right\rangle$$

$$D_a = \frac{\tau_t}{\tau_c} = \frac{\tau_t}{\tau_k} \frac{\tau_k}{\tau_c} \approx \frac{\tau_t}{\tau_k} \frac{2}{\tilde{\chi}_{st} \tau_c} \approx 2\sqrt{\text{Re}_t D_a^{fl}}$$

$$D_a^{\text{LFA}} \approx 2D_a^{\text{SE}}$$

$$D_a^{\text{ext}} \approx 0.4D_a^{\text{SE}}$$

Regime diagram for nonpremixed turbulent combustion

Combustion Laboratory POHANG UNIVERSITY OF SCIENCE AND TECHNOLOGY POSTECH

# Turbulent Diffusion Flame

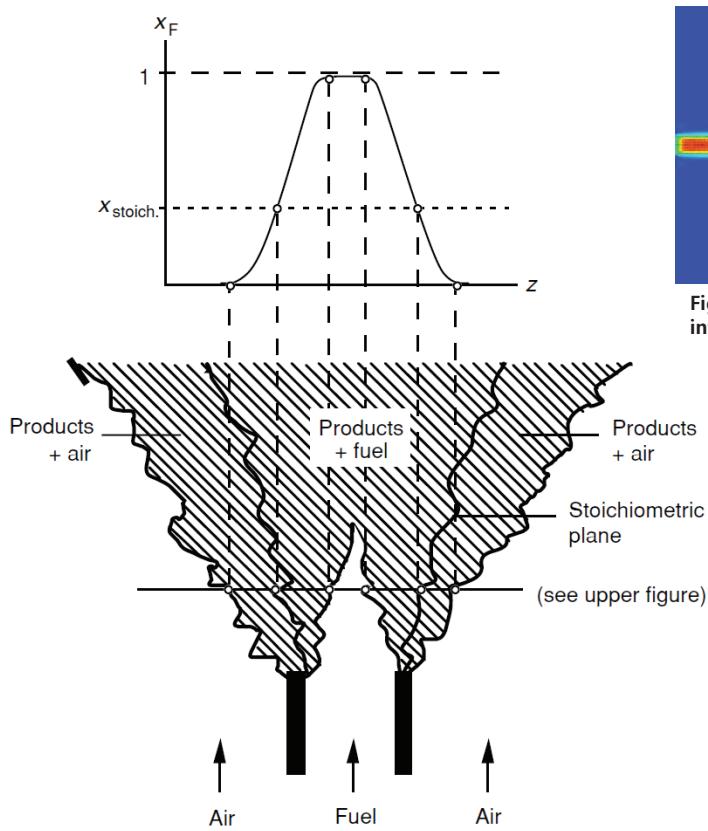


Fig. Schematic drawing of a momentary picture of a turbulent nonpremixed jet flame

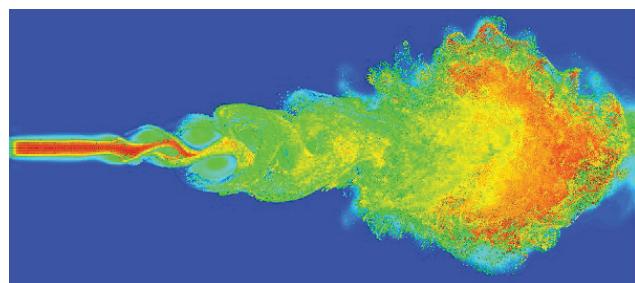


Fig. Large eddy simulation of a turbulent diffusion flame interacting with evaporating water droplets.

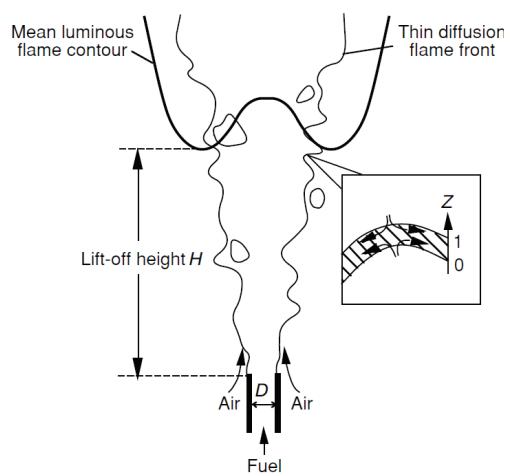


Fig. Schematic illustration of the lift-off behavior of a turbulent jet nonpremixed flame

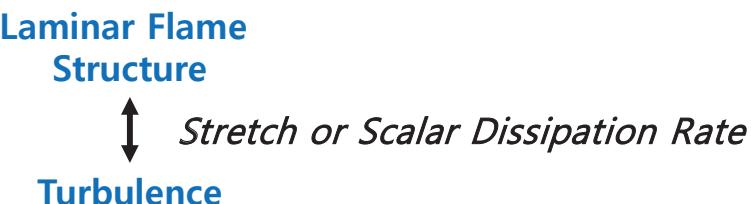
Combustion Laboratory POHANG UNIVERSITY OF SCIENCE AND TECHNOLOGY POSTECH

# How to Resolve Turbulent/Chemistry Coupling – Nonpremixed Comb.

## (1) Equilibrium Assumption – Infinitely fast chemistry

## (2) Laminar Flamelet Model

- Steady – SLFM(Stationary Laminar Flamelet Model)
- Transient – RIF(Representative Interactive Flamelet)



## (3) Conditional Averaging

### CMC(Conditional Moment Closure)

- Deterministic relationships between mixture fraction and all other reactive scalars.
- 1<sup>st</sup> order closure for chemical reaction rate.

# How to Resolve Turbulent/Chemistry Coupling – Nonpremixed Comb.

## (4) Tabulation

→ Scalars given as a function of a few local parameters

- SLFM(Stationary Laminar Flamelet Model)
- FPI(Flame Prolongation of ILDM) or FGM(Flamelet Generated Manifold)
- ISAT(In Situ Adaptive Tabulation)

## (5) ILDM(Intrinsic Low Dimensional Manifold)

- Fast processes in Local Equilibrium
- Integration along the attracting manifold of larger chemical time scales

## (6) Direct Integration

- DNS with Detailed Chemistry – Not feasible

# Combustion

## Applications

### Conventional Devices

- ① Fast chemistry
- ② Transport by turbulence
- ③ Chemistry not relevant for the mean reaction rate

### New Devices

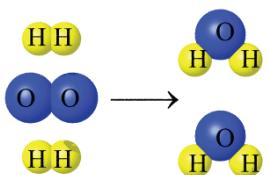
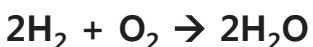
- ① Finite chemistry – Emissions and Heat Release
- ② Both transport and chemistry : Rate controlling steps

## Chemistry as a Controlling Step

|    | Combustion                    | Examples                 |
|----|-------------------------------|--------------------------|
| 1. | Emissions                     | NO <sub>x</sub> , PM, CO |
| 2. | Ignition/Extinction           | Knocking, Auto-ignition  |
| 3. | Low Temperature Combustion    | LTC Diesel               |
| 4. | Lean Premixed Combustion      | DLN G/T, HCCI            |
| 5. | High EGR(FGR)+Preheating      | MILD                     |
| 6. | Partially Premixed Combustion | SI GDI                   |

# Combustion Chemistry

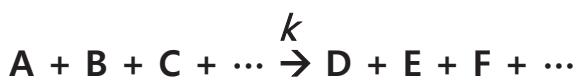
## Global reaction step



## Elementary reaction step

- ①  $\text{H} + \text{O}_2 \rightarrow \text{OH} + \text{O}$
- ②  $\text{H}_2 + \text{O} \rightarrow \text{OH} + \text{H}$
- ③  $\text{OH} + \text{OH} \rightarrow \text{H}_2\text{O} + \text{O}$

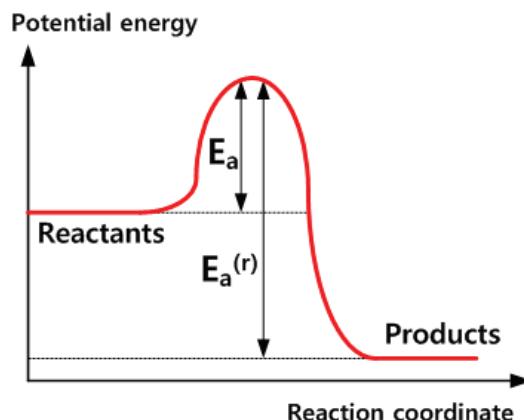
## Reaction rate



$$\frac{d[A]}{dt} = -k[A]^a[B]^b[C]^c\dots$$

## Arrhenius law

$$k = AT^\beta \exp(-E_a/RT)$$



# Reaction Mechanism

## Detailed chemical mechanism

Complete reaction steps (Too large to handle computationally)

## Skeletal chemical mechanism

Rate limiting reaction steps (Eliminate unimportant steps)

## Reduced chemical mechanism

Quasi-Steady analysis → Species  
Partial Equilibrium analysis → Elementary steps

(Exp.) Reduction for n-Heptane (Combustion and Flame 154 (2008) 153-163)

### Detailed mechanism

561 species

2539 reactions

- ① DRG
- ② DRGASA
- ③ Reaction Elimination
- ④ Isomer Lumping

### Skeletal mechanism

68 species

283 reactions

- ① QSS Reduction
- ② QSS Graph
- ③ Diffusive Species Bundling

### Reduced mechanism

55 species  
51 global steps  
14 diffusion species

# Radical Chain Reaction

- Radicals: OH, O, H, HO<sub>2</sub>, etc

**Chain reactions of very reactive radicals form the basis of combustion processes.**

- Major chain reactions for hydrocarbon fuels

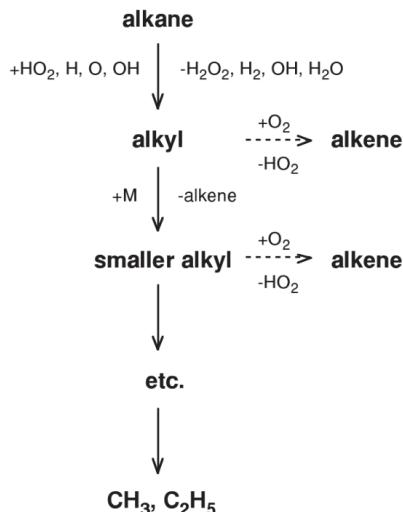
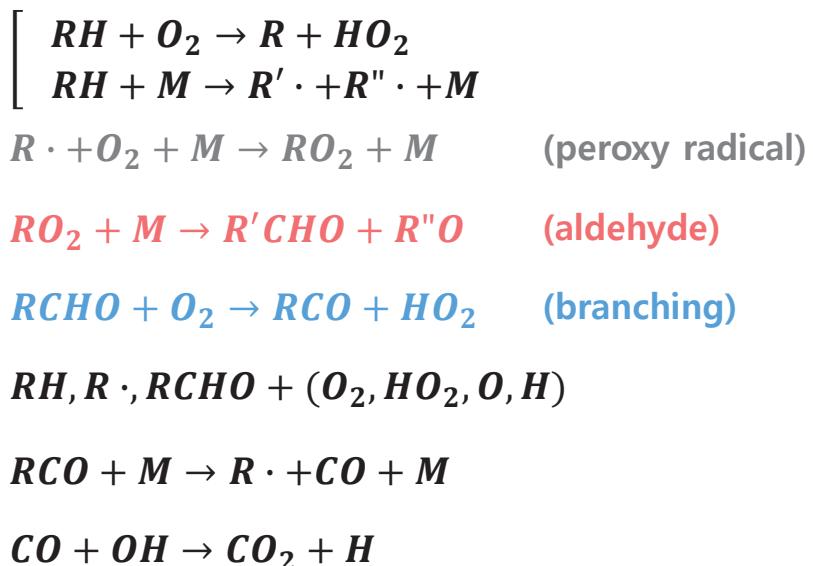


Fig. Schematic mechanism of the radical pyrolysis of large aliphatic hydrocarbons



Combustion Laboratory POSTECH

# Hierarchical structure of HC Oxidation

- Different mechanisms for lean and *rich* mixtures
  - High HC's
- Kinetics of C<sub>3</sub> and higher HC's is different from that of CH<sub>4</sub> due to faster kinetics of C<sub>2</sub>H<sub>5</sub>.

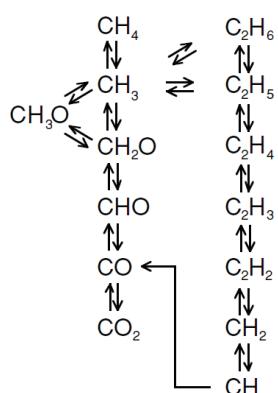


Fig. Schematic mechanism of C1 and C2 hydrocarbon oxidation

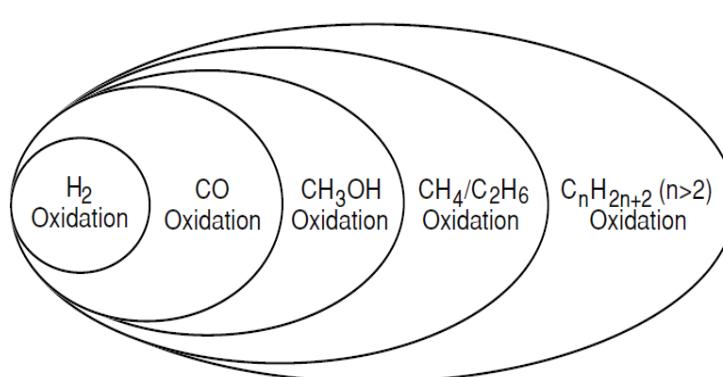


Fig. Hierarchy in the reaction mechanism describing aliphatic hydrocarbon combustion

Combustion Laboratory POSTECH

# Physical and Life Sciences Directorate

[Contact Us](#) | [S&T](#) | [Site Map](#)

Search

GO

Science/Technology

About PLS

Jobs and Internships

News and Events

[Overview](#) | [Physics](#) | **Chemistry** | [Materials](#) | [Earth](#) | [Life Sciences](#)

## Science and Technology

[Print View](#)

- Hydrogen
- Ethanol
- Butanol isomers
- Dimethyl Ether
- CH<sub>4</sub>, C<sub>2</sub>H<sub>4</sub>, C<sub>2</sub>H<sub>6</sub>, C<sub>3</sub>H<sub>8</sub>, and nC<sub>4</sub>H<sub>10</sub>
- CH<sub>4</sub>, C<sub>2</sub>H<sub>4</sub>, C<sub>2</sub>H<sub>6</sub>, C<sub>3</sub>H<sub>6</sub>, C<sub>3</sub>H<sub>8</sub>, and NO<sub>x</sub>
- C<sub>8</sub>-C<sub>16</sub> n-Alkanes
- Cyclohexane
- Methylcyclohexane
- Methyl Butanoate and Methyl Formate
- Methyl Decanoate
- Methyl Decenoates
- Biodiesel Surrogates
- Dimethyl Carbonate
- Heptane, Detailed Mechanism
- Heptane, Reduced Mechanism
- iso-Octane
- Gasoline Surrogate
- 2-Methyl Alkanes
- Primary Reference Fuels: iso-Octane / n-Heptane Mixtures
- 2,2,4,4,6,8,8-Heptamethylnonane
- Organophosphorus Compounds under Incineration Conditions
- Organophosphorus Compounds in Propane Flames
- Organophosphorus Compounds Effect on Flame Speeds

[home](#) > [science and technology](#) > [chemistry](#) > [combustion](#)

### Combustion Chemistry

[Go Directly to Mechanisms...](#)

The central feature of the Combustion Chemistry project at LLNL is our development, validation, and application of detailed chemical kinetic reaction mechanisms for the combustion of hydrocarbon and other types of chemical fuels. For the past 30 years, our group has built hydrocarbon mechanism for fuels from hydrogen and methane through much larger fuels including heptanes and octanes. Other classes of fuels for which models have been developed include flame suppressants such as halons and organophosphates, and air pollutants such as soot and oxides of nitrogen and sulfur.

Reaction mechanisms have been tested and validated extensively through comparisons between computed results and measured data from laboratory experiments (e.g., shock tubes, laminar flames, rapid compression machines, flow reactors, stirred reactors) and from practical systems (e.g., diesel engines, spark-ignition engines, homogeneous charge, compression ignition (HCCI) engines). We have used these kinetic models to examine a wide range of combustion systems.

### Lawrence Livermore National Laboratory

[https://www-pls.llnl.gov/?url=science\\_and\\_technology-chemistry-combustion](https://www-pls.llnl.gov/?url=science_and_technology-chemistry-combustion)

Chemical kinetic modeling is an essential tool in understanding and predicting performance and emissions from these three very different types of internal combustion engines.

[\[top\]](#)

### Science in Support of National Objectives

## Turbulent Nonpremixed Combustion Models – Phenomenological

### EDM (Eddy Dissipation Model) (Magnussen et al.)

$$\left. \begin{aligned} R_f &= A \cdot \bar{Y}_f (\varepsilon/k) \\ R_f &= A(\bar{Y}_{O_2}/r_f)(\varepsilon/k) \\ R_f &= A \cdot B \{\bar{Y}_P/(1+r_f)\}(\varepsilon/k) \end{aligned} \right\} \begin{array}{l} \text{Minimum } R_f \\ \longrightarrow \end{array} \begin{array}{l} \text{Local mean rate} \\ \text{of combustion} \end{array}$$

$A, B$  : 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

$$R_{i,r} = v'_{i,r} M_{w,i} A \frac{\rho \varepsilon}{k} \min\left(\frac{Y_R}{v'_{R,r} M_{w,R}}\right)$$

$$R_{i,r} = v'_{i,r} M_{w,i} AB \frac{\rho \varepsilon}{k} \frac{\sum_p Y_p}{\sum_j v'_{j,r} M_{w,j}}$$

#### Arrhenius

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

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

for species  $i$  and reaction  $r$

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

# Turbulent Nonpremixed Combustion Models – Phenomenological

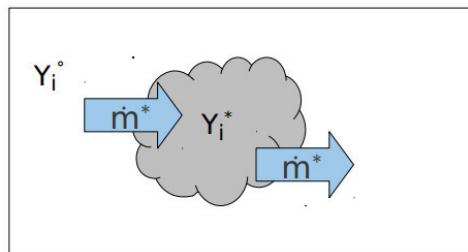
## EDC (Eddy Dissipation Concept) (Magnussen 1981)

- It is an extension of EDM to include turbulence-chemistry interaction with detailed chemical mechanism. Only when fast chemistry is not valid.
- Chemical reaction and molecular mixing/dissipation occur in intermittent regions in space – vortex tubes
- It is assumed that reaction occurs in small turbulent structures called fine scales.
- The length fraction of the fine scales is modeled as

$$\xi^* = C_\xi \left( \frac{\nu \epsilon}{k^2} \right)^{1/4}$$

$C_\xi$  : constant = 2.1377  
 $\nu$  : kinematic viscosity

$(\xi^*)^3$  : Volume fraction of the fine scales



Fraction of the flow occupied by fine structures:

$$\gamma^* = (\xi^*)^3$$

Relation of mean, fine structure and surrounding state:

$$\bar{Y}_i = \gamma^* Y^* + (1 - \gamma^*) Y^o$$

Combustion Laboratory Pohang University of Science and Technology POSTECH

# Turbulent Nonpremixed Combustion Models – Phenomenological

## EDC (Eddy Dissipation Concept) (Magnussen 1981)

- Species are assumed to react in the fine structures over a time scale according to given chemical mechanisms.

$$\tau^* = C_\tau \left( \frac{\nu}{\epsilon} \right)^{1/2}$$

$C_\tau$  : time scale constant = 0.4082

- The source term for the mean species  $i$  is modeled as

$$\bar{R}_i = \frac{\bar{\rho}(\xi^*)^2}{\tau^* [1 - (\xi^*)^3]} (-\bar{Y}_i + Y_i^*)$$

$$\bar{Y}_i = (\xi^*)^3 Y_i^* + (1 - (\xi^*)^3) Y_i^o$$

$Y_i^*$  : fine scale species mass fraction after reaction over the time  $\tau^*$

# Turbulent Nonpremixed Combustion Models – Phenomenological

## PaSR (Partially Stirred Reactor) (Golovitchev 2001)

- A computational cell is divided into the reacting and the non-reacting part.

$$Y_i = (1 - \kappa) Y_i^N + \kappa Y_i^R, \quad \kappa = \frac{\tau_{ch}}{\tau_{ch} + \tau_m} \quad \kappa : \text{molecularly mixed fraction of the cell volume}$$

- Micro mixing time

$$\tau_m = \sqrt{\frac{k}{\varepsilon} \left( \frac{\nu}{\varepsilon} \right)^{\frac{1}{2}}} = \left( \frac{1}{Re_t} \right)^{\frac{1}{4}} \frac{k}{\varepsilon} = C_{mix} \sqrt{\frac{\mu_{eff}}{\rho \varepsilon}} \quad (\text{OF})$$

$$C_{mix} = \sqrt{\frac{1}{1 + C_\mu Re_t}}$$

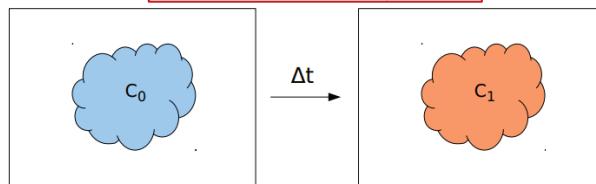
$C_{mix} = 1$  laminar  
 $C_{mix} = 0$   $Re_t = \infty$

- Reaction time

$$\frac{1}{\tau_{ch}} = \max \left( -\frac{R_{fuel}}{\rho Y_{fuel}}, -\frac{R_{O2}}{\rho Y_{O2}} \right)$$

- The reaction zone is treated as a perfectly stirred reactor.
- The reaction rate is determined in a transient manner as

$$\frac{dY_i}{dt} = \dot{\omega}_i + \frac{Y_i - Y_i^*}{\tau_m}$$



Combustion Laboratory POSTECH

# Turbulent Nonpremixed Combustion Models

## Infinitely Fast Chemistry

### One-step irreversible reaction (SCRS)

- Rich condition ( $\xi > \xi_{st}$ )

$$Y_F(\xi) = \xi Y_F^0 + (\xi - 1) \frac{Y_O^0}{S} = Y_F^0 \frac{\xi - \xi_{st}}{1 - \xi_{st}}$$

$$Y_O(\xi) = 0.$$

$$T(\xi) = \xi T_F^0 + (1 - \xi) T_O^0 + \frac{Q Y_F^0}{C_p} \xi_{st} \frac{1 - \xi}{1 - \xi_{st}}$$

- Lean condition ( $\xi < \xi_{st}$ )

$$Y_F(\xi) = 0.$$

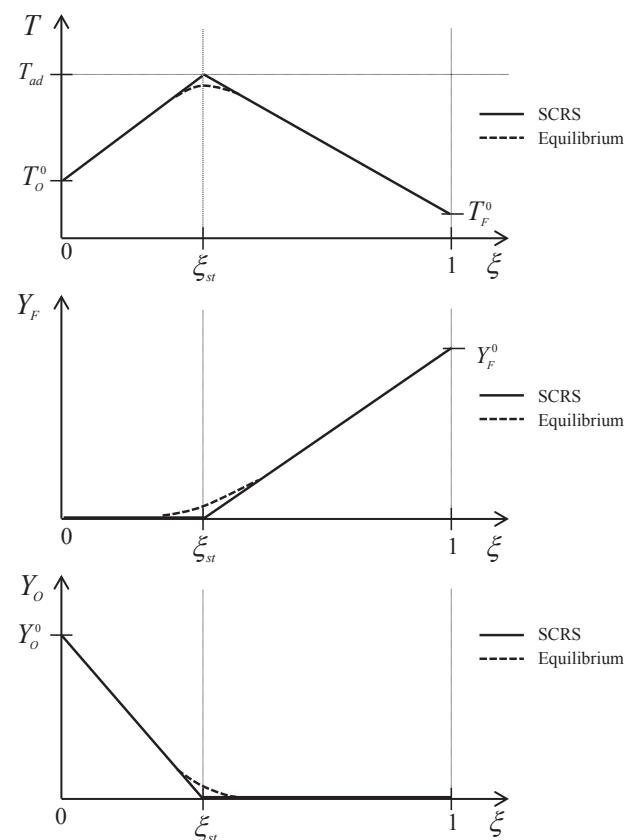
$$Y_O(\xi) = Y_O^0 \left( 1 - \frac{\xi}{\xi_{st}} \right)$$

$$T(\xi) = \xi T_F^0 + (1 - \xi) T_O^0 + \frac{Q Y_F^0}{C_p} \xi$$

- Stoichiometric condition ( $\xi = \xi_{st}$ )

$$s = \frac{\nu_O W_O}{\nu_F W_F} \quad (\nu_F F + \nu_O O \rightleftharpoons \nu_P P)$$

$$\xi_{st} = \frac{1}{1 + \frac{s Y_F^0}{Y_O^0}}$$



Combustion Laboratory POSTECH

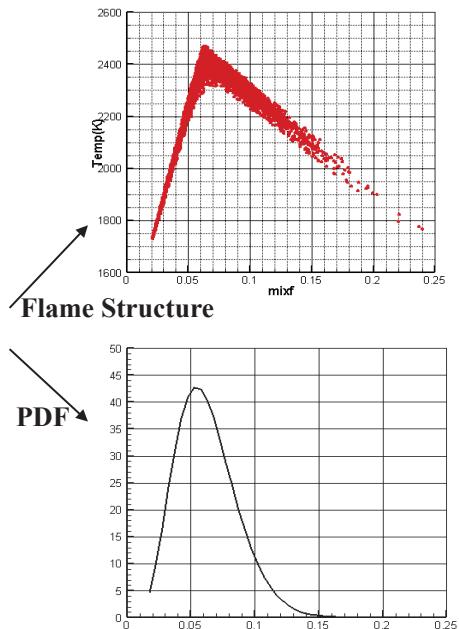
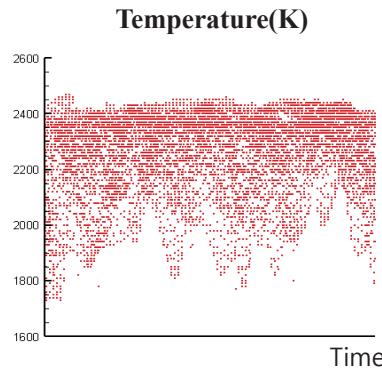
# Turbulent Nonpremixed Combustion Models - Systematic

## Conditional Averaging Concept

- For a fluctuating variable  $\Phi$ , there exists a fluctuating variable  $x$  which is closely related with fluctuation of  $\Phi$

$$\sum_{i=1}^n P_i = 1 , \quad \sum_{i=1}^n P_i \langle \phi | X_i \rangle = \bar{\phi}$$

| X            | Probability | $\langle \phi   X \rangle$   |
|--------------|-------------|------------------------------|
| $\Delta X_1$ | $P_1$       | $\langle \phi   X_1 \rangle$ |
| $\Delta X_2$ | $P_2$       | $\langle \phi   X_2 \rangle$ |
| ⋮            | ⋮           | ⋮                            |
| $\Delta X_n$ | $P_n$       | $\langle \phi   X_n \rangle$ |



# Turbulent Nonpremixed Combustion Models - Systematic

## CMC vs. LFM

### ▪ CMC

Based on rigorously derived transport eq. for conditional average variables  
Modeling of conditional average velocity and scalar dissipation terms, etc.

### ▪ LFM

Based on ad-hoc assumption of a flamelet structure  
Lagrangian handling of a transient effect (RIF)  
Applicable range?

Major uncertainties in engineering combustion problems are in the pdf's due to turbulent mixing, rather than in the conditional flamelet structures.

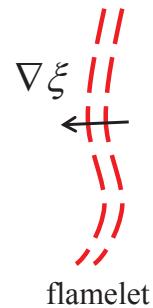
# Turbulent Nonpremixed Combustion Models - Systematic

## Laminar Flamelet Model

- Crocco Transformation :  $(\underline{x}, t) \rightarrow (\xi, t)$

$$\rho \frac{\partial Y_k}{\partial t} + \rho v \cdot \nabla Y_k = \nabla \cdot (\rho D \nabla Y_k) + \rho \dot{\omega}_k$$

$$\rightarrow \rho \frac{\partial Y_k}{\partial t} = \rho D (\nabla \xi)^2 \frac{\partial^2 Y_k}{\partial \xi^2} + \rho \dot{\omega}_k$$



- Steady Laminar Flamelet

$$D(\nabla \xi)^2 \frac{\partial^2 Y_k}{\partial \xi^2} + \dot{\omega}_k = 0$$

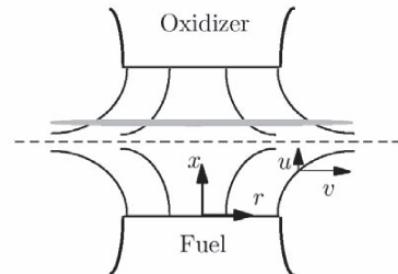
- Counterflow diffusion flame

$$\xi = \frac{1}{2} \left( 1 - \text{erf} \left( \frac{\chi}{\sqrt{2D/a}} \right) \right), \quad u = ax, \quad v = -ay$$

$$\chi = 2D(\nabla \xi)^2 \quad (\text{SDR : Scalar Dissipation Rate})$$

$$\frac{\chi}{F(\xi)} = \frac{\chi_{st}}{F(\xi_{st})}, \quad F(\xi) = \exp \left[ -2 \left\{ \text{erf}^{-1}(2\xi - 1) \right\}^2 \right]$$

$$\chi_{st} = \frac{a}{\pi} \exp \left[ -2 \left\{ \text{erfc}^{-1}(2\xi_{st}) \right\}^2 \right]$$



Lutz et al., 1997

Combustion Laboratory POSTECH

# Turbulent Nonpremixed Combustion Models - Systematic

## Laminar Flamelet Model

$$\bar{\rho} \tilde{Y}_k = \int_0^\infty \int_0^1 \rho Y_k(\xi, \chi_{st}) P(\xi, \chi_{st}) d\xi d\chi_{st}$$

$\rho Y_k(\xi, \chi_{st})$  from the steady laminar flamelet library

Assume

$$P(\xi, \chi_{st}) = P(\xi) P(\chi_{st})$$

$$P(\chi_{st}) = \delta(\chi_{st} - \tilde{\chi}_{st}) \quad (1)$$

$$P(\chi_{st}) = \frac{1}{\chi_{st} \sigma \sqrt{2\pi}} \exp \left( -\frac{(\ln \chi_{st} - \mu)^2}{2\sigma^2} \right) \quad (2)$$

$$\sigma \sim O(1)$$

From Eq (1)

$$\bar{\rho} \tilde{Y}_k = \int_0^1 \rho Y_k(\xi, \tilde{\chi}_{st}) P(\xi) d\xi$$

$P(\xi)$  : Assumed  $\beta$ -pdf in terms of local  $\xi$  and  $\tilde{\xi}^{n/2}$

$Y_k(\xi, \tilde{\chi}_{st})$  : from laminar flamelet library

Combustion Laboratory POSTECH

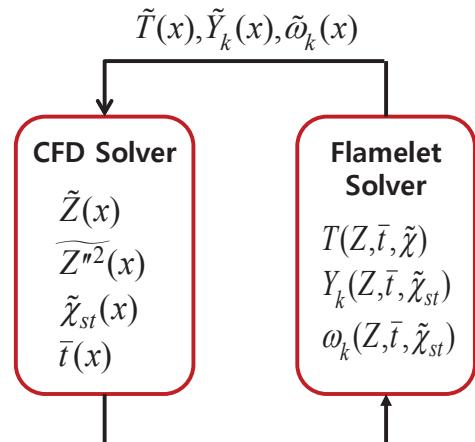
# Turbulent Nonpremixed Combustion Models - Systematic

## Representative Interactive Flamelet (RIF)

- The unsteady term in the flamelet equation must be retained if the SDR change rapidly compared to the flame chemical time scale.

$$\bar{t} = \int_0^x \frac{1}{\tilde{u}(x)} dx : \text{residence time of a Lagrangian flamelet}$$

$$\tilde{Y}_k = \int_0^1 Y_k(\xi, \bar{t}, \tilde{x}_{st}) P(\xi) d\xi$$



▲ Iterative loop in the RIF model

## Eulerian Particle Flamelet Model (EPFM)

$$\frac{\partial}{\partial t} (\bar{\rho} \tilde{I}_l) + \nabla \cdot (\bar{\rho} \tilde{v} \tilde{I}_l) - \nabla \cdot \left( \bar{\rho} \frac{V_t}{Sc_t} \tilde{I}_l \right) = 0$$

$\tilde{I}_l$  : mass-weighted fraction of the  $l$ -th flamelet following unsteady evolution to determine the probability of finding the  $l$ -th flamelet at each location

# Turbulent Nonpremixed Combustion Models - Systematic

## Conditional Moment Closure (CMC)

- Conditional mean mass fraction

$$Q_i(\eta, \mathbf{x}, t) = \langle Y_i(\mathbf{x}, t) | \xi(\mathbf{x}, t) = \eta \rangle \quad \text{where } \eta : \text{sampling variable of mixture fraction, } \xi$$

- Conditional mass fraction and enthalpy equation

$$\underbrace{\frac{\partial Q_i}{\partial t}}_{\text{time rate of change}} + \underbrace{\langle u | \eta \rangle \cdot \nabla Q_i}_{\text{spatial transport}} + \underbrace{\frac{\nabla \cdot (\langle u'' Y_i'' | \eta \rangle P(\eta) \rho_\eta)}{P(\eta) \rho_\eta}}_{\text{scalar dissipation rate}} = \underbrace{\langle N | \eta \rangle \frac{\partial^2 Q_i}{\partial \eta^2}}_{\text{reaction rate}}$$

$$\underbrace{\frac{\partial Q_h}{\partial t}}_{\text{time rate of change}} + \underbrace{\langle u | \eta \rangle \cdot \nabla Q_h}_{\text{spatial transport}} + \underbrace{\frac{\nabla \cdot (\langle u'' h'' | \eta \rangle P(\eta) \rho_\eta)}{P(\eta) \rho_\eta}}_{\text{scalar dissipation rate}} = \underbrace{\langle N | \eta \rangle \frac{\partial^2 Q_h}{\partial \eta^2}}_{\text{pressure change}} + \underbrace{\left\langle \frac{1}{\rho} \frac{\partial p}{\partial t} \right\rangle | \eta \rangle}_{\text{pressure change}}$$

# Turbulent Nonpremixed Combustion Models - Systematic

## Conditional Moment Closure (CMC)

- Beta function PDF

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

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

- Favre mean mass fraction of the i-th species

$$\bar{Y}_i(\mathbf{x}, t) = \int_0^1 \underbrace{Q_i(\eta, \mathbf{x}, t)}_{\text{conditional flame structure}} \underbrace{\tilde{P}(\eta)}_{\text{PDF}} d\eta$$

- Mean mixture fraction and its variance

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

$$\frac{\partial(\bar{\rho}\tilde{\xi}''^2)}{\partial t} + \nabla \cdot (\bar{\rho}\tilde{u}\tilde{\xi}''^2) = \nabla \cdot \left[ \frac{\mu_t}{Sc_{\tilde{\xi}''^2}} \tilde{\xi}''^2 \right] + \frac{2\mu_t}{Sc_{\tilde{\xi}''^2}} (\nabla \tilde{\xi})^2 - 2\bar{\rho}\tilde{N} + 2C\bar{\rho}\tilde{\xi}''\tilde{s}_{\tilde{\xi}}(1-\tilde{\xi})$$

where  $\mu_t$  : Turbulent viscosity

$Sc_{\tilde{\xi}}$  : Schmidt number for mixture fraction

$Sc_{\tilde{\xi}''^2}$  : Schmidt number for mixture fraction variance

$\tilde{s}_{\tilde{\xi}}$  : Evaporation source term for mixture fraction

C : Correlation coefficient (= 0.5)

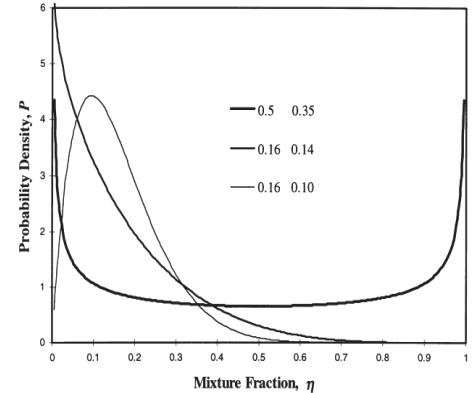


Figure. Beta PDFs with different mean mixture fractions and its variances

Combustion Laboratory POSTECH

# Turbulent Nonpremixed Combustion Models - Systematic

## Conditional Moment Closure (CMC)

- Lagrangian CMC

Conditional species mass fraction,  $Q_{i,j}$ , of the  $i$ -th species in the  $j$ -th flame group

$$\frac{DQ_{i,j}}{Dt} = \langle N | \eta \rangle_j \frac{\partial^2 Q_{i,j}}{\partial \eta^2} + \langle w_i | \eta \rangle_j$$

Conditional enthalpy,  $Q_{h,j}$ , for the  $j$ -th flame group

$$\frac{DQ_{h,j}}{Dt} = \langle N | \eta \rangle_j \frac{\partial^2 Q_{h,j}}{\partial \eta^2} + \left\langle \frac{1}{\rho} \frac{\partial p}{\partial t} | \eta \right\rangle$$

- Eulerian CMC

Conditional mass fraction,  $Q_i$ , of the  $i$ -th species

$$\frac{\partial Q_i}{\partial t} + \langle u_i | \eta \rangle \cdot \nabla Q_i = \langle N | \eta \rangle \frac{\partial^2 Q_i}{\partial \eta^2} + \langle w_i | \eta \rangle$$

Conditional enthalpy,  $Q_h$

$$\frac{\partial Q_h}{\partial t} + \langle u_i | \eta \rangle \cdot \nabla Q_i = \langle N | \eta \rangle \frac{\partial^2 Q_h}{\partial \eta^2} + \left\langle \frac{1}{\rho} \frac{\partial p}{\partial t} | \eta \right\rangle$$

- Modeling

Conditional reaction rate (1<sup>st</sup> order closure)

$$\langle w_i | \eta \rangle \approx w_i(Q_i, \dots, Q_n, Q_h)$$

Conditional scalar dissipation rate :  $\langle N | \eta \rangle = \frac{\widetilde{N} \exp(-2[\text{erf}^{-1}(2\eta-1)]^2)}{\int_0^1 \exp(-2[\text{erf}^{-1}(2\eta-1)]^2) \tilde{P}(\eta) d\eta}$  where  $\widetilde{N} \equiv D(\nabla \tilde{\xi})^2 = \frac{\tilde{\varepsilon}}{\tilde{k}} \tilde{\xi}''^2$  (amplitude mapping closure, AMC)

Conditional velocity :

$$\langle u_i | \eta \rangle = \tilde{u}_i + \frac{\tilde{u}_i \tilde{\xi}''}{\tilde{\xi}''^2} (\eta - \tilde{\xi}) \quad \text{where } \tilde{u}_i \tilde{\xi}'' = -D_i \frac{\partial \tilde{\xi}}{\partial x_i}$$

Combustion Laboratory POSTECH

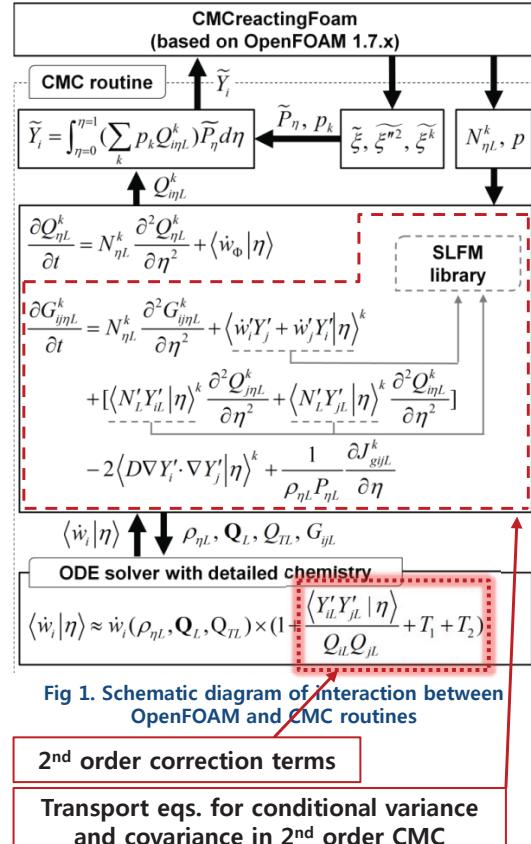
# Turbulent Nonpremixed Combustion Models - Systematic

## Conditional Moment Closure (CMC) – Numerical Algorithms

### 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

(2<sup>nd</sup> 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



Combustion Laboratory POSTECH

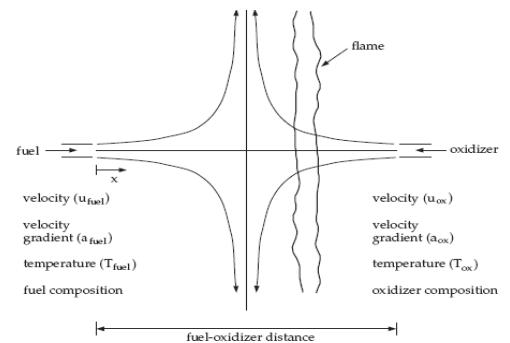
# Turbulent Nonpremixed Combustion Models - Systematic

## Steady Laminar Flamelet Model ( SLFM )

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

$$\text{Assumed beta-function PDF} \quad \tilde{P}(\eta) = \frac{\zeta^{\alpha-1} (1-\zeta)^{\beta-1}}{\Gamma(\alpha)\Gamma(\beta)} \Gamma(\alpha+\beta)$$

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



### Mixture fraction

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

### Mixture fraction variance

$$\frac{\partial(\bar{\rho}\tilde{\xi}^{\tilde{\eta}^2})}{\partial t} + \nabla \cdot (\bar{\rho}\tilde{u}\tilde{\xi}^{\tilde{\eta}^2}) = \nabla \cdot \left[ \frac{\mu_t}{Sc_{\tilde{\xi}^{\tilde{\eta}^2}}} \nabla \tilde{\xi}^{\tilde{\eta}^2} \right] + \frac{2\mu_t}{Sc_{\tilde{\xi}^{\tilde{\eta}^2}}} (\nabla \tilde{\xi})^2 - \bar{\rho}\tilde{\chi}$$

### Mean Reaction rate

$$\tilde{\phi}_i(x, t) = \int_0^1 \phi_i(\eta, x, t) \tilde{P}(\eta, x, t) d\eta$$

# Turbulent Premixed Combustion Models

## EBU (Eddy Break Up)

Turbulent mixing controls the mean reaction rate – No effects of chemical kinetics

$$\overline{\dot{\omega}_c} = C_{EBU} \bar{\rho} \frac{\tilde{\varepsilon}}{k} \tilde{c}(1-\tilde{c})$$

$$\bar{c} = (1 + \tau) \tilde{c} / (1 + \tau \tilde{c}), \quad \tau = \rho_u / \rho_b - 1$$

$$\tau_t = \frac{k}{\tilde{\varepsilon}} \quad C_{EBU} : \text{Tuning constant}$$

## BML (Bray Moss Libby) [1]

Bimodal pdf for burned and unburned gas

$$\begin{aligned}\overline{\dot{\omega}_c} &= \frac{1}{2c_m - 1} \left( 2\rho D \frac{\partial c}{\partial x_i} \frac{\partial \bar{c}}{\partial x_i} \right) = \frac{\bar{\rho} \tilde{\chi}_c}{2c_m - 1} \\ \overline{\rho \chi_c} &= \bar{\rho} \tilde{c}^{\prime\prime 2} / \tau_t \quad \bar{\rho} \tilde{c}^{\prime\prime 2} = \bar{\rho} \tilde{c} (1 - \tilde{c}) \\ c_m &= \frac{\overline{c \dot{\omega}_c}}{\overline{\dot{\omega}_c}} = \frac{\int_0^1 c \dot{\omega}_c f(c) dc}{\int_0^1 \dot{\omega}_c f(c) dc}\end{aligned}$$

Systematic derivation of EBU

- Same form as EBU

$$\overline{\dot{\omega}_c} = \frac{1}{2c_m - 1} \bar{\rho} \frac{\tilde{k}}{\tilde{\varepsilon}} \tilde{c} (1 - \tilde{c})$$

$$C_{EBU} = \frac{1}{2c_m - 1} \sim O(1)$$

[1] P.A.Libby and F.A. Williams, "Turbulent Reacting Flows", 1994

Combustion Laboratory  POSTECH

# Turbulent Premixed Combustion Models

## FSD (Flame Surface Density)

$$\overline{\dot{\omega}_c} = \rho_0 \langle S_c \rangle_s \Sigma$$

$$\langle S_c \rangle_s = I_0 S_L^0$$

$I_0$  : stretch factor

$S_L^0$  : unstretched laminar flame speed

### Algebraic expressions

#### Measurement

$$\Sigma = n_y / \sigma_y$$

$$\Sigma = \frac{g}{\sigma_y} \frac{\bar{c}(1 - \bar{c})}{L_y} = \frac{g}{\sigma_y} \frac{1 + \tau}{(1 + \tau \tilde{c})^2} \frac{\tilde{c}(1 - \tilde{c})}{L_y}, \quad L_y = C_l l_t \left( S_L^0 / u' \right)^n$$

$$\sigma_y \approx 0.5$$

$n_y$  : number of flamelet crossings per unit length along constant  $\bar{c}$  direction

$\sigma_y$  : flamelet orientation factor, mean cosine angle with iso- $\bar{c}$  surface

$g$  : model constant of order unity  $L_y$  : wrinkling length scale of the flame front

$l_t$  : integral length scale,  $n \sim O(1)$

#### Fractal theories

$$\Sigma = \frac{1}{L_{outer}} \left( \frac{L_{outer}}{L_{inner}} \right)^{D-2} \quad (2 < D < 3)$$

$L_{outer}$  : outer cut-off scale,  $f(l_t)$

$L_{inner}$  : inner cut-off scale,  $f(\delta_f, Ka)$

Combustion Laboratory  POSTECH

# Turbulent Premixed Combustion Models

## FSD (Flame Surface Density)

- Balance equation for the flame surface density

$$\frac{\partial \Sigma}{\partial t} + \frac{\partial u_i \Sigma}{\partial x_i} = \frac{\partial}{\partial x_i} \left( \frac{v_t}{\sigma_c} \frac{\partial \Sigma}{\partial x_i} \right) + \kappa_m \Sigma + \kappa_t \Sigma - D$$

$$\left\langle \left( \delta_{ij} - n_i n_j \right) \frac{\partial u_i}{\partial x_j} \right\rangle_s = \underbrace{\left( \delta_{ij} - \langle n_i n_j \rangle_s \right) \frac{\partial \tilde{u}_i}{\partial x_j}}_{\kappa_m} + \underbrace{\left\langle \left( \delta_{ij} - n_i n_j \right) \frac{\partial u''_i}{\partial x_j} \right\rangle_s}_{\kappa_t}$$

$\kappa_m$  : strain rate acting of the flame surface and induced by the mean flow field

$\kappa_t$  : strain rate acting of the flame surface and induced by the turbulent motions

$D$  : consumption term

Simple closure :  $\kappa_m + \kappa_t = \alpha \frac{\epsilon}{k}$

$$D = \beta_0 \langle S_c \rangle_s \Sigma^2 / (1 - \tilde{c})$$

# Turbulent Premixed Combustion Models

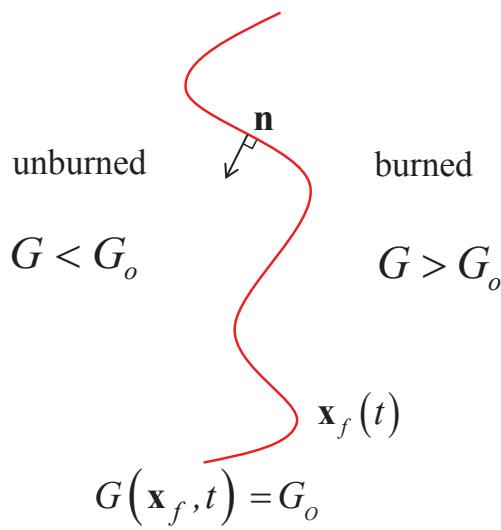
## FSD (Flame Surface Density) [1]

| MODEL                                                    | $\kappa_m \Sigma$                                                            | $\kappa_t \Sigma$                                                                                                                                                     | $D$                                                                                                                                                 |
|----------------------------------------------------------|------------------------------------------------------------------------------|-----------------------------------------------------------------------------------------------------------------------------------------------------------------------|-----------------------------------------------------------------------------------------------------------------------------------------------------|
| CPB<br>Cant et al. <sup>85</sup>                         | $A_k \frac{\partial \tilde{u}_k}{\partial x_i} \Sigma$                       | $\alpha_0 C_A \sqrt{\frac{\epsilon}{V}} \Sigma$                                                                                                                       | $\beta_0 \langle s_c \rangle_s \frac{2 + e^{-\Theta}}{3(1-\Theta)} \Sigma^2$<br>$R = \frac{(1-\Theta)\epsilon}{\Sigma \langle s_c \rangle_s k}$     |
| CFM1<br>CFM2-a<br>CFM2-b<br>Duclos et al. <sup>151</sup> | $A_k \frac{\partial \tilde{u}_k}{\partial x_i} \Sigma$                       | $\alpha_0 \frac{\epsilon}{k} \Sigma$                                                                                                                                  | $\beta_0 \frac{\langle s_c \rangle_s + C\sqrt{k}}{(1-\Theta)} \Sigma^2$                                                                             |
|                                                          | $A_k \frac{\partial \tilde{u}_k}{\partial x_i} \Sigma$                       | $\alpha_0 \Gamma \kappa \frac{\epsilon}{k} \Sigma$                                                                                                                    | $\beta_0 \frac{\langle s_c \rangle_s + C\sqrt{k}}{(1-\Theta)} \Sigma^2$                                                                             |
|                                                          | $A_k \frac{\partial \tilde{u}_k}{\partial x_i} \Sigma$                       | $\alpha_0 \Gamma \kappa \frac{\epsilon}{k} \Sigma$                                                                                                                    | $\beta_0 \frac{\langle s_c \rangle_s + C\sqrt{k}}{\Theta(1-\Theta)} \Sigma^2$                                                                       |
| CFM3<br>Veynante et al. <sup>532</sup>                   | $A_k \frac{\partial \tilde{u}_k}{\partial x_i} \Sigma$                       | $\alpha_0 \Gamma \kappa \frac{\epsilon}{k} \Sigma$                                                                                                                    | $\beta_0 \langle s_c \rangle_s \frac{\Theta - \tilde{\Theta}}{\Theta(1-\Theta)} \Sigma^2$                                                           |
| MB<br>Mantel and Borghi <sup>333</sup>                   | $E \frac{\tilde{u}_i'' \frac{\partial \tilde{u}_k}{\partial x_i}}{k} \Sigma$ | $\alpha_0 \sqrt{Re_t} \frac{\epsilon}{k} \Sigma$<br>$+ \frac{F}{\langle s_c \rangle_s} \frac{\epsilon}{k} \frac{\tilde{u}_i'' \partial \tilde{\Theta}}{\partial x_i}$ | $\frac{\beta_0 \langle s_c \rangle_s \sqrt{Re_t} \Sigma^2}{\Theta(1-\Theta) \left( 1 + c \frac{\langle s_c \rangle_s}{\sqrt{k}} \right)^{2\gamma}}$ |
| CD<br>Cheng and Diringer <sup>95</sup>                   |                                                                              | $\alpha_0 \lambda \frac{\epsilon}{k} \Sigma \text{ if } \kappa_t \leq \alpha_0 K \frac{\langle s_c \rangle_s}{\delta_L}$                                              | $\beta_0 \frac{\langle s_c \rangle_s}{1-\Theta} \Sigma^2$                                                                                           |
| CH1<br>CH2<br>Choi and Huh <sup>98</sup>                 |                                                                              | $\alpha_0 \sqrt{\frac{\epsilon}{15\nu}} \Sigma$<br>$\alpha_0 \frac{u'}{l_e} \Sigma$                                                                                   | $\beta_0 \frac{\langle s_c \rangle_s}{\Theta(1-\Theta)} \Sigma^2$<br>$\beta_0 \frac{\langle s_c \rangle_s}{\Theta(1-\Theta)} \Sigma^2$              |

[1] T. Poinsot and D. Veynante,  
“Theoretical and Numerical Combustion”,  
Philadelphia: Edwards, 2005.

# Turbulent Premixed Combustion Models

## G – equation for infinitesimally thin flamelets



$$\mathbf{n} = \frac{-\nabla G}{|\nabla G|}$$

$$\frac{d\mathbf{x}_f}{dt} = \mathbf{v}_f + \mathbf{n} S_L$$

$$\rightarrow \boxed{\frac{\partial G}{\partial t} + \mathbf{v}_f \cdot \nabla G = S_L |\nabla G|}$$

$$S_L = S_L^0 - S_L^0 \mathcal{L} \kappa - \mathcal{L} S$$

$\kappa$  : curvature ( $= -\nabla \cdot \mathbf{n}$ )

$S$  : rate of strain ( $= -\mathbf{n} \cdot \nabla \mathbf{v} \cdot \mathbf{n}$ )

$\mathcal{L}$  : Markstein length

# Turbulent Premixed Combustion Models

## G – equation for infinitesimally thin flamelets

$$G(\underline{x}, t) = G_0$$

$P(G; \underline{x}, t)$  : pdf for  $G$  at  $(\underline{x}, t)$

$$\overline{G} = \int G P dG,$$

$$\overline{G'^2} = \int (G - \overline{G})^2 P dG$$

$$P(G_0; \underline{x}, t) = \int \delta(G - G_0) P dG = P(\underline{x}, t) \quad \text{Probability of finding the flame at } (\underline{x}, t)$$

Assume a 1-D statistical steadiness flame brush

$$P(\underline{x}, t) = P(\underline{x}), \quad \int P(x) dx = 1 \quad \longrightarrow \quad x_f = \int x P(x) dx : \text{mean flame position}$$

Define  $G(\underline{x}, t)$  such that

$$\overline{G}(x) - G_0 = x - x_f$$

$$\text{Set } G = G_0, \quad G' = -(x - x_f)$$

$$\overline{(G'^2)}_0 = \overline{(x - x_f)^2}$$

$G$  conditioned at  $G = G_0$  corresponds to the spatial fluctuation of the flame front positon

# Turbulent Premixed Combustion Models

## G – equation for infinitesimally thin flamelets

$$G = \tilde{G} + G'', \quad \nu = \tilde{\nu} + \nu''$$

$$\bar{\rho} \frac{\partial \tilde{G}}{\partial t} + \bar{\rho} \tilde{\nu} \cdot \nabla \tilde{G} = (\bar{\rho} S_T^0) |\nabla \tilde{G}| - \bar{\rho} D_t \tilde{\kappa} |\nabla \tilde{G}| \quad : \text{mean flame position } (\tilde{G} = G_0)$$

$$\bar{\rho} \frac{\partial \widetilde{G''^2}}{\partial t} + \bar{\rho} \tilde{\nu} \cdot \nabla \widetilde{G''^2} = \nabla \cdot (\bar{\rho} D_t \nabla \widetilde{G''^2}) + 2\bar{\rho} D_t (\nabla \tilde{G})^2 - c_s \bar{\rho} \frac{\tilde{\varepsilon}}{\tilde{k}} \widetilde{G''^2} \quad : \text{flame brush thickness}$$

Closure for  $S_T^\theta$  (or  $\sigma$ ) required

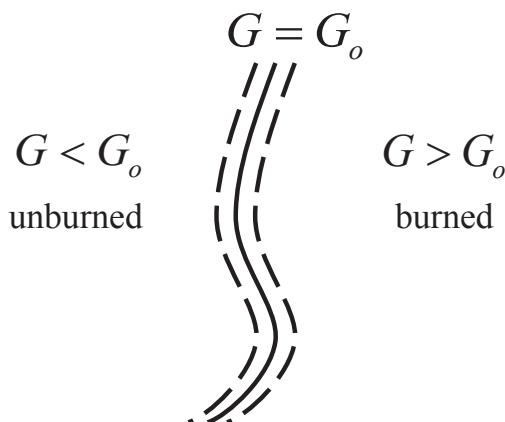
$$\bar{\rho} S_T^0 |\nabla \tilde{G}| = \rho \sigma S_L^0 |\nabla \tilde{G}|$$

$$-\nabla \left( \bar{\rho} \widetilde{v'' G''^2} \right) = \nabla \left( \bar{\rho} D_t \nabla \widetilde{G''^2} \right) \quad -\widetilde{v'' G''} \cdot \nabla \tilde{G} = D_t (\nabla \tilde{G})^2 \quad \text{gradient transport assumption}$$

$$-\nabla \cdot (\bar{\rho} \widetilde{v'' G''}) = -\bar{\rho} D_t \tilde{\kappa} |\nabla \tilde{G}|$$

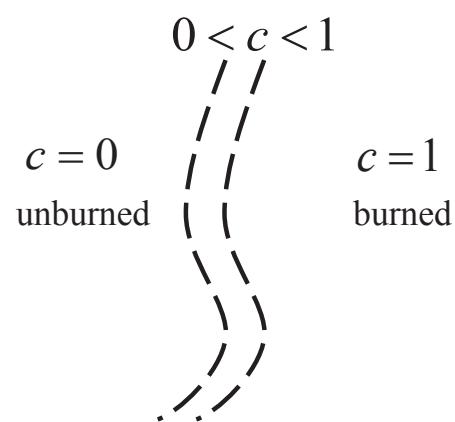
Combustion Laboratory  POSTECH  
POHANG UNIVERSITY OF SCIENCE AND TECHNOLOGY

## G-Eqn Approach vs. Conditional Averaging



$$\frac{\partial G}{\partial t} + \nu \cdot \nabla G = S_d |\nabla G|$$

**G-Equation Approach**



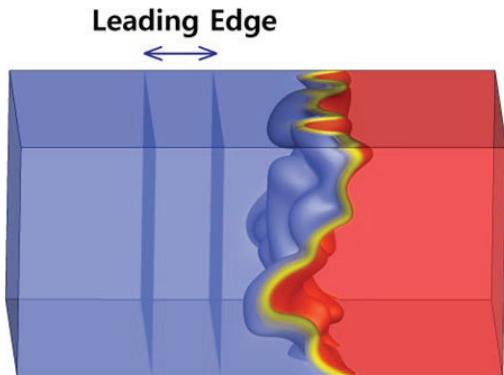
$$\frac{\partial c}{\partial t} + \nu \cdot \nabla c = S_d \Sigma_f'$$

$$\Sigma_f' = |\nabla c|$$

**Conditional Averaging**

Combustion Laboratory  POSTECH  
POHANG UNIVERSITY OF SCIENCE AND TECHNOLOGY

# Statistics at the Leading Edge



$$\nabla(\rho c) = \rho \nabla c + c \nabla \rho \quad \text{as } c \rightarrow 0$$

$$\text{since } \rho \nabla c \sim O(c)$$

$$c \nabla \rho \sim O(c^2) \quad (\text{ideal gas})$$

therefore,

Incompressible with constant  $\rho$  at LE

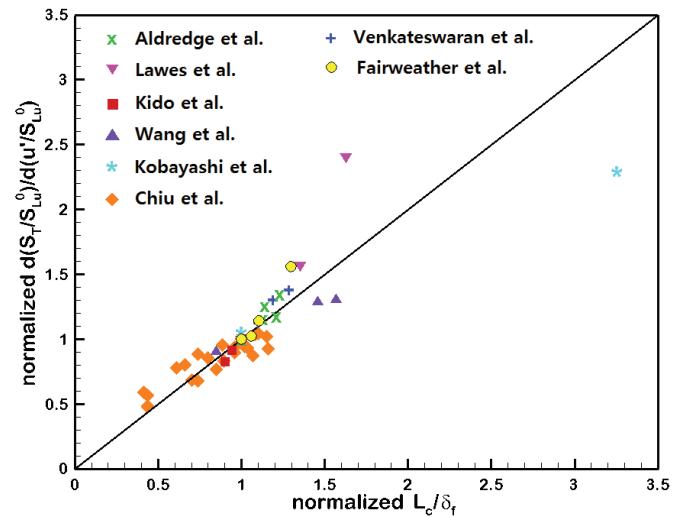
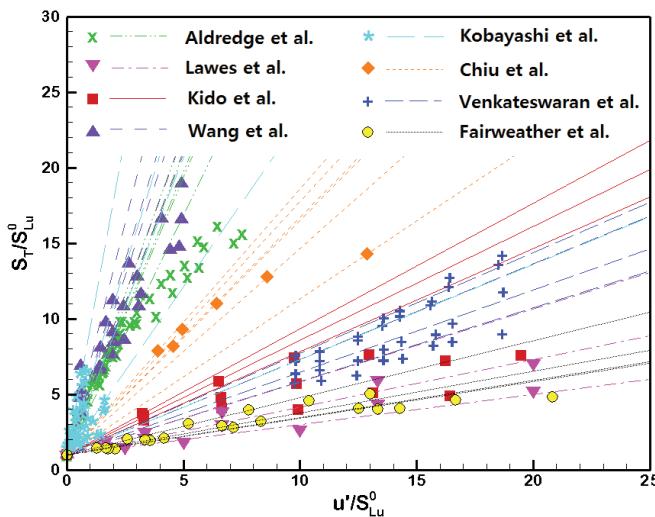
- $\bar{c}$  decays exponentially
- $c$  conditional in a flamelet decays exponentially

$$1/L_w = \left( \frac{1}{\bar{c}} \frac{d\bar{c}}{dx} \right) \quad \text{as } \bar{c} \rightarrow 0$$

$$1/L_m^* = \left\langle \frac{1}{c} \frac{dc}{dx} \right\rangle_f \quad \text{as } c \rightarrow 0$$

## $S_T$ Validation by Published Measurement Data

### Normalized $dS_T/du'$ and $L_c/\delta_f$



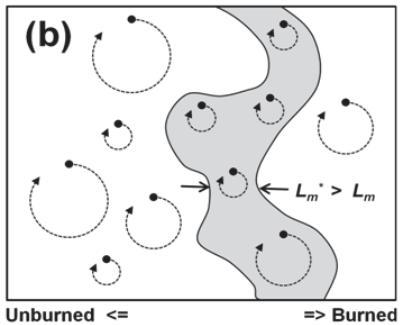
(a) Measured (symbols) and fitted (lines)  $S_T/S_{Lu}^0$  with respect to  $u'/S_{Lu}^0$  and (b)  $dS_T/du'$  with respect to  $L_c/\delta_f$  with both normalized by those of the reference cases for 35 datasets

$$S_T/S_{Lu}^0 = \frac{KCL_c}{C_s \delta_f} \frac{u'}{S_{Lu}^0} + 1 \Rightarrow \frac{d(S_T/S_{Lu}^0)}{d(u'/S_{Lu}^0)} = \frac{dS_T}{du'} \propto \frac{L_c}{\delta_f}$$

$$\frac{(dS_T/du')}{(dS_T/du')_{ref}} \propto \frac{(L_c/\delta_f)}{(L_c/\delta_f)_{ref}}$$

# Asymptotic Expressions for $S_T$ at the Leading Edge

## Arbitrary Flamelet Regime

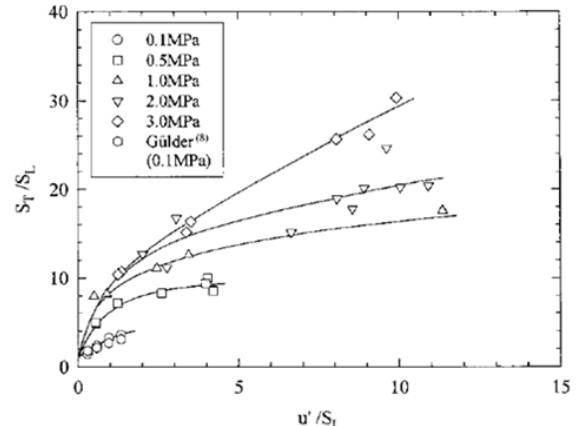


$$S_T = \frac{(D_{mu} + D_{tu})}{L_w}, \quad \frac{1}{L_w} = \frac{1}{L_m^*} - \langle \nabla \cdot \mathbf{n} \rangle_f$$

$$S_T / S_{Lu}^0 = \left( \frac{D_{tu}}{D_{mu}} + 1 \right) \left( \frac{L_m}{L_m^*} - L_m \langle \nabla \cdot \mathbf{n} \rangle_f \right)$$

- Bending due to decreasing  $\left( \frac{L_m}{L_m^*} - L_m \langle \nabla \cdot \mathbf{n} \rangle_f \right)$

- Positive mean curvature,  $\langle \nabla \cdot \mathbf{n} \rangle_f$
- Broadened  $L_m^*$  by curvature and unsteadiness



Lee and Huh, Combust. Flame, 159 (2012).

Combustion Laboratory POSTECH

# Turbulent Premixed Combustion Models

## Xi Foam

**Reaction progress variable :**  $c = \frac{T - T_f}{T_b - T_f}$  or  $c = \frac{Y_{Fu} - Y_F}{Y_{Fu} - Y_{Fb}}$

**Rhegic variable :**  $b = 1 - c = \begin{cases} 1 & \text{unburned gas} \\ 0 & \text{burned gas} \end{cases}$

$$\frac{\partial}{\partial t}(\rho b) + \nabla \cdot (\rho \mathbf{u} b) = \nabla \cdot \left( \frac{\mu_t}{Sc_t} \nabla b \right) - \rho S_c$$

$\rho S_c = \rho_u S_u \psi | \nabla b |$        $\rho_u$  : unburned gas density

$S_u$  : laminar flame speed

$\psi = S_T / S_u$        $S_T$  : turbulent burning velocity

### Three models for $\Psi$

- 1) Fixed  $\Psi$
- 2) Algebraic

$$\psi_{eq}^* = 1 + 0.62 \sqrt{\frac{u'}{S_u}} R_\eta \quad R_\eta : \text{Kolmogorov Re}$$

$$\psi_{eq} = 1 + 2(1-b)(\psi_{eq}^* - 1)$$

- 3) Transport equation for  $\Psi$  (LES) [1]

[1] H.G. Weller et al., "Application of a flame-wrinkling LES combustion model to a turbulent mixing layer", Proc. Combust. Inst. 27, pp. 899-907, 1998.

Combustion Laboratory POSTECH

# Turbulent Premixed Combustion Models

## ■ Coherent Flamelet Model (CFM) [1]

$$\frac{\partial \Sigma}{\partial t} + \nabla \cdot (\mathbf{u} \Sigma) = \nabla \cdot (\Gamma \nabla \Sigma) + \alpha K_t \Sigma - D \Sigma^2$$

$$\Gamma = \Gamma_m + \Gamma_t$$

$$K_t : \text{turbulent stretch } (= \sqrt{\frac{\varepsilon}{\mu}}, \frac{\varepsilon}{k}, \text{ ITNFS })$$

$D$  : consumption coeff.

### ▪ ITNFS (Intermittent Turbulence Net Flame Stretch) model

$\Gamma_k$  : dimensionless net flame stretch function

$$\Gamma_k = \frac{K_t}{\varepsilon/k} = f\left(\frac{u'}{S_L}, \frac{l_t}{\delta_f}\right) = \Gamma_p - b\Gamma_q$$

$\Gamma_p - b\Gamma_q$  : flame production and quenching due to stretch

[1] Meneveau, C. and T. Poinsot, "Stretching and quenching of flamelets in premixed turbulent combustion", *Combustion and Flame* **86**, pp. 311-332, 1991.

# Turbulent Partially-Premixed Combustion Models

## ■ Xi FlameletsFoam (Partially-Premixed Coherent Flamelet Model (PCFM) [1])

$$\begin{aligned} \frac{\partial}{\partial t} (\bar{\rho} \tilde{c}) + \nabla \cdot (\bar{\rho} \tilde{u} \tilde{c}) &= \nabla \cdot (\Gamma_c \nabla \tilde{c}) + \tilde{\omega}_c + 2 \frac{\Gamma_c}{(\tilde{Y}_{Fu} - \tilde{Y}_{Fb})} \nabla \tilde{c} \cdot \nabla (\tilde{Y}_{Fu} - \tilde{Y}_{Fb}) \\ \frac{\partial}{\partial t} (\bar{\rho} \tilde{\xi}) + \nabla \cdot (\bar{\rho} \tilde{u} \tilde{\xi}) &= \nabla \cdot (\Gamma_\xi \nabla \tilde{\xi}) \end{aligned}$$

$$\tilde{\omega}_c = \rho_u S_L \psi |\nabla \tilde{c}|$$

### ▪ Favre mean scalars, $\tilde{\phi}$

$$\tilde{\phi}(\tilde{\xi}, \tilde{c}; x, t) = \int_0^1 \int_0^1 \phi(\xi, c) P(\xi, c; x, t) d\xi dc$$

Assume statistical independence of  $\xi$  and  $c$  and bimodal  $P_c$

$$\begin{aligned} P(\xi, c) &= P(\xi) (\tilde{c} \delta(1-c) + (1-\tilde{c}) \delta(c)) \\ \tilde{\phi} &= (1-\tilde{c}) \int_0^1 \phi(\xi, 0) P(\xi) d\xi + \tilde{c} \int_0^1 \phi(\xi, 1) P(\xi) d\xi \\ &= (1-\tilde{c}) \tilde{\phi}_u(\tilde{\xi}) + \tilde{c} \tilde{\phi}_b(\tilde{\xi}) \end{aligned}$$

[1] Y. Zhang and R. Rawat, "Simulation of Turbulent Lifted Flames Using a Partially Premixed Coherent Flame Model", *J. Eng. Gas Turbines. Power*, **131**, 2009.

# Turbulent Premixed Combustion Models

## Turbulent Premixed Combustion (Conditional Averaging)– In Progress

- $\bar{c}$  or  $\tilde{c}$  transport

$$\frac{\partial}{\partial t}(\bar{\rho}\tilde{c}) + \nabla \cdot (\bar{\rho}\tilde{v}\tilde{c} + \bar{\rho}\tilde{v}\tilde{c}') = \nabla \cdot (\bar{\rho}D_m \nabla \bar{c}) + \bar{\rho}\tilde{w}_c$$

$$\frac{\partial \bar{c}}{\partial t} = \langle v_n \rangle + S_d \Sigma_f$$

$$v_n = \mathbf{v} \cdot \mathbf{n} \quad \Sigma_f = \langle \Sigma_f' \rangle \quad \Sigma_f' = |\nabla c|$$

- Zone conditional averaging

- Conditional continuity equations

$$\frac{\partial}{\partial t}(\bar{c}\rho_b) + \nabla \cdot (\bar{c}\rho_b \langle \mathbf{v} \rangle_b) = \rho_b S_{Lb} \Sigma_f$$

$$\frac{\partial}{\partial t}[(1-\bar{c})\rho_u] + \nabla \cdot ((1-\bar{c})\rho_u \langle \mathbf{v} \rangle_u) = -\rho_u S_{Lu} \Sigma_f$$

- Conditional momentum equations

$$\frac{\partial}{\partial t}(\bar{c}\rho_b \langle \mathbf{v} \rangle_b) + \nabla \cdot (\bar{c}\rho_b \langle \mathbf{v} \rangle_b \langle \mathbf{v} \rangle_b) = -\bar{c} \nabla \langle p \rangle_b - \nabla \cdot (\bar{c}\rho_b \langle \mathbf{v}'_b \mathbf{v}'_b \rangle) + \rho_b S_{Lb} \langle \mathbf{v} \rangle_b \Sigma_f$$

$$\frac{\partial}{\partial t}[(1-\bar{c})\rho_u \langle \mathbf{v} \rangle_u] + \nabla \cdot ((1-\bar{c})\rho_u \langle \mathbf{v} \rangle_u \langle \mathbf{v} \rangle_u) = -(1-\bar{c}) \nabla \langle p \rangle_u - \nabla \cdot ((1-\bar{c})\rho_u \langle \mathbf{v}'_u \mathbf{v}'_u \rangle) - \rho_u S_{Lu} \langle \mathbf{v} \rangle_u \Sigma_f$$

# OpenFOAM Solver Development

## CMCreactingFoam

- Transient solver with newly implemented Conditional Moment Closure.
- CMC model based on rigorous mathematical procedure for conditional averaging.
- Conditional sub-models required for conditional velocity, scalar dissipation rate, etc.

## Conditional sub-models

### Conditional species concentration and enthalpy

$$\begin{aligned} Q_i(\eta, \bar{x}, t) &= \langle Y_i | \eta \rangle \equiv \langle Y_i(\bar{x}, t) | \xi(\bar{x}, t) = \eta \rangle \\ \frac{\partial Q_i}{\partial t} + \langle u | \eta \rangle \cdot \nabla Q_i + \frac{\nabla \cdot (\langle u'' Y'' | \eta \rangle P(\eta) \rho_\eta)}{P(\eta) \rho_\eta} &= \langle N | \eta \rangle \frac{\partial^2 Q_i}{\partial \eta^2} + \langle w_i | \eta \rangle \\ \frac{\partial Q_h}{\partial t} + \langle u | \eta \rangle \cdot \nabla Q_h + \frac{\nabla \cdot (\langle u'' h'' | \eta \rangle P(\eta) \rho_\eta)}{P(\eta) \rho_\eta} &= \langle N | \eta \rangle \frac{\partial^2 Q_h}{\partial \eta^2} + \left\langle \frac{1}{\rho_\eta} \frac{\partial p}{\partial t} \right\rangle | \eta \rangle \end{aligned}$$

### Conditional reaction rate

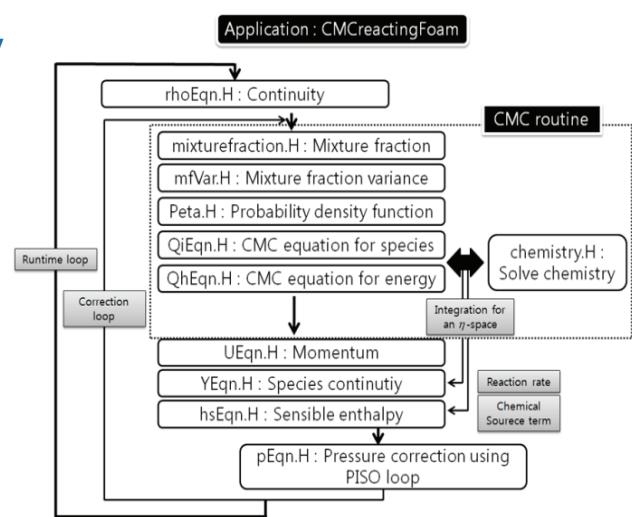
$$\langle w_i | \eta \rangle \approx w_i(Q_i, \dots, Q_n, Q_h)$$

### Conditional Velocity

$$\langle u_i | \eta \rangle = \tilde{u}_i + \frac{\tilde{u}_i \tilde{\xi}'}{\tilde{\xi}^2} (\eta - \tilde{\xi}) \quad \text{where } \tilde{u}_i \tilde{\xi}' = -D_t \frac{\partial \tilde{\xi}}{\partial x_i}$$

### Conditional scalar dissipation rate

$$\langle N | \eta \rangle = \frac{\tilde{N} \exp(-2[erf^{-1}(2\eta-1)]^2)}{\int_0^1 \exp(-2[erf^{-1}(2\eta-1)]^2) \tilde{P}(\eta) d\eta} \quad \text{where } \tilde{N} \equiv D(\nabla \tilde{\xi})^2 = \frac{\tilde{\epsilon}}{k} \tilde{\xi}^{n^2}$$

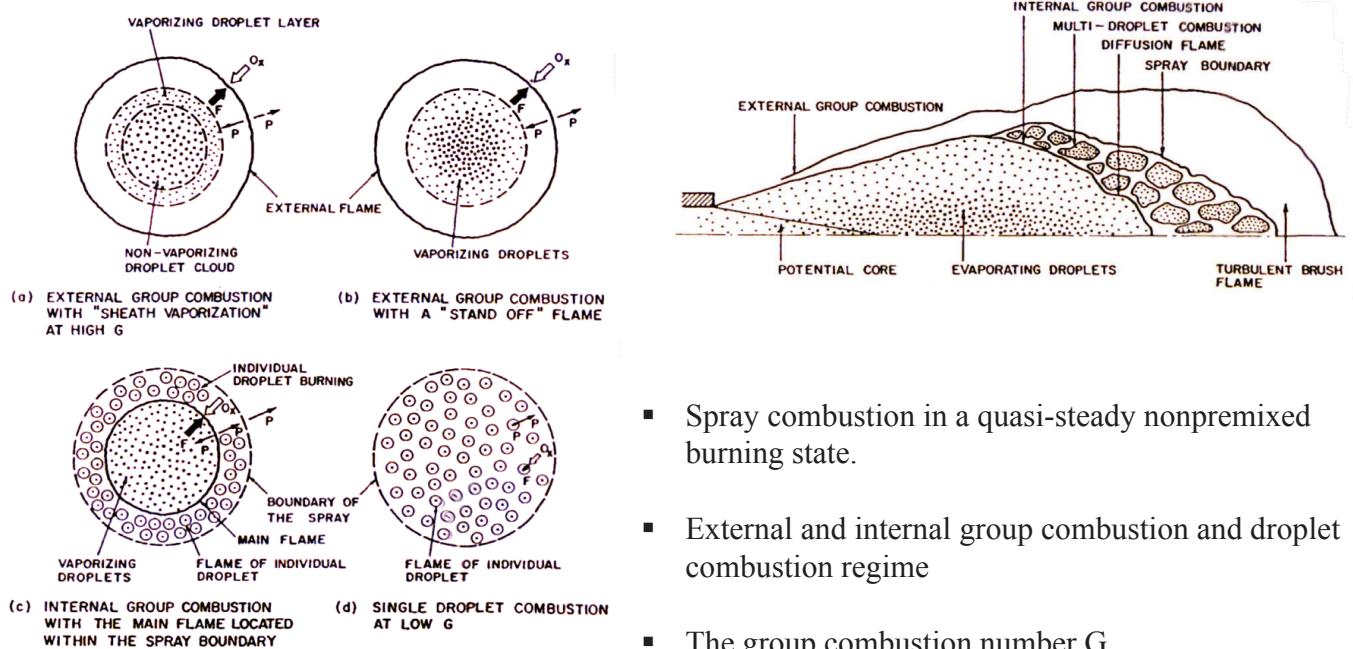


Structure of CMCreatingFoam

# Spray Combustion

## Spray Combustion Regime Map

- Group combustion number [Chiu, 1977]



- Spray combustion in a quasi-steady nonpremixed burning state.
- External and internal group combustion and droplet combustion regime
- The group combustion number G

Combustion Laboratory POSTECH

# Spray Combustion

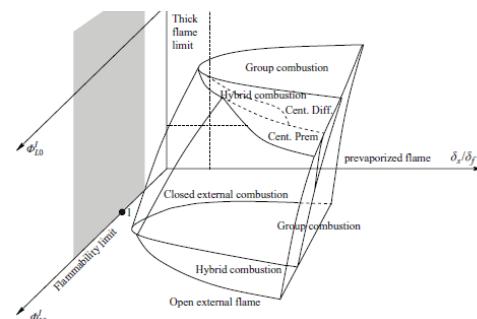
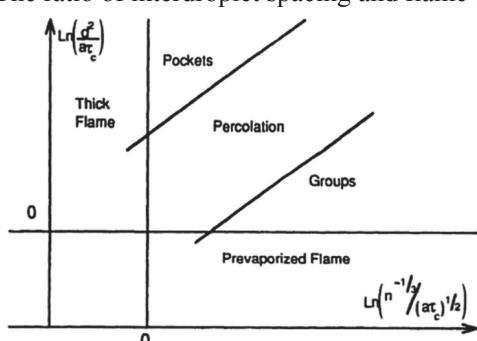
## Spray Combustion Regime Map

- Spray jet flow [Réveillon and Vervisch, 2005]

- Weakly turbulent spray-jet with coflowing preheated air
- External, group and hybrid combustion regimes
- Global equivalence ratio and ratio between mean inter-droplet distance and flamelet thickness

## Regime map [Borghi, 1996]

- The diffusion reaction zone behind a premixed flame
- Group, pocket, percolation, thick and prevaporized flame
- The ratio of interdroplet spacing and flame thickness



## Percolation theory [Kertein and Law, 1982]

- Randomly distributed reaction zones in the fast chemistry limit
- Cluster and percolate combustion regimes

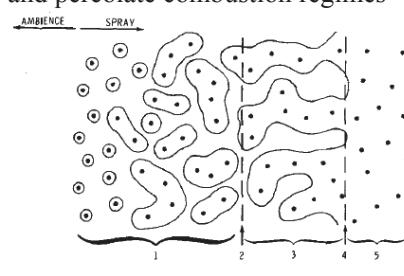
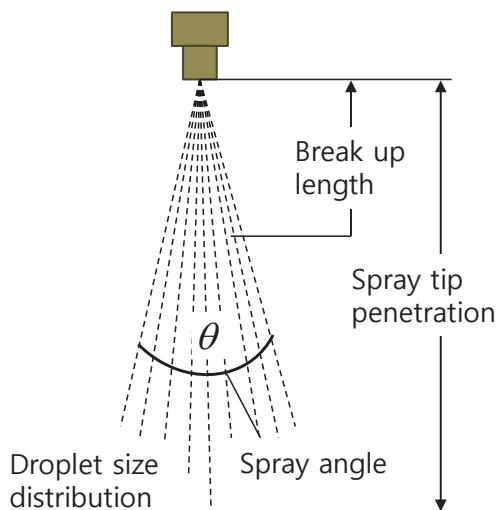


FIG. 1. Combustion regimes in a non-premixed spray (schematic). 1. Cluster combustion. 2. Transition boundary. 3. Percolate combustion. 4. Leading edge. 5. Vaporization.

Combustion Laboratory POSTECH

# Spray Combustion

## Spray Characteristic



### Spray tip penetration

- The maximal distance measured from the injector to spray tip
- Maximum penetration length achieved by the droplets in the center of the spray

### Spray angle

- Angle between two straight lines originating from the orifice exit of the nozzle and being tangent to the spray outline
- Usually, ranges from 5 to 30°

### Droplet size distribution

- Usually measured on an average basis by medium diameter of the droplet  $d_{32}$ , called the Sauter mean diameter
- A quantity can be used to estimate the quality of atomization of the fuel

### 1. For incomplete spray

$$SMD = 0.38d(\text{Re})^{0.25}(\text{We})^{-0.32}\left(\frac{\mu_l}{\mu_g}\right)^{0.37}\left(\frac{\rho_l}{\rho_g}\right)^{-0.47}$$

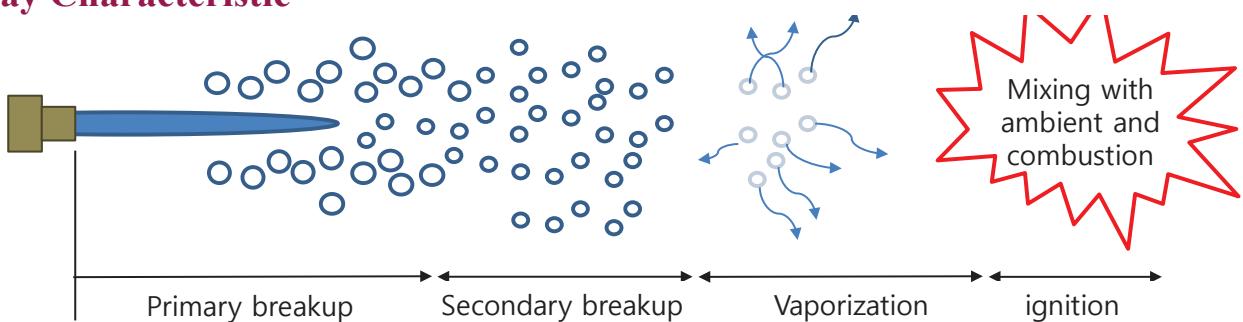
### 2. For complete spray

$$SMD = 8.7(\text{Re}_l \text{We}_l)^{-0.28} d_0$$

Combustion Laboratory  POSTECH

# Spray Combustion

## Spray Characteristic



### Primary breakup

- A break up of liquid core into droplets, shortly after nozzle exit

### Secondary breakup

- In the second stage, the formed droplets break up into smaller droplets

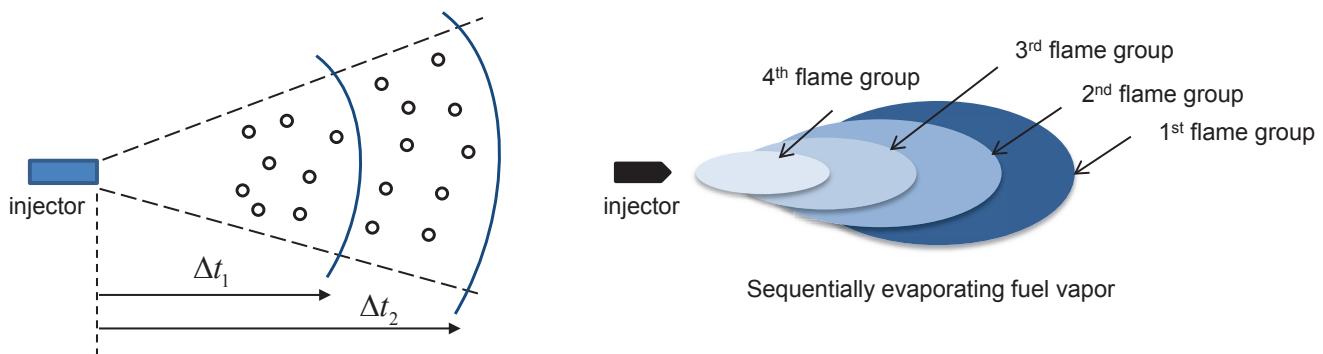
### Collision / Coalescence

- Drop collision and coalescence phenomena is important in dense spray
- Coalescence occur at low drop collision weber numbers
- Separation of colliding drops occurs as the weber number is increased beyond a critical value

# Spray Combustion

## Lagrangian CMC concept

- Use fluid sample of the given residence time at locations of the corresponding Lagrangian particles
- Multiple flame groups may be defined for fuel injected sequentially at the nozzle and tracked with Lagrangian identities in the domain
- Fuel of the same residence time shares approximately the same conditional flame structure in many problems including transient autoignition and steady parabolic 1D jet flame



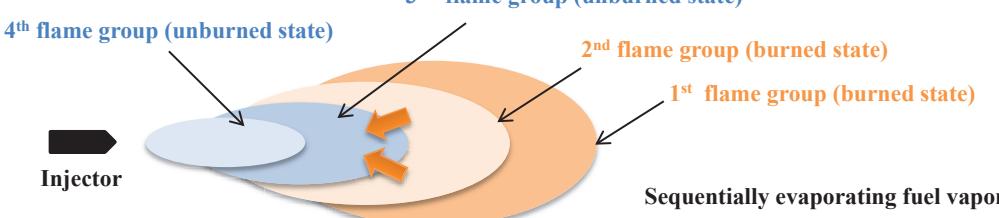
# Spray Combustion

## Lagrangian CMC with flame group interaction

- Flame group interaction is modeled as propagating premixed combustion by the EBU model
- The mean reaction progress variable is defined as
$$\tilde{c} \equiv \widetilde{\xi_B} / \widetilde{\xi} \quad \text{where} \quad \widetilde{\xi_B} + \widetilde{\xi_{UB}} = \widetilde{\xi} \quad \text{and} \quad (1 - \tilde{c})\widetilde{\xi} = \sum_{\text{all } j \text{ unburned}} \widetilde{\xi_j}$$
- The source term for  $\tilde{c}$  is given by the EBU model for premixed combustion as
$$\dot{\tilde{w}_c} \equiv K \frac{\tilde{c}(1 - \tilde{c})}{\tau_t} \quad \text{where integral time scale} \quad \tau_t \equiv \tilde{k} / \widetilde{\varepsilon}$$
- Transport equation for fuel fraction of the  $j$ -th fuel group

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

3rd flame group (unburned state)

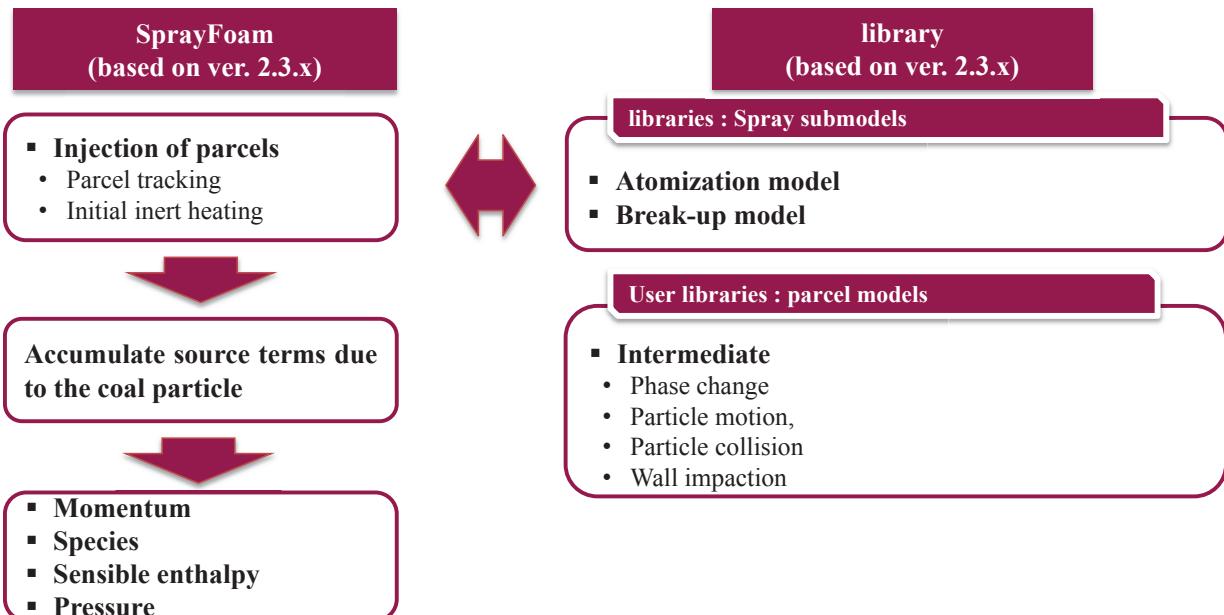


# Spray Combustion – OpenFOAM algorithm

## OpenFOAM solver structure

### SprayFoam

- Transient PIMPLE solver for compressible, laminar or turbulent flow with spray parcels
- Spray submodels : Atomization model, Break-up model

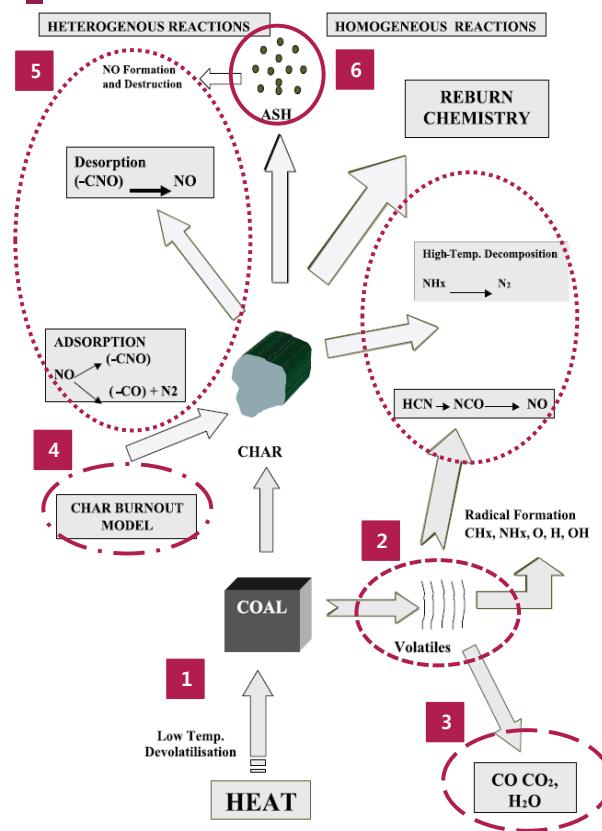


The schematic diagram of the SprayFoam

Combustion Laboratory POHANG UNIVERSITY OF SCIENCE AND TECHNOLOGY POSTECH

# Coal Combustion

## Pulverized coal combustion model



(1) Analysis of particle trajectory and particle heating

(2) Devolatilization model

(3) Gas phase reaction model

(4) Char combustion model

(5) NOx model

(6) Ash deposition and slagging model

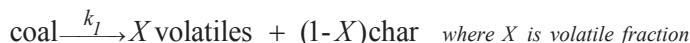
The schematic diagram of pulverized coal combustion

Combustion Laboratory POHANG UNIVERSITY OF SCIENCE AND TECHNOLOGY POSTECH

# Coal Combustion – Devolatilization Model

## Empirical model

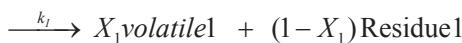
### Single step model



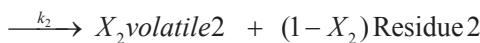
$$\frac{dX(t)}{dt} = k_1(X^\infty - X(t)) \quad k_1 = A_1 \exp(-E_1 / RT)$$

### Two competing rates model (at low and high temperatures)

- Heating rate affects both devolatilization rate and yield



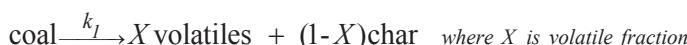
coal



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

### Distributed Activation Energy Model (DAEM)

- Volatiles can be released from coal of different activation energies in parallel

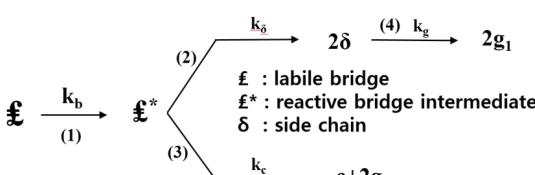
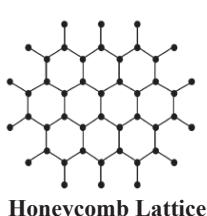


$$1 - \frac{X(t)}{X^\infty} = \int_0^\infty \exp\left[-k_0 \int_0^t e^{-E/RT} dt\right] f(E) dE \quad \int_0^\infty f(E) dE = 1$$

# Coal Combustion – Devolatilization Model

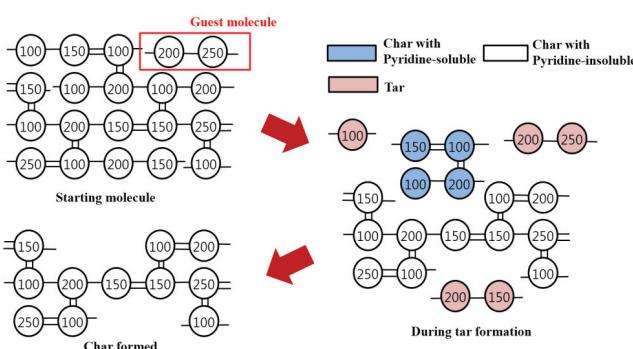
## Network model

### CPD (Chemical Percolation Devolatilization)

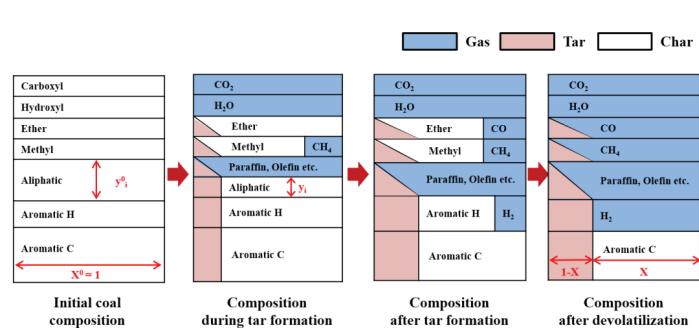


- (1) formation of a reactive bridge intermediate
- (2) formation of a side chain
- (3) formation of char bridge and gas
- (4) conversion of side chain into light gases

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



DVC model



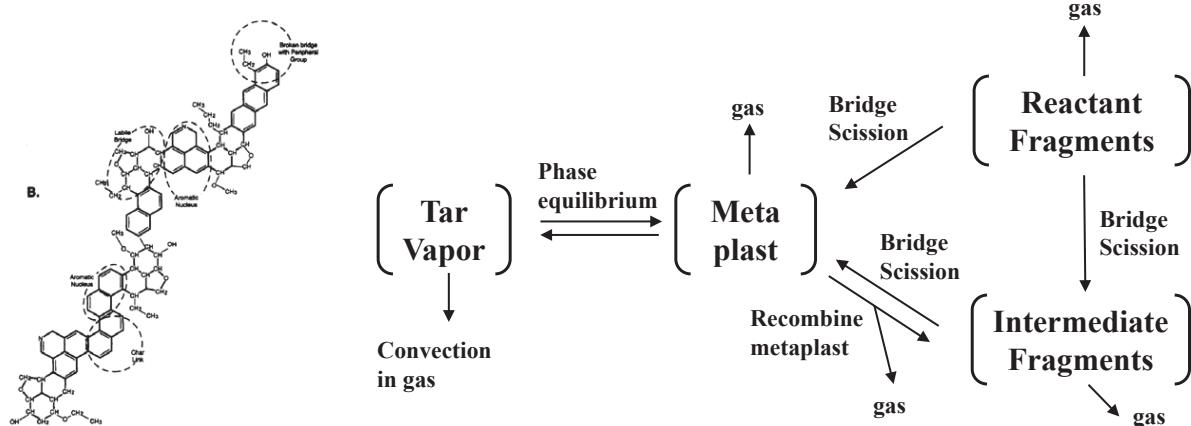
FG model

# 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

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

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

$D_{AB}$  : diffusion coefficient,

$Sh$  : Sherwood number,

$MW_c$  : carbon molecular weight,

$P_\infty$  : bulk pressure,

$\psi$  : char burnout ratio

### Apparent rate / Diffusion rate model

- Most often used for char combustion model in CFD

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

$$k_d = \left( \frac{2}{1+\psi} \right) \frac{D_{AB} MW_c Sh}{d_p RT_g} \quad \xleftrightarrow{\text{Diffusion}} \quad \underbrace{k_c = A \exp(-E / RT)}_{\text{chemical kinetics}}$$

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

$$\therefore \frac{dm_p}{dt} = -\pi d_p^2 k P_\infty = -\pi d_p^2 \left( \frac{k_d k_c}{k_d + k_c} \right) P_\infty$$

### Global n-th order model

- It's appropriate to use the model in specific condition

$$q = k_s P_s^n \quad \text{where } k_s = A \exp(-E / RT_p)$$

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

$P_s$  : saturated pressure of oxygen 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  $E$  ( $T_g > 1500K$ ,  $75 \mu m < d_p < 200 \mu m$ ,  $P_{O_2} > 0.03 \text{ atm}$ )

# 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

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

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

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

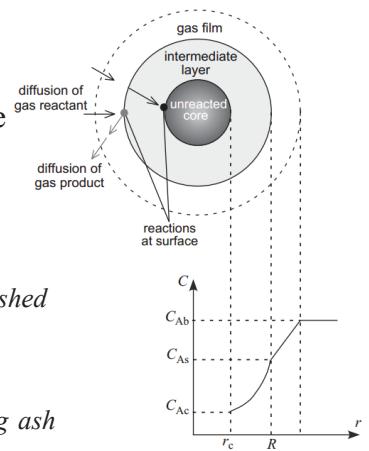
$f$  : conversion when pyrolysis is finished

$\varepsilon$  : 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

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

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

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

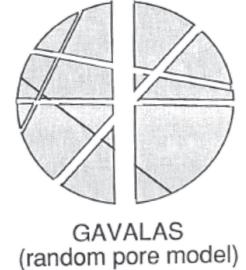
$\psi$  : initial pore structure,

$L_0$  : initial pore length,

$\varepsilon_0$  : initial porosity,

$S_0$  : initial specific surface area

$x$  : conversion ratio of char

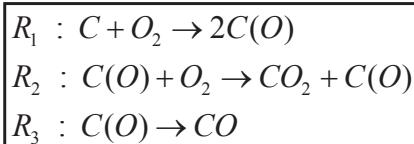


# Coal Combustion – Char Combustion Model

## Intrinsic reactivity model (based on BET)

### CBK/E (Char Burnout kinetic / extended) model

- Assume oxidation reaction to 3 step intrinsic kinetics



$\theta$  : the fraction of sites occupied by the absorbed oxygen complex

$M_T$  : Thiele modulus

Where  $k_1, k_2, k_3$  is Arrhenius form

$$q_1 = k_1 P_{O_2} (1-\theta)$$

$$A_3 = 10^{14.38 - 0.0764 C_{daf}}$$

$$E_1 = 25 \text{ KJ / mole}$$

Ratio of CO and  $CO_2$  generation

$$q_2 = k_2 P_{O_2} \theta$$

$$A_2 / A_3 = 5.0 \times 10^4 \text{ and } E_2 = 117 \text{ KJ / mole}$$

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

$$q_3 = k_3 \theta$$

$$A_3 / A_1 = 1.0 \times 10^{-6}$$

$$E_3 = 133.8 \text{ KJ / mole}$$

Overall oxidation rate

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

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

# 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

Combustion Laboratory  POSTECH

## Thank you



Combustion Laboratory  POSTECH



# 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



OpenFOAM in Industrial Combustion Simulations, Pohang University Feb/2015

Turbulence and Thermodynamics Modelling Libraries – p. 1

## 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 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()`

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

```
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();
```

## Interface to the Momentum Equation: Historical Form

- This is the original implementation (1994), but it is not sufficiently flexible. In fact, it is wrong:  $\nabla \mu_t \cdot \nabla \mathbf{u}$  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 the 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();
```

## 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!

## 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

## 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  $\epsilon$  and  $\omega$  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 ( $\epsilon$ ) in the near-wall cell

## Implementation of Wall Functions

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

## 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



## 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  $\psi$

$$\rho = \psi p$$

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



## 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>>>
    >
>;
```



## 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

26-27<sup>th</sup> Feb. 2015  
POSCO International Center, Pohang, Korea

Kang Y. Huh  
Pohang University of Science and Technology



## Contents

### I Introduction

### II Turbulent Nonpremixed Flames

- TNF flames - Sandia Flame D & E, Bluff-body Flame
- Heat Recovery Steam Generator

### III Turbulent Partially Premixed Flames

- TNF flames – Sydney Swirl Flame SMA1
- 5MWe Micro Gas Turbine

### IV Spray Combustion Modeling

- ECN
- Diesel Engine
- Heavy-oil Furnace

## Contents

### V Solid Combustion Modeling

- IFRF MMF 5-2 flame
- 500MWe Tangential Firing Pulverized Coal Furnace

### VI Material Processing Furnace

- FINEX
- Rotary Klin

### VII Implement New Combustion Models

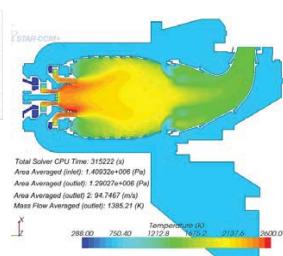
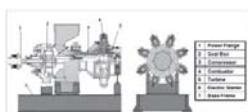
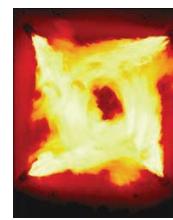
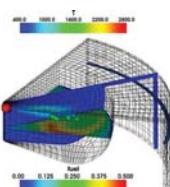
- Structure of reactingParcelFoam and PaSR model
- How to implement New Combustion Models

## Introduction

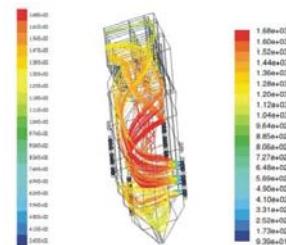
Most industrial combustion devices operate in the regime of turbulent combustion



Diesel engine



Gas turbine



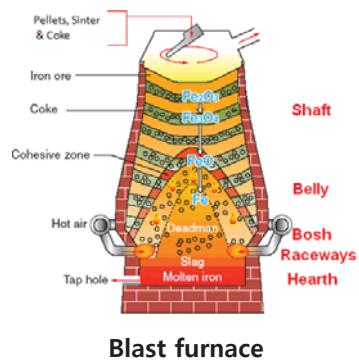
Pulverized coal power plant

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

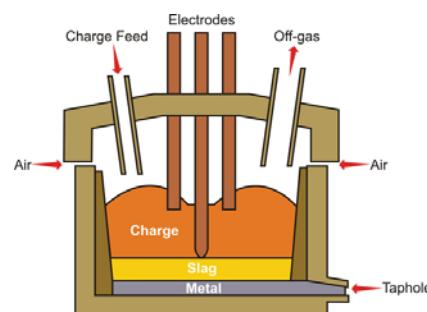
# Introduction

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<br>Pulverized coal furnace |



Blast furnace

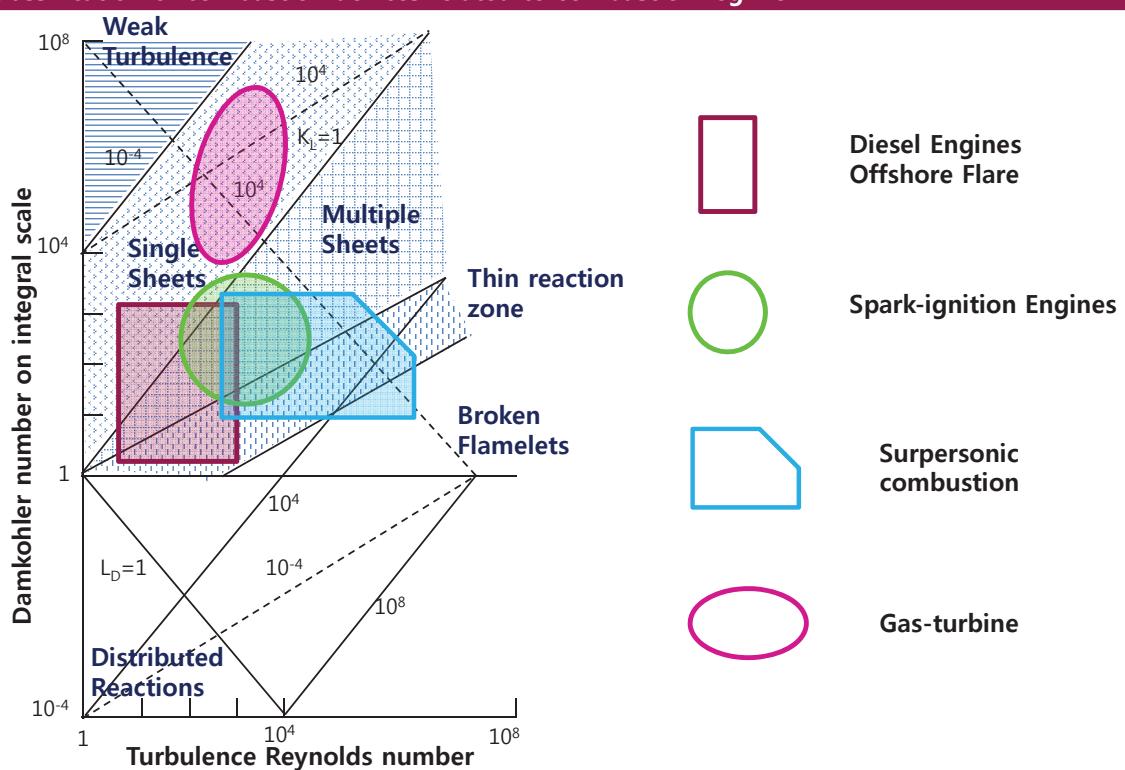


Electric arc furnace

Combustion Laboratory POSTECH

# Introduction

Classification of combustion devices related to combustion regime



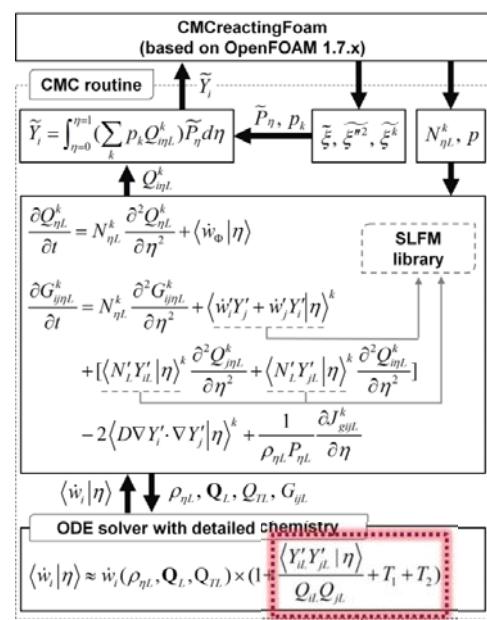
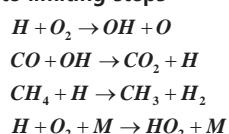
Combustion Laboratory POSTECH

# 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
  - (2<sup>nd</sup> 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

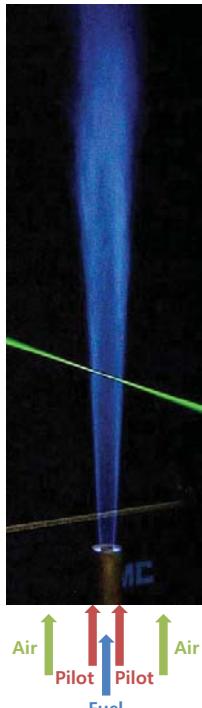


Schematic diagram of interaction between  
OpenFOAM and CMC routines

Second order correction terms

# Basic Flame - Sandia Flame D & E

## Case description



### Sandia/TUD Piloted CH<sub>4</sub>/Air Jet Flames

Fuel : 25% CH<sub>4</sub>, 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

Fuel velocity = 49.6m/s

Pilot velocity = 11.4m/s

Reynolds number = 22,400

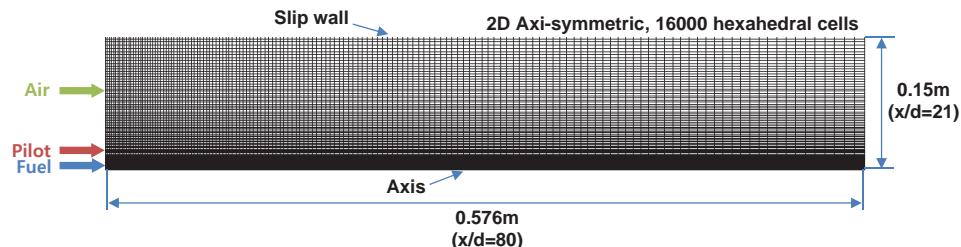
#### Flame E

Fuel velocity = 74.4m/s

Pilot velocity = 17.1m/s

Reynolds number = 33,600

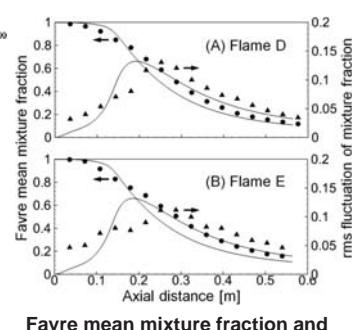
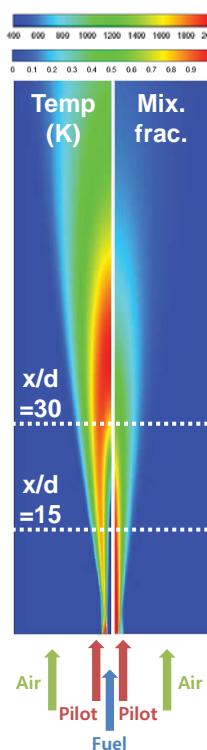
R. S. Barlow and J. H. Frank, Proc. Combust. Inst. 27:1087-1095 (1998)  
R. S. Barlow, J. H. Frank, A. N. Karpetis and J. Y. Chen, Combust. Flame 143:433-449 (2005)  
Ch. Schneider, A. Dreizler, J. Janicka, Combust. Flame 135:185-190 (2003)



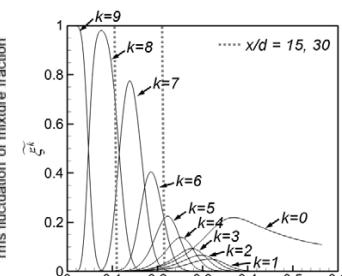
Combustion Laboratory POSTECH  
Pohang University of Science and Technology

# Basic Flame - Sandia Flame D & E

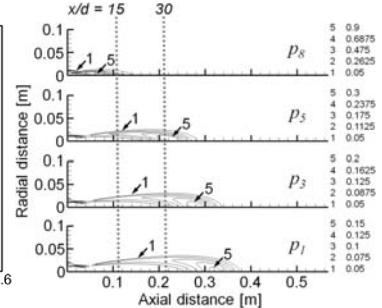
## Results



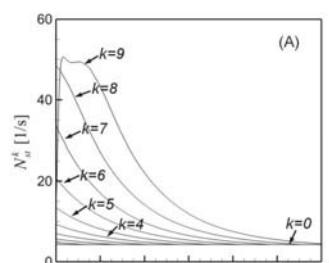
Favre mean mixture fraction and rms fluctuations of mixture fraction along the axis



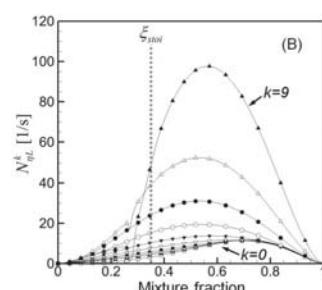
Distribution of  $\bar{x}_L$  for ten flame groups along the axis



Axial distribution of  $P_k$  for the 1<sup>st</sup>, 3<sup>rd</sup>, 5<sup>th</sup> and 8<sup>th</sup> flame groups in Flame D



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

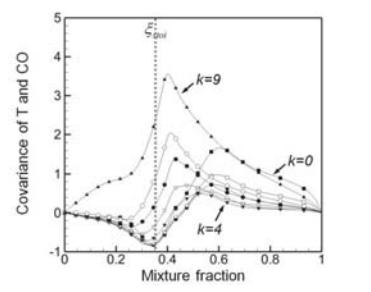
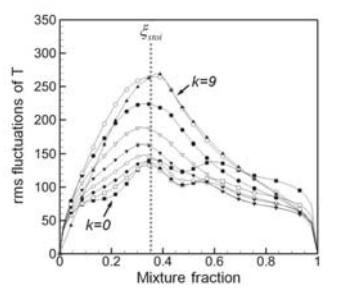
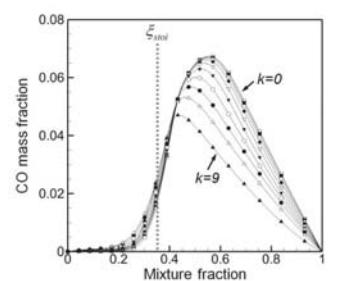
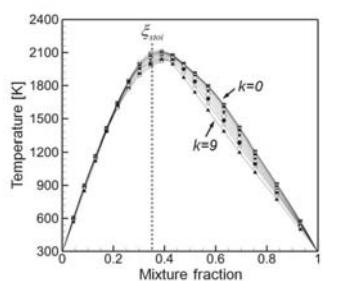
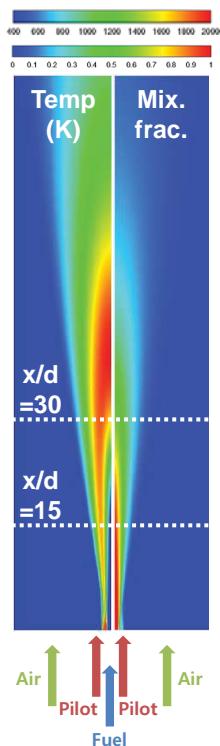


Conditional SDR's for ten flame groups in Flame D

Combustion Laboratory POSTECH  
Pohang University of Science and Technology

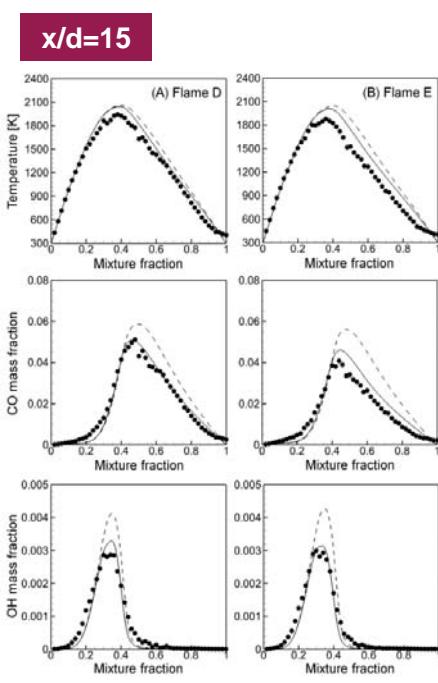
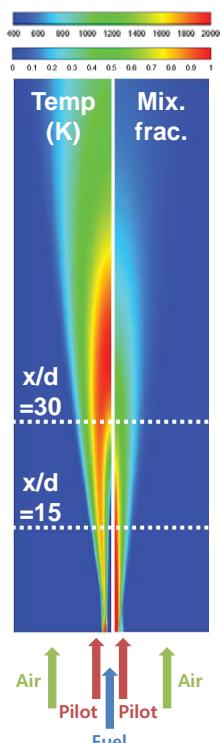
# Basic Flame - Sandia Flame D & E

## Results



# Basic Flame - Sandia Flame D & E

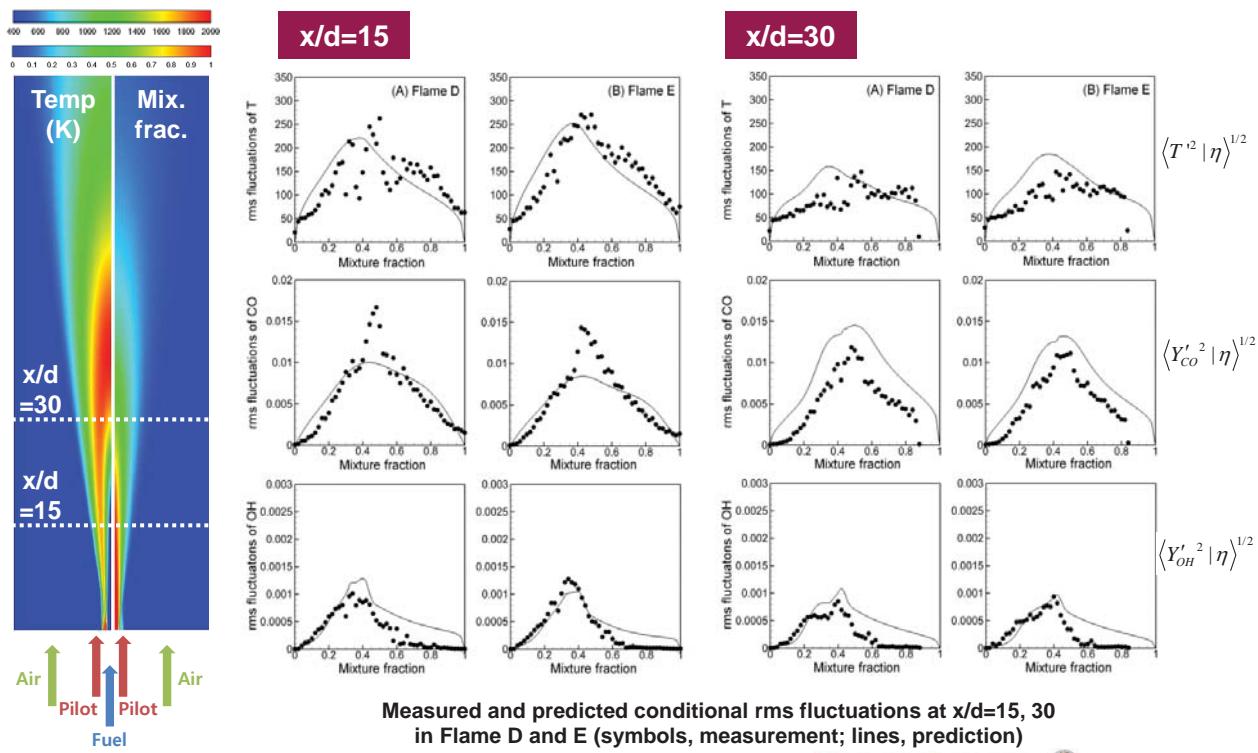
## Results



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

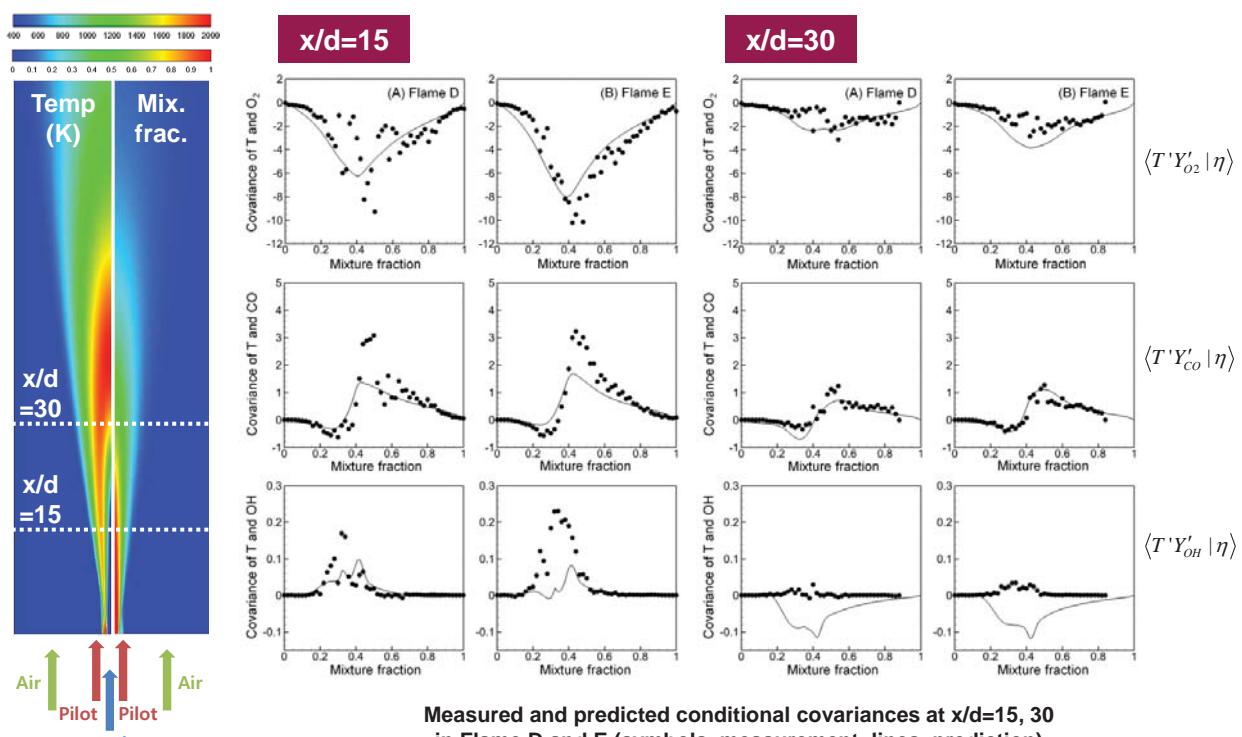


Combustion Laboratory

POSTECH

# Basic Flame - Sandia Flame D & E

## Results

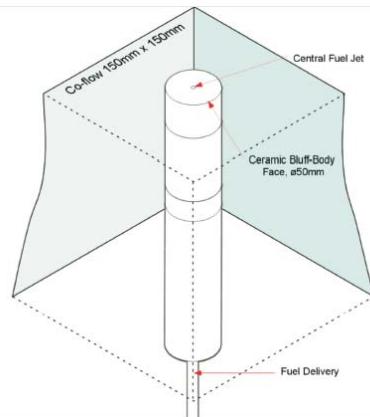


Combustion Laboratory

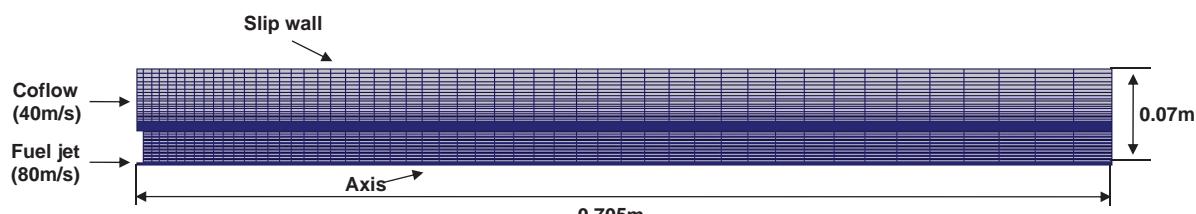
POSTECH

# Basic Flame - Bluff Body Flame

## Case description



| Description                     | Specification                           |
|---------------------------------|-----------------------------------------|
| Fuel                            | $\text{CH}_3\text{OH}(\text{methanol})$ |
| 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                                   |

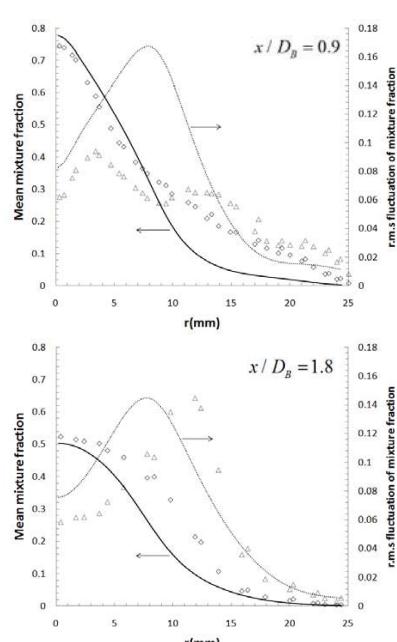
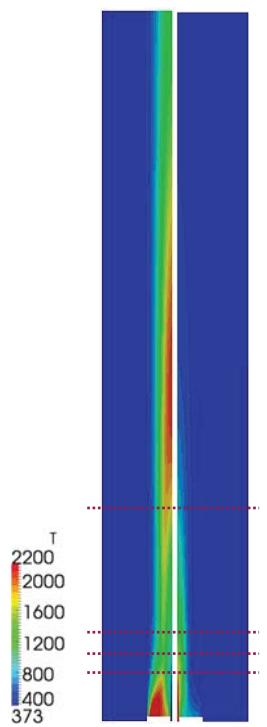


"Instantaneous and mean compositional structure of bluff body stabilized non-premixed flames", B. B. Dally, A. R. Masri, R. S. Barlow, G. J. Fiechtner, Combustion and Flame, 114:119-148 (1998)

Combustion Laboratory POSTECH

# Basic Flame - Bluff Body Flame

## Results

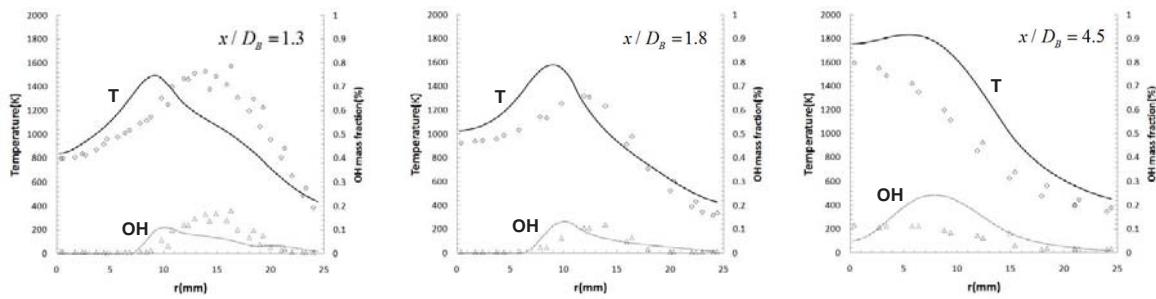


Radial distributions of the Favre mean mixture fraction and r.m.s fluctuation of the mixture fraction

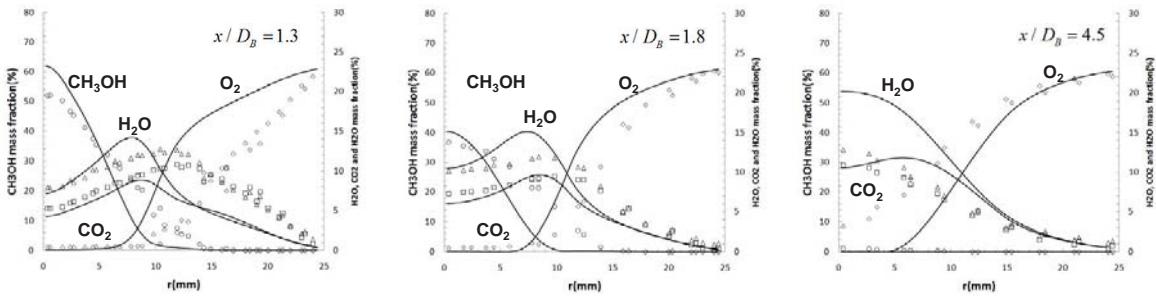
Combustion Laboratory POSTECH

# Basic Flame - Bluff Body Flame

## Results



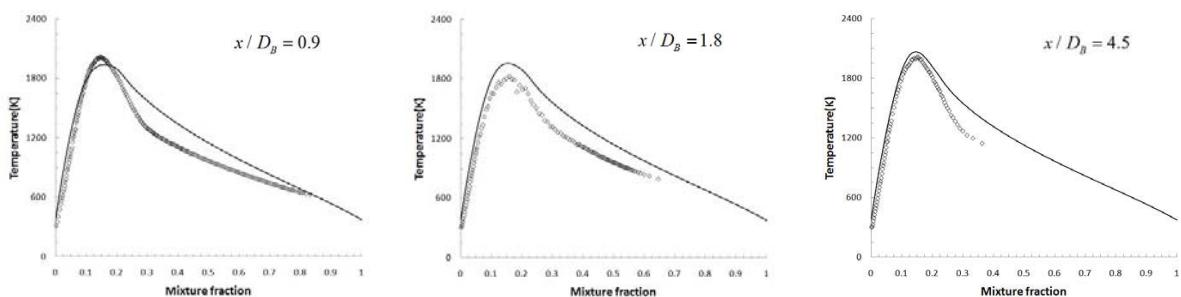
Radial distributions of the Favre mean temperature and OH mass fraction<sup>a</sup>



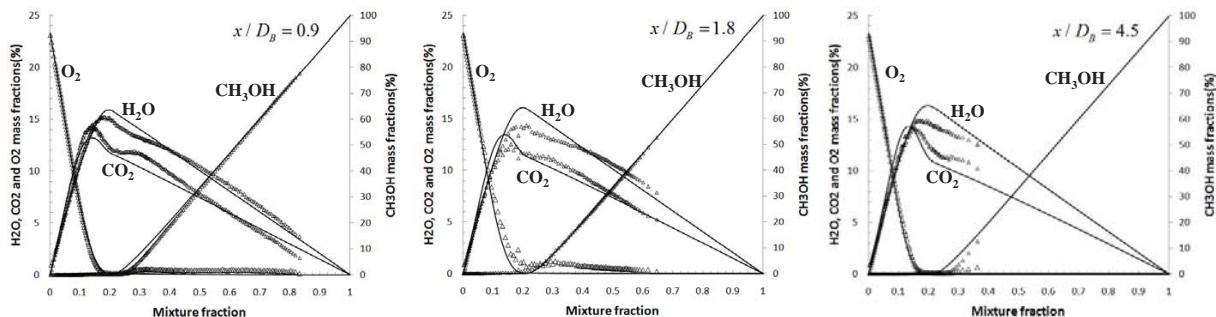
Radial distributions of the major species mass fractions

# Basic Flame - Bluff Body Flame

## Results



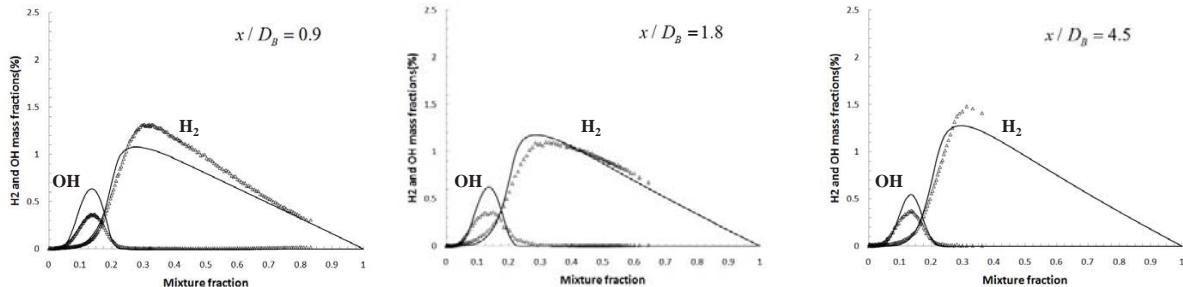
Conditional mean temperature with respect to the mixture fraction



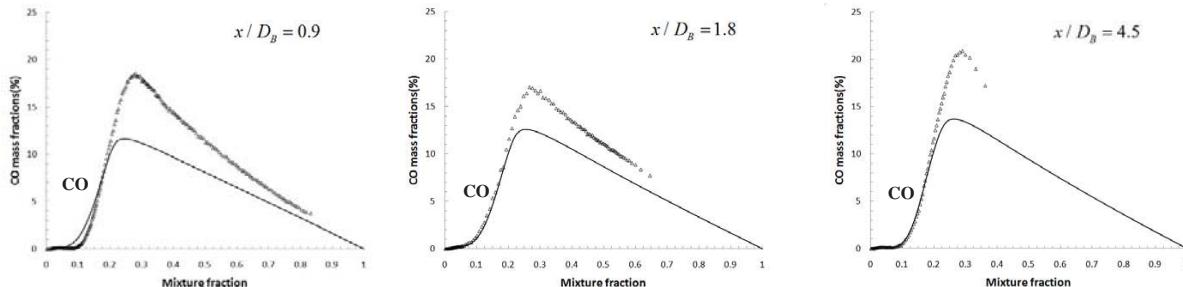
Major species mass fractions with respect to the mixture fraction

# Basic Flame - Bluff Body Flame

## Results



H<sub>2</sub> 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?)

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

• Integrate scalars ( $Y_i, T, h, \dots$ ) from SLFM and PDF library, make 3D Look-up table (mf, mfVar, SDR) before calculation



### PDF Library

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

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

- 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

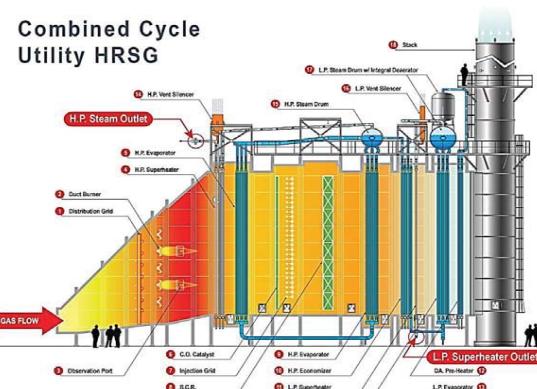
# Industrial Furnace - Heat Recovery Steam Generator

## 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

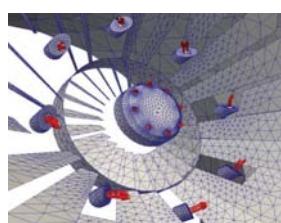
- Silencer**  
Attenuates noise level to meet government and site requirements
- Integral Deaerator**  
Uses low temperature heat to deaerate feed-water for improved thermal efficiency
- CO Catalyst**  
Reduces carbon monoxide in the flue gas
- Divertor Valve**  
Modulates steam production in the bypass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



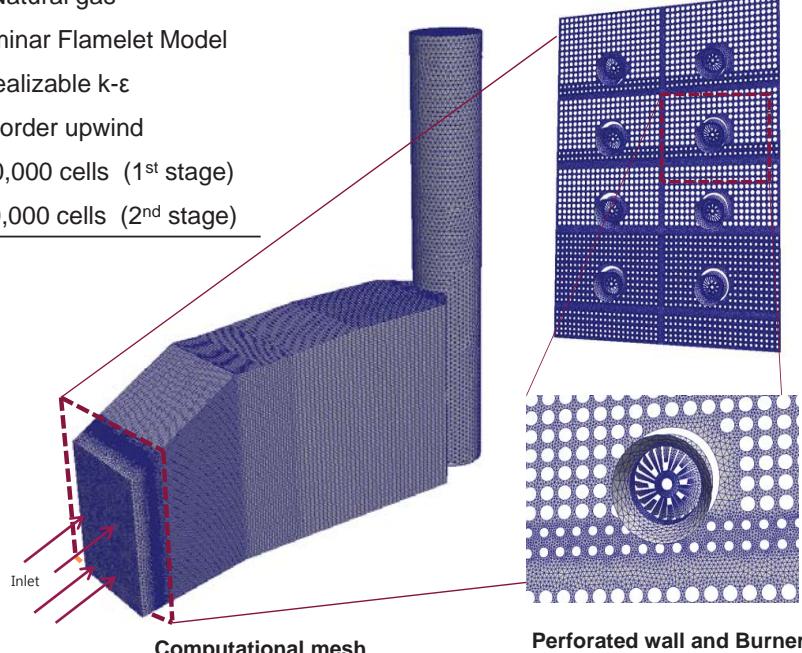
# Industrial Furnace - Heat Recovery Steam Generator

## Case description

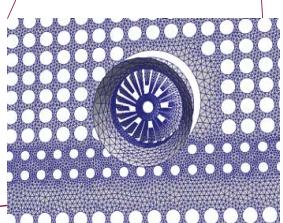
| Description      | Specification                                                                                  |
|------------------|------------------------------------------------------------------------------------------------|
| Fuel             | Natural gas                                                                                    |
| Combustion model | Steady Laminar Flamelet Model                                                                  |
| Turbulence model | Realizable k- $\epsilon$                                                                       |
| Discretization   | 2 <sup>nd</sup> order upwind                                                                   |
| Mesh             | About 7,000,000 cells (1 <sup>st</sup> stage)<br>About 9,000,000 cells (2 <sup>nd</sup> stage) |



Fuel nozzle



Computational mesh

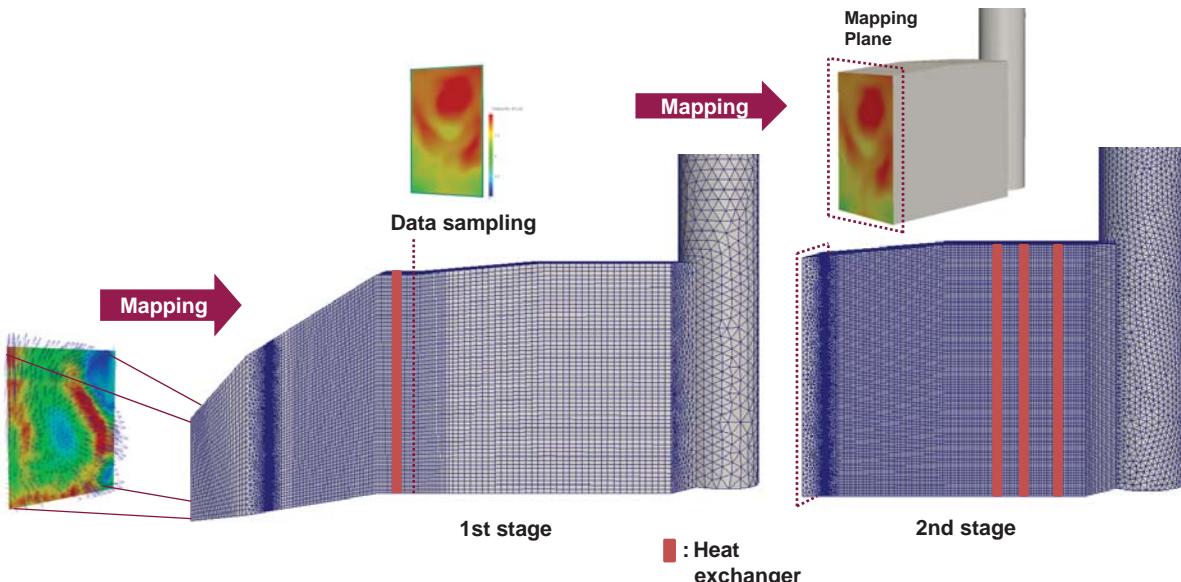


Perforated wall and Burner

# 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; 1<sup>st</sup> stage and 2<sup>nd</sup> stage
- A input of 2<sup>nd</sup> stage use sampled data from 1<sup>st</sup> stage result

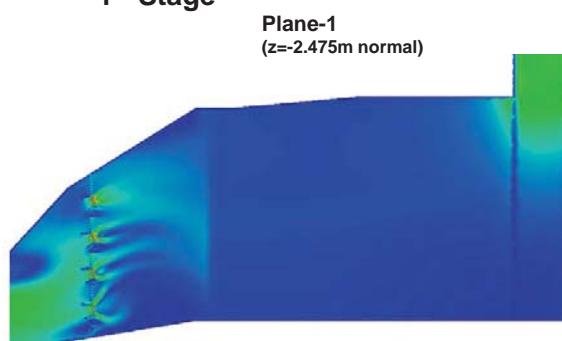


Combustion Laboratory POSTECH

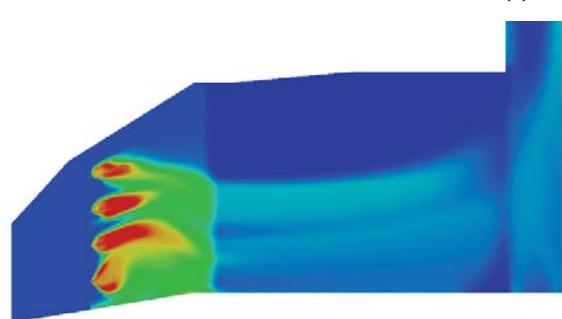
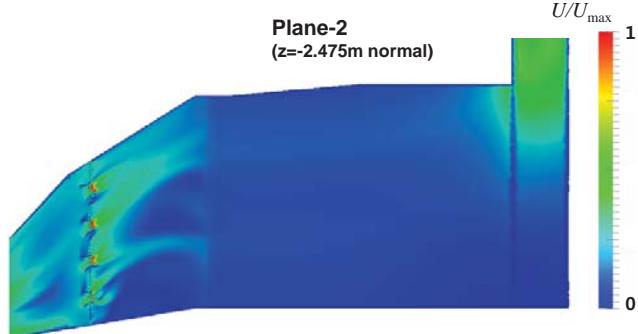
# Industrial Furnace - Heat Recovery Steam Generator

## Results

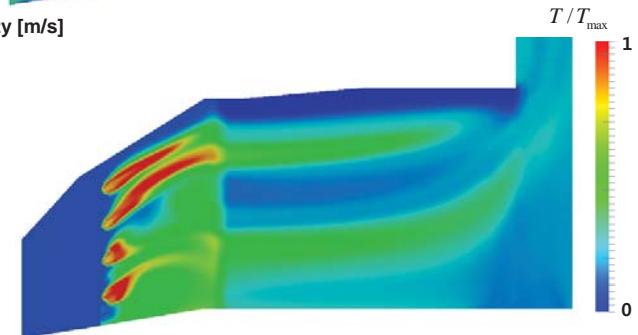
### ▪ 1<sup>st</sup> Stage



(a) Velocity [m/s]



(b) Temperature [K]

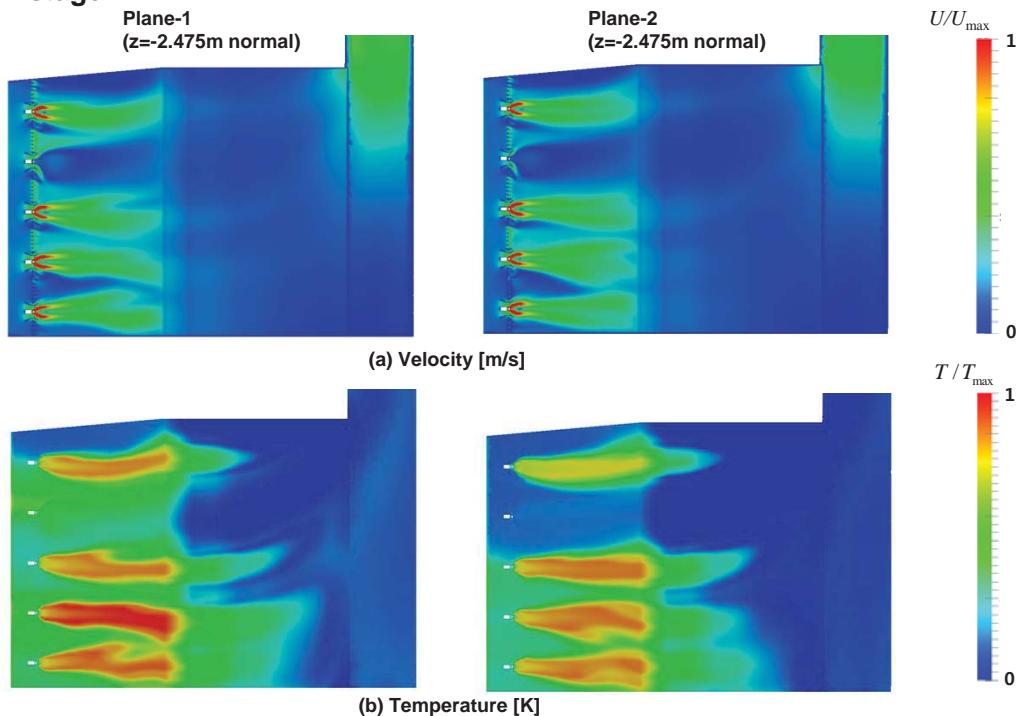


Combustion Laboratory POSTECH

# Industrial Furnace - Heat Recovery Steam Generator

## Results

### ▪ 2<sup>nd</sup> stage



# Industrial Furnace - Heat Recovery Steam Generator

## Results

### ▪ Comparison with the result for FLUENT

| Case        | Units                            | Fluent | OpenFOAM | Measured / Performance | Ratio  |
|-------------|----------------------------------|--------|----------|------------------------|--------|
|             |                                  | Outlet | Outlet   | Outlet                 | Fluent |
| Temperature | K                                |        |          | 0.96                   | 1.12   |
|             |                                  |        |          | 0.94                   | 0.92   |
|             |                                  |        |          | 1.05                   | 1.13   |
|             |                                  |        |          | 1.07                   | 0.99   |
| Composition | mass fraction                    |        |          | 1.00                   | 1.00   |
|             |                                  |        |          | 0.99                   | 0.97   |
|             |                                  |        |          | 0.49                   | 0.75   |
|             | ppm vd<br>@actual O <sub>2</sub> |        |          | 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

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

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

#### Transport equation of progress variable

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

Modified for partially premixed combustion

$$\tilde{c} = \frac{\tilde{Y}_{fu} - \tilde{Y}_f}{\tilde{Y}_{fu} - \tilde{Y}_{fb}}$$
$$\tilde{\omega}_c = \rho_u S_L \Xi |\nabla c|$$

#### Unburned state

$$\tilde{\Phi}_u(\tilde{t}, \tilde{x}, t) = \Phi_0 + \tilde{k}(\Phi_f - \Phi_0)$$

#### Burned state

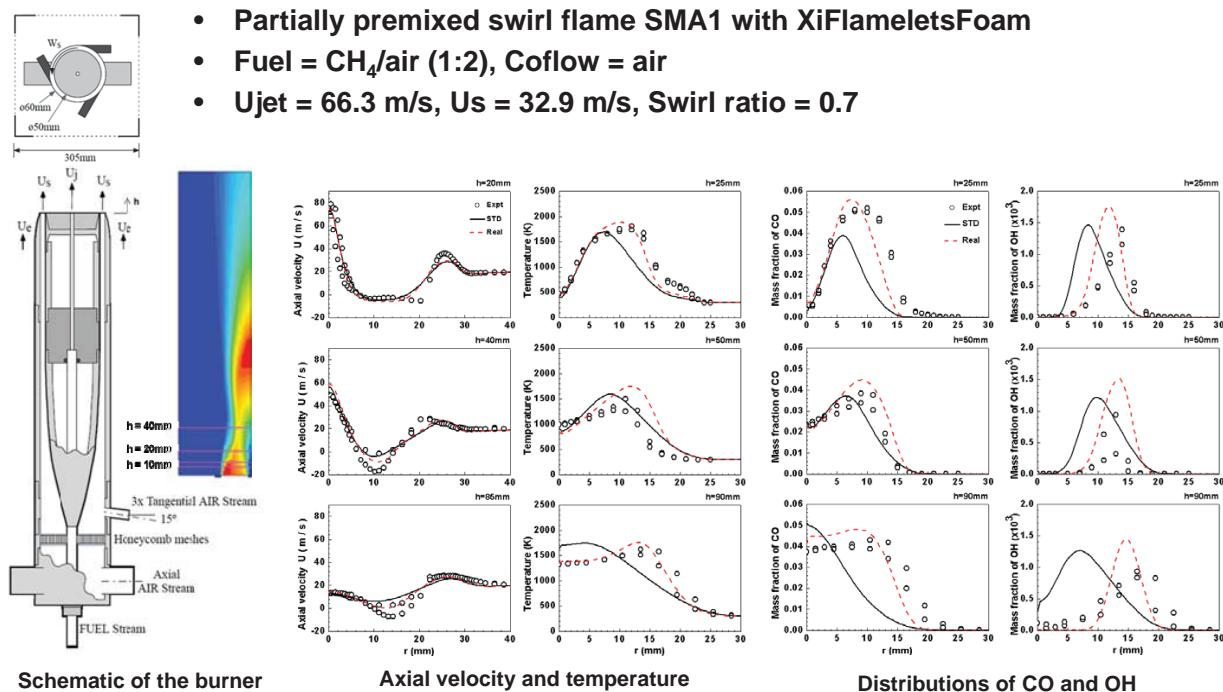
$$\tilde{\Phi}_b(\tilde{t}, \tilde{x}, \tilde{x}, t) = \iint \Phi(f, \chi) P(f, \chi) df d\chi$$

#### Mean conserved scalar

$$\tilde{\Phi}(\tilde{t}, \tilde{x}, \tilde{x}, \tilde{x}, t) = (1 - \tilde{\delta})\tilde{\Phi}_u(\tilde{t}, \tilde{x}, t) + \tilde{\delta}\tilde{\Phi}_b(\tilde{t}, \tilde{x}, \tilde{x}, t)$$

# Basic Flame – Sydney Swirl Flame SMA1

## Results



Combustion Laboratory POSTECH Pohang University of Science and Technology

# Industrial Furnace – 5MWe Micro Gas Turbine

## Case description

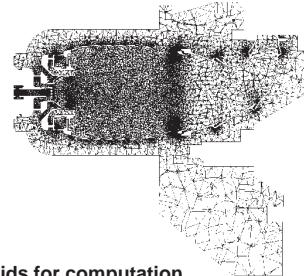
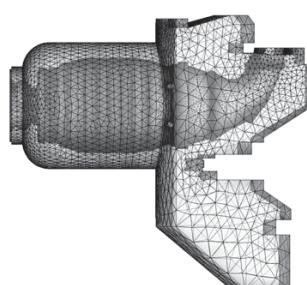
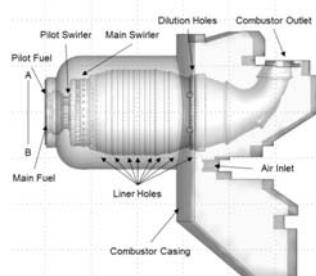
- Reverse-flow semi silo type with compressed air entering through pilot and main nozzles
- Two coaxial annular nozzles with radial swirlers
- Fuel is injected to be partially premixed with air in both nozzles

## 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)

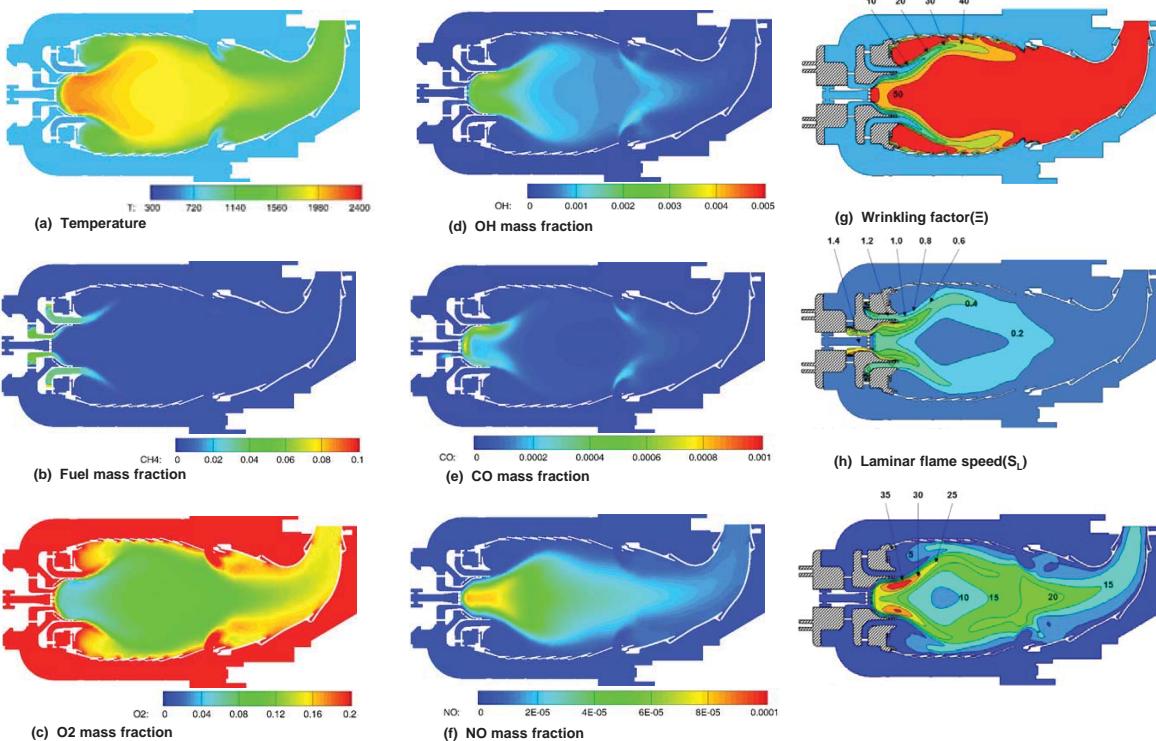


Grids for computation

Combustion Laboratory POSTECH Pohang University of Science and Technology

# Industrial Furnace – 5MWe Micro Gas Turbine

## Results

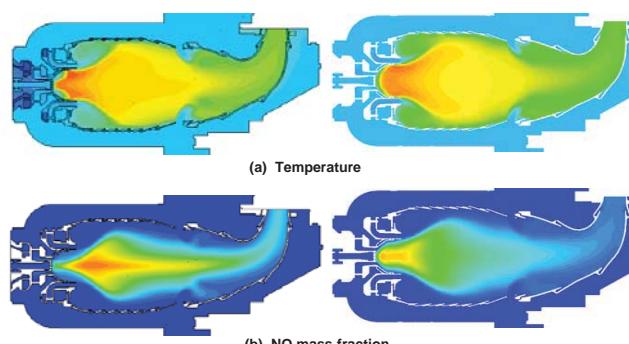


Combustion Laboratory POSTECH

# 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%, O<sub>2</sub> < 6%
  - reasonable order of degree for CO



| Specification               | Measured | PCFM<br>(STAR-CCM+) | XiFlamelet<br>(OpenFOAM) |
|-----------------------------|----------|---------------------|--------------------------|
| Temperature                 |          |                     |                          |
| O <sub>2</sub>              |          |                     |                          |
| CO<br>(@15%O <sub>2</sub> ) |          |                     |                          |

PCFM  
(STAR-CCM+)

XiFlameletsFOAM  
(OpenFOAM)

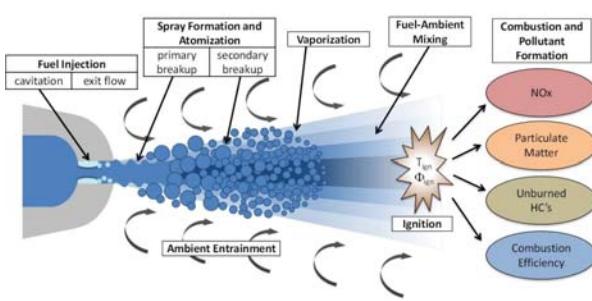
Measured and calculated scalars at the outlet

Combustion Laboratory POSTECH

# Spray Combustion Modeling

## Spray Combustion Modeling

### CMC equation with spray



$$\text{CMC} \quad \frac{\partial Q_\eta}{\partial t} = \langle N | \eta \rangle \frac{\partial^2 Q_\eta}{\partial \eta^2} + \langle w_\eta | \eta \rangle$$

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

Assumed beta-function PDF

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

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

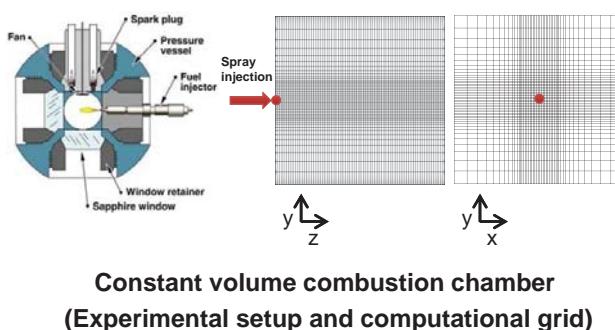
$$\text{Mixture fraction} \quad \frac{\partial(\bar{\rho}\tilde{\xi})}{\partial t} + \nabla \cdot (\bar{\rho}\tilde{\mathbf{u}}\tilde{\xi}) = \nabla \cdot \left[ \frac{\mu_t}{Sc_{\tilde{\xi}}} \nabla \tilde{\xi} \right] + \bar{\rho}\tilde{s}_{\tilde{\xi}}$$

$$\text{Mixture fraction variance} \quad \frac{\partial(\bar{\rho}\tilde{\xi}^{\prime\prime 2})}{\partial t} + \nabla \cdot (\bar{\rho}\tilde{\mathbf{u}}\tilde{\xi}^{\prime\prime 2}) = \nabla \cdot \left[ \frac{\mu_t}{Sc_{\tilde{\xi}^{\prime\prime 2}}} \nabla \tilde{\xi}^{\prime\prime 2} \right] + \frac{2\mu_t}{Sc_{\tilde{\xi}^{\prime\prime 2}}} (\nabla \tilde{\xi})^2$$

-  $2\bar{\rho}\tilde{\xi}^{\prime\prime} s_{\tilde{\xi}} (1 - \tilde{\xi}) - \bar{\rho}\tilde{\xi}^{\prime\prime} s_{\tilde{\xi}} - \bar{\rho}\tilde{\chi}$

## 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/>



### Injector Specification

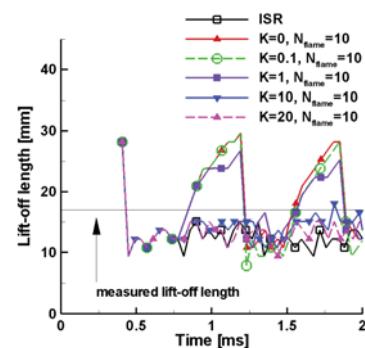
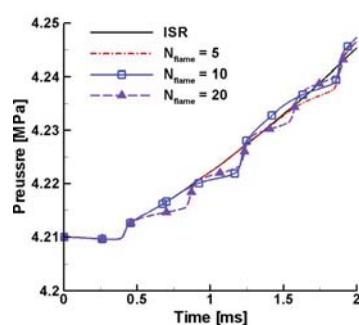
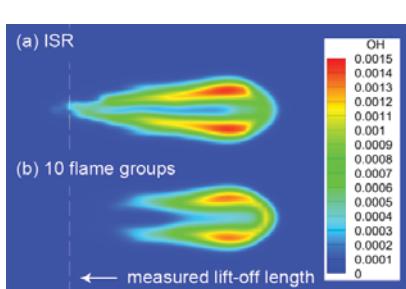
| Parameters                    | Values               |
|-------------------------------|----------------------|
| Injector type                 | Common-rail injector |
| Nozzle outlet diameter [mm]   | 0.1                  |
| Nozzle K factor               | 1.5                  |
| Nozzle shaping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

### Simulation Cases (n-heptane spray)

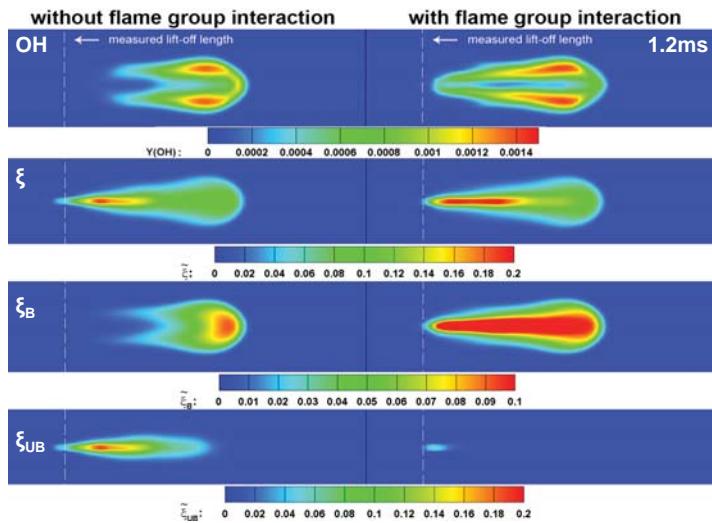
| Case | O <sub>2</sub> [%] | Ambient Temperature [K] | Ambient Density [kg/m <sup>3</sup> ] | Measured Lift-off Length [mm] |
|------|--------------------|-------------------------|--------------------------------------|-------------------------------|
| 1    | 21                 | 1000                    | 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                          |

## 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

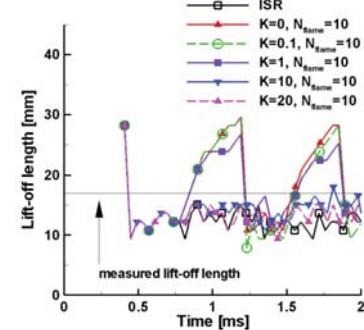


## Results



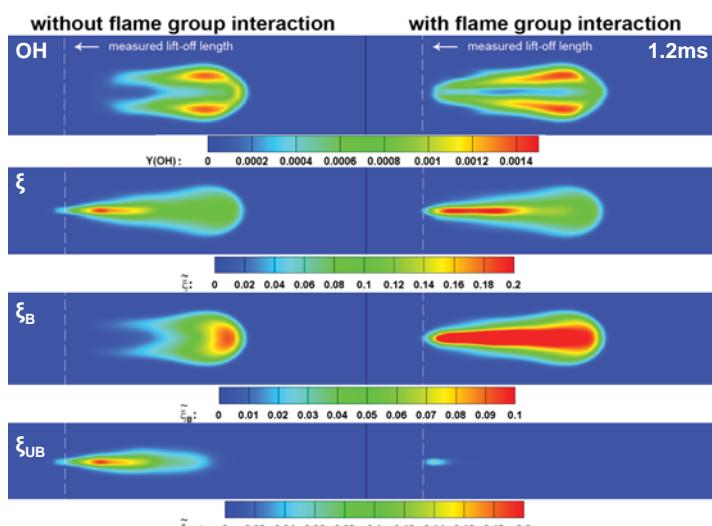
$$\frac{\partial(\bar{\rho}\tilde{\xi}_j)}{\partial t} + \nabla \cdot (\rho\tilde{\mathbf{v}}\tilde{\xi}_j) = \nabla \cdot \left[ \frac{\mu_t}{Sc_{\tilde{\xi}_j}} \nabla \tilde{\xi}_j \right] + \bar{\rho}\tilde{s}_{\xi,j} + \bar{\rho}\tilde{\xi}_j K \frac{(1-\tilde{c})}{\tau_t} \quad \text{for the burned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

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



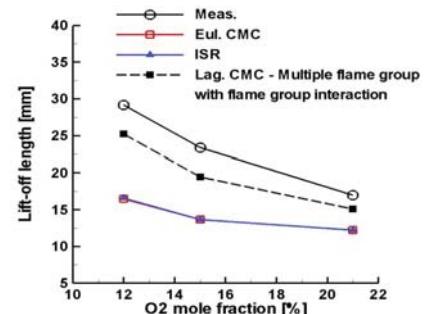
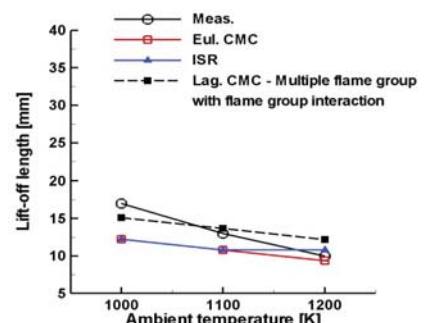
| Case                                  | Lift-off length [mm] |
|---------------------------------------|----------------------|
| Measured                              | 17.0                 |
| ISR                                   | 12.2                 |
| K=0 (without flame group interaction) | 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                 |

## Results



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

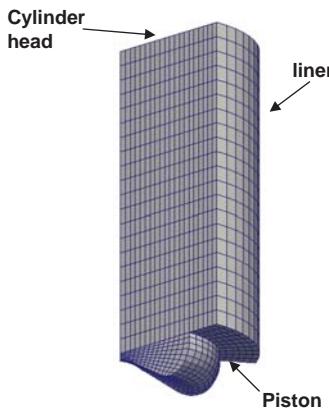
$$\frac{\partial(\bar{\rho}\tilde{\xi}_j)}{\partial t} + \nabla \cdot (\rho\tilde{\mathbf{v}}\tilde{\xi}_j) = \nabla \cdot \left[ \frac{\mu_t}{Sc_{\tilde{\xi}_j}} \nabla \tilde{\xi}_j \right] + \bar{\rho}\tilde{s}_{\xi,j} - \bar{\rho}\tilde{\xi}_j K \frac{\tilde{c}}{\tau_t} \quad \text{for the unburned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



# 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

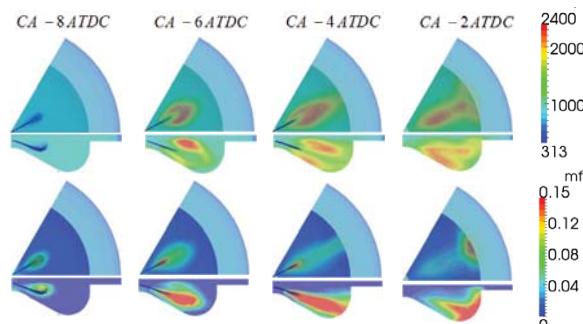
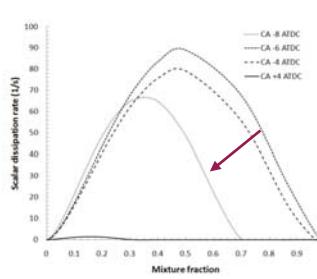
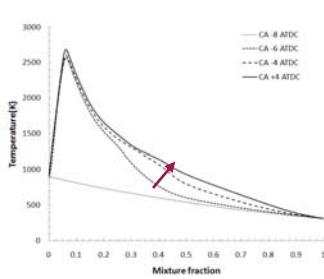
| Description                  | Specification                                 |
|------------------------------|-----------------------------------------------|
| Engine                       | Caterpillar 3401E                             |
| Engine speed (rpm)           | 821                                           |
| 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                     |

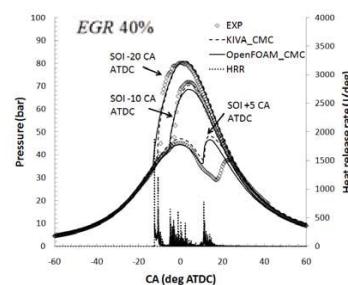
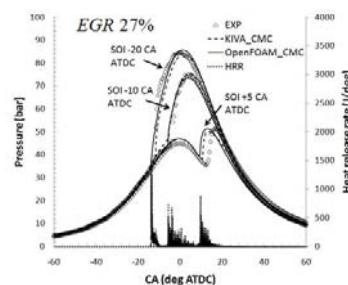
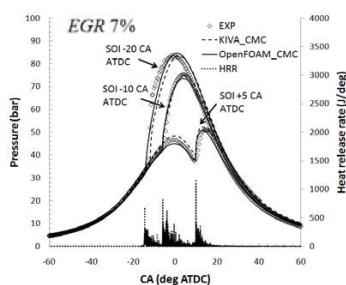
# Diesel Engine – ERC

## Results



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

Spatial distributions of the temperature and mean mixture fraction

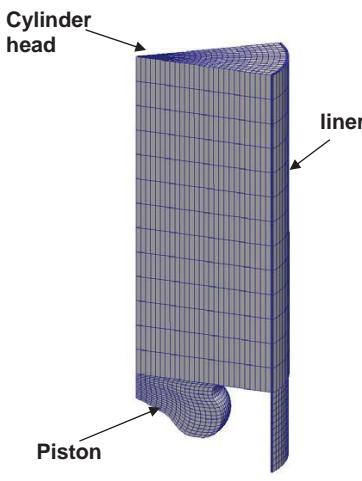


Pressure trace w.r.t different EGR (%)

# Diesel Engine – D1

## Case description

- Geometry



- ✓ D1 diesel engine
- ✓ Multiple fuel injection

- Engine specification

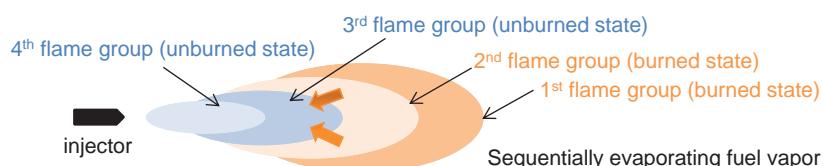
| Description                     | Specification |
|---------------------------------|---------------|
| Engine                          | D1            |
| Engine speed (rpm)              |               |
| Bore (mm) x Stroke (mm)         |               |
| Compression ratio               |               |
| Displacement (cm <sup>3</sup> ) |               |
| 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

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

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

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

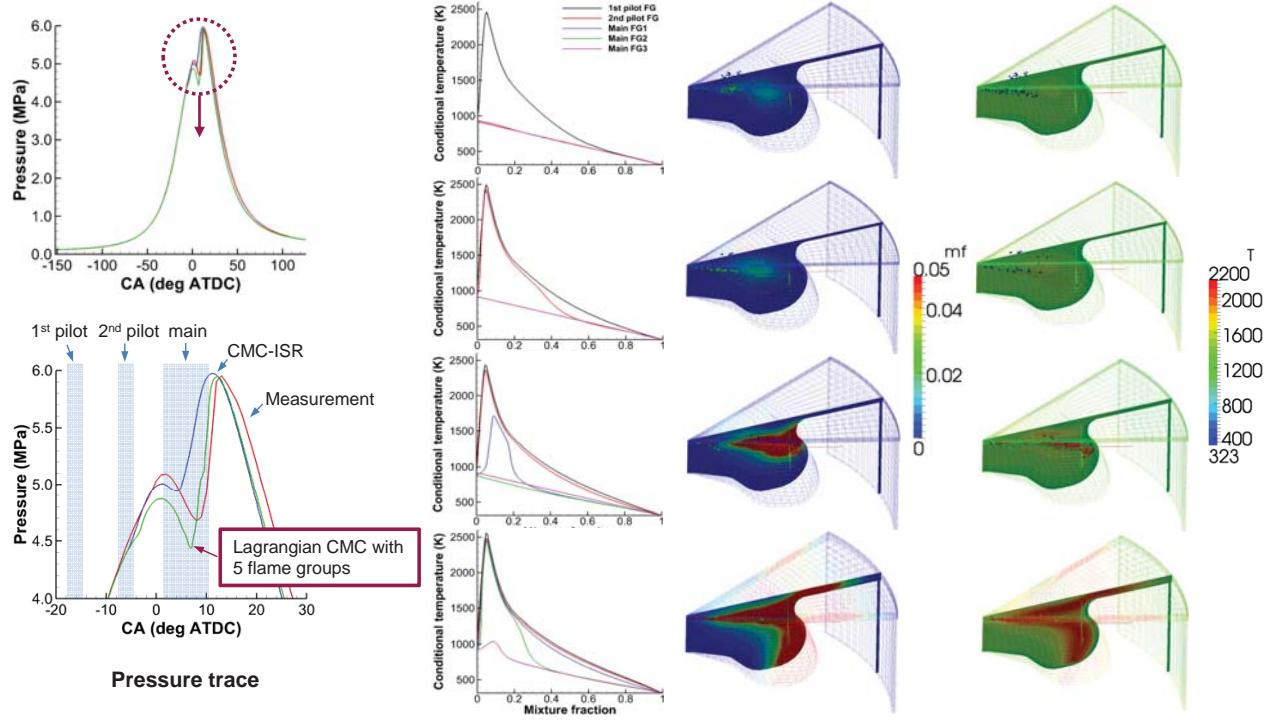
- Transport equation for fuel fraction of the j-th fuel group

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

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

# Diesel Engine - D1

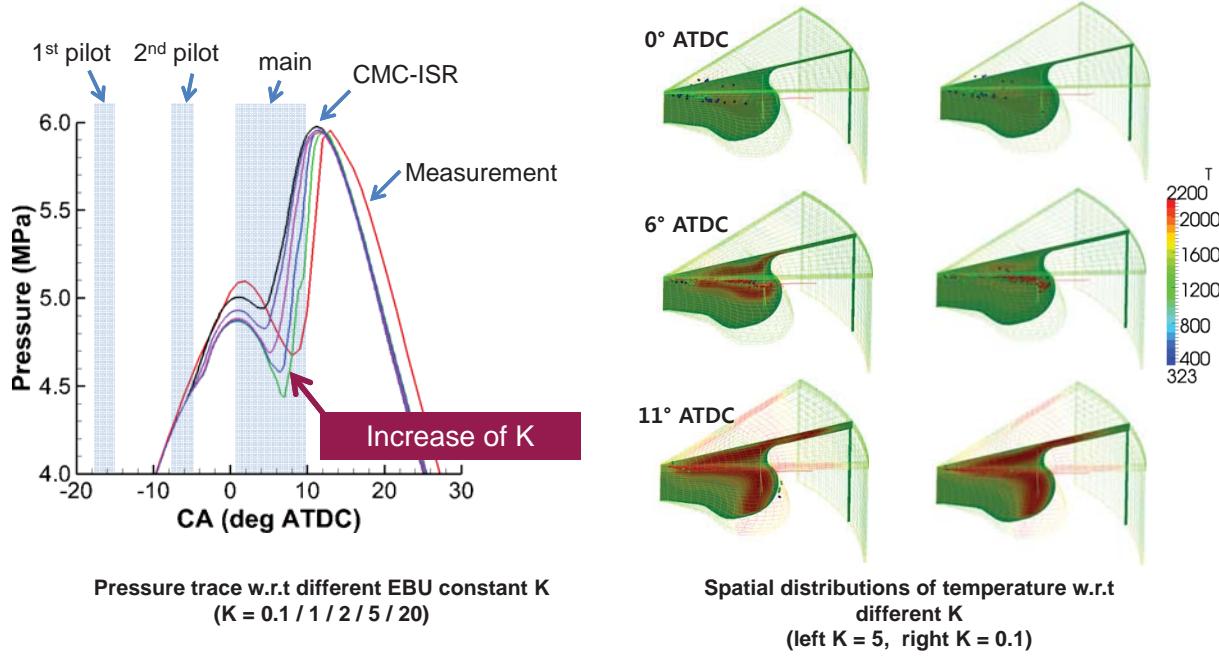
## Results



# Diesel Engine - D1

## Results

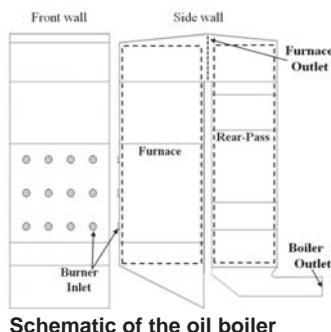
- The first peak pressure is increased by higher constant K which is corresponding to intensive flame propagation between flame groups



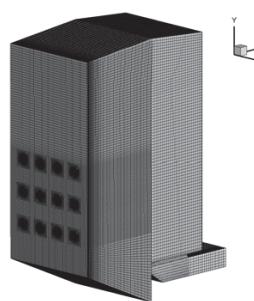
# Heavy Oil Furnace - Full Scale

## 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- $\epsilon$  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

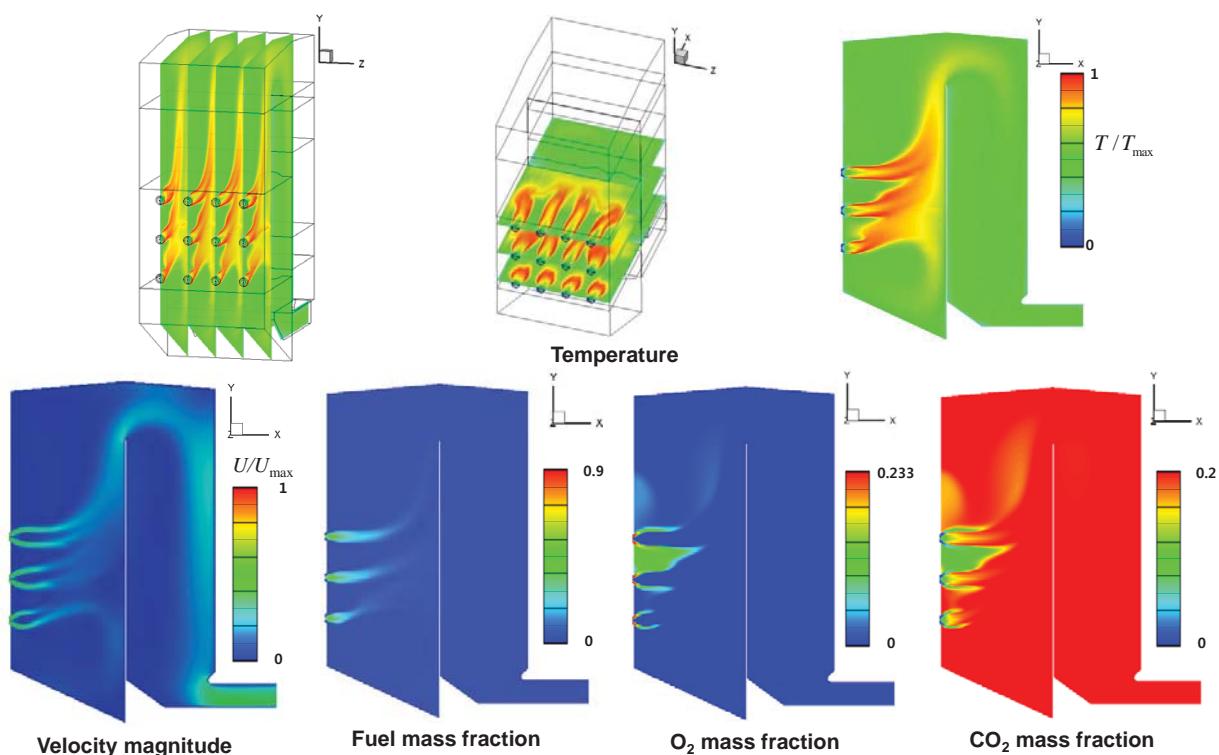


Grids for computation

Combustion Laboratory POSTECH

# Heavy Oil Furnace - Full Scale

## Results



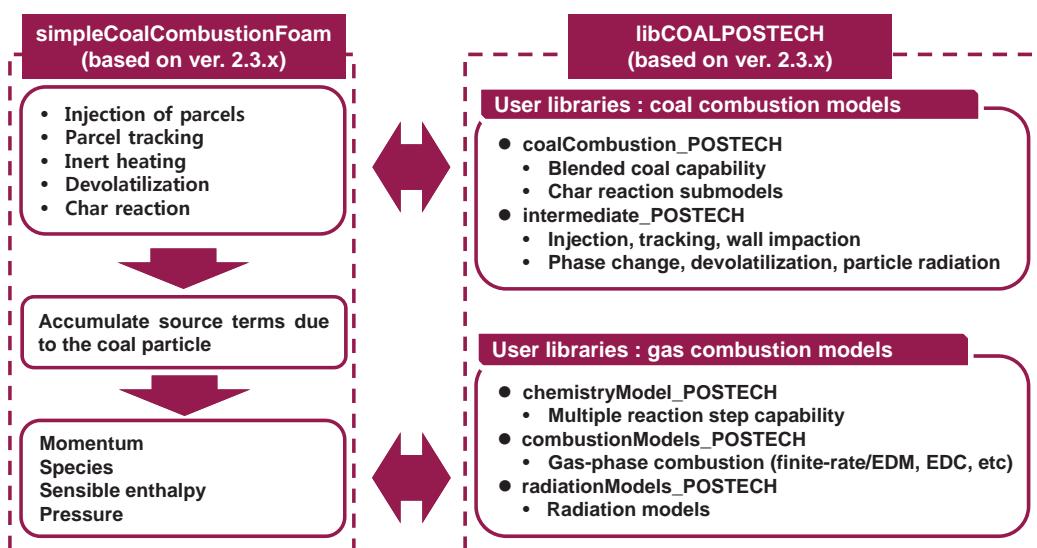
Combustion Laboratory POSTECH

# 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

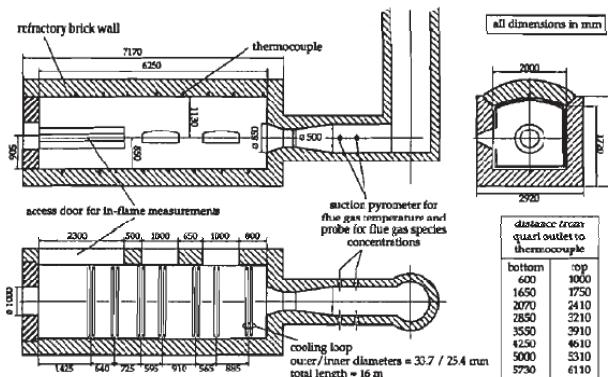


The schematic diagram of the simpleCoalCombustionFoam

# 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, O<sub>2</sub>, CO<sub>2</sub>, 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/200.

All dimensions in millimeters

burner outlet  
 $r(x) = -18.20x^3 + 129x^2 + 6.41x^{10} - 117$

calibration port

PC & TA

CA

Mass flow rate(kg/h) V(m/s) T(°C)

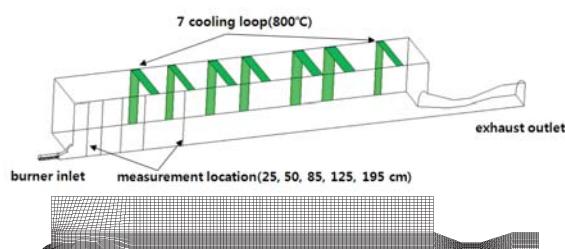
|                 | Mass flow rate(kg/h) | V(m/s)              | T(°C) |
|-----------------|----------------------|---------------------|-------|
| Pulverized Coal | 263                  | 23                  | 70    |
| Transport Air   | 421                  | 23                  | 70    |
| Combustion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

Combustion Laboratory POSTECH Pohang University of Science and Technology

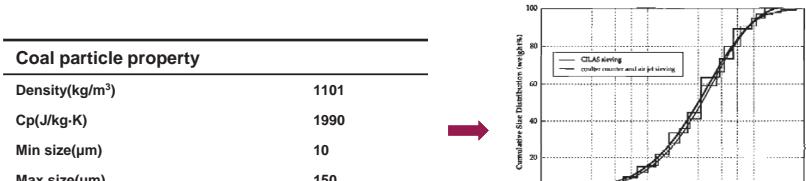
# IFRF MMF 5-2 Flame

## Case description

- Coal HCV(J/kg) = 2.79e+07 (AR)
- Vaporization T(K) = 773
- VM molecular weight(kg/kmol) = 45.6
- P1 radiation ( $\alpha$  : 0.1,  $E_{\text{particle}}$  = 0.8,  $S_{\text{particle}}$  = 0.5)
- Standard k-e model with 1st order upwind
- Stochastic particle dispersion
- 2-step eddy dissipation
- $\frac{1}{4}$  sector periodic B.C, 50,000 structured cells



| Proximate analysis (weight %)     |      | Coal particle property        |         |
|-----------------------------------|------|-------------------------------|---------|
| VM                                | 37   | Density(kg/m <sup>3</sup> )   | 1101    |
| Fixed Carbon                      | 52.5 | Cp(J/kg·K)                    | 1990    |
| Ash                               | 8.5  | Min size(μm)                  | 10      |
| Moisture                          | 2    | Max size(μm)                  | 150     |
| Ultimate analysis (weight %, daf) |      | Mean size(μm)                 | 60      |
| C                                 | 79.3 | Spread parameter              | 1.13    |
| H                                 | 4.7  | Swelling Index                | 1       |
| O                                 | 13.7 | Vaporization T(°C)            | 500     |
| N                                 | 2.3  | 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)

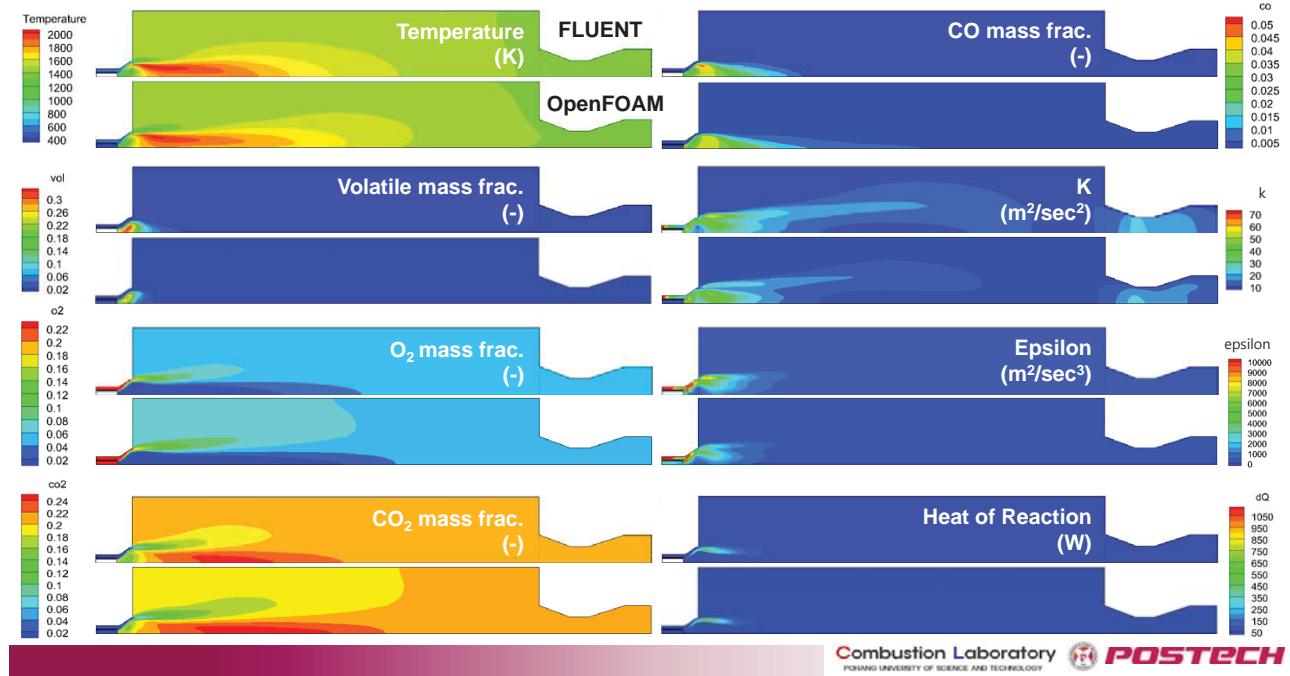
single rate devolatilization

Combustion Laboratory POSTECH Pohang University of Science and Technology

# IFRF MMF 5-2 Flame

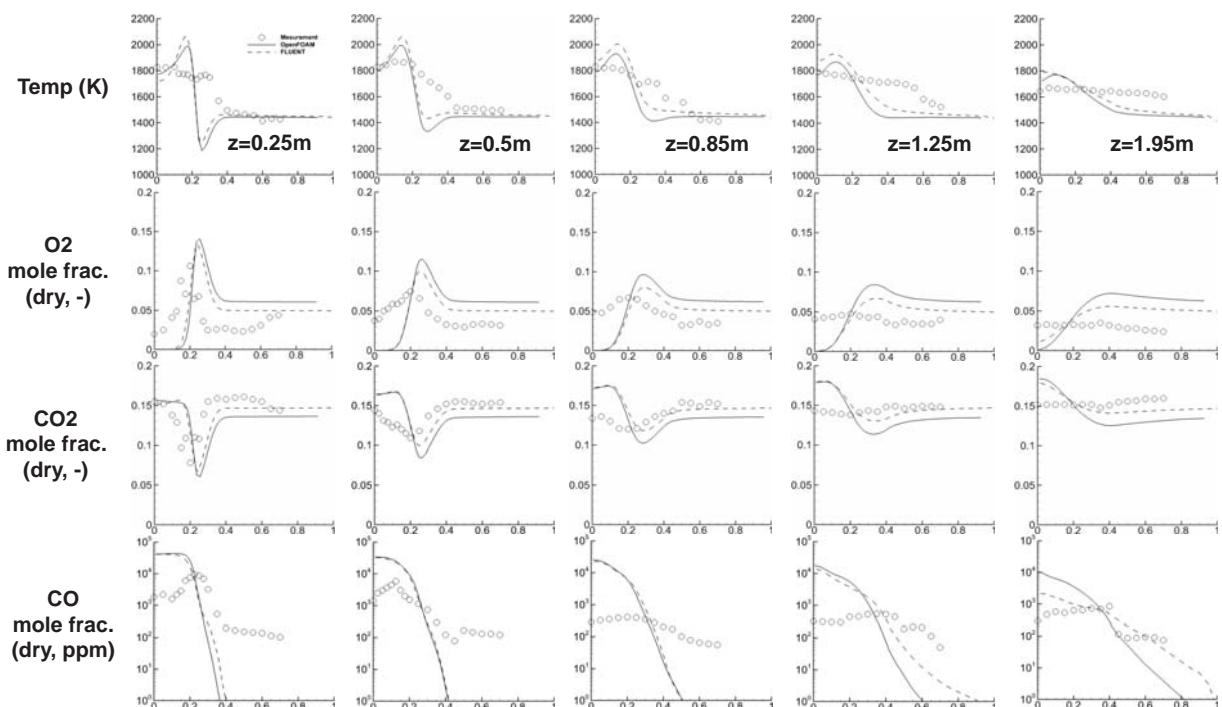
## Results

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



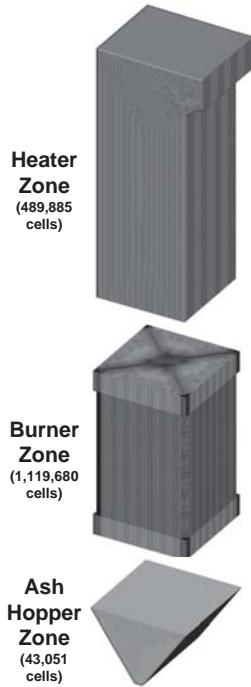
# IFRF MMF 5-2 Flame

## Results



# 500MWe Tangentially fired pulverized-coal boiler

## Case description



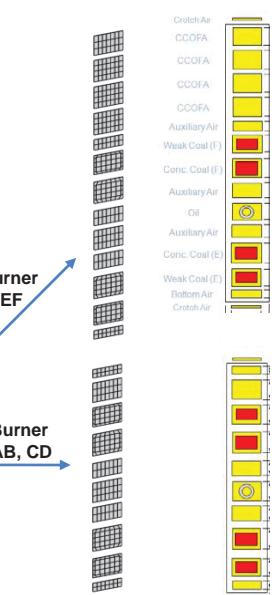
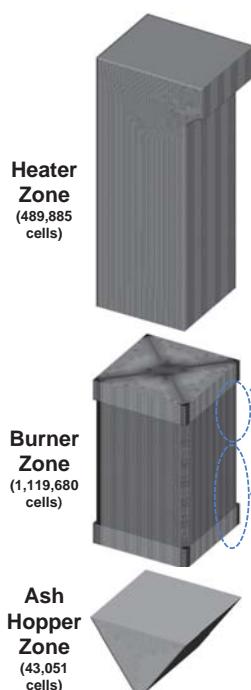
- 500MWe tangentially fired pulverized coal boiler
- Coal HCV(J/kg) = 2.68e+07 (AR)
- VM molecular weight(kg/kmol) = 41.3
- P1 radiation ( $\alpha : 0.1$ ,  $E_{\text{particle}} = 0.8$ ,  $S_{\text{particle}} = 0.1$ )
- Standard k-e model with 1st order upwind
- Stochastic particle dispersion
- 2-step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

| Proximate analysis (weight %)     |      | Coal particle property        |         |
|-----------------------------------|------|-------------------------------|---------|
| VM                                | 28   | Density(kg/m <sup>3</sup> )   | 1184    |
| Fixed Carbon                      | 52   | Cp(J/kg·K)                    | 1960    |
| Ash                               | 15   | Min size(μm)                  | 10      |
| Moisture                          | 5    | Max size(μm)                  | 150     |
| Ultimate analysis (weight %, daf) |      | Mean size(μm)                 | 60      |
| C                                 | 82   | Spread parameter              | 1.13    |
| H                                 | 5.1  | Swelling Index                | 1       |
| O                                 | 10.3 | Vaporization T(°C)            | 500     |
| N                                 | 1.7  | Kinetics limited, A           | 6.7     |
| S                                 | 0.9  | Kinetics limited, Ea(MJ/kmol) | 113.82  |
|                                   |      | Single rate, A                | 3.6e+05 |
|                                   |      | Single rate, Ea(J/kmol)       | 3.8e+07 |

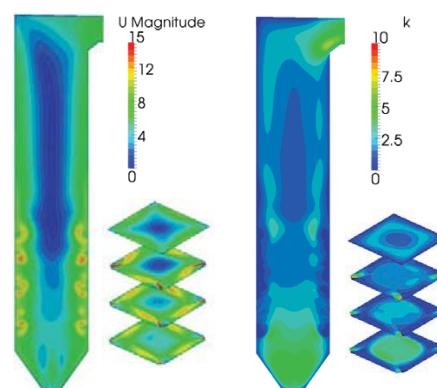
Combustion Laboratory POSTECH

# 500MWe Tangentially fired pulverized-coal boiler

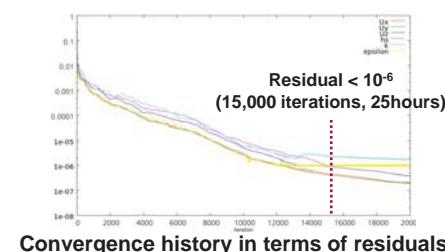
## Results



Burner configurations



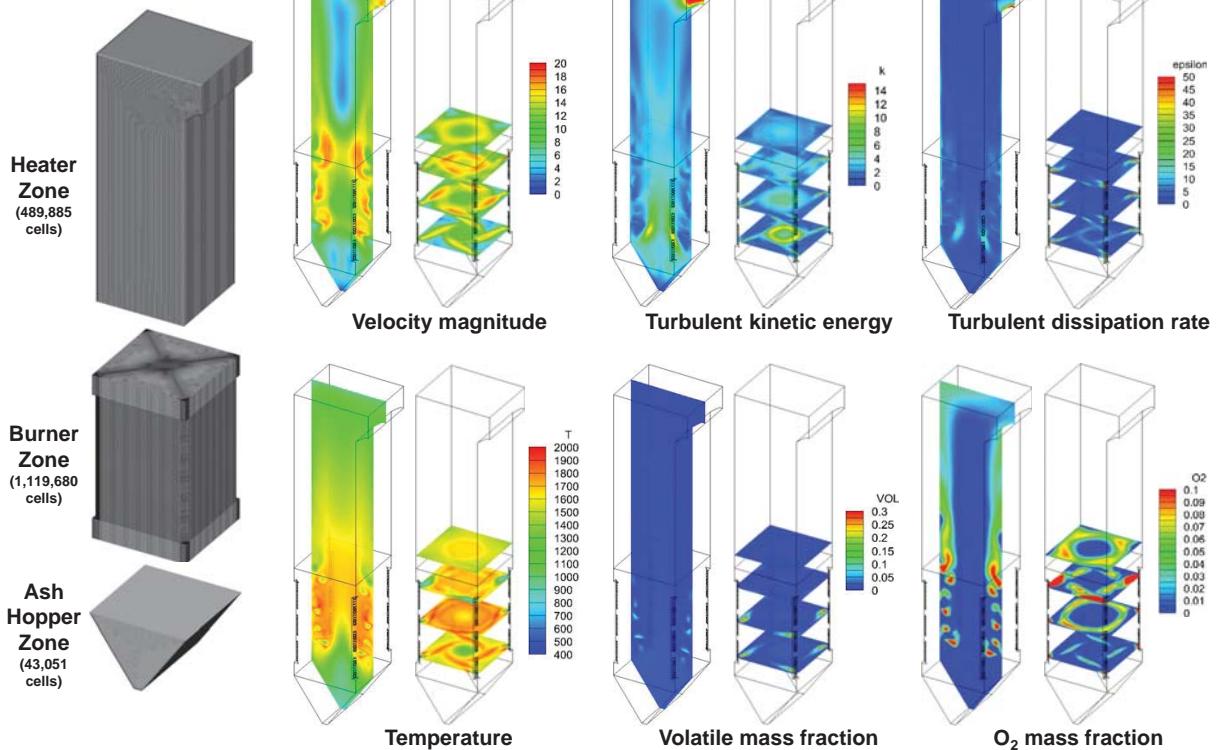
Distributions of velocity magnitude and turbulent kinetic energy



Combustion Laboratory POSTECH

# 500MWe Tangentially fired pulverized-coal boiler

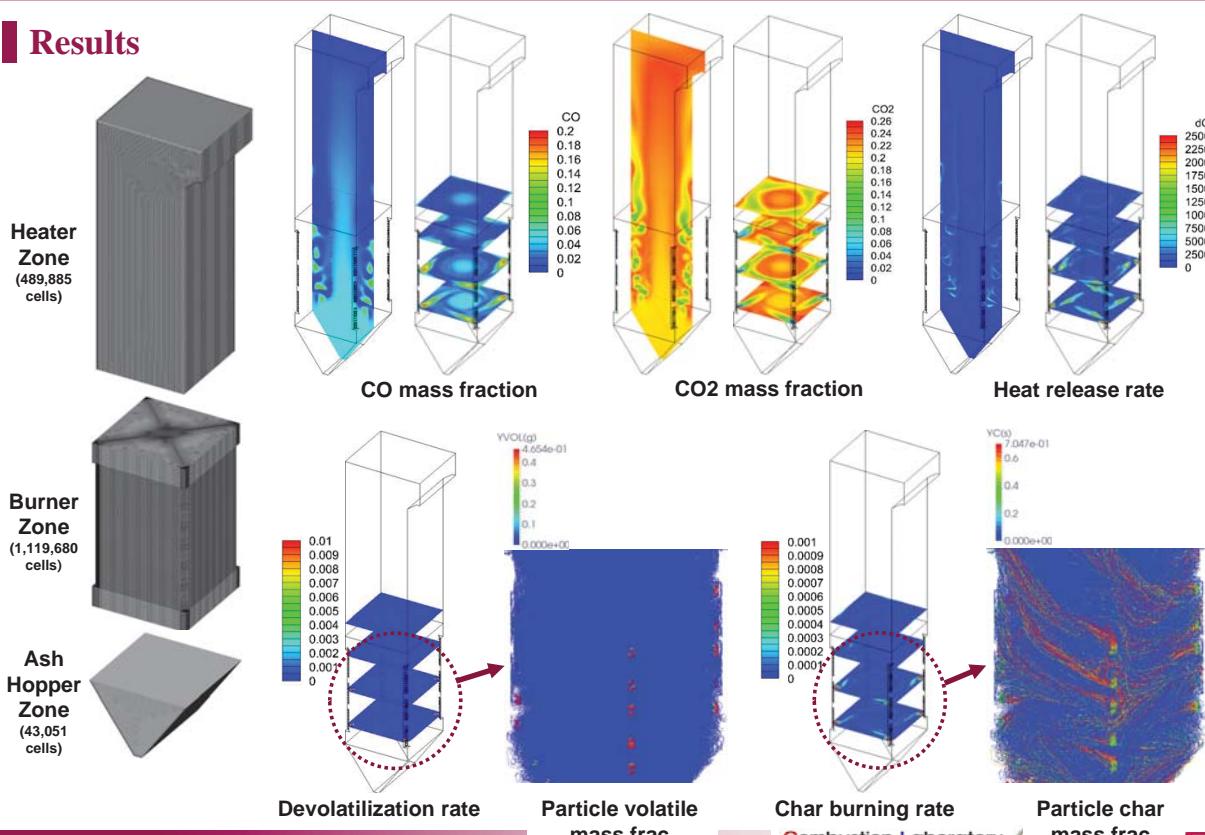
## Results



Combustion Laboratory POSTECH Pohang University of Science and Technology

# 500MWe Tangentially fired pulverized-coal boiler

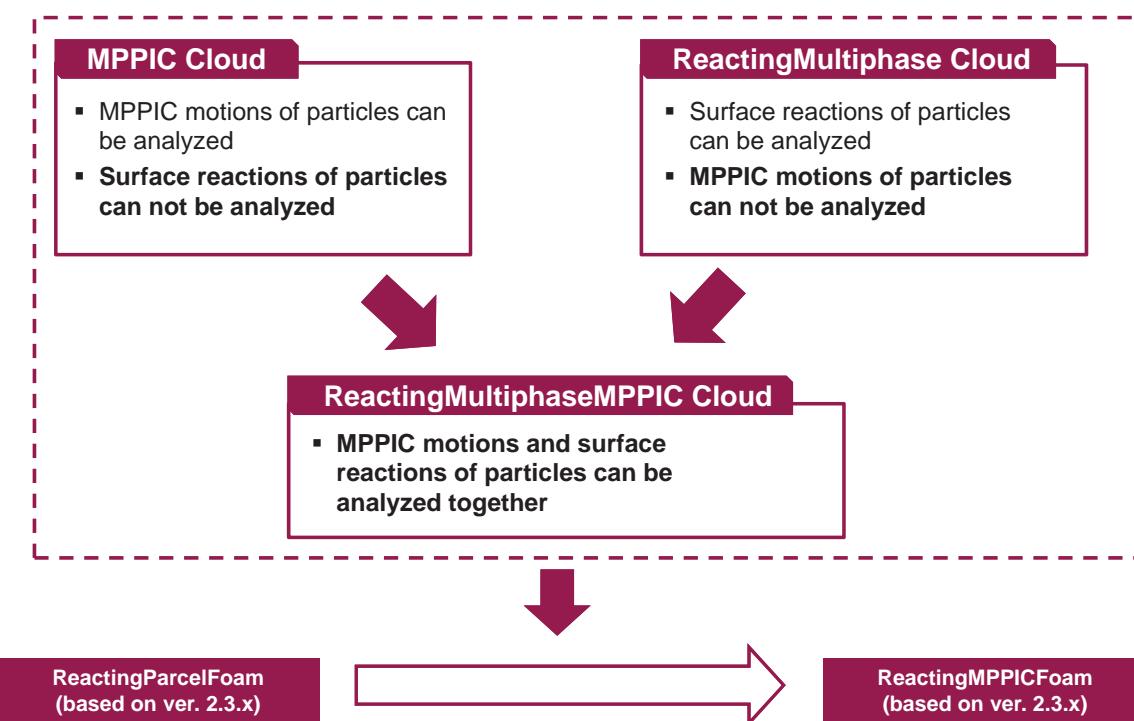
## Results



Combustion Laboratory POSTECH Pohang University of Science and Technology CH

# *Material Processing Furnace*

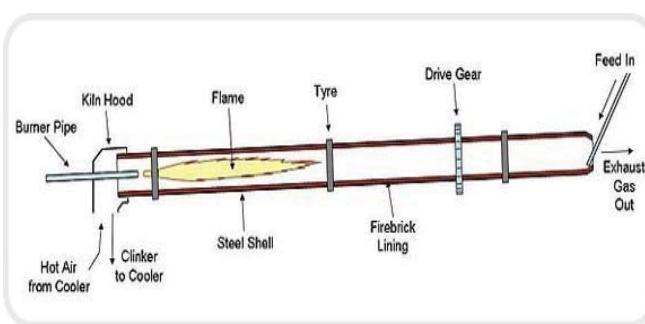
## ReactingMPPICFoam



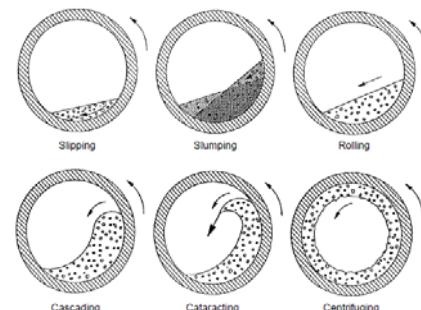
# Rotary Klin

## 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

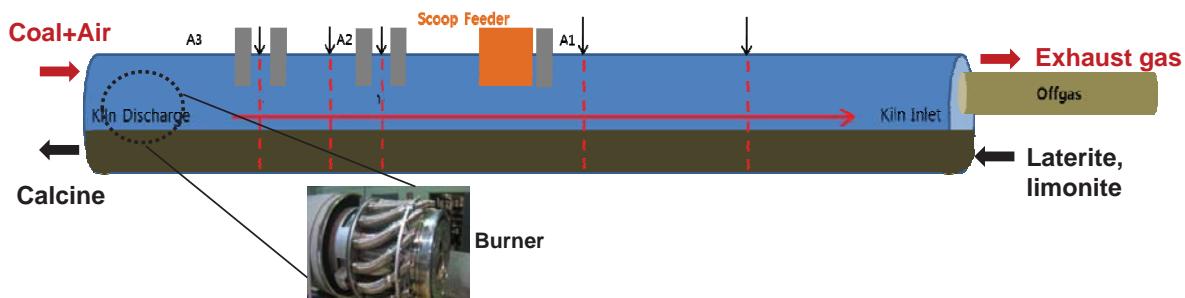


Different modes of operation in the transversal mixing plane

Combustion Laboratory POSTECH

# 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 <sub>2</sub>               | 39.8     |         |
| MgO                            | 22.7     |         |
| Al <sub>2</sub> O <sub>3</sub> | 1.2      |         |
| IL                             | 10.4     |         |
| Moisture                       | 20.8     |         |

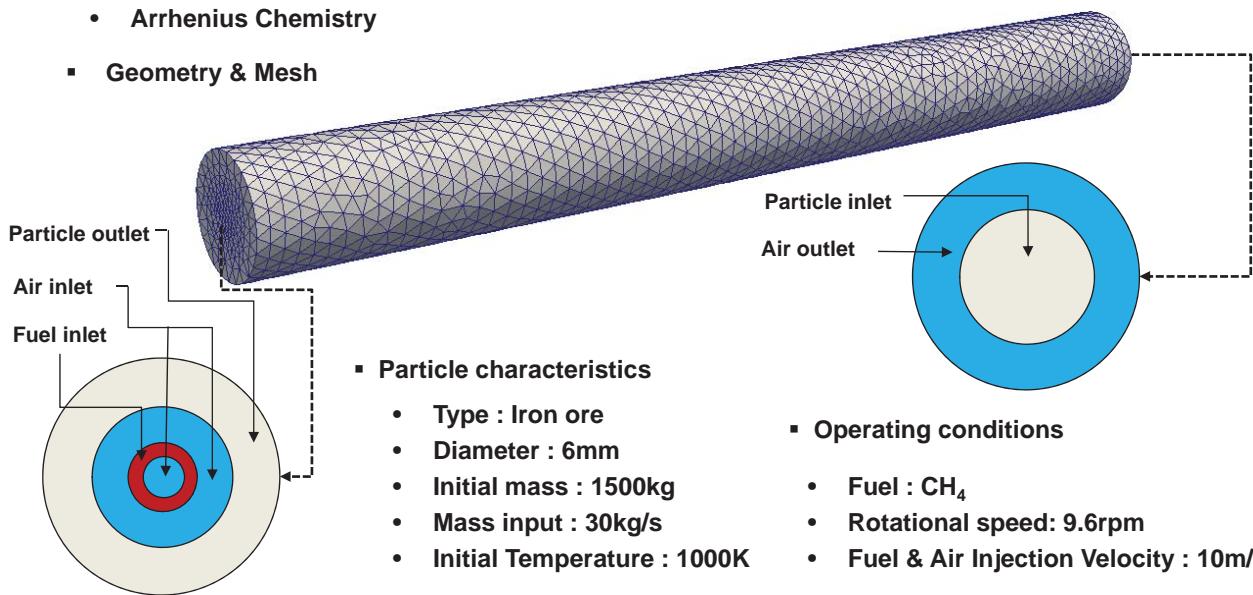
Composition of material

Combustion Laboratory POSTECH

# Rotary Klin

## Test Case

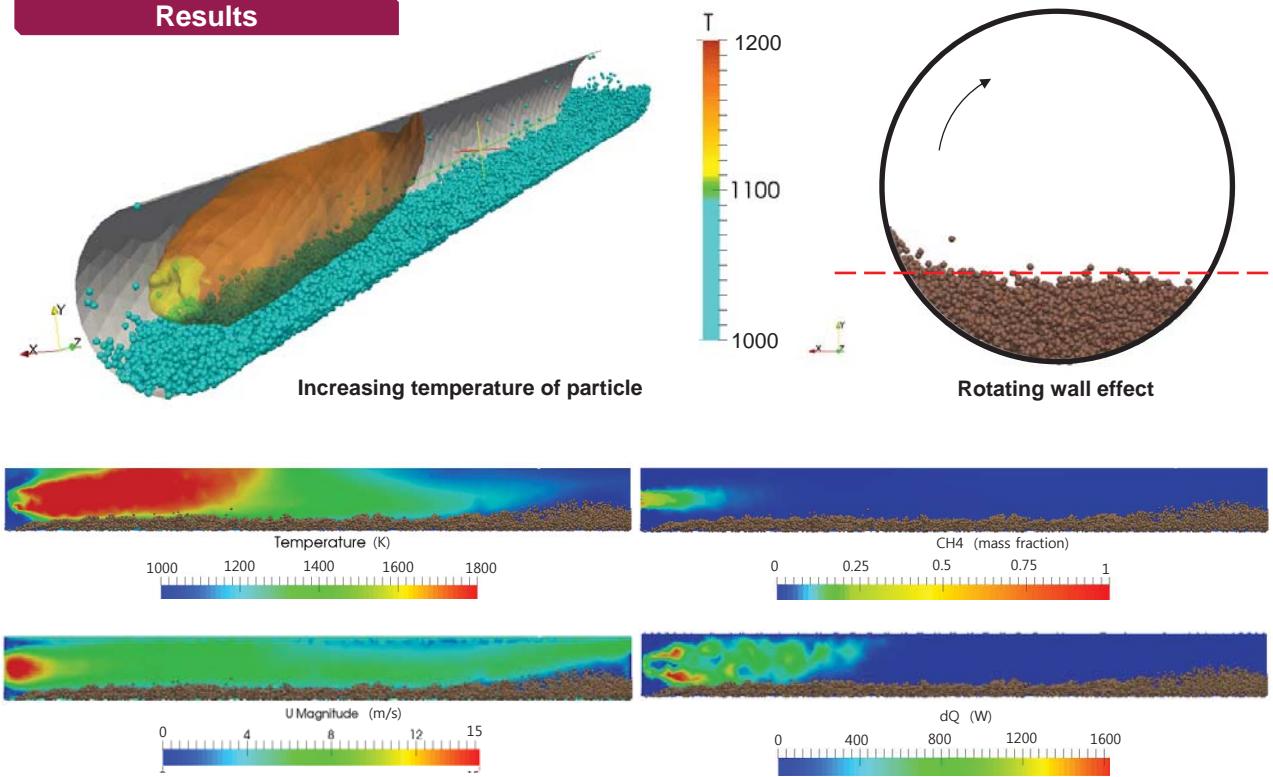
- Numerical Method
  - Eulerian(gas phase) - Lagrangian(solid phase) approach
  - MP-PIC (Multi Phase – Particle in Cell) method for particle motion
  - Arrhenius Chemistry
- Geometry & Mesh



Combustion Laboratory POSTECH

# Rotary Klin

## Results

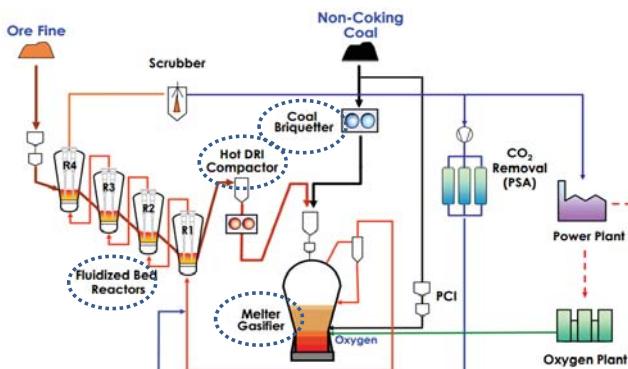


Combustion Laboratory POSTECH

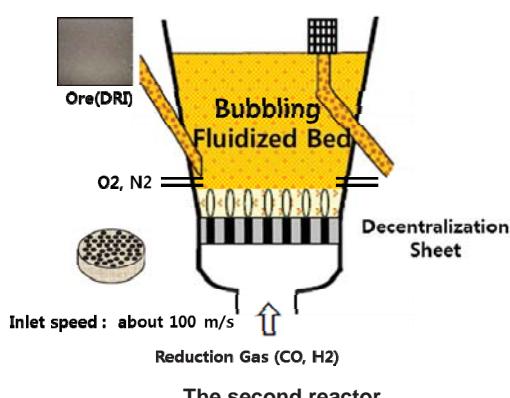
# FINEX: R2

## Case description

- A part of the iron-making process
- The biggest fluidized bed reactor in the world (12m toll)
- Multiphase flow: Gases(CO, H<sub>2</sub>, N<sub>2</sub> ...) – Particles (HCl) flow
- Chemical reaction: reducing process, non-premixed combustion  
Both homogeneous and heterogeneous reactions are included
- Some tricky phenomenon such as sticking around O<sub>2</sub> nozzle
- Wide size distribution of particles



Whole process flow of the FINEX

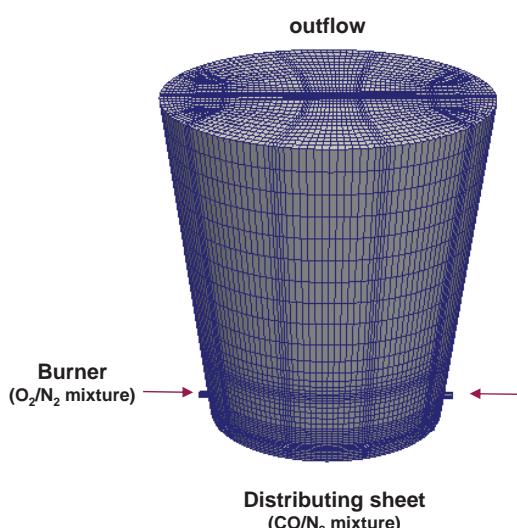


Combustion Laboratory POSTECH

# 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

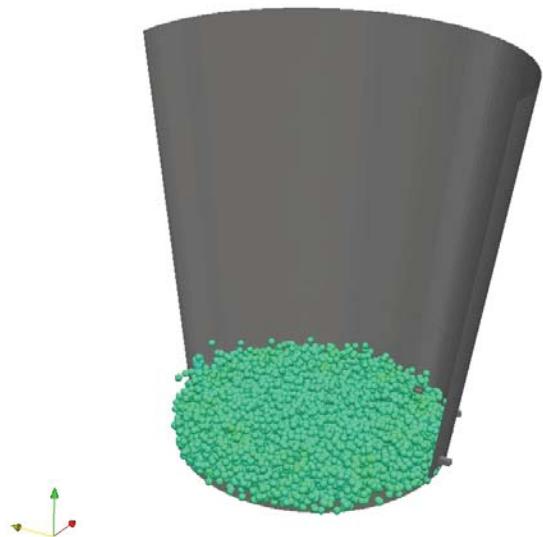


- Domain description
  - Fluidized bed reactor (particle I/O included)
  - Combustion (O<sub>2</sub> burner 10 pcs)
  - Distributing sheet
  - Height : 3m
- Particle characteristics
  - Type : Iron ore
  - Diameter : 1 mm (expectation)
  - Parcel Number : ~ 30000
  - Initial Temperature : 300K
- Operating conditions
  - CO/N<sub>2</sub> : 20 m/s through distributing sheet
  - O<sub>2</sub>/N<sub>2</sub> : 10 m/s at burner nozzle

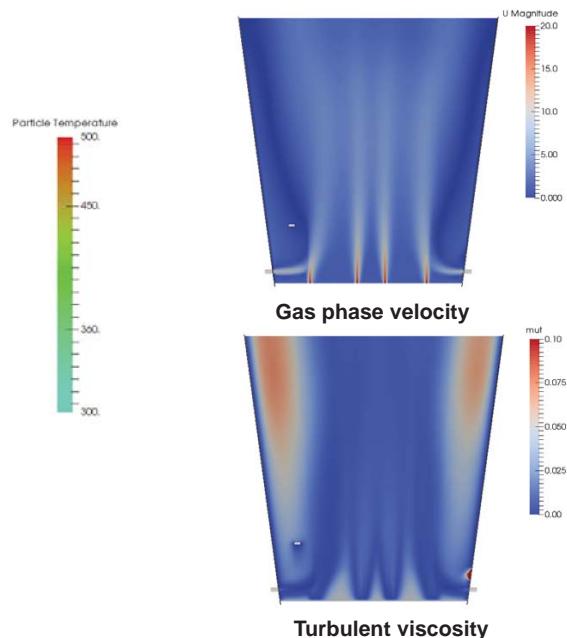
Combustion Laboratory POSTECH

# FINEX: R2

## Results



Increasing temperature of particle



Gas phase velocity

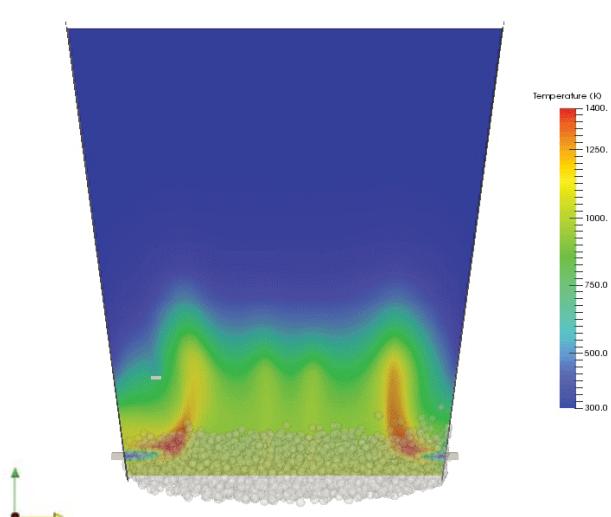
Turbulent viscosity

Combustion Laboratory

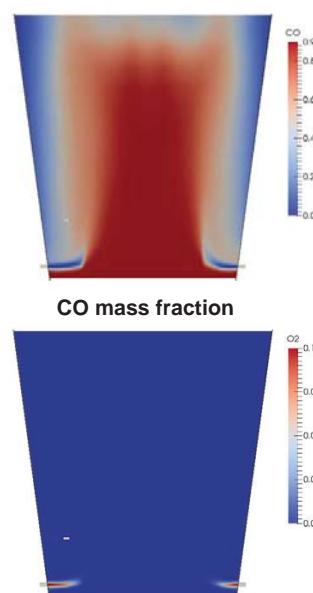
POSTECH

# FINEX: R2

## Results



Temperature contour



CO mass fraction

O<sub>2</sub> mass fraction

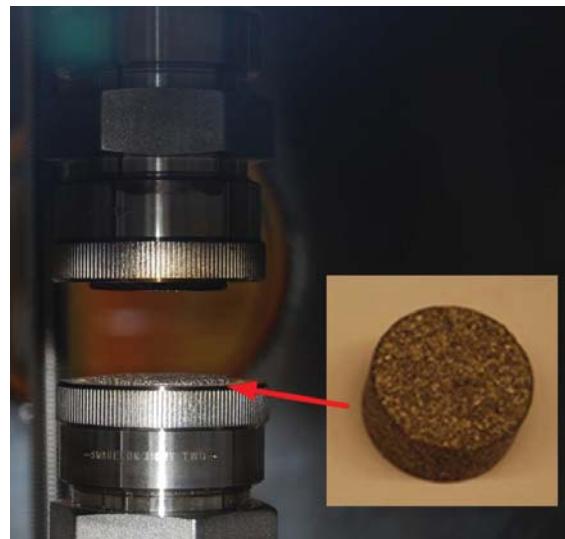
Combustion Laboratory

POSTECH

# 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<sub>4</sub> and O<sub>2</sub> 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



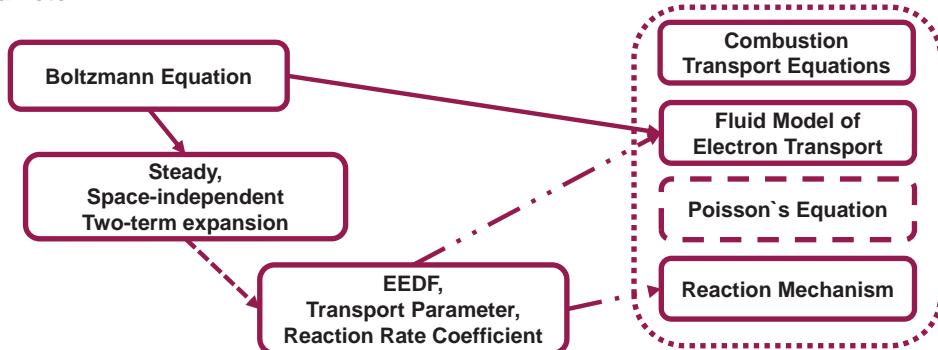
Counter flow burner with stainless steel porous electrodes (photograph)

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

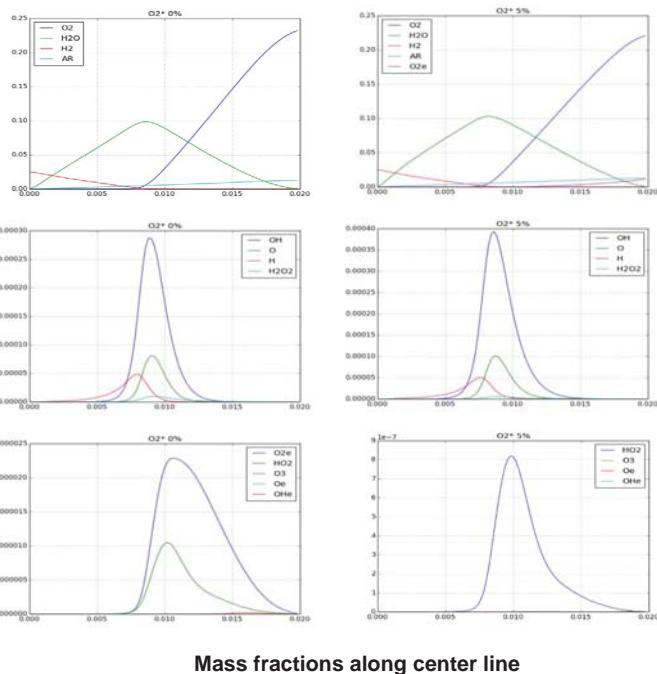
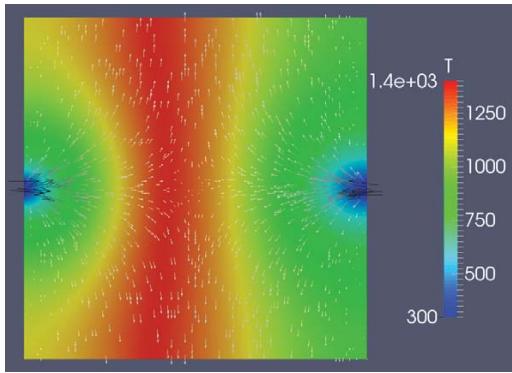


A Schematic diagram of plasma assisted combustion

# Plasma Assisted Combustion

## Results

- H<sub>2</sub> is diluted with N<sub>2</sub> ( $Y_{H_2} = 2.5\%$ )
- Oxidizer is air
- Discharge is applied before oxidizer nozzle
- 5% O<sub>2</sub> 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 H<sub>2</sub>-O<sub>2</sub> flames involving electronically-excited species O<sub>2</sub>(a<sup>1</sup>Δg), O(1D) and OH(2Σ<sup>+</sup>). Proceedings of the Combustion Institute, 32 II, pp. 3171-3179

Combustion Laboratory POSTECH

## 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, Fluidized 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.

Combustion Laboratory POSTECH

# ***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

### I 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)
);
```

parcels.evolve();

```
#include "rhoEqn.H"

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

**Need source term due to combustion**

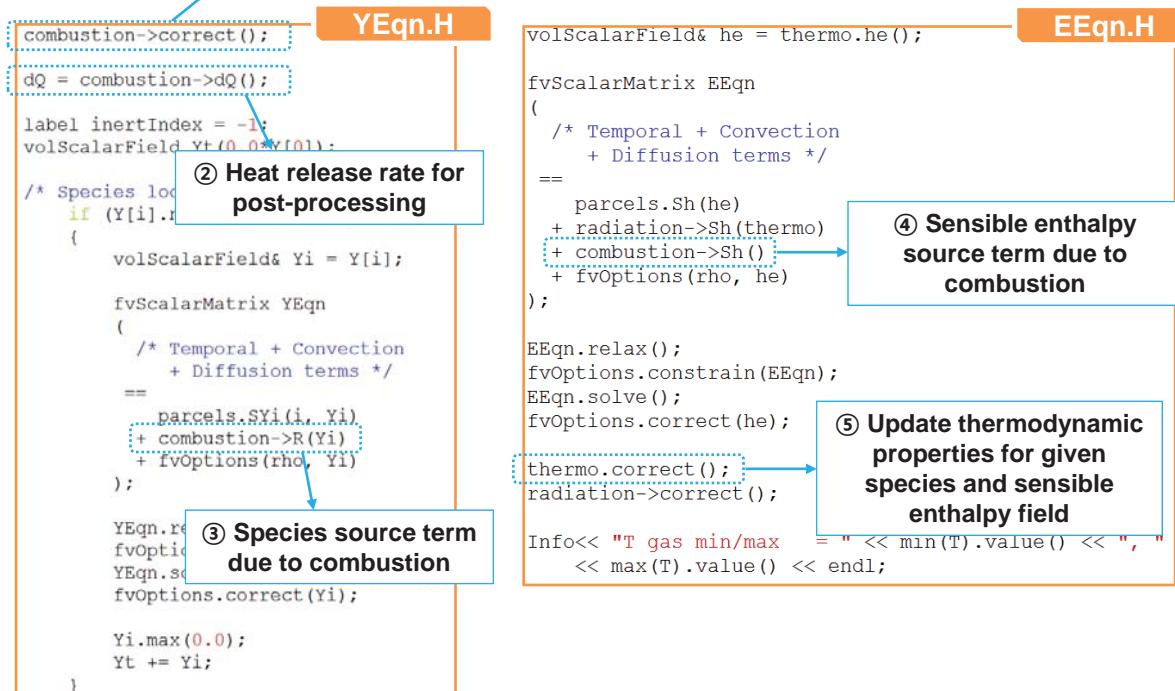
```
// --- Pressure corrector loop
while (pimple.correct())
{
    #include "pEqn.H"

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

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

## Structure of reactingParcelFoam

- Combustion-related source terms for sensible enthalpy equations
- Member functions for Y and hs Eqns (previously constructed) are located in src/combustionModels



## 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)

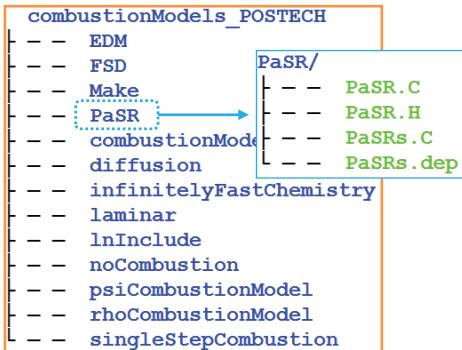
# Structure of PaSR model

## 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

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

src/combustionModels



combustionModels/Make/files

```

combustionModel/combustionModel.C
/* Some models in here */

diffusion/diffusions.C

infinitelyFastChemistry/ininitelyFastChemistrys.C

EDM/EDMs.C

PaSR/PaSRs.C (highlighted)

laminar/laminars.C

/* Some models in here */

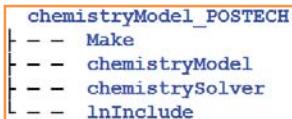
FSD/FSDs.C

noCombustion/noCombustions.C

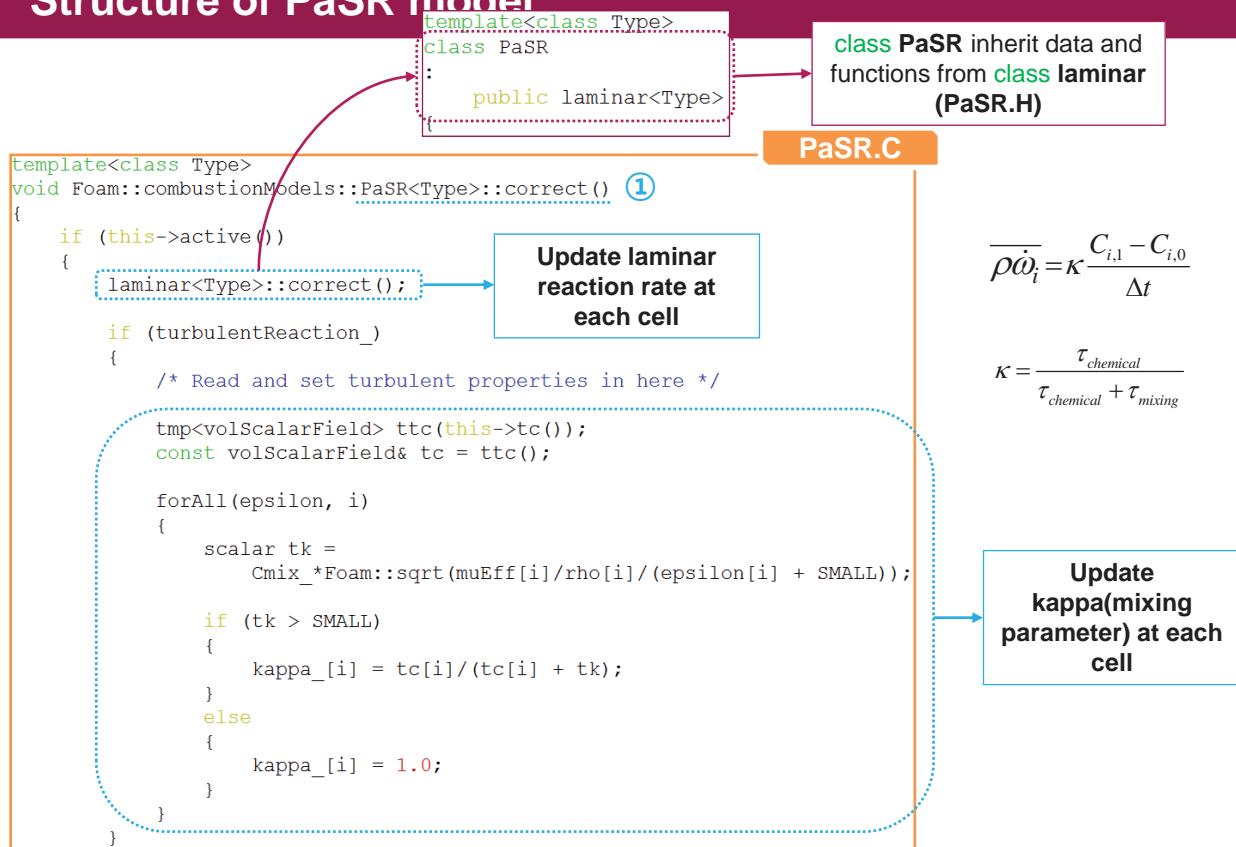
LIB = $(FOAM_USER_LIBBIN)/libcombustionModels_POSTECH

```

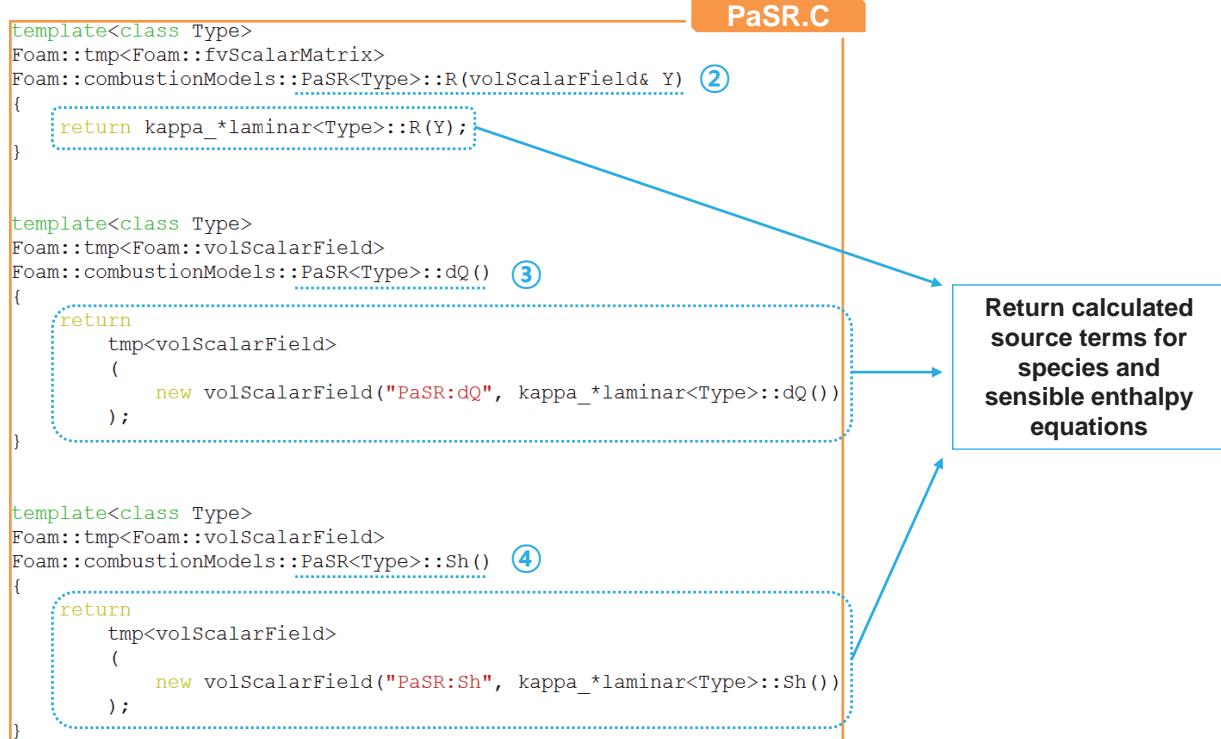
src/thermophysicalModel/chemistryModel



# Structure of PaSR model



## Structure of PaSR model



## Structure of PaSR model



### 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'  
3. Auto-pointer 'chemistryPtr\_' refers to above Object

## Structure of PaSR model

```

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_->update();
        }
    }
}

template<class CompType, class ThermoType>
template<class DeltaTTType>
Foam::scalar Foam::chemistryModel<CompType, ThermoType>::solve(const DeltaTTType& 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;
}

```

laminar.C

chemistryModel.C

Update laminar reaction rate at each cell

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

## Structure of PaSR model

```

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;
}

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();
    }

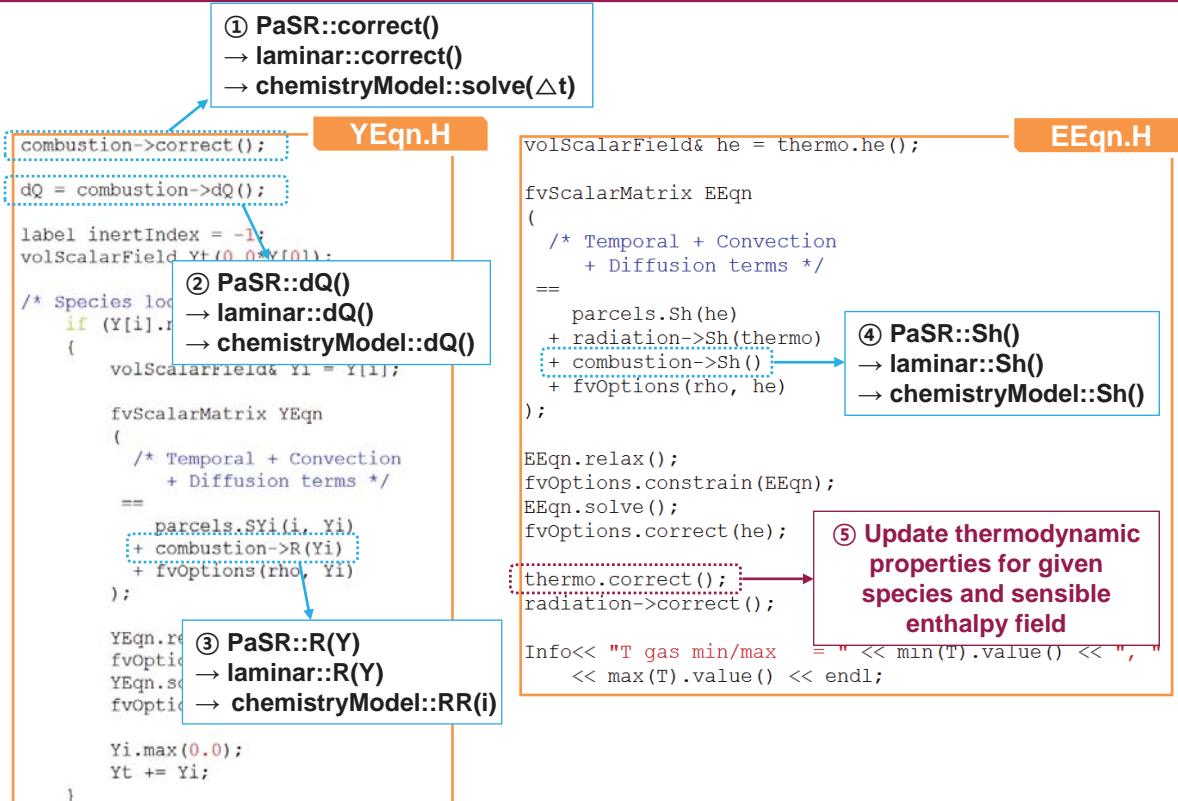
    return tSh;
}

```

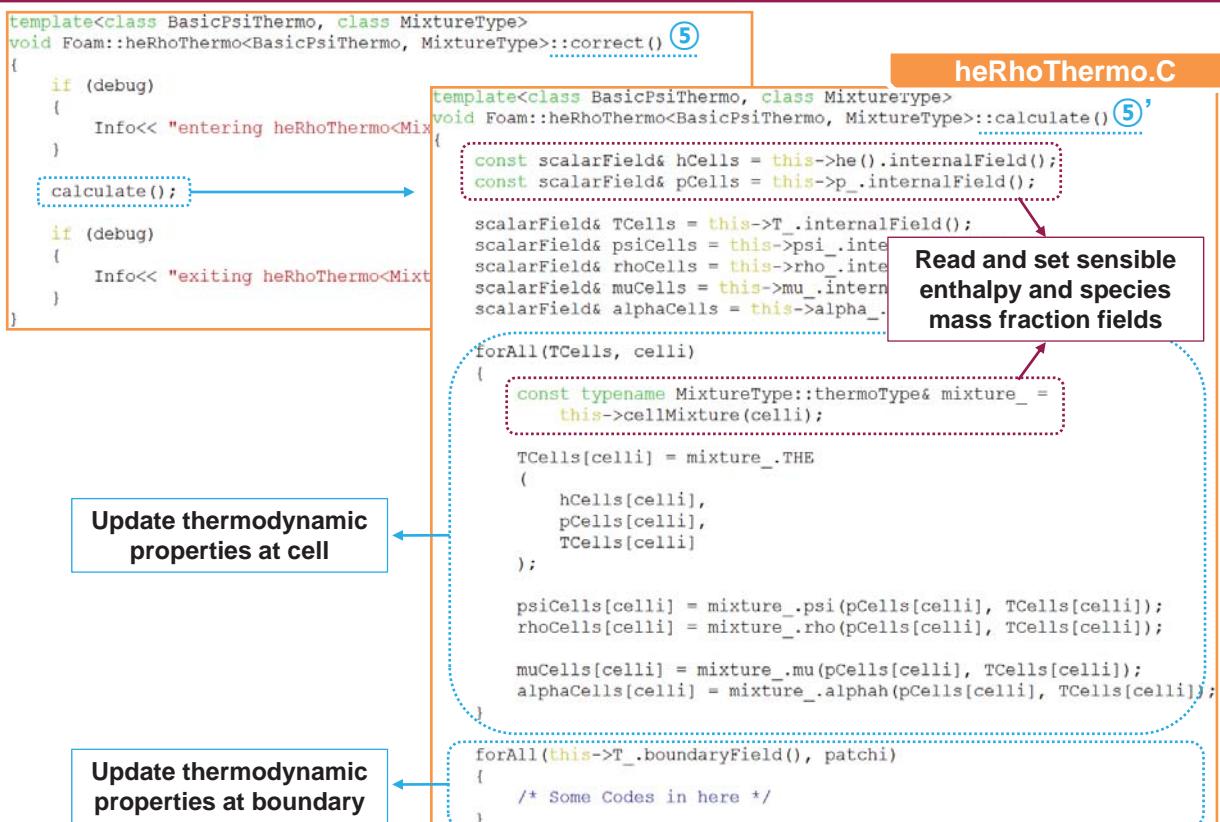
laminar.C

Return laminar reaction source terms at each cell

## Structure of PaSR model



## Structure of PaSR 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)

## How to Implement New Combustion Models (EDM)

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

$$\left. \begin{array}{l} R_f = A \cdot \bar{Y}_f (\varepsilon/k) \\ R_f = A(\bar{Y}_{O_2}/r_f)(\varepsilon/k) \\ R_f = A \cdot B \{\bar{Y}_P/(1+r_f)\}(\varepsilon/k) \end{array} \right\} \text{Minimum } R_f \longrightarrow \text{Local mean rate of combustion}$$

$A, B$ : 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

$$R_{i,r} = v'_{i,r} M W_i A \rho \frac{\varepsilon}{k} \min\left(\frac{Y_{\text{Reactant}}}{v'_{\text{Reactant},r} M W_{\text{Reactant}}}\right)$$
$$R_{i,r} = v'_{i,r} M W_i A B \rho \frac{\varepsilon}{k} \min\left(\frac{\sum Y_{\text{Product}}}{\sum v''_{\text{Product},r} M W_{\text{Product}}}\right)$$

#### Arrhenius

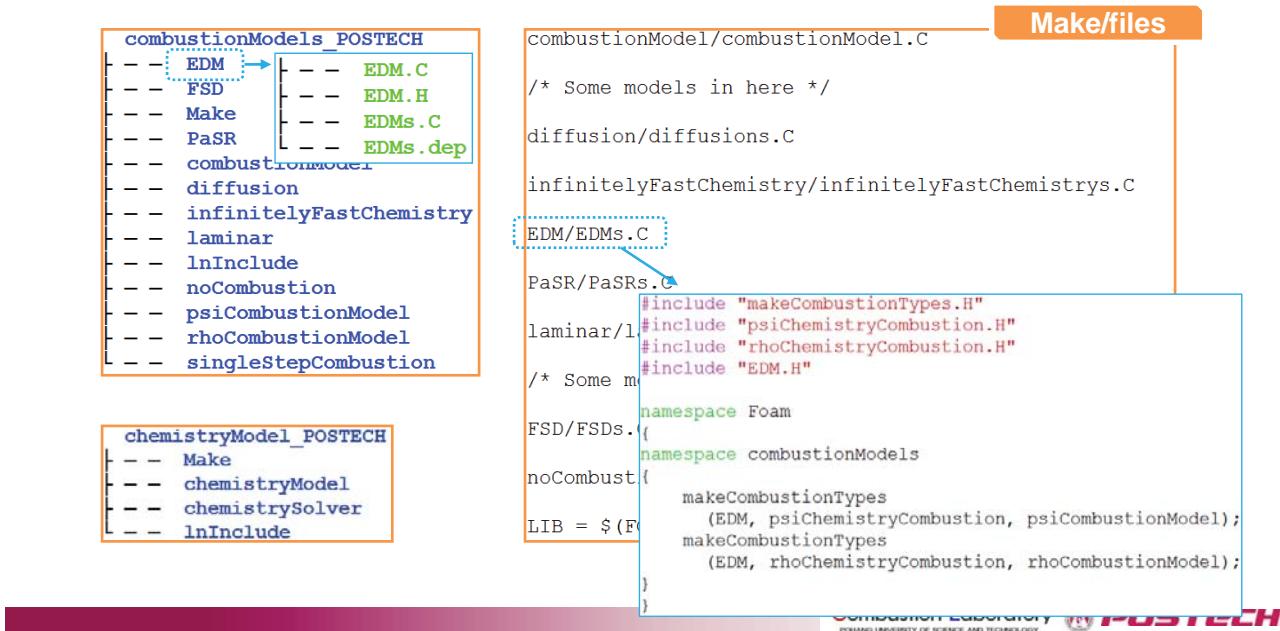
$$R_{i,r} = \Gamma(v''_{i,r} - v'_{i,r}) \left( k_{f,r} \prod_{j=1}^N [C_{j,r}]^{v_j} - k_{b,r} \prod_{j=1}^N [C_{j,r}]^{v'_j} \right)$$
$$k_r = A_r T^{b_r} \exp(-E_r / RT)$$

for species  $i$  and reaction  $r$

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

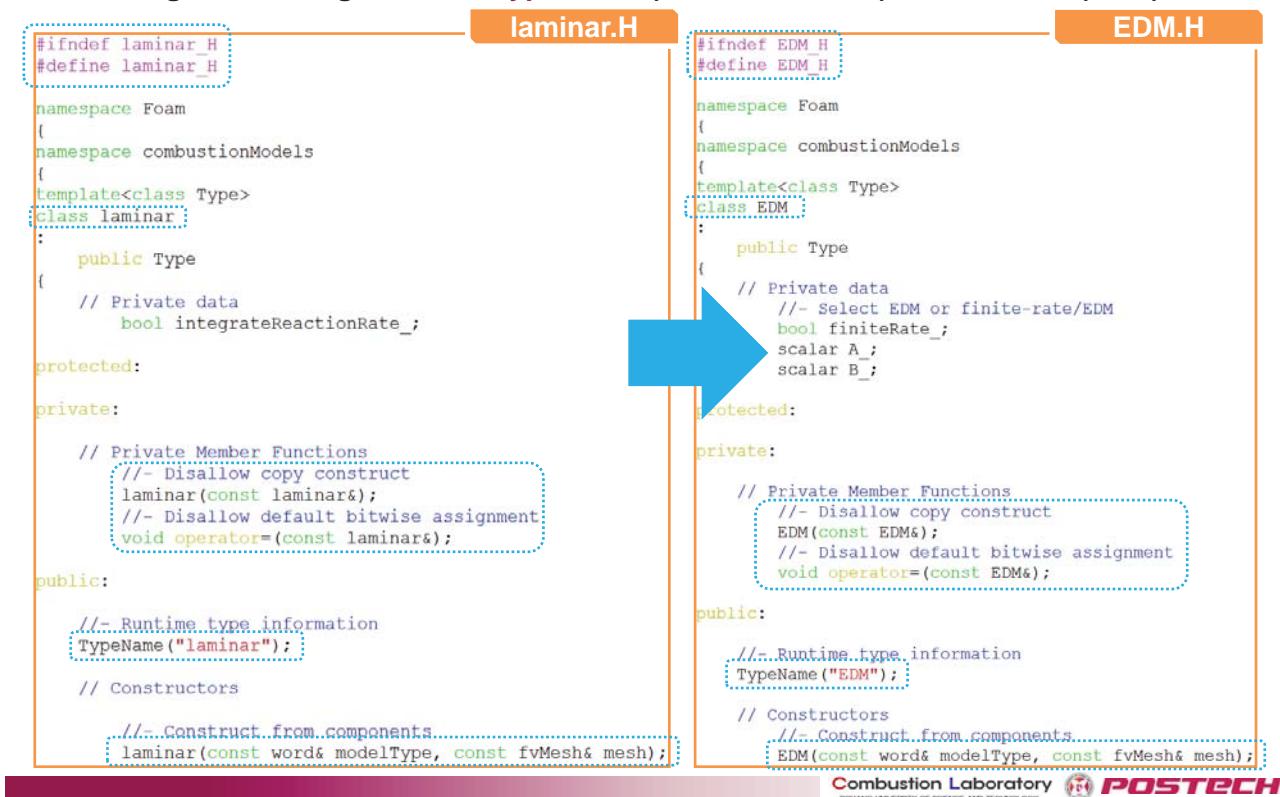
# How to Implement New Combustion Models (EDM)

- 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



# How to Implement New Combustion Models (EDM)

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



# How to Implement New Combustion Models (EDM)

```
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;
    }
}
```

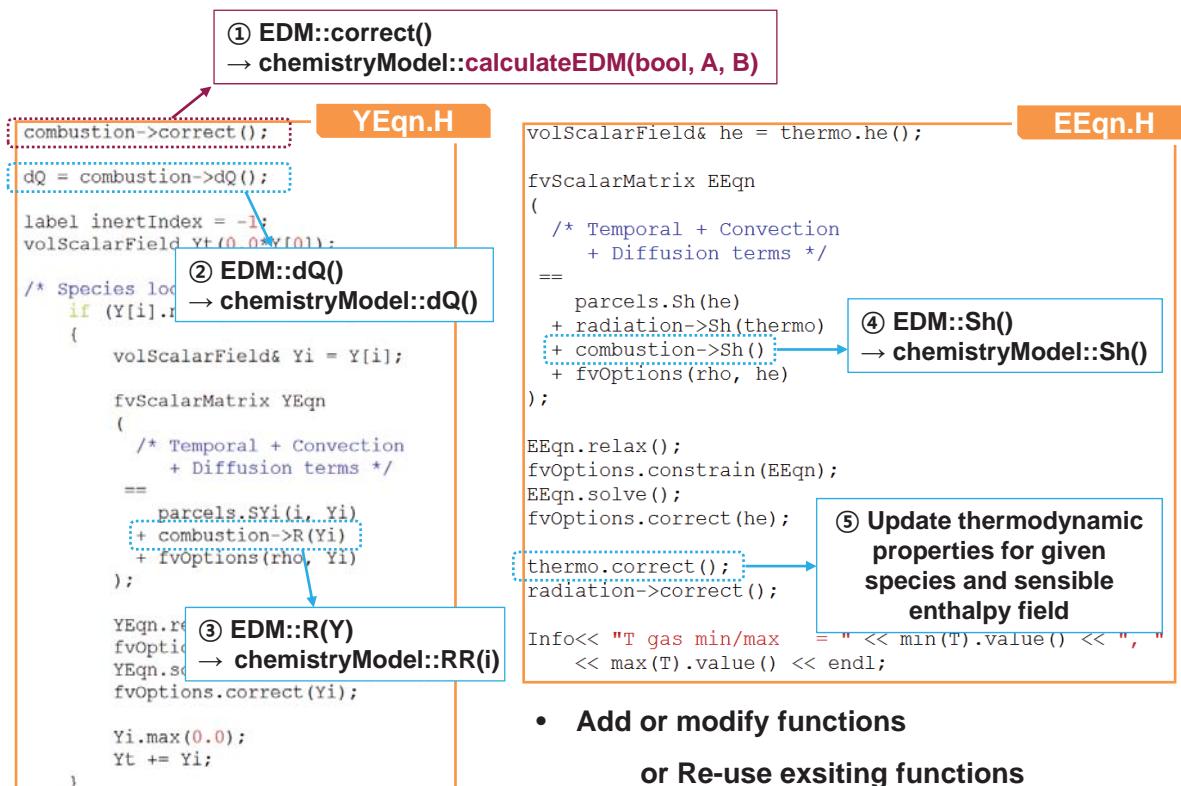
## Constructor at EDM.C

- Add flag or modeling coefficient

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

$$R_{i,r} = \nu'_{i,r} M W_i A B \rho \frac{\varepsilon}{k} \min\left(\frac{\sum Y_{\text{Product}}}{\sum \nu''_{\text{Product},r} M W_{\text{Product}}}\right)$$

# How to Implement New Combustion Models (EDM)



# How to Implement New Combustion Models (EDM)

```
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;
}
```

EDM.C

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)

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

EDM.C

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

/* Some functions */

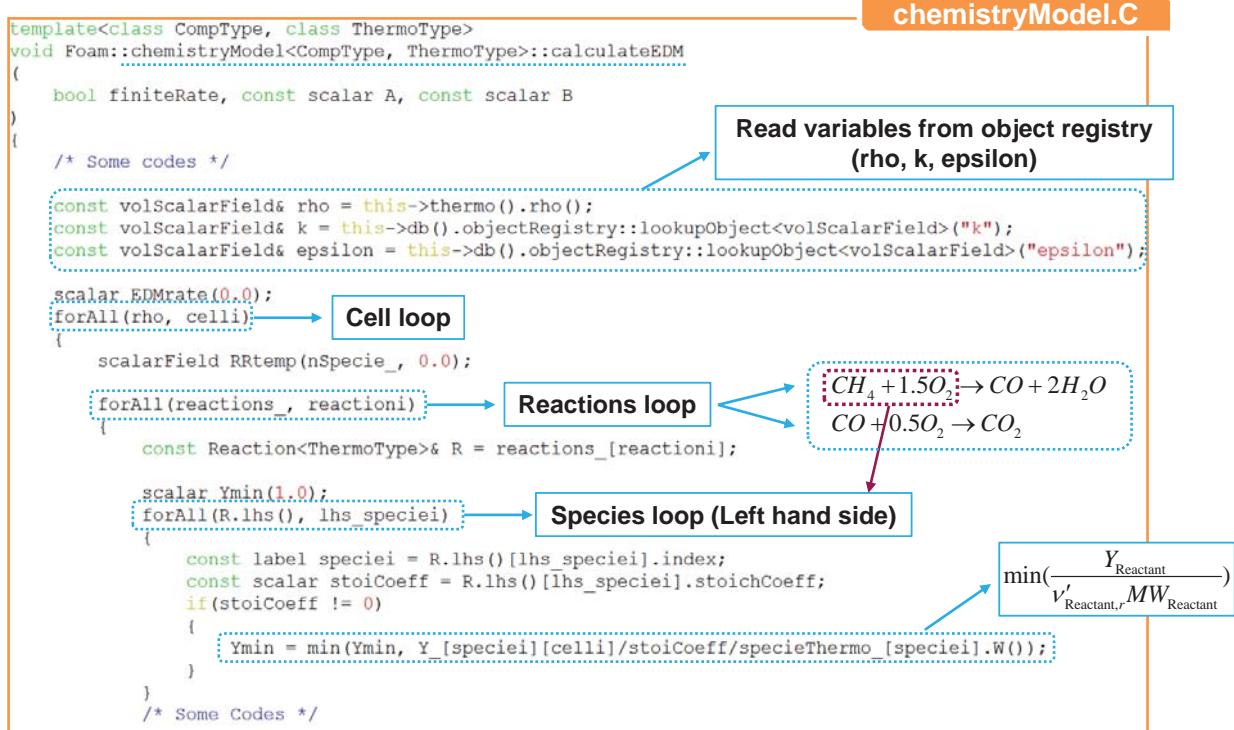
// Calculates the reaction rates (EDM or finite-rate/EDM)
// 12.Dec.2014 Karam Han
virtual void calculateEDM(bool finiteRate, const scalar A, const scalar B);
```

chemistryModel.H

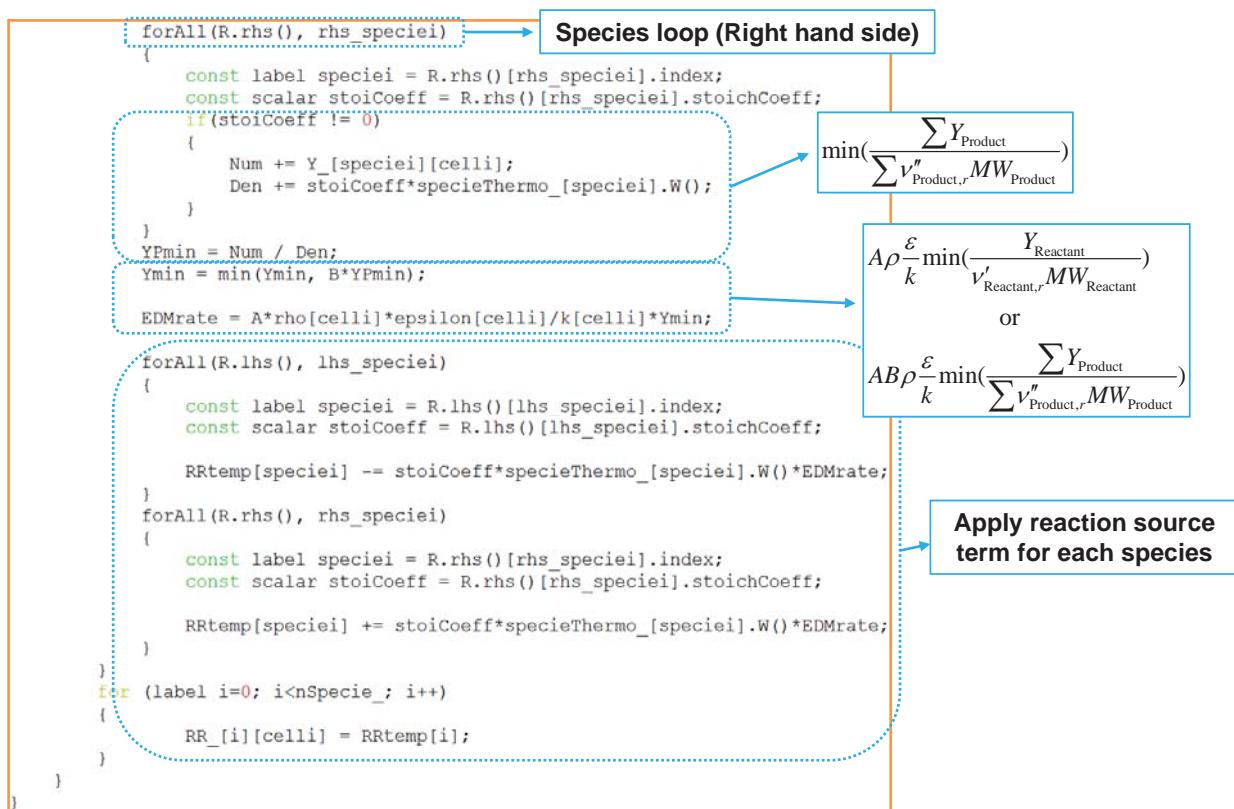
```
// Calculates the reaction rates (EDM or finite-rate/EDM)
// 12.Dec.2014 Karam Han
virtual void calculateEDM(bool finiteRate, const scalar A, const scalar B) = 0;
```

basicChemistryModel.H

# How to Implement New Combustion Models (EDM)



# How to Implement New Combustion Models (EDM)

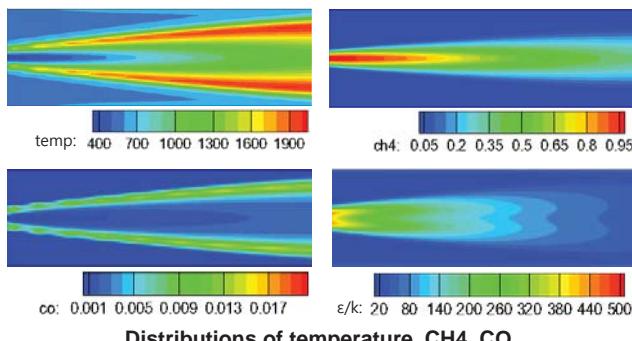


# How to Implement New Combustion Models (EDM)

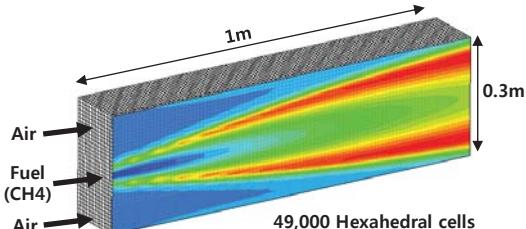
```
/*----- C++ -----*/
| \ \ / r i e l d | OpenFOAM: The Open Source CFD Toolbox
| \ \ / O p e r a t i o n | Version: 2.3.x
| \ \ / A n d | Web: www.OpenFOAM.org
| \ \ / M a n i p u l a t i o n |
FoamFile
{
    version 2.0;
    format ascii;
    class dictionary;
    location "constant";
    object combustionProperties;
}
// * * * * *
active true;

combustionModel EDM<rhoChemistryCombustion>;
EDMCoeffs
{
    A 4.0;
    B 0.5; //1e15;
    finiteRate false;
}
// * * * * *

CombustionProperties
```

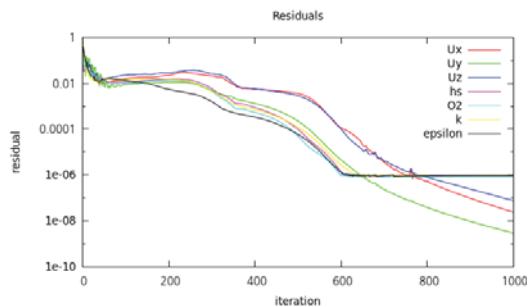


Distributions of temperature, CH4, CO



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

## Geometry and operating conditions



Convergence history in terms of residuals

Combustion Laboratory POSTECH

# Implement New Combustion Models



## Structure of reactingParcelFoam and PaSR model

- reactingParcelFoam
- Partially stirred reactor model



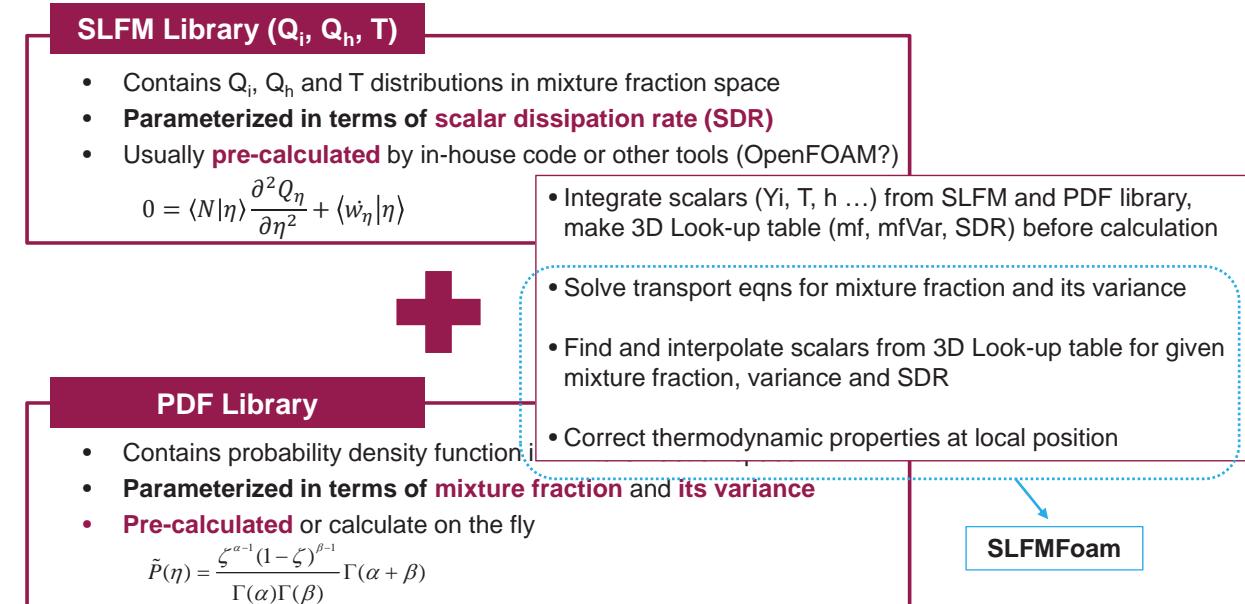
## How to implement New Combustion Models

- 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



Combustion Laboratory POSTECH

# How to Implement New Combustion Models (SLFM)

```
#include "fvCFD.H"
#include "turbulenceModel.H"
#include "basicReactingMultiphaseCloud.H"
#include "rhoCombustionModel.H"
#include "radiationModel.H"
#include "fvOptionList.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 "createCloud.H"
    #include "createVOF.H"
    #include "initCont.H"
    #include "readTime.H"
    #include "compressibleRecurcure.H"
    #include "setInitialDeltaT.H"
}

parcels.evolve();

// --- Pressure-velocity PIMPLE corrector loop
while (pimple.loop())
{
    #include "UEqn.H"
    #include "YFcorr.H"
    #include "ELeqn.H" → No need to be solved

    // --- Mixture corrector loop
    while (le.correct())
    {
        #include "MixtureFraction.H"
        #include "MixtureFractionVar.H"
    }
}
```

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

**No need to be solved**

**Add new transport equations for  $mf$  and  $mfVar$**

Combustion Laboratory POSTECH

# 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)

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

Solve transport eqns for mixture fraction and its variance
#include "setRootCase.H"

Find and interpolate scalars from 3D Look-up table for given mixture fraction, variance and SDR
#include "readGravitationalAcceleration.H"

Correct thermodynamic properties at local position
#include "createFields.H"
#include "createRadiationModel.H"
#include "createC1"
#include "createC2"
#include "initCont"
#include "readTimed"
#include "compressibleRecurance.H"
#include "setInitialDeltaT.H"
```

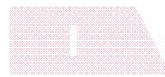
```
SLFMDom.C
parcels.evolve();

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

    // --- Pressure corrector loop
    while (pimple.correct())
    {
        #include "pEqn.H"
    }
    pimple.turbCorr();
    turbulence->correct();
}
```

Add new volScalarfield **mf, mfVar**

# Implement New Combustion Models



## 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

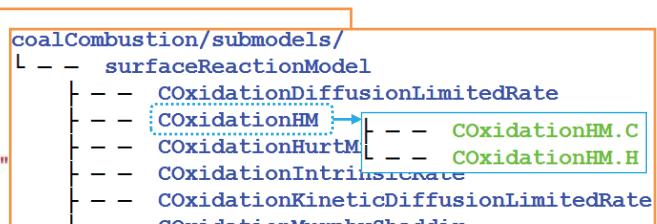
```
#ifndef makeCoalParcelSurfaceReactionModels_H
#define makeCoalParcelSurfaceReactionModels_H

#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
```



# How to Implement New Combustion Models (Char Reaction)

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

```
#include "SurfaceReactionModel.H"
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 constant
const scalar C1_;
```

**COxidationKineticDiffusionLimited.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_;
```

**COxidationHM.H**

Combustion Laboratory POSTECH

# 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_*W02_;
    const scalar dmCO = dOmega*(WC_ + Sb_*W02_);

    // 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



OpenFOAM in Industrial Combustion Simulations, Pohang University Feb/2015

OpenFOAM: Code Development, Debugging and Trouble-Shooting – p. 1

## 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

## 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



## 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

## 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

## 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

## 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

## 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

## 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];
internalField   uniform (0 0 0);

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

## 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 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 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 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 `fvPatchFields` from a dictionary
- (Do you really want to know how this is done?)

## 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())
```



## 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_;
}
```



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
  - `functionObjects` 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



## 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;

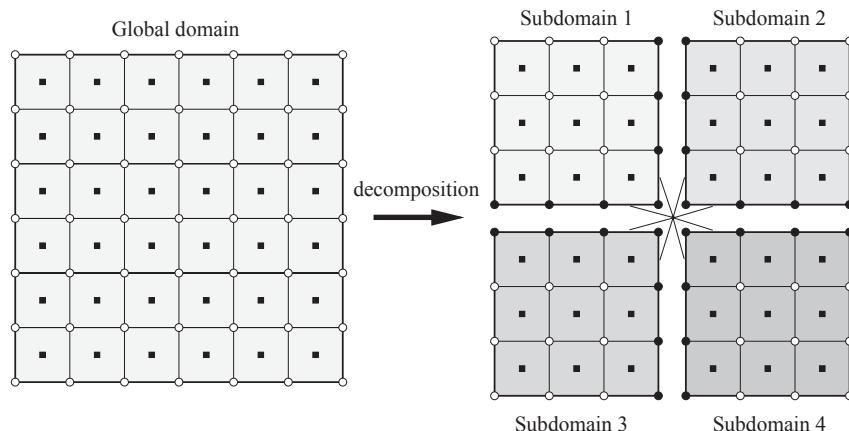
    virtual bool read(const dictionary& dict) = 0;
};
```



# 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, it is essential that all processors **follow the same execution patch**



## git: A Distributed Version Control System

- Keep track of your code development, know current 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

$$\alpha = 3$$

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

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

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

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

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

## 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



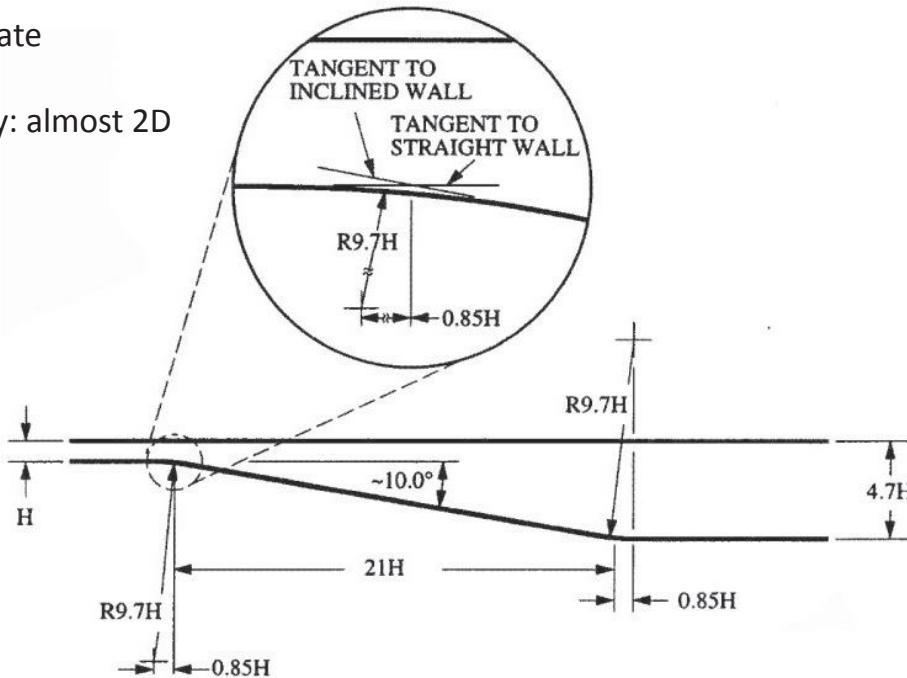
- The problem: flow simulation through an asymmetric planar diffuser.
- The solver: simpleFoam
- Case setup: mesh conversion, definition of initial conditions, dictionaries
- Effects of turbulence models and wall functions
- Validation with experimental data : velocity field, ...

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](#)

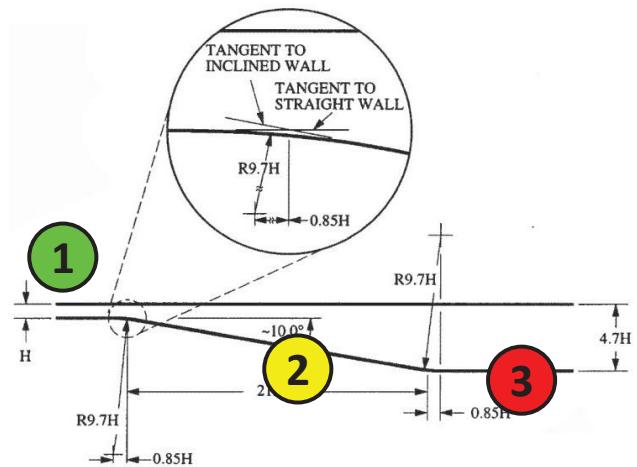
T. Lucchini: Incompressible, steady-state turbulent flow simulations in OpenFOAM®

# Flow problem

## Flow through an asymmetric plane diffuser

This flow includes three important features:

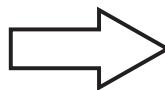
- 1) The well defined inlet conditions. The inlet channel flow is turbulent and fully-developed with a  $Re=20,000$  based on the centreline velocity and the channel height.
- 2) A smooth-wall separation due to an adverse pressure gradient is involved. The prediction of the separation point and the extent 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



simpleFoam

`$FOAM_SOLVERS/incompressible/simpleFoam`

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

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

## simpleFoam.C

```
int main(int argc, char *argv[])
{
    #include "setRootCase.H" → Identification of case directory
    #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;
}
```

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;
}

```

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

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;
}

```

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*

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."
    #include "initContinuityE
    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



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

# 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.

# 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:  $\vec{U} \cdot \vec{n} S_f$

# 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"), pRefCell, pRefValue);
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

# 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             |
| LauderGibsonRSTM    |
| LauderSharmaKE      |
| LienCubicKE         |
| LienCubicKELowRe    |
| LienLeschzinerLowRe |
| LRR                 |
| NonlinearKEShah     |
| 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 implementation related references are reported.

...  
OpenFOAM®

## 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?

- Discretize in a suitable way convection, stress and gradient terms
  - $\text{div}(\phi, \mathbf{U})$
  - $\text{turbulence}->\text{divDevReff}(\mathbf{U})$
  - $\text{grad}(p)$
- Provide proper under-relaxation factors for momentum equation:  $\text{UEqn}().\text{relax}()$
- Define suitable parameters for solution of momentum equation:
  - $\text{solve}(\text{UEqn}() == -\text{fvc}::\text{grad}(p))$

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

# 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

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

# 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();`

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

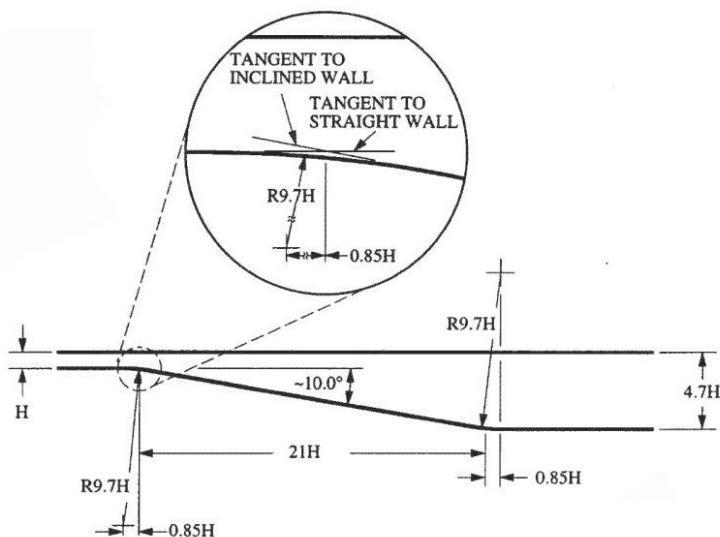
# simpleFoam solver



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

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

# 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.

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

# 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
```

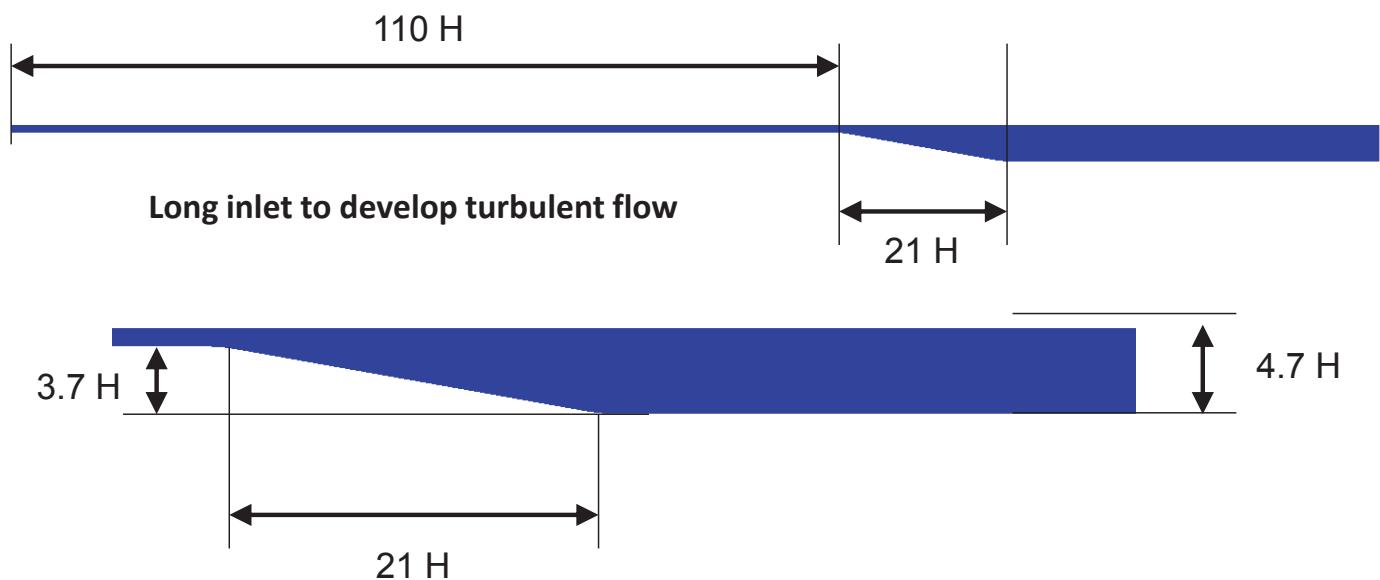
- Check the mesh and see the log file

```
> checkMesh -case diffuserBase
```

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

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

# Diffuser simulation: first case



## First attempt setup

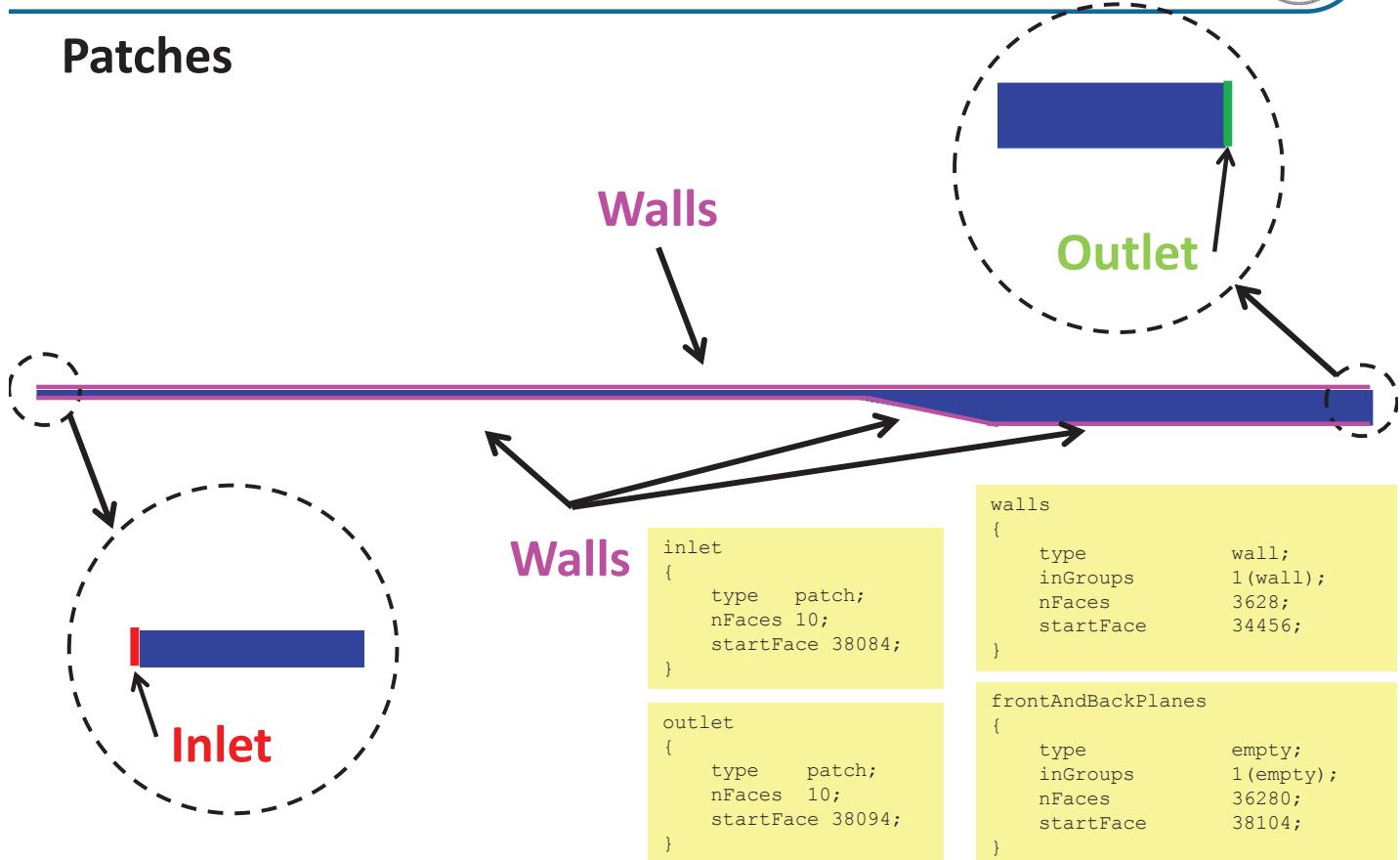
- Mesh size from user experience
- Standard  $k-\varepsilon$  model for turbulence and standard wall functions
- First order convection schemes
- Standard OpenFOAM settings for equation solution

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

# Diffuser simulation: mesh



## Patches

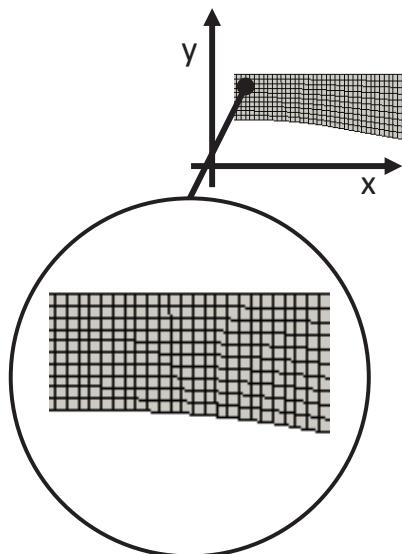


T. Lucchini: Incompressible, steady-state turbulent flow simulations in OpenFOAM®

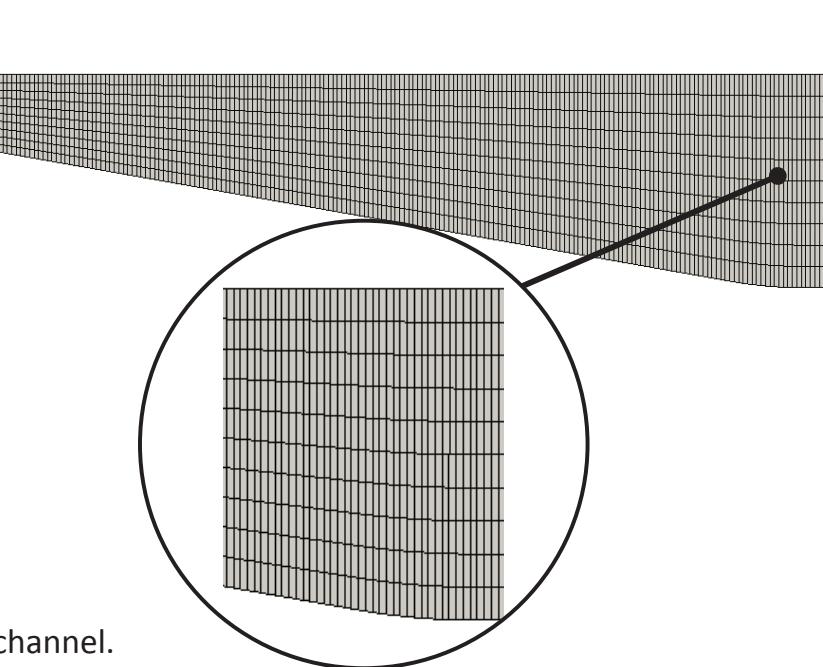
# Diffuser simulation: mesh



## Mesh in the diffuser region



## Block structured grid



- Cell size:  $0.1 H \times 0.1 H$  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....

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

# 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.

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

# 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  $\nu_{air} = 10^{-4} \text{ m}^2/\text{s}$  it is possible to estimate the inlet velocity:

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

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

# 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 $\mathbf{U}$

### 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;
    }
}
```

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

# Diffuser simulation: initial conditions



## Velocity field $\mathbf{U}$

### 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  $p/\rho$  and the pressure is relative. Which is why pressure dimensions are  $\text{m}^2/\text{s}^2$ .
- 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;
    }
}
```

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

# 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;
    }
}
```

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

# 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  $k-\varepsilon$  model is used,  $k$  and  $\varepsilon$  fields must be initialized.
- Specification of the  $\nu_t$  (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  $\varepsilon$ .

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

# 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;
    }
}
```

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

# 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,  $\varepsilon$  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 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];
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 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];
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           ;
}
```

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           ;
}
```

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)

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

# 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:

- 1<sup>st</sup> 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)

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

# 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:

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

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.

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

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

# 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-orthogonal 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.

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

# 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.

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

# 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*;

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

# Diffuser simulation: directory system



## fvSolution - 3

### 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;
}
```

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

# Diffuser simulation: directory system



## fvSolution - 5

### The SIMPLE subdictionary

```
SIMPLE
{
    nNonOrthogonalCorrectors 0;

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

Number of non-orthogonal correctors is nNonOrthogonalCorrectors.

Since the pressure equation is:

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

And Laplacian is decomposed into two parts:

- orthogonal: implicit
- non-orthogonal: 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

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

# Diffuser simulation: directory system



## fvSolution - 7

### The relaxationFactors subdictionary

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

- 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  $\alpha$ ,  $0 < \alpha \leq 1$ , specifies the amount of under-relaxation, ranging from not at all ( $\alpha = 1$ ) and increasing its strength as it tends to zero.

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

# 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  $\alpha$  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;
    }
}
```

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

# Diffuser simulation: directory system



## fvSolution - 9

## The relaxationFactors subdictionary

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

- Very conservative, slow convergence

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

# Diffuser simulation: directory system



## fvSolution - 10

### The relaxationFactors subdictionary

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

- Conservative, convergence ok

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

# Diffuser simulation: directory system



## fvSolution - 11

### The relaxationFactors subdictionary

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

- 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
```

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

# 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](#)

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



# 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
```

---

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

# 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...)

---

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

# 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).

---

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

# 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.

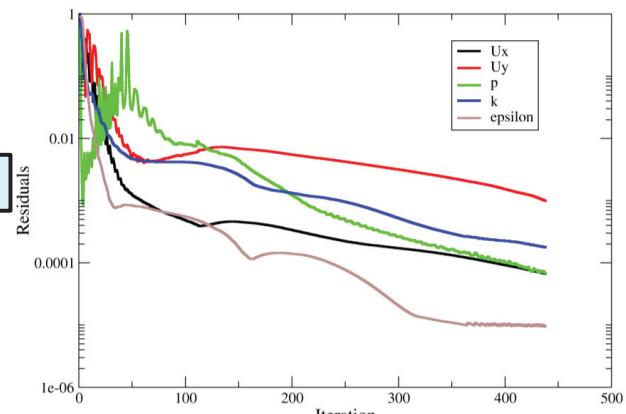
- 2) Check the residuals to see if they smoothly decreased for any of the solved field. Run:

```
> 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.



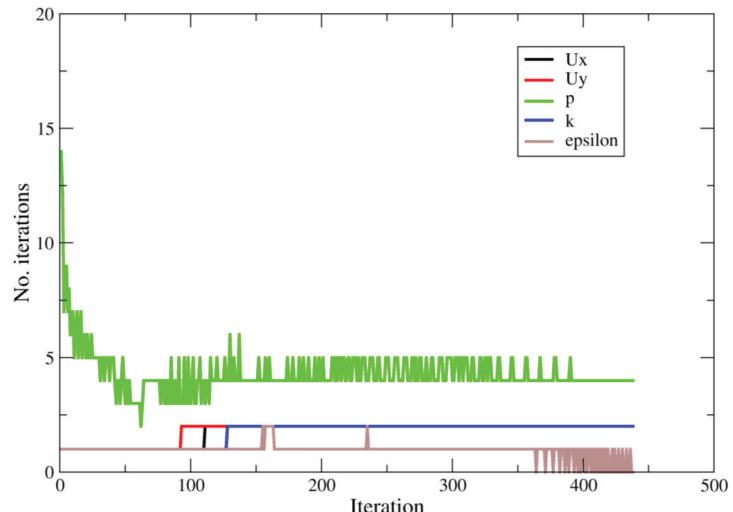

---

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

# 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...

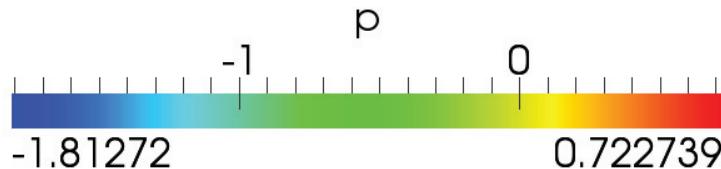
T. Lucchini: Incompressible, steady-state turbulent flow simulations in OpenFOAM®

# Diffuser simulation



## Fields

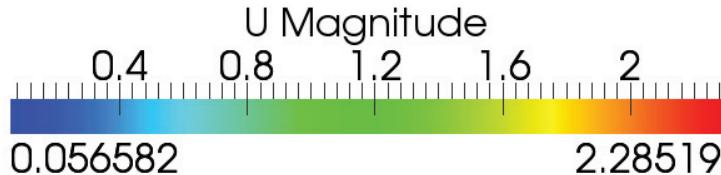
pressure



Pressure decreases inside the duct

Pressure recovery inside the diffuser

velocity



Re-development of the boundary layer

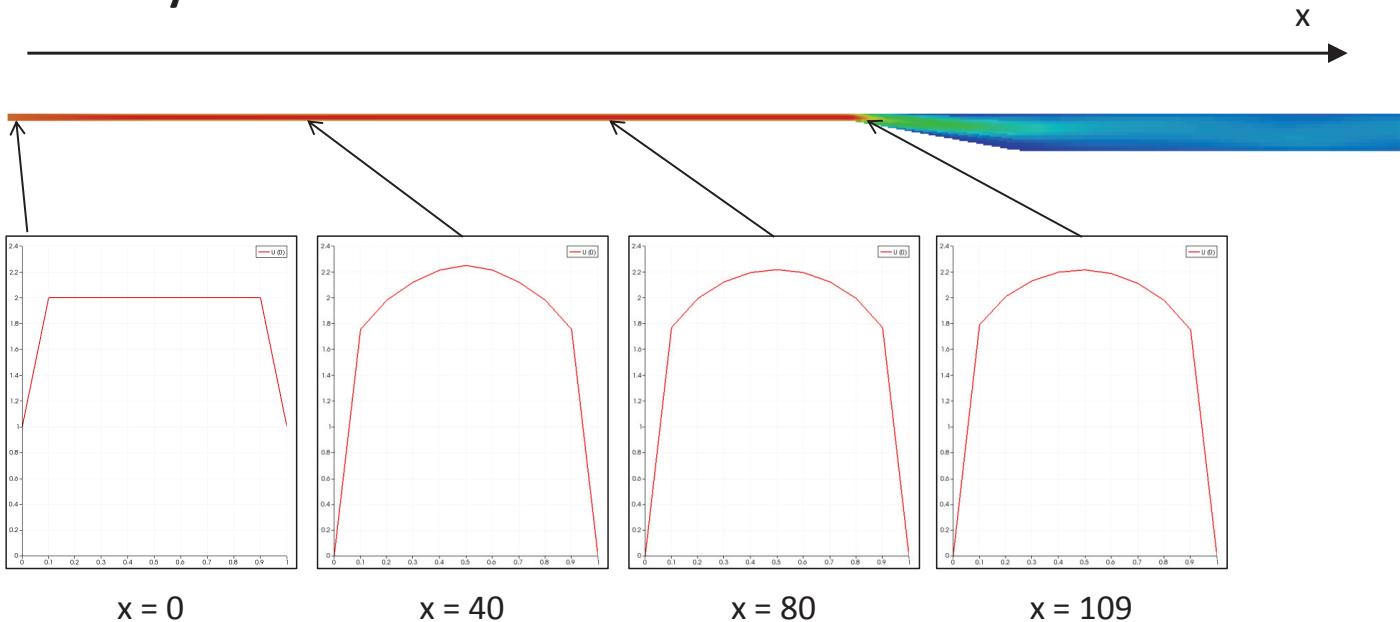
Turbulent flow development in the inlet pipe

Flow detachment due to adverse pressure gradient

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

# 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

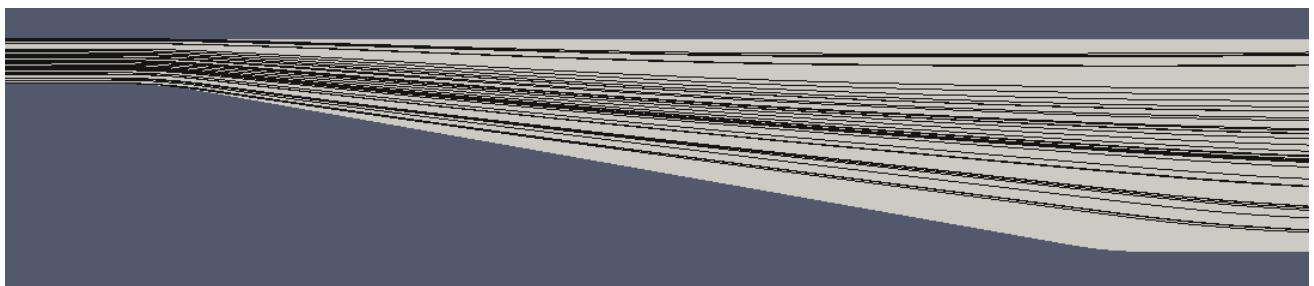
T. Lucchini: Incompressible, steady-state turbulent flow simulations in OpenFOAM®

# 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

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

# 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

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

# 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

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

# 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
```

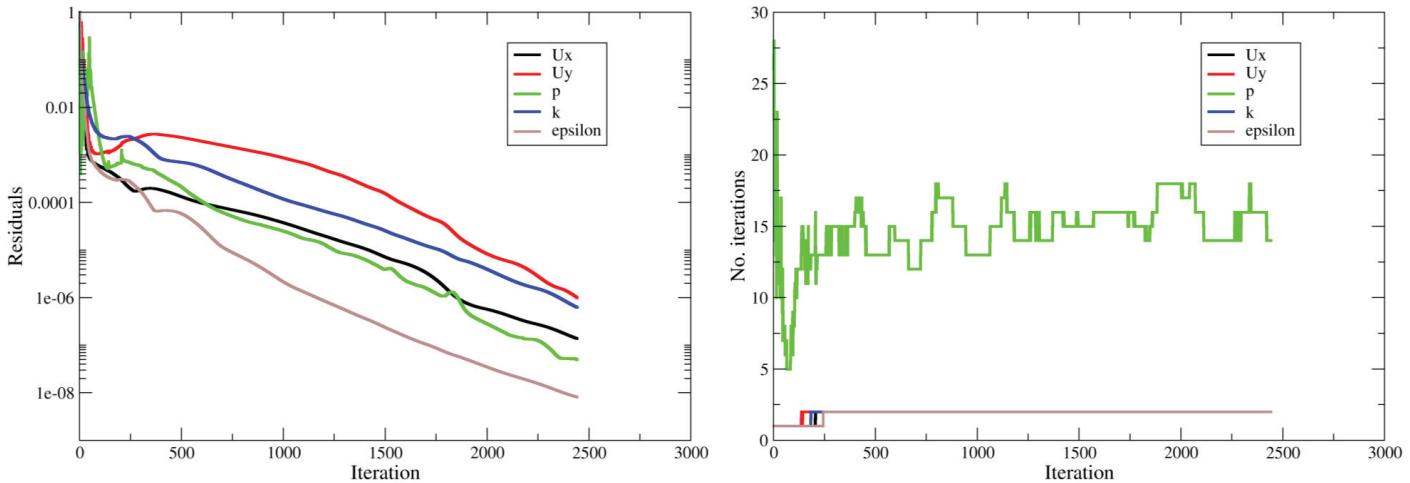
- Initial conditions are the same as `diffuserBase` case.
  - ⇒ Use `mapFields` utility (if you want, not necessary) to initialize the flow field with the one of the latest time of the `diffuserBase` case. This is expected to reduce the number of iterations required to reach convergence.

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

# 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`.

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

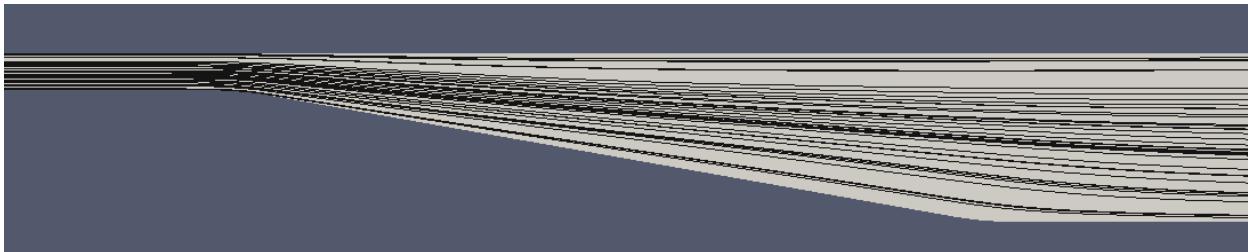
# Diffuser simulation

## 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...)

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

# 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®

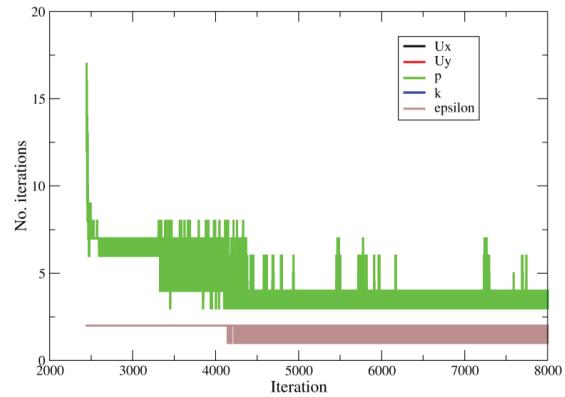
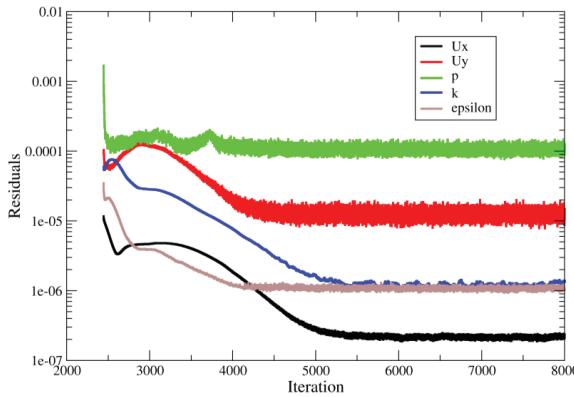
# Diffuser simulation

## 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 of all the fields are stable. Run `reconstructPar` to visualize results at the latest time-step



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

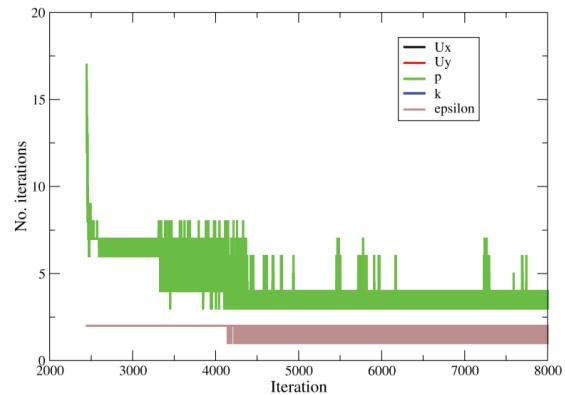
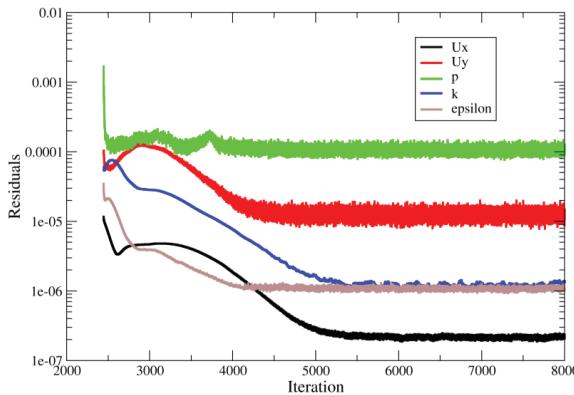
# Diffuser simulation

## 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
```

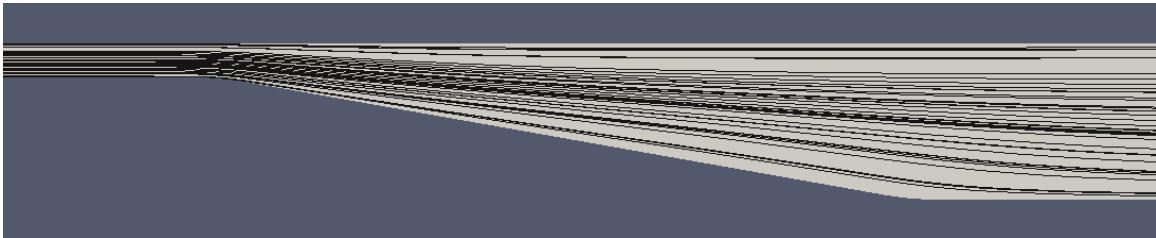
- Convergence (to user tolerance) is not reached, but residuals of all the fields are stable. Run `reconstructPar` to visualize results at the latest time-step



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

# 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?

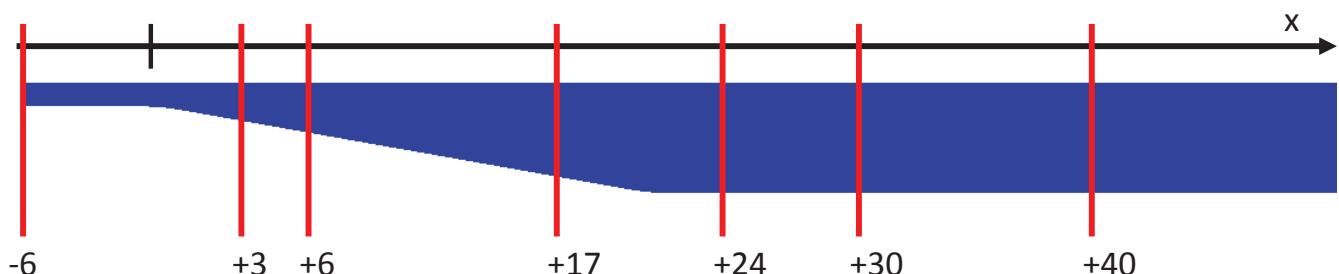
- 1) Check the  $y_+$  values to see if they are in the proper range
  - If not rerun the simulation with a more refined mesh at the walls
- 2) Compare results with experimental data (if available)
- 3) See if results improve when using lower tolerances for residuals
- 4) Use high-order numerical schemes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

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

# 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

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

# 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;
}
)
```

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

# 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
```

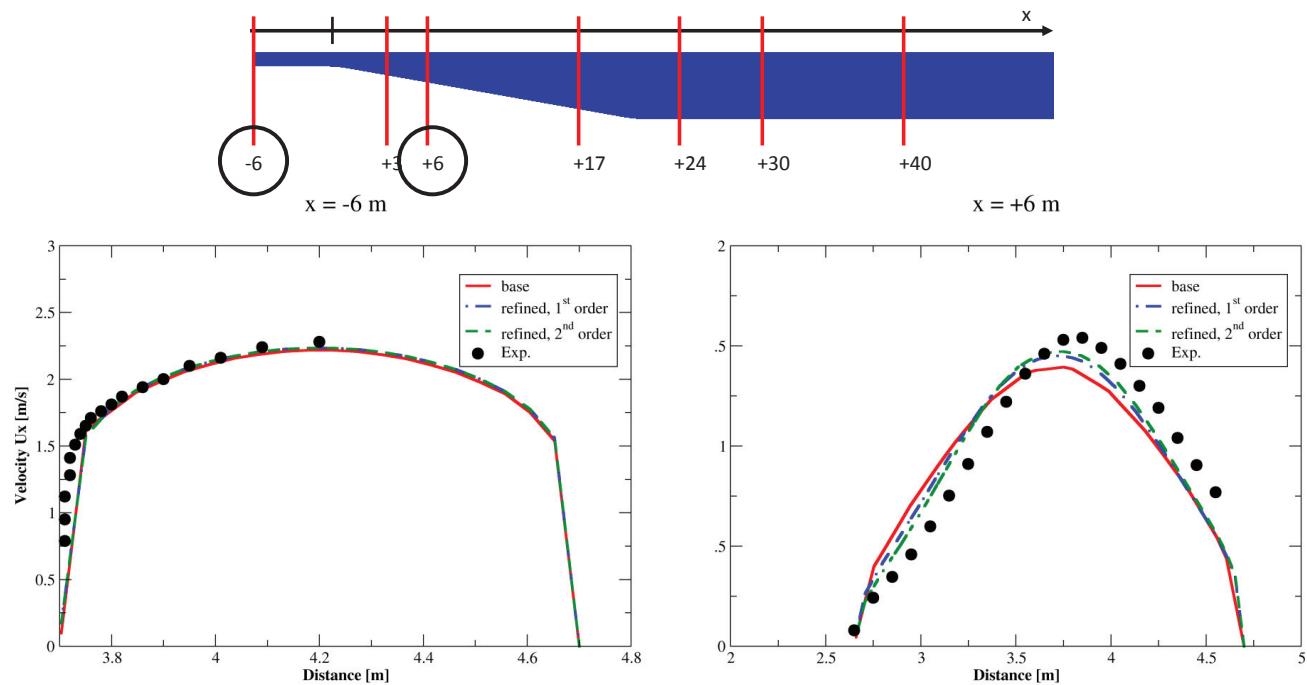
- Sub-directories with the names of the time-steps where post-processing was performed appear including the sampled velocity profiles along the different lines.

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

# Diffuser simulation



## Comparison with experimental data: velocity profiles

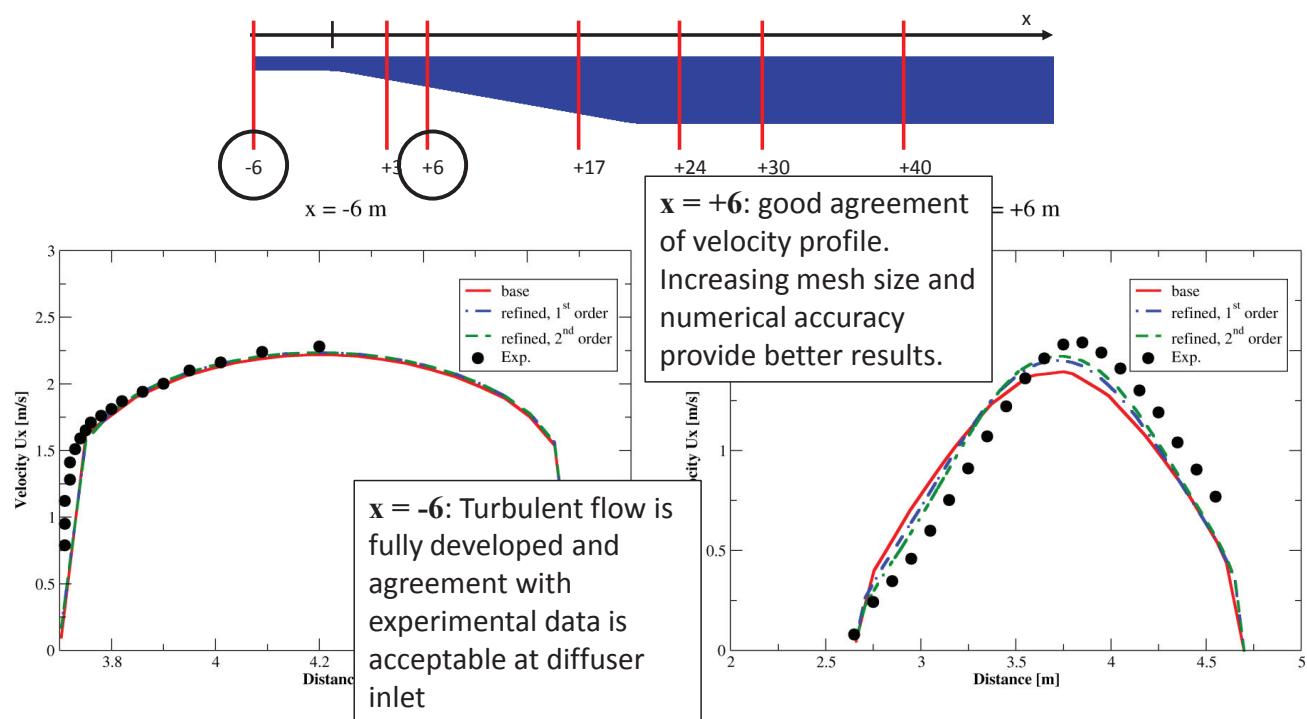


T. Lucchini: Incompressible, steady-state turbulent flow simulations in OpenFOAM®

# Diffuser simulation



## Comparison with experimental data: velocity profiles

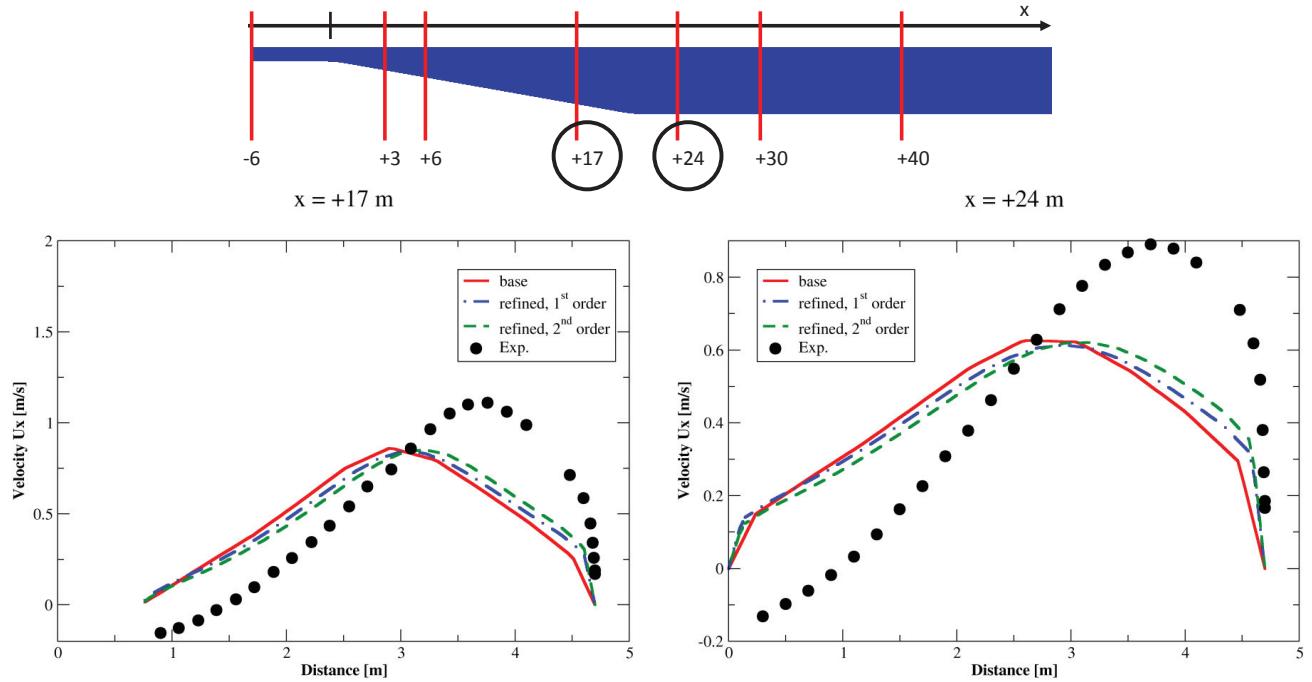


T. Lucchini: Incompressible, steady-state turbulent flow simulations in OpenFOAM®

# Diffuser simulation



## Comparison with experimental data: velocity profiles

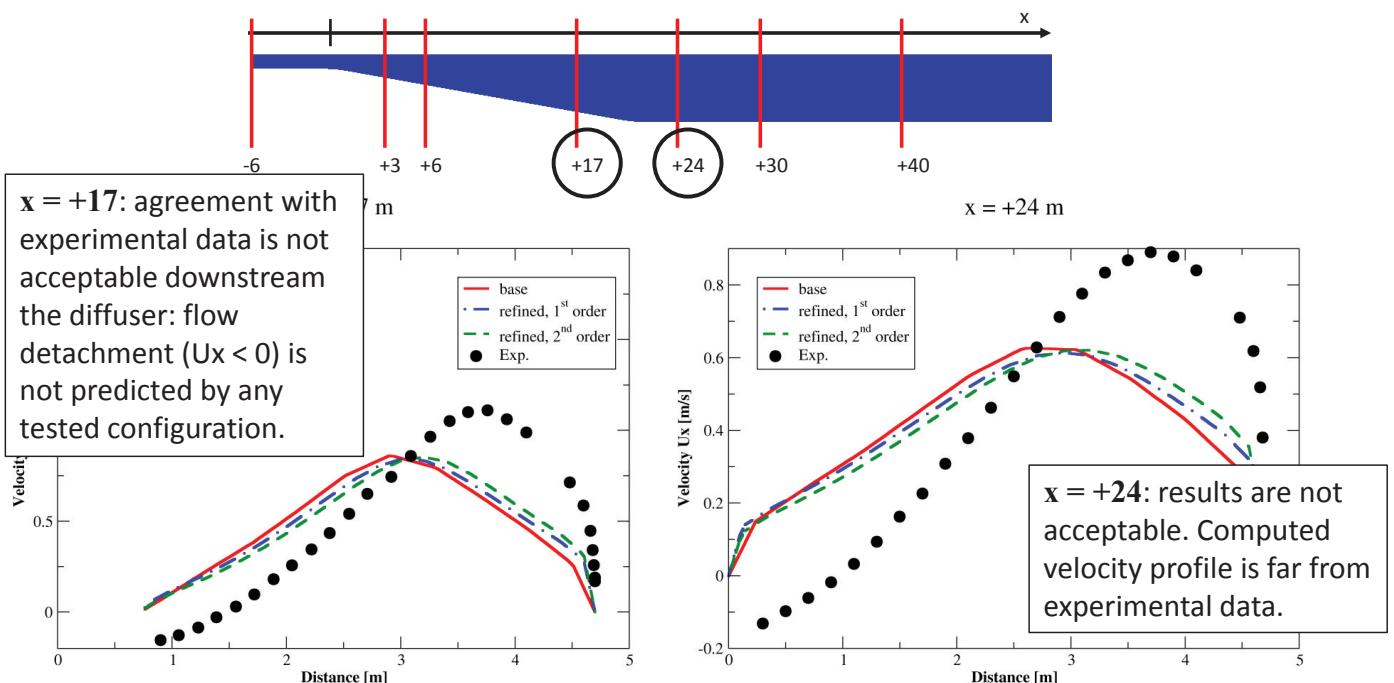


T. Lucchini: Incompressible, steady-state turbulent flow simulations in OpenFOAM®

# Diffuser simulation



## Comparison with experimental data: velocity profiles

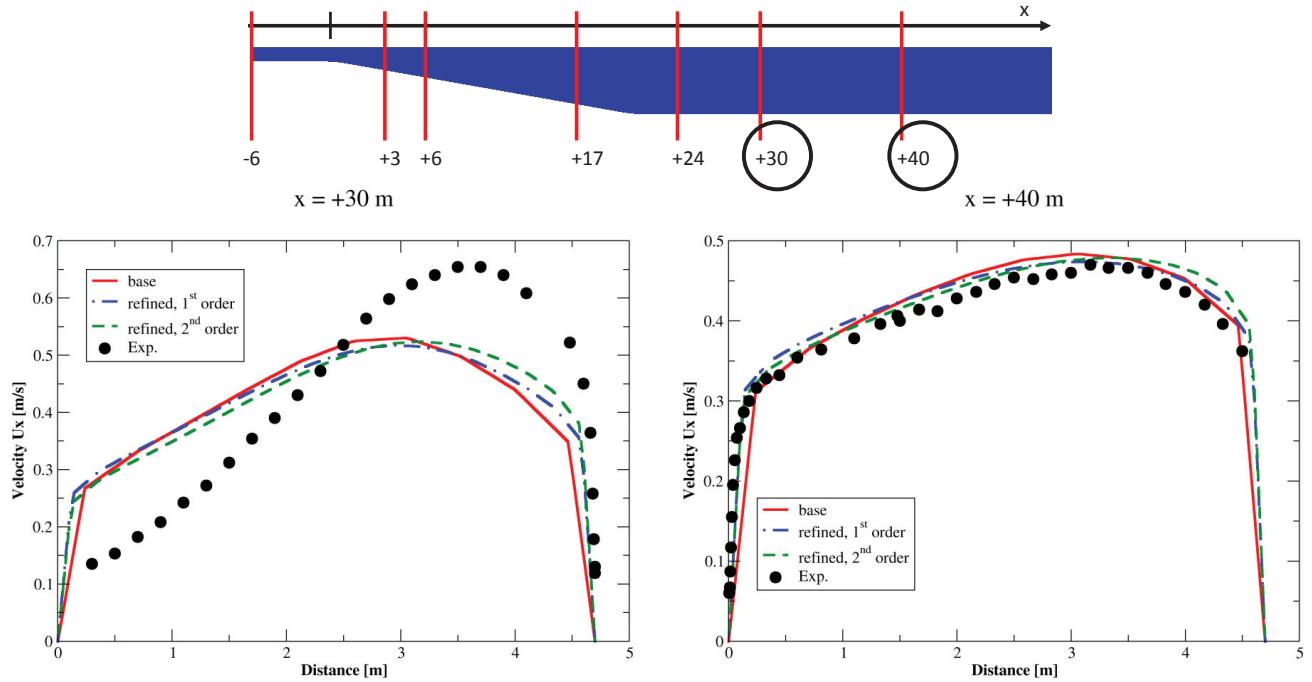


T. Lucchini: Incompressible, steady-state turbulent flow simulations in OpenFOAM®

# Diffuser simulation



## Comparison with experimental data: velocity profiles

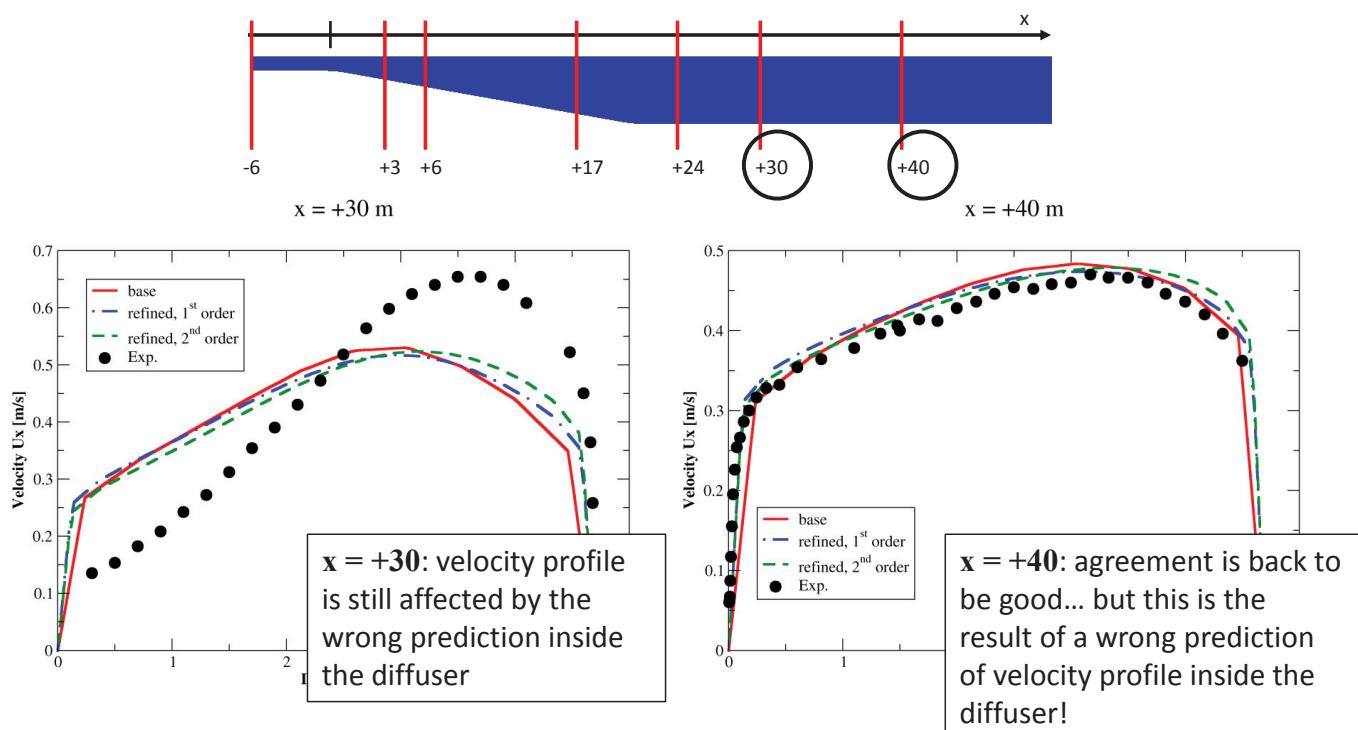


T. Lucchini: Incompressible, steady-state turbulent flow simulations in OpenFOAM®

# Diffuser simulation



## Comparison with experimental data: velocity profiles



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



# 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

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

# Diffuser simulation



## Run the case with the kOmegaSST turbulence model

- Initialization of the omega field

- Since  $\omega = \frac{\epsilon}{k}$ , 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.

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

# Diffuser simulation



## Turbulent specific dissipation rate omega

- Dimensions are inverse of time
- Initial internal value: ratio between  $\epsilon$  and  $k$  of the previous case.
- At the `inlet` boundary,  $\omega$  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;
    }
}
```

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

# 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;
    }
}
```

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

# Diffuser simulation

## Run the case with the kOmegaSST turbulence model

- Running the case in two-steps:

- 1) Get convergence with 1<sup>st</sup> order schemes:

- Add `div(phi, omega)` in `fvSchemes` (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;
    }
}
```

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

# Diffuser simulation

## Run the case with the kOmegaSST turbulence model

- Running the case in two-steps:

- 2) Then restart the case with 2<sup>nd</sup> 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)

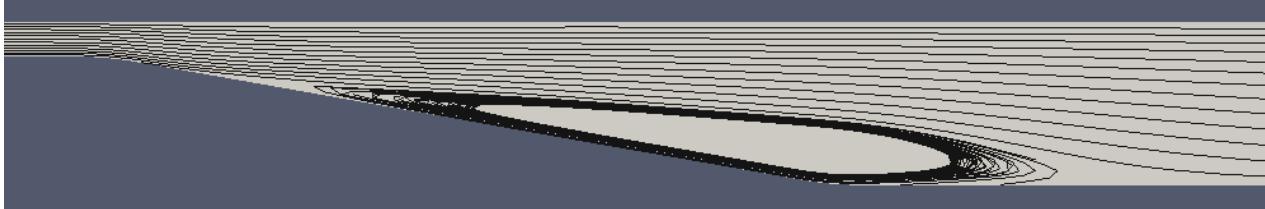
```
Patch 0 named walls y+ : min: 16.9854 max: 65.4602 average: 31.0476
```

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

# Diffuser simulation

## Run the case with the kOmegaSST turbulence model

- Post-processing with paraFoam: streamlines



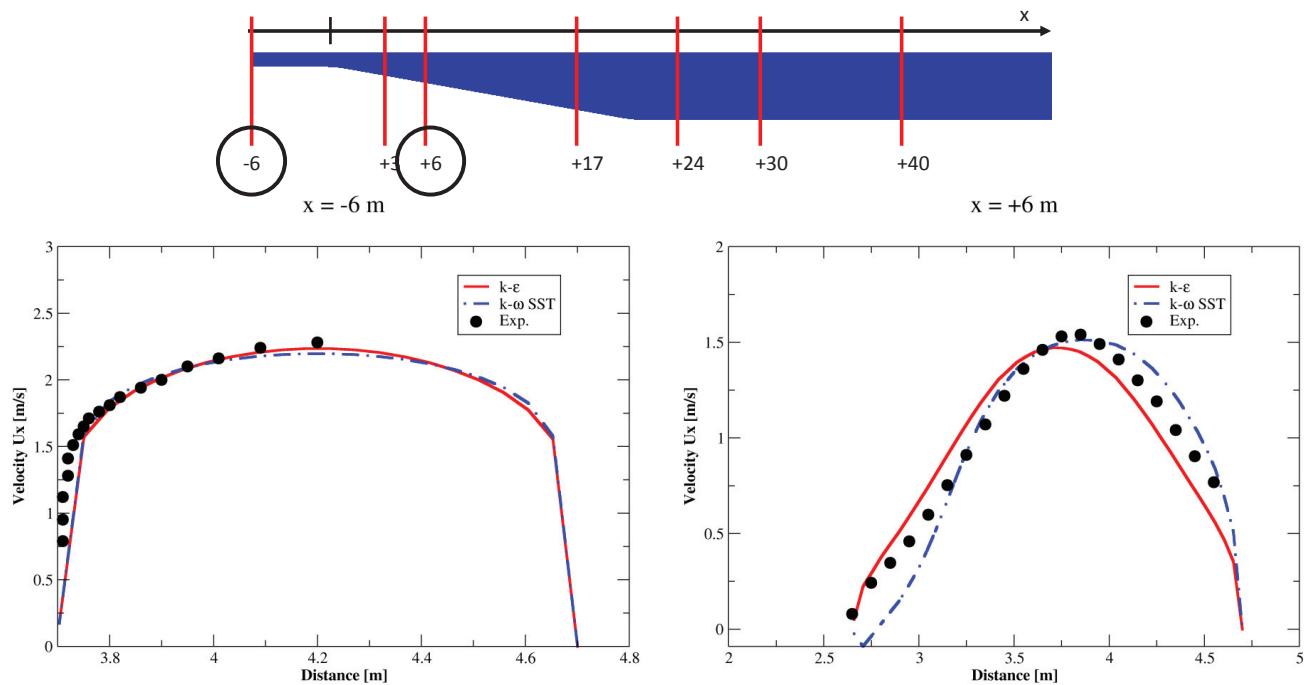
- Flow detachment and recirculation inside the diffuser are now properly predicted.
- Changing the turbulence model qualitatively improved the computed results.
- Let's compare again computed and experimental data...
  - Standard  $k-\epsilon$  (last iteration of the diffuserRefined case)
  - $k-\omega$  SST (last iteration of the diffuserKOmegaSST case)

⇒ Since same mesh, tolerances and numerical methods were used, the comparison is consistent.

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

# Diffuser simulation

## Comparison with experimental data: velocity profiles

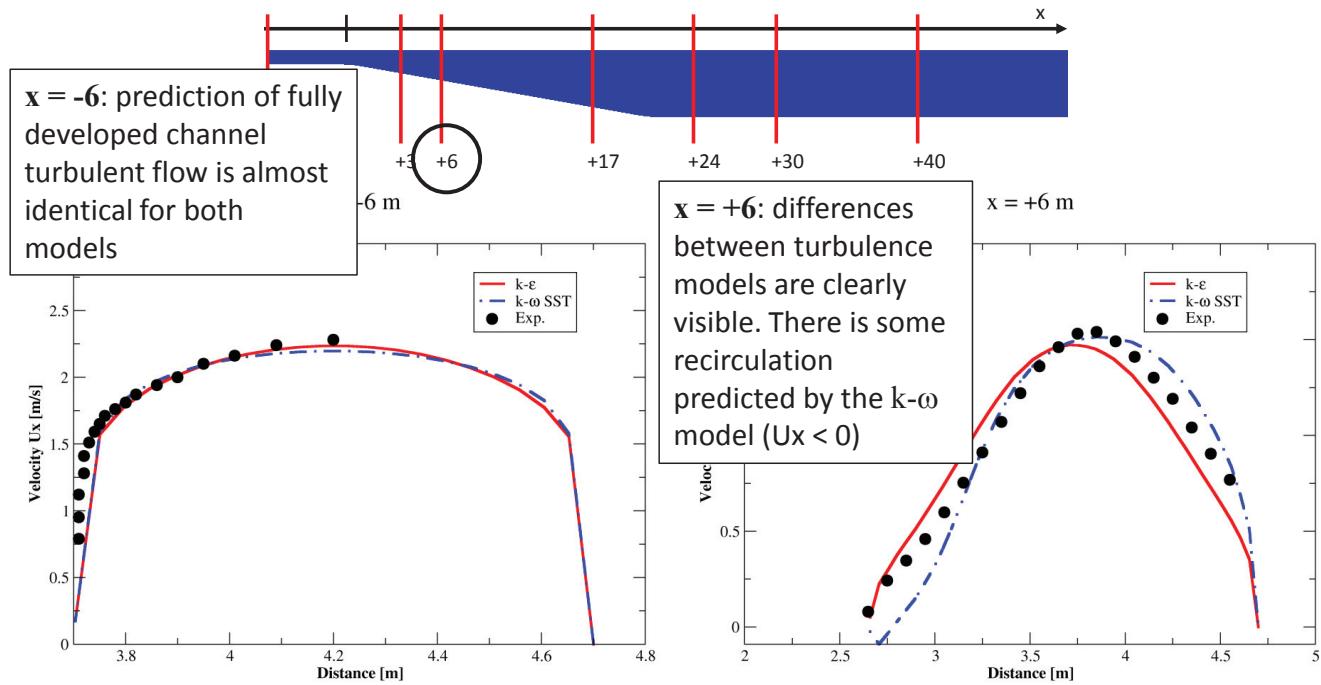


T. Lucchini: Incompressible, steady-state turbulent flow simulations in OpenFOAM®

# Diffuser simulation



## Comparison with experimental data: velocity profiles

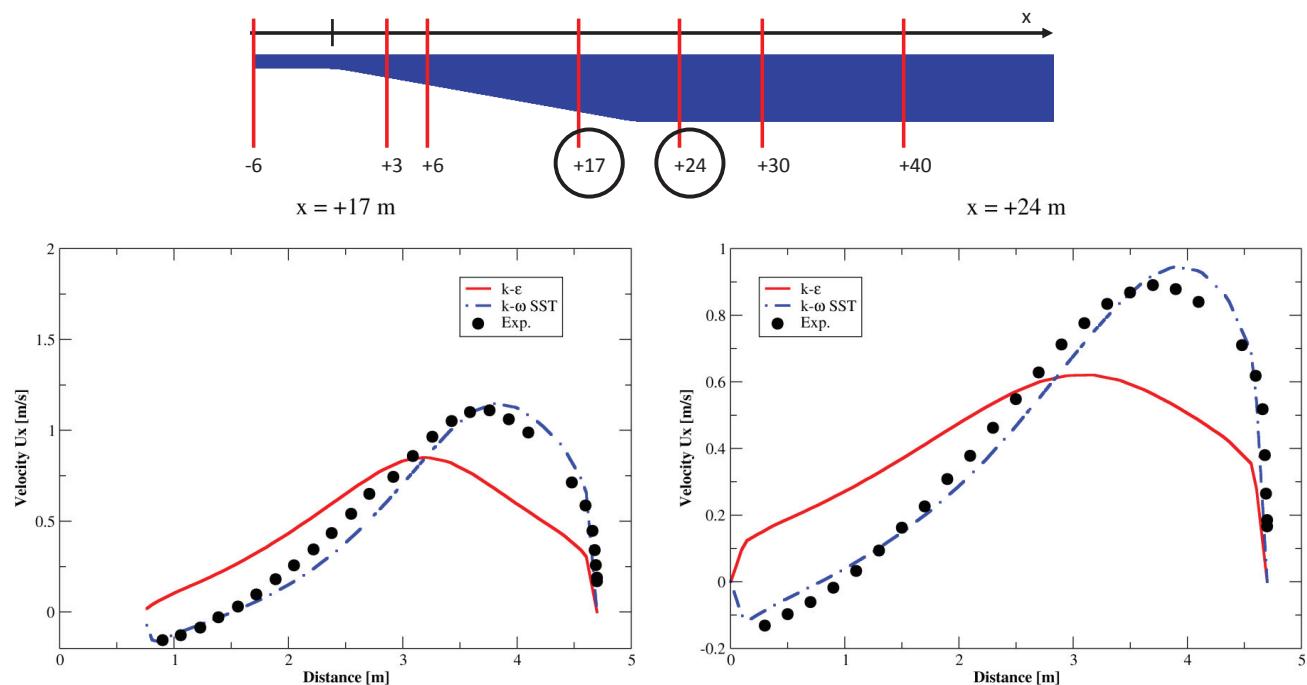


T. Lucchini: Incompressible, steady-state turbulent flow simulations in OpenFOAM®

# Diffuser simulation



## Comparison with experimental data: velocity profiles

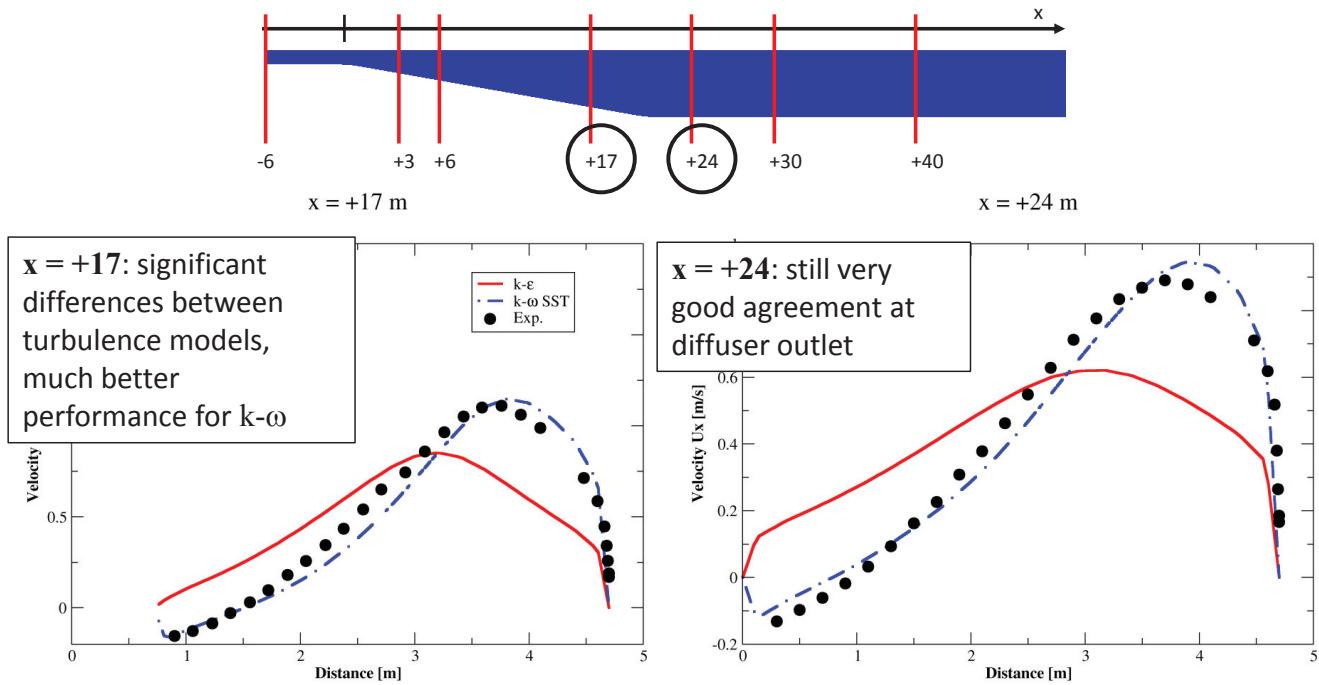


T. Lucchini: Incompressible, steady-state turbulent flow simulations in OpenFOAM®

# Diffuser simulation



## Comparison with experimental data: velocity profiles

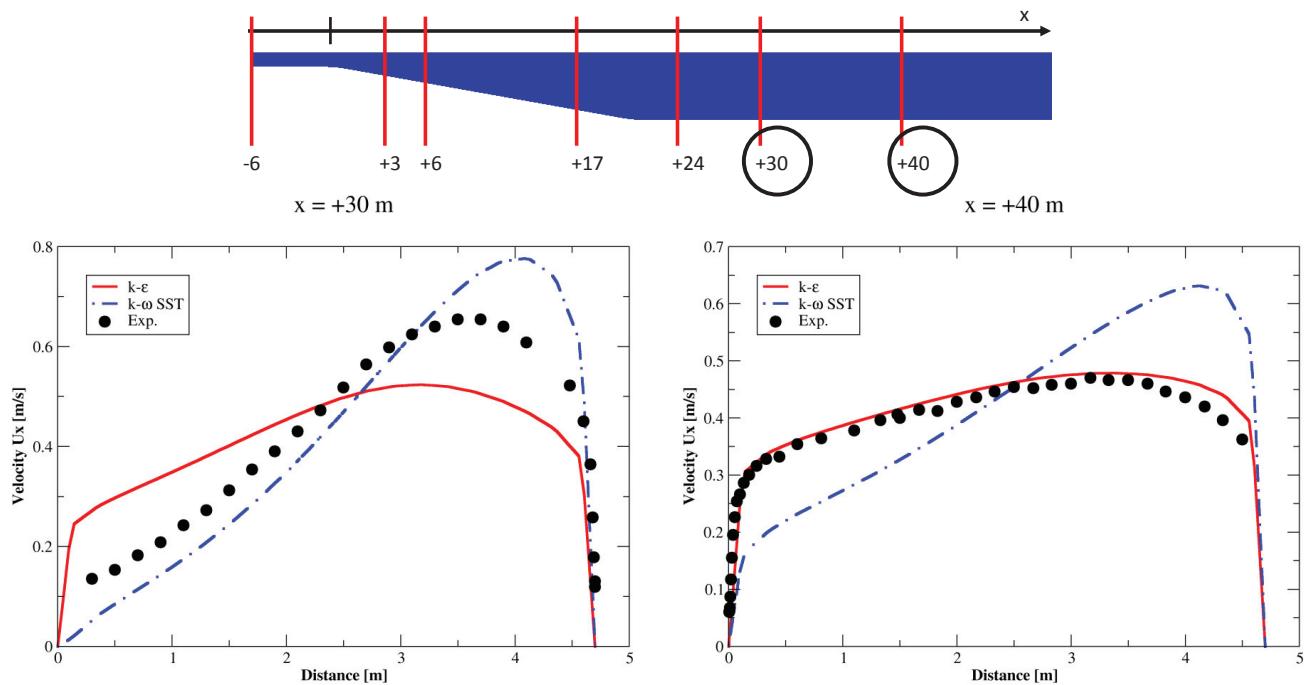


T. Lucchini: Incompressible, steady-state turbulent flow simulations in OpenFOAM®

# Diffuser simulation



## Comparison with experimental data: velocity profiles

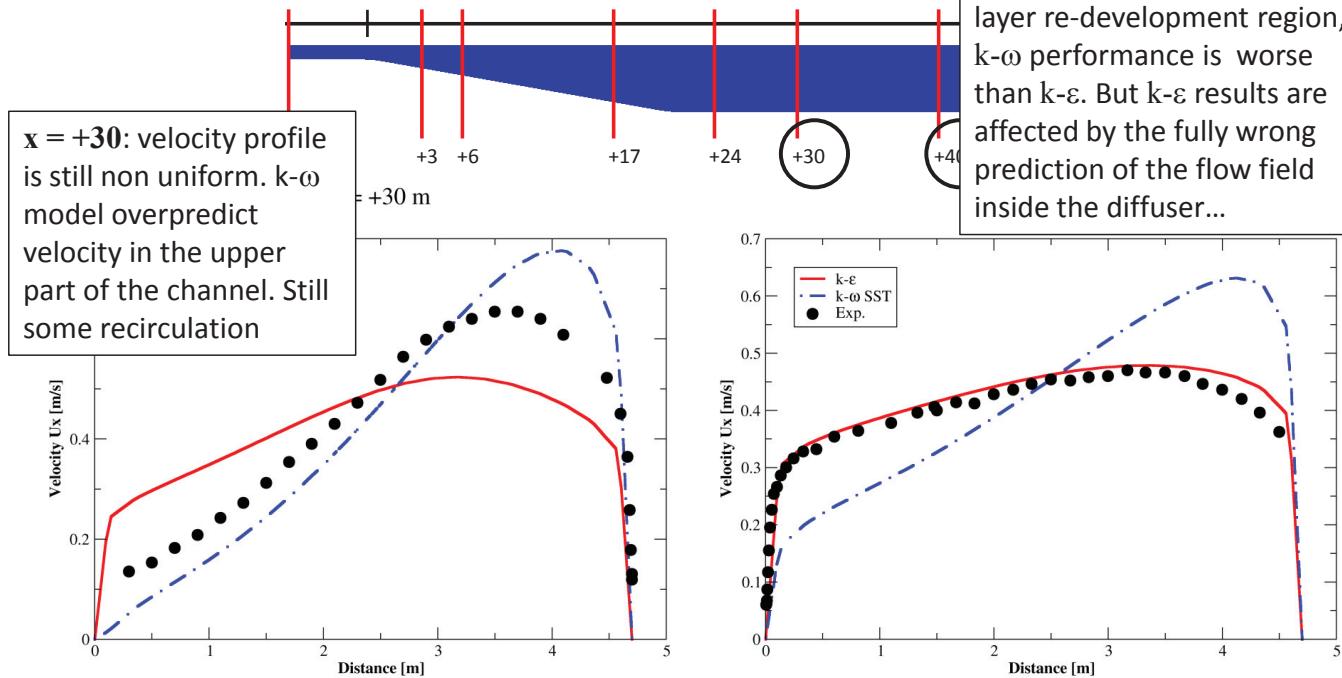


T. Lucchini: Incompressible, steady-state turbulent flow simulations in OpenFOAM®

# Diffuser simulation



## Comparison with experimental data: velocity profiles



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

# 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...

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

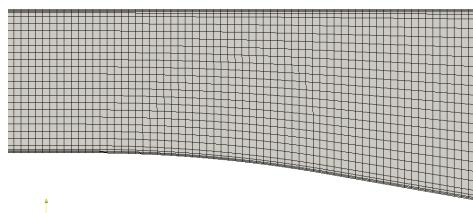
# Diffuser simulation

## Improving (?) results of the ERCOFTAC diffuser case:

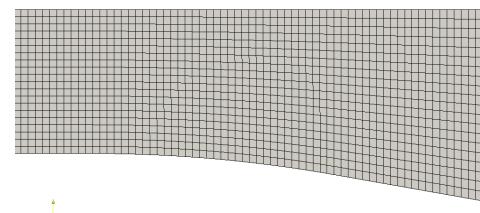
- Use a more refined boundary layer:
  - Create a new case, called `diffuserKOmegaSSTBL` starting from the `diffuserKOmegaSST` one.
  - Use the `refineWallLayer` utility to refine cells next to patches in the normal direction. It requires a patch name and a weight factor. Run it three times with the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



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

# 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
```

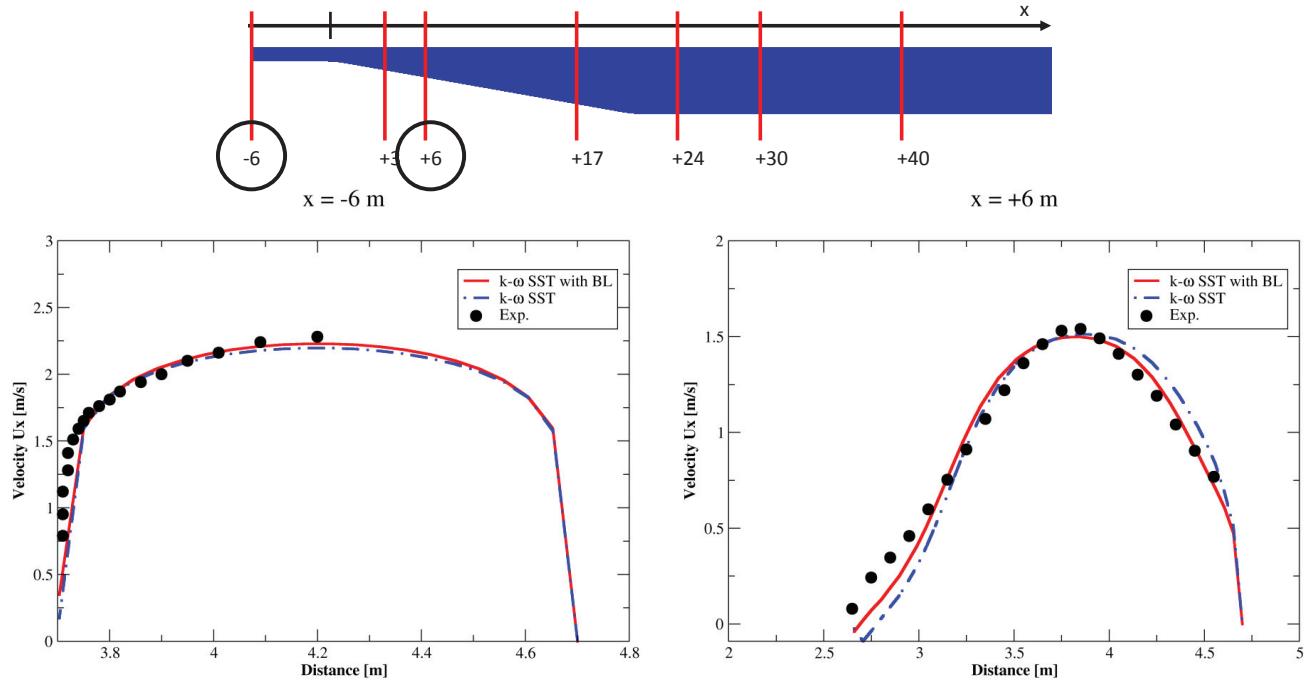
- $y+$  is much lower than before. Let's see the results...

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

# Diffuser simulation



## Comparison with experimental data: velocity profiles

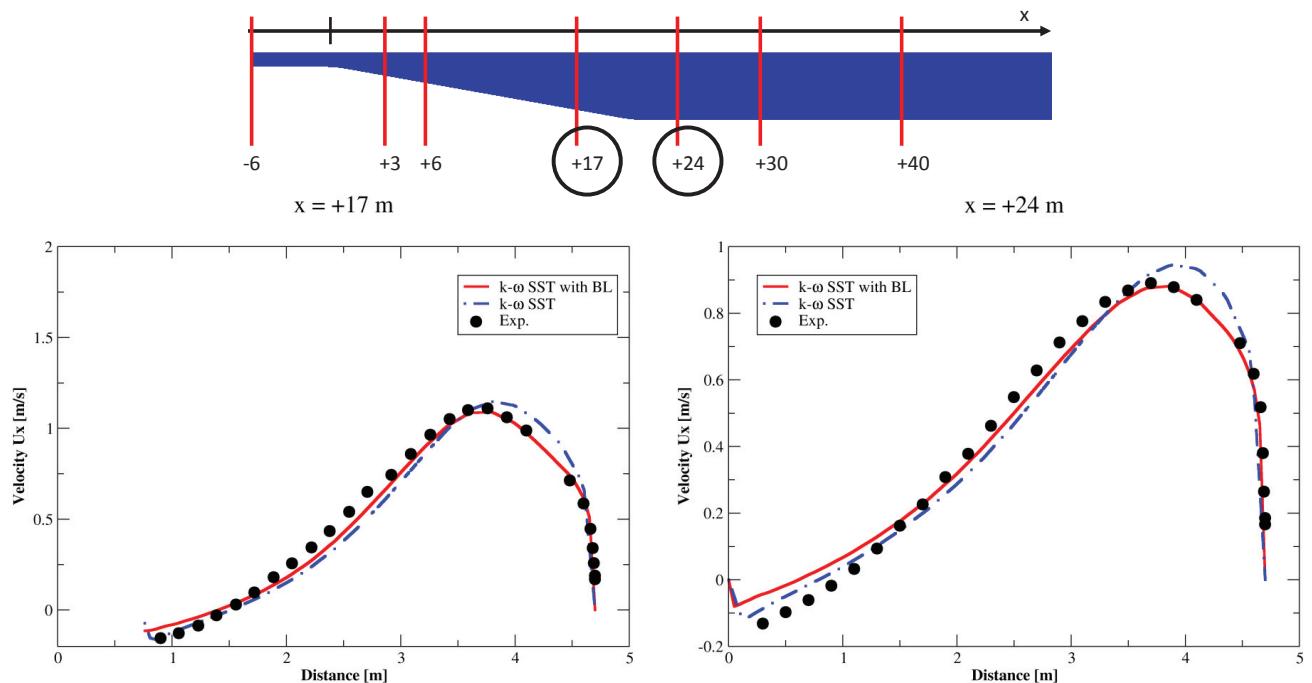


T. Lucchini: Incompressible, steady-state turbulent flow simulations in OpenFOAM®

# Diffuser simulation



## Comparison with experimental data: velocity profiles

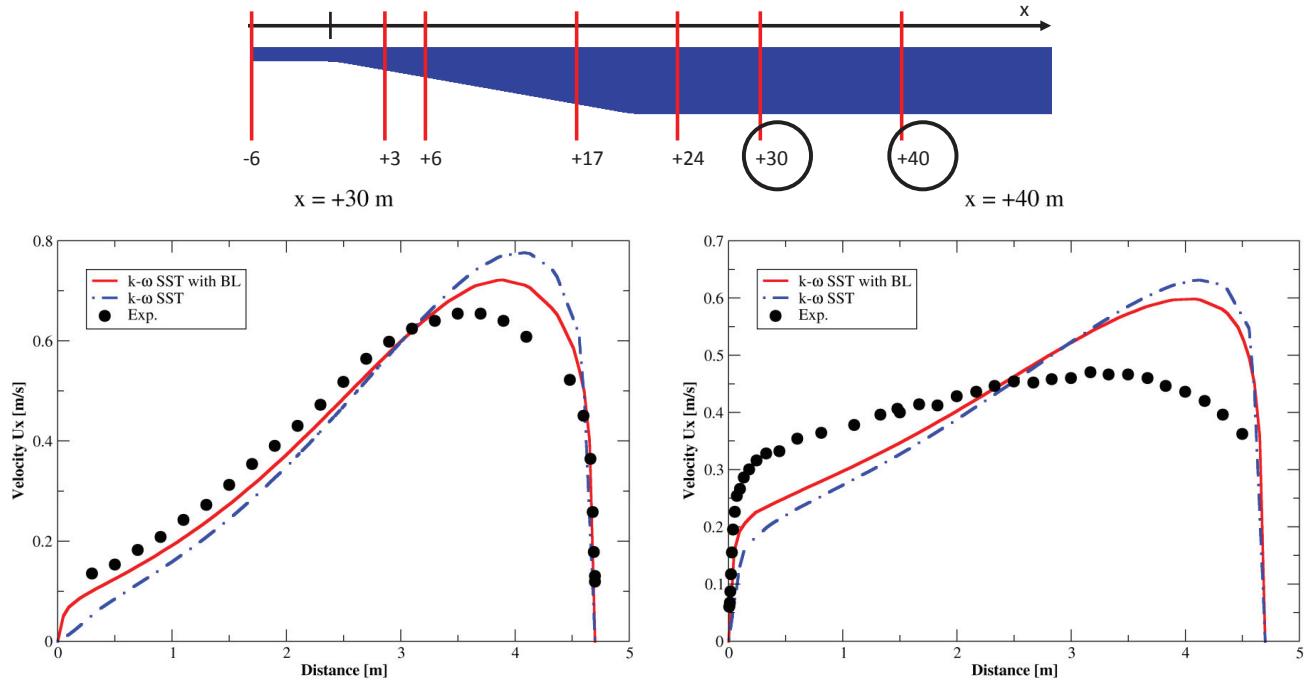


T. Lucchini: Incompressible, steady-state turbulent flow simulations in OpenFOAM®

# Diffuser simulation



## Comparison with experimental data: velocity profiles



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

# 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  $k-\varepsilon$  or  $k-\omega$ .

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

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

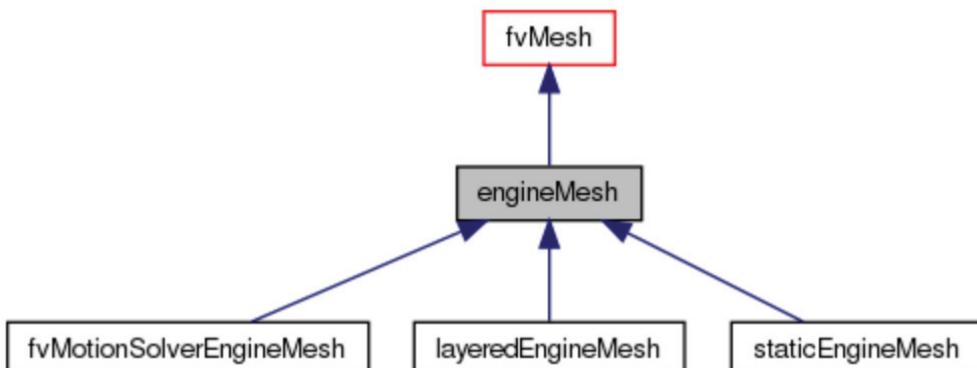
# Customise OpenFOAM to model an engine with moving valves

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 developments 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;
}

```

Database 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 read as inputs 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

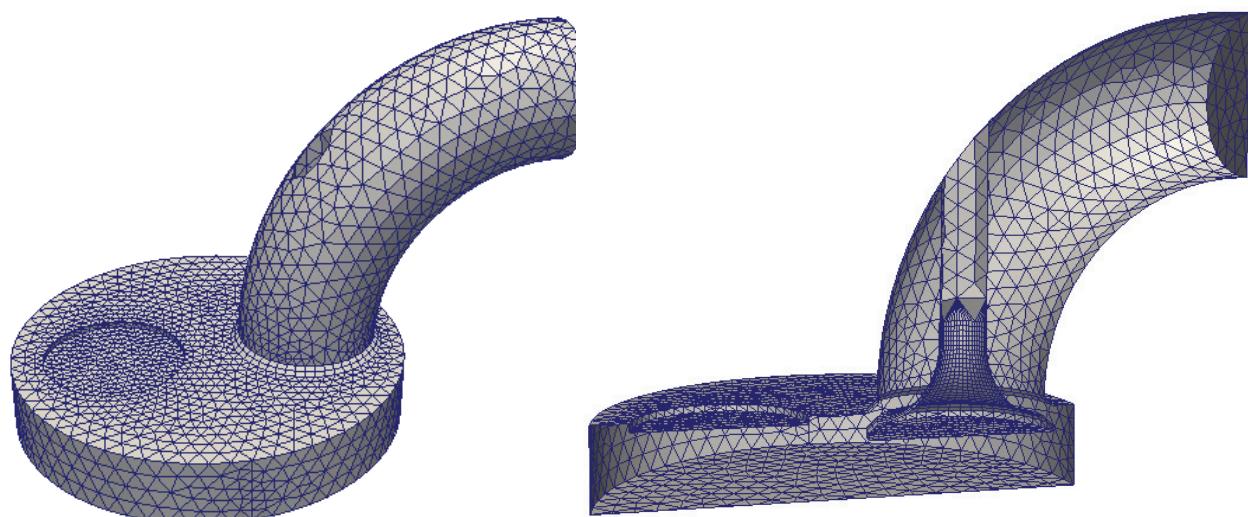
Liner points are assigned interpolating the piston motion along the piston axis

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(const
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 centered 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

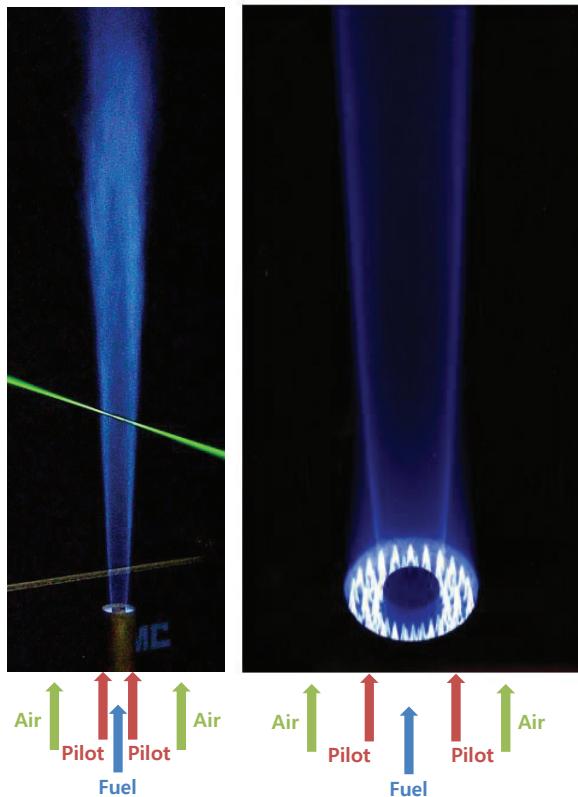
- SLFM (Stationary Laminar Flamelet Model)
- Code Structure
- Tutorial

### III EdmParcelFoam

- Case Description
- Code Structure
- Tutorial

# Case description

## Piloted CH<sub>4</sub>/Air Flame



### ● Burner Dimensions

- Main jet inner diameter,  $d = 7.2 \text{ mm}$
- 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% CH<sub>4</sub>, 75% dry air by volume
- Main jet kinematic viscosity = 1.58e-05  $\text{m}^2/\text{s}$
- 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)

Combustion Laboratory Pohang University of Science and Technology POSTECH

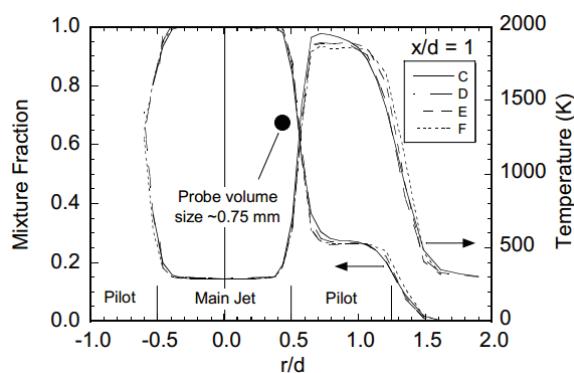
# Case description

## Piloted CH<sub>4</sub>/Air Flame

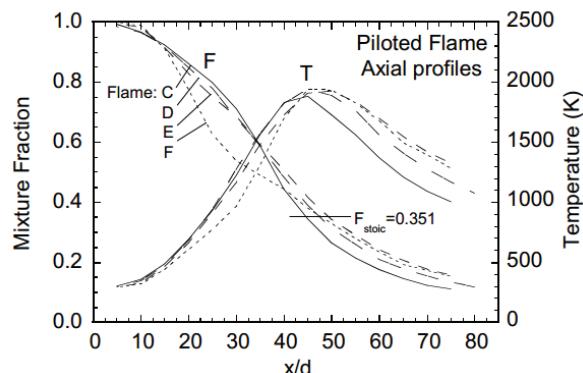
### ● Composition

- Main jet (mass fraction) – N<sub>2</sub> : 0.647, CH<sub>4</sub> : 0.156, O<sub>2</sub> : 0.197
- Pilot (mass fraction) – N<sub>2</sub> : 0.7342, O<sub>2</sub> : 0.0540, O : 7.47e-4, H<sub>2</sub> : 1.29e-4, H : 2.48e-5  
H<sub>2</sub>O : 0.0942, CO : 4.07e-3, CO<sub>2</sub> : 0.1098, OH : 0.0028, NO : 4.80e-6
- Co-flow (mass fraction) – N<sub>2</sub> : 0.767, O<sub>2</sub> : 0.233

### ● Measured profiles



Measured radial profiles of Favre-average mixture fraction and temperature at  $x/d=1$  in the four turbulent piloted flames

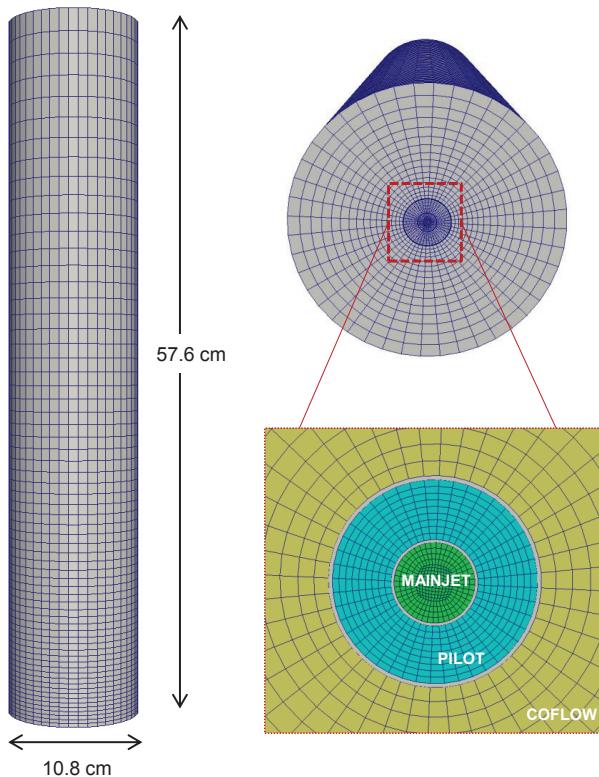


Axial profiles of measured mixture fraction and temperature (Favre average) in piloted flames C, D, E, and F

Combustion Laboratory Pohang University of Science and Technology POSTECH

# 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        |        |
| <br>Overall number of cells of each type:                      |          |        |
| hexahedra:                                                     | 102245   |        |
| prisms:                                                        | 0        |        |
| wedges:                                                        | 0        |        |
| pyramids:                                                      | 0        |        |
| tet wedges:                                                    | 0        |        |
| tetrahedra:                                                    | 0        |        |
| polyhedra:                                                     | 0        |        |
| <br>Checking patch topology for multiply connected surfaces... |          |        |
| Patch                                                          | Faces    | Points |
| MAINJET                                                        | 297      | 320    |
| PILOT                                                          | 396      | 440    |
| COFLOW                                                         | 704      | 748    |
| OUTLET                                                         | 1573     | 1596   |
| CASING                                                         | 2860     | 2904   |
| INNERWALL                                                      | 88       | 132    |
| OUTERWALL                                                      | 88       | 132    |

Combustion Laboratory POSTECH  
POHANG UNIVERSITY OF SCIENCE AND TECHNOLOGY

# simpleEdmFoam

## Governing Equations

$$\frac{\partial \bar{\rho}}{\partial t} + \frac{\partial}{\partial x_k} \bar{\rho} \tilde{v}_k = 0$$

$$\frac{\partial}{\partial t} \bar{\rho} \tilde{v}_i + \frac{\partial}{\partial x_k} \bar{\rho} \boxed{\textcolor{red}{v}_k \textcolor{red}{v}_i} = - \frac{\partial \bar{p}}{\partial x_i} + \frac{\partial}{\partial x_k} \bar{\tau}_{ik} + \bar{g}_i$$

$$\frac{\partial}{\partial t} \bar{\rho} \tilde{Y}_i + \frac{\partial}{\partial x_k} \bar{\rho} \boxed{\textcolor{red}{v}_k Y_i} = - \frac{\partial}{\partial x_k} \bar{J}_{ik} + \boxed{\textcolor{red}{\dot{w}_i}} \rightarrow \text{Nonlinear Reaction Term}$$

$$\frac{\partial}{\partial t} \bar{\rho} \tilde{h} + \frac{\partial}{\partial x_k} \bar{\rho} \boxed{\textcolor{red}{v}_k h} = \frac{\partial \bar{p}}{\partial t} + \frac{\partial}{\partial x_k} [\frac{\mu}{\sigma} \frac{\partial \bar{h}}{\partial x_k} + \mu \sum_{i=1}^N (\frac{1}{Sc_i} - \frac{1}{\sigma}) \bar{h}_i \frac{\partial \bar{Y}_i}{\partial x_k}]$$

$$\bar{p} = \bar{\rho} R \tilde{T}$$

Nonlinear Convection Term

$$\tilde{h} = h(\tilde{Y}_i, \bar{p}, \tilde{T})$$

Combustion Laboratory POSTECH  
POHANG UNIVERSITY OF SCIENCE AND TECHNOLOGY

# simpleEdmFoam

## EDM(Eddy Dissipation Model)

### ● EDM (Eddy Dissipation Model)

$$\left. \begin{array}{l} R_f = A \cdot \bar{Y}_f (\varepsilon / k) \\ R_f = A(\bar{Y}_{O_2} / r_f)(\varepsilon / k) \\ R_f = A \cdot B(\bar{Y}_P / (1+r_f))(\varepsilon / k) \end{array} \right\} \text{Minimum } R_f \rightarrow \text{Local mean rate of combustion}$$

$A, B$  : 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

$$R_{i,r} = v'_{i,r} M_{w,i} A \frac{\rho \varepsilon}{k} \min\left(\frac{Y_R}{v'_{R,j} M_{w,R}}\right)$$
$$R_{i,r} = v'_{i,r} M_{w,i} AB \frac{\rho \varepsilon}{k} \frac{\sum_P Y_P}{\sum_j^N v'_{j,r} M_{w,j}}$$

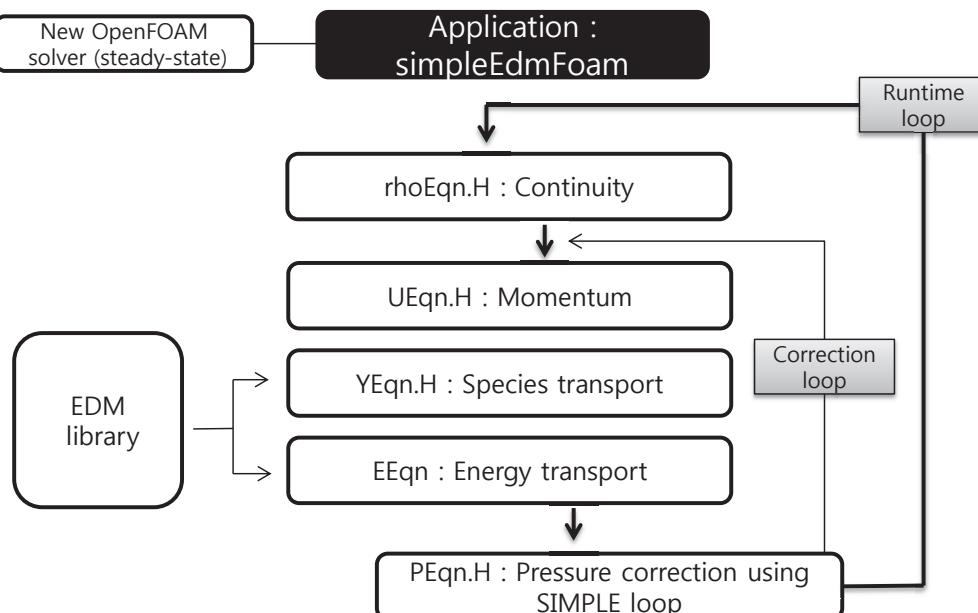
Arrhenius

$$R_{i,r} = \Gamma(v''_{i,r} - v'_{i,r}) \left( k_{f,r} \prod_{j=1}^N [C_{j,r}]^{v'_j} - k_{b,r} \prod_{j=1}^N [C_{j,r}]^{v''_j} \right)$$
$$k_r = A_r T^{b_r} \exp(-E_r / RT)$$

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

# simpleEdmFoam

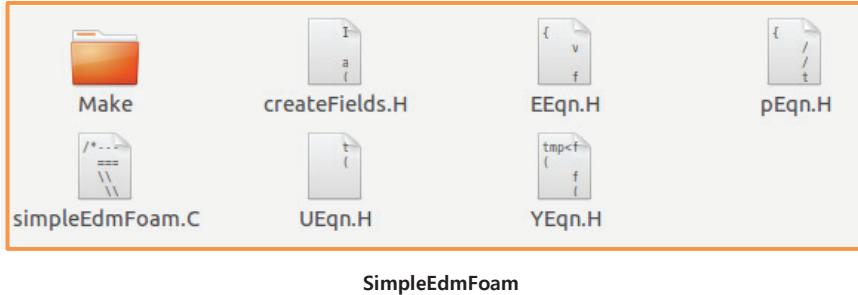
## Code Structure



# simpleEdmFoam

## Code Structure

/solvers/simpleEdmFoam



/libs/combustionModels\_POSTECH/EDM



# 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

```
tmp<fvVectorMatrix> UEqn  
(  
    fvm::div(phi, U)  
    + turbulence->divDevRhoReff(U)  
==  
    rho.dimensionedInternalField()*g  
);
```

```
volScalarField& Yi = Y[i];  
  
fvScalarMatrix YEqn  
(  
    mvConvection->fvmDiv(phi, Yi)  
    - fvm::laplacian(turbulence->muEff(), Yi)  
==  
    combustion->R(Yi)  
    + fvOptions(rho, Yi)  
);
```

```
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();
    }
}
```

**EDM.C**

Combustion Laboratory POSTECH

# 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_POSTECH/lnInclude
```

**options**

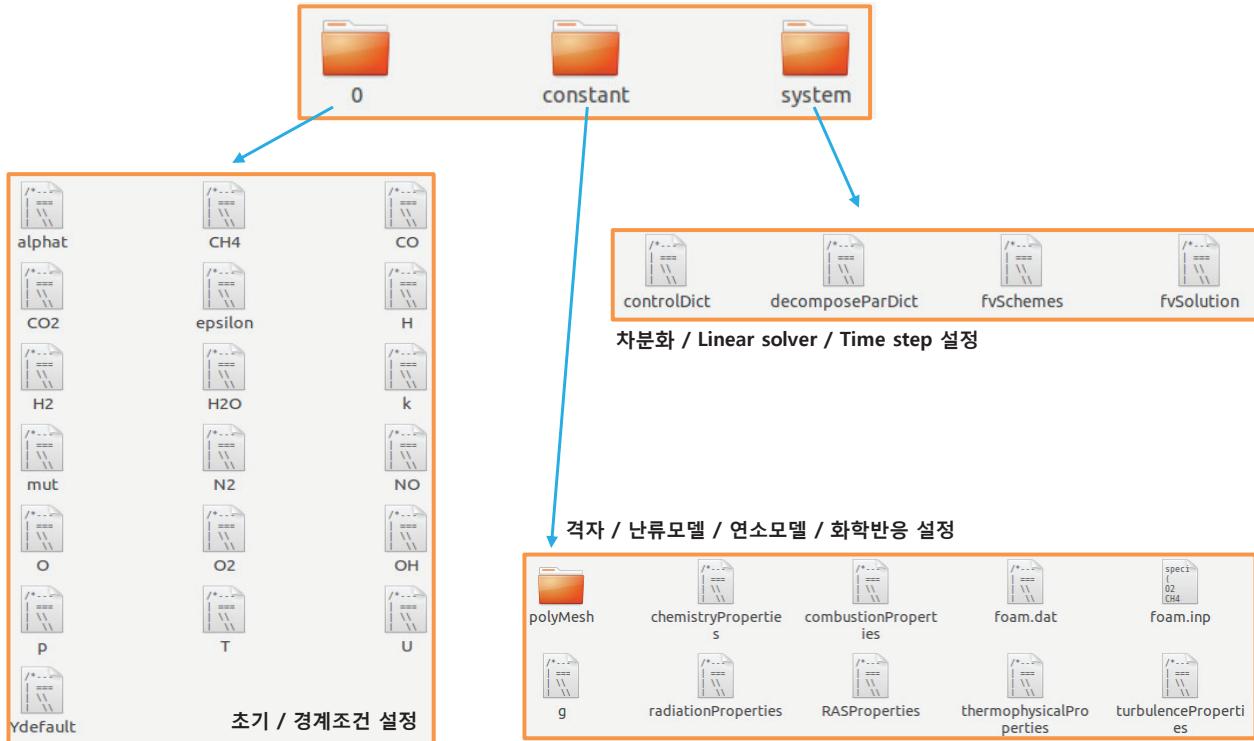
```
EXE_LIBS = \
    -L$(FOAM_USER_LIBBIN) \
    -lcombustionModels_POSTECH \
    -lchemistryModel_POSTECH \
    -lfiniteVolume \
    -lmeshTools \
    -lcompressibleTurbulenceModel \
    -lcompressibleRASModels \
    -lcompressibleLESModels \
    -lspecie \
    -lfluidThermophysicalModels \
    -lliquidProperties \
    -lliquidMixtureProperties \
    -lsolidProperties \
    -lsolidMixtureProperties \
    -lthermophysicalFunctions \
    -lreactionThermophysicalModels \
    -lSLGThermo \
    -lradiationModels \
    -LODE \
    -lregionModels \
    -lsurfaceFilmModels \
    -lfvOptions \
    -lsampling|
```

Combustion Laboratory POSTECH

# simpleEdmFoam

## Case Folder

/tutorials/EDM\_reacting\_flow/

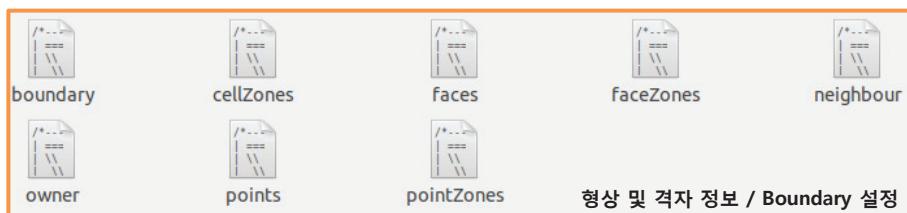


Combustion Laboratory  
POHANG UNIVERSITY OF SCIENCE AND TECHNOLOGY POSTECH

# simpleEdmFoam

## PolyMesh

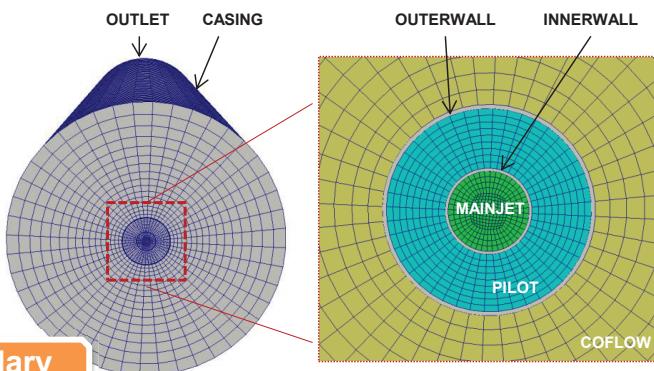
/tutorials/EDM\_reacting\_flow/constant/polyMesh



```
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          patch;
    inGroups      1(wall);
    nFaces        2860;
    startFace     306702;
}
INNERWALL
{
    type          patch;
    inGroups      1(wall);
    nFaces        88;
    startFace     309562;
}
OUTERWALL
{
    type          patch;
    inGroups      1(wall);
    nFaces        88;
    startFace     309650;
}
```

**boundary**



Combustion Laboratory  
POHANG UNIVERSITY OF SCIENCE AND TECHNOLOGY POSTECH

# 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          value
        fixedValue;  uniform 0.1966623;
    }
    PILOT
    {
        type          value
        fixedValue;  uniform 0.0540244;
    }
    COFLOW
    {
        type          value
        fixedValue;  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          value
        fixedValue;  uniform 0.6473821;
    }
    PILOT
    {
        type          value
        fixedValue;  uniform 0.7342;
    }
    COFLOW
    {
        type          value
        fixedValue;  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          value
        fixedValue;  uniform 0.1559556;
    }
    PILOT
    {
        type          value
        fixedValue;  uniform 0.0;
    }
    COFLOW
    {
        type          value
        fixedValue;  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**

# simpleEdmFoam

## '0' Folder

- U(velocity), T(temperature), p壓力)

```
dimensions [0 1 -1 0 0 0];
internalField uniform (0 0.9 0);
boundaryField {
    MAINJET {
        type value
        value fixedValue;
        uniform (0 49.6 0);
    }
    PILOT {
        type value
        value fixedValue;
        uniform (0 11.4 0);
    }
    COFLOW {
        type value
        value fixedValue;
        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 value
        value fixedValue;
        uniform 294.0;
    }
    PILOT {
        type value
        value fixedValue;
        uniform 1880.0;
    }
    COFLOW {
        type value
        value fixedValue;
        uniform 291.0;
    }
    OUTLET {
        type zeroGradient;
    }
    CASING {
        type slip;
    }
    INNERWALL {
        type zeroGradient;
    }
    OUTERWALL {
        type zeroGradient;
    }
}
```

**T**

```
dimensions [1 -1 -2 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**

# 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";
inertSpecies

```

**thermophysicalProperties**

```

reactions
{
    methaneReaction1
    {
        type      irreversibleArrheniusReaction;
        reaction "CH4^(0.7) + 1.502^(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 + 0.502 = 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 5.336483e-10 -3.146
                      (-5.14911 -0.0136622 4.91454e-05 -4.84247e-08 1.666
        lowCpCoeffs (-1.65326 0.0100263 -3.31661e-06 5.336483e-10 -3.146
                      (-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
                      (-3.57953 -0.000610354 1.01681e-06 9.07006e-10 -9.0
        lowCpCoeffs (-3.04849 0.00135173 -4.85794e-07 7.88536e-11 -4.69
                      (-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*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**

# simpleEdmFoam

## Solver

- Calculation / MPI(Message Passing Interface)

```
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
```

controldict

```
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         0 1 2 3
```

decomposePardict

Combustion Laboratory  
POHANG UNIVERSITY OF SCIENCE AND TECHNOLOGY



# simpleEdmFoam

## Tutorial

- Open ‘Terminal’ : click

- ~\$ cd tutorials/EDM\_reacting\_flow :~\$ cd tutorials/EDM\_reacting\_flow

- ~\$ 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.
```

- ~\$ mpirun -np 4 simpleEdmFoam -parallel

```
:~/tutorials/EDM_reacting_flow$ mpirun -np 4 simpleEdmFoam -parallel
```

Combustion Laboratory  
POHANG UNIVERSITY OF SCIENCE AND TECHNOLOGY



# 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 |
|------|---------------|----------|
| Time | min/max value | Residual |

# simpleEdmFoam

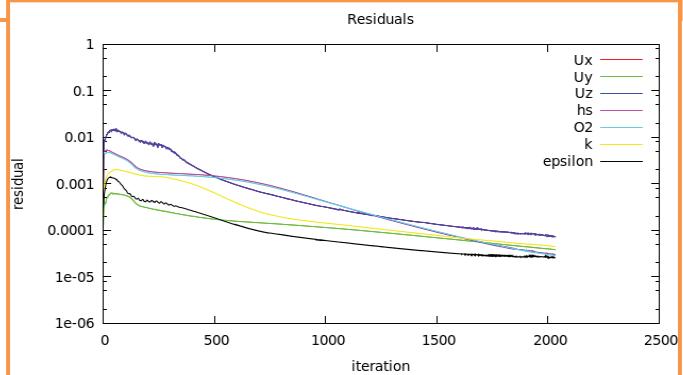
## Tutorial

- Calculating

```
Time = 2000  
④  
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**



Combustion Laboratory POSTECH

# simpleEdmFoam

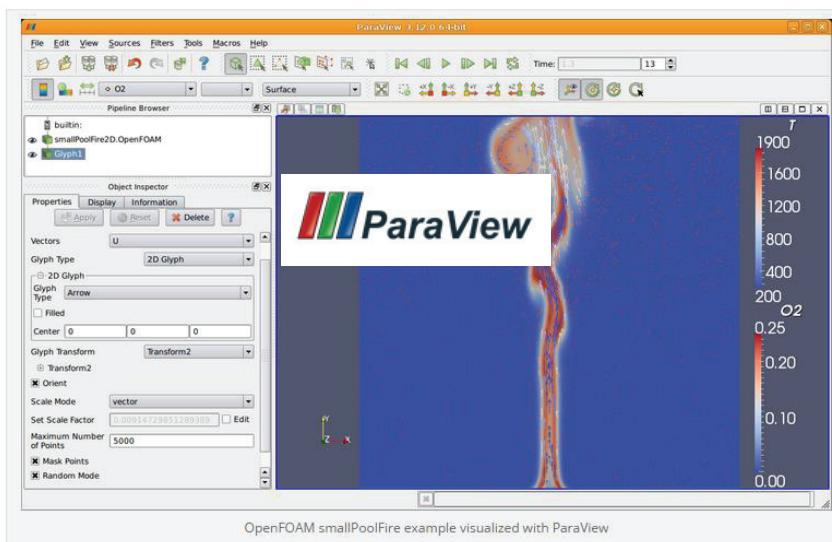
## Post Processing

- ~\$ reconstructPar

```
:~/tutorials/EDM_reacting_flow$ reconstructPar
```

- ~\$ cd tutorials/EDM\_reacting\_flow/paraFoam

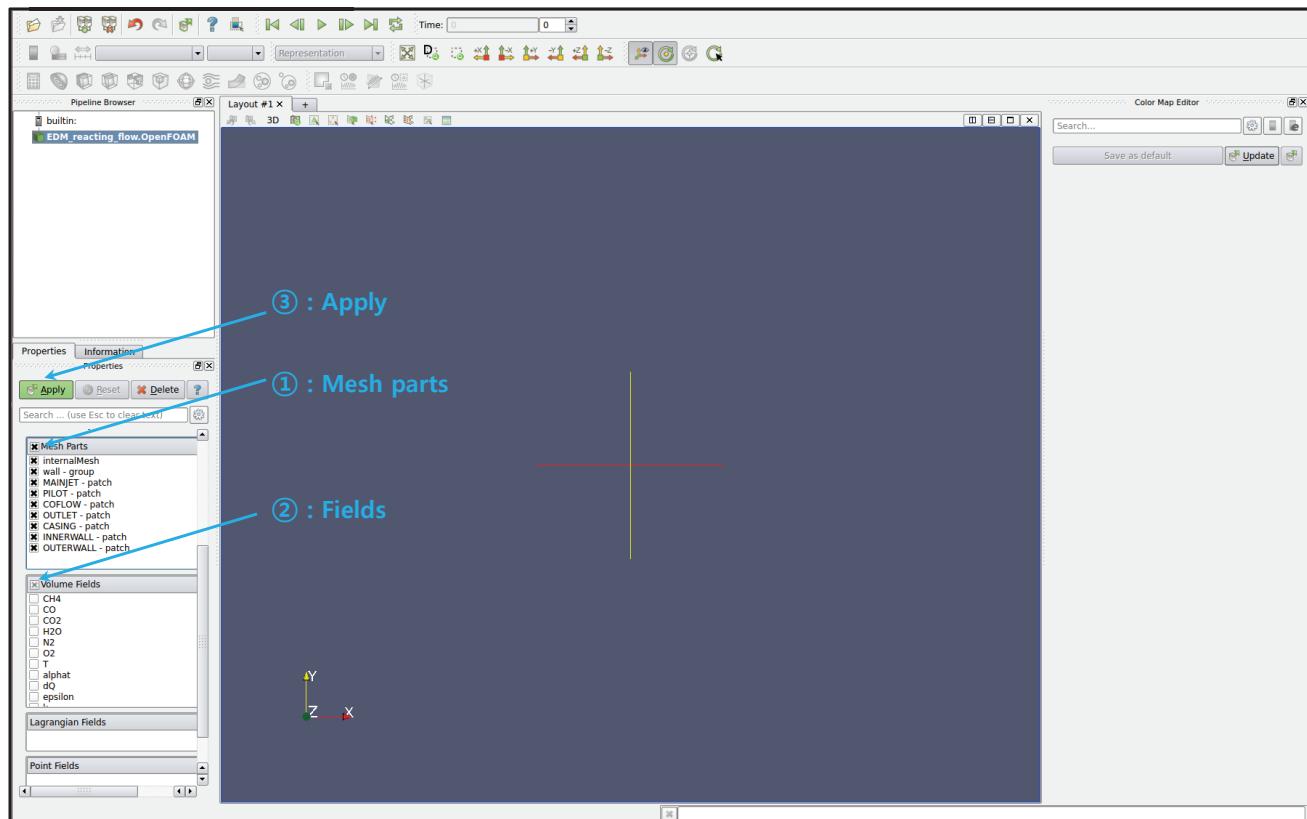
```
:~/tutorials/EDM_reacting_flow$ paraFoam
```



Combustion Laboratory POSTECH

# simpleEdmFoam

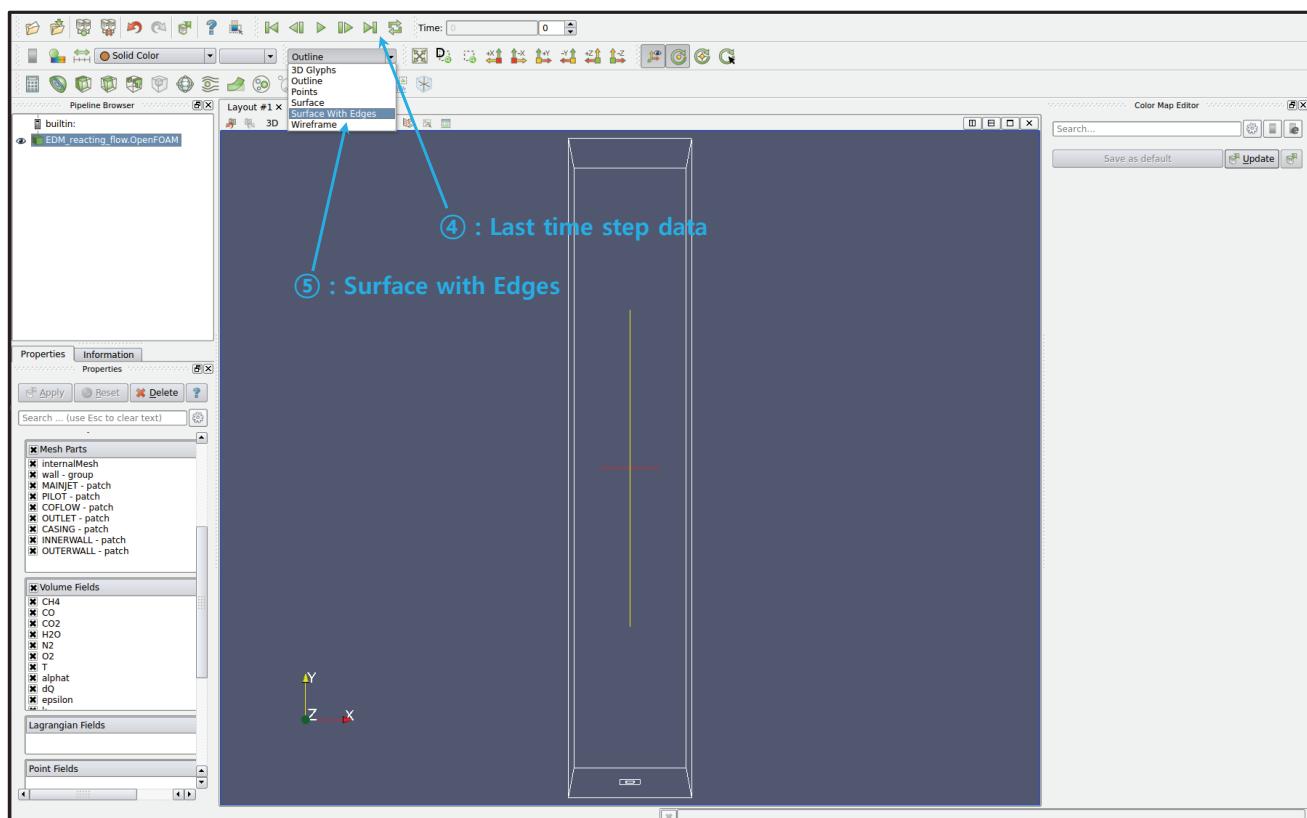
## Post Processing



Combustion Laboratory POHANG UNIVERSITY OF SCIENCE AND TECHNOLOGY POSTECH

# simpleEdmFoam

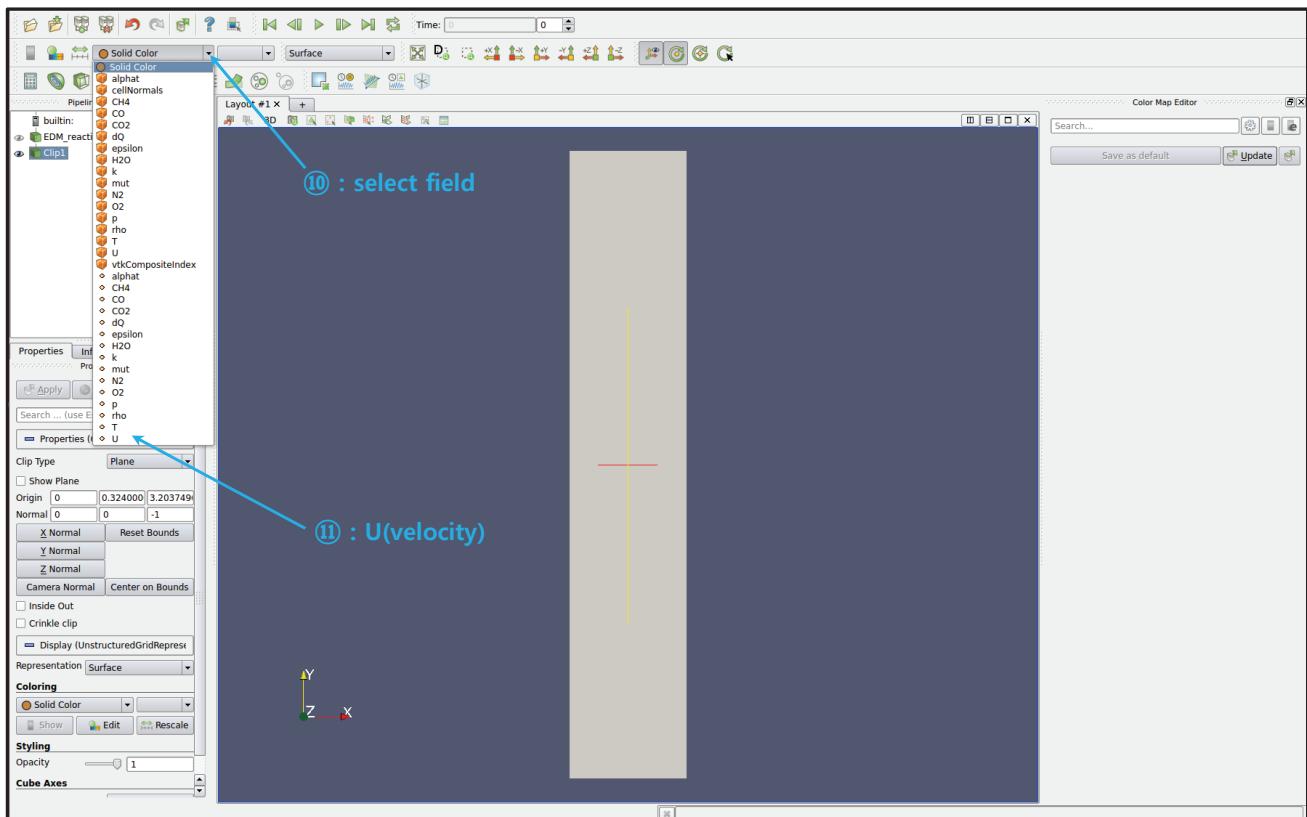
## Post Processing



Combustion Laboratory POHANG UNIVERSITY OF SCIENCE AND TECHNOLOGY POSTECH

# simpleEdmFoam

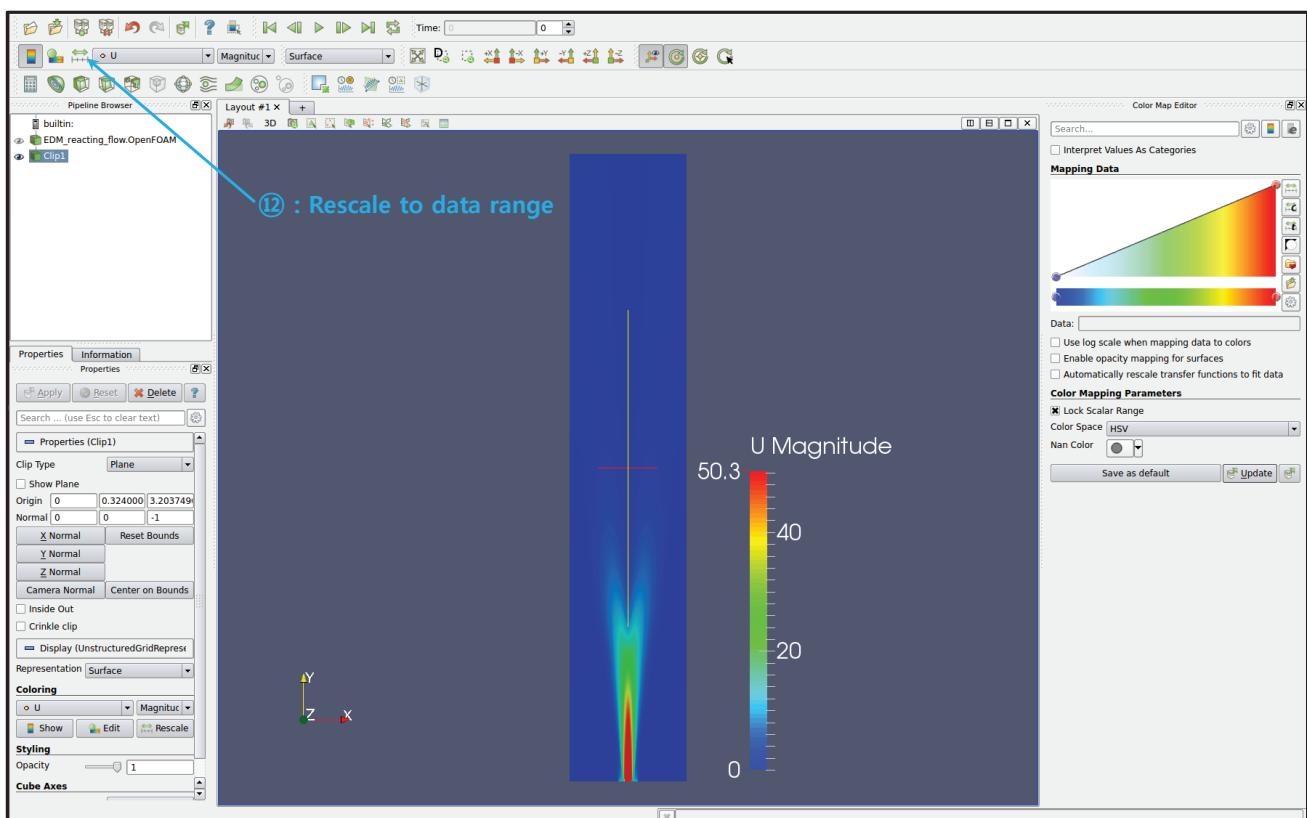
## Post Processing



Combustion Laboratory POHANG UNIVERSITY OF SCIENCE AND TECHNOLOGY POSTECH

# simpleEdmFoam

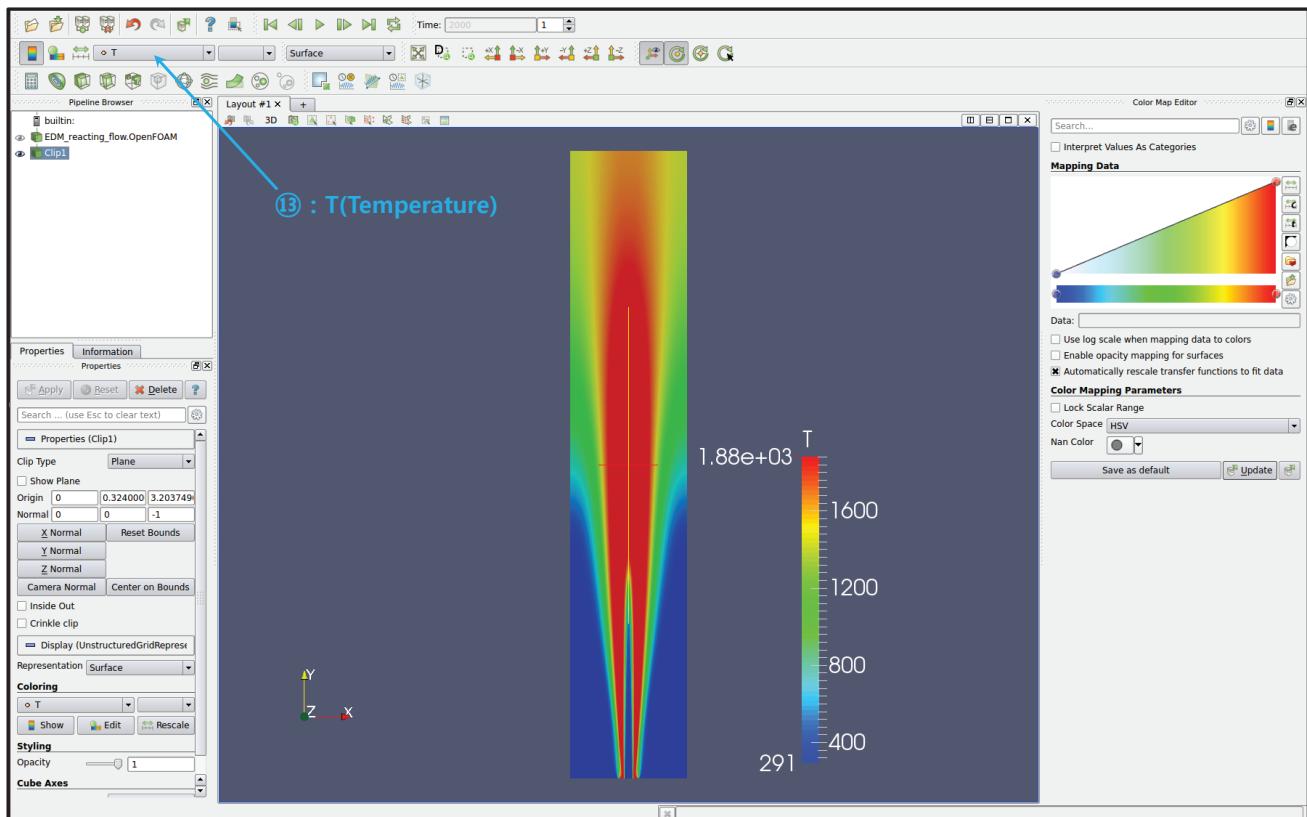
## Post Processing



Combustion Laboratory POHANG UNIVERSITY OF SCIENCE AND TECHNOLOGY POSTECH

# simpleEdmFoam

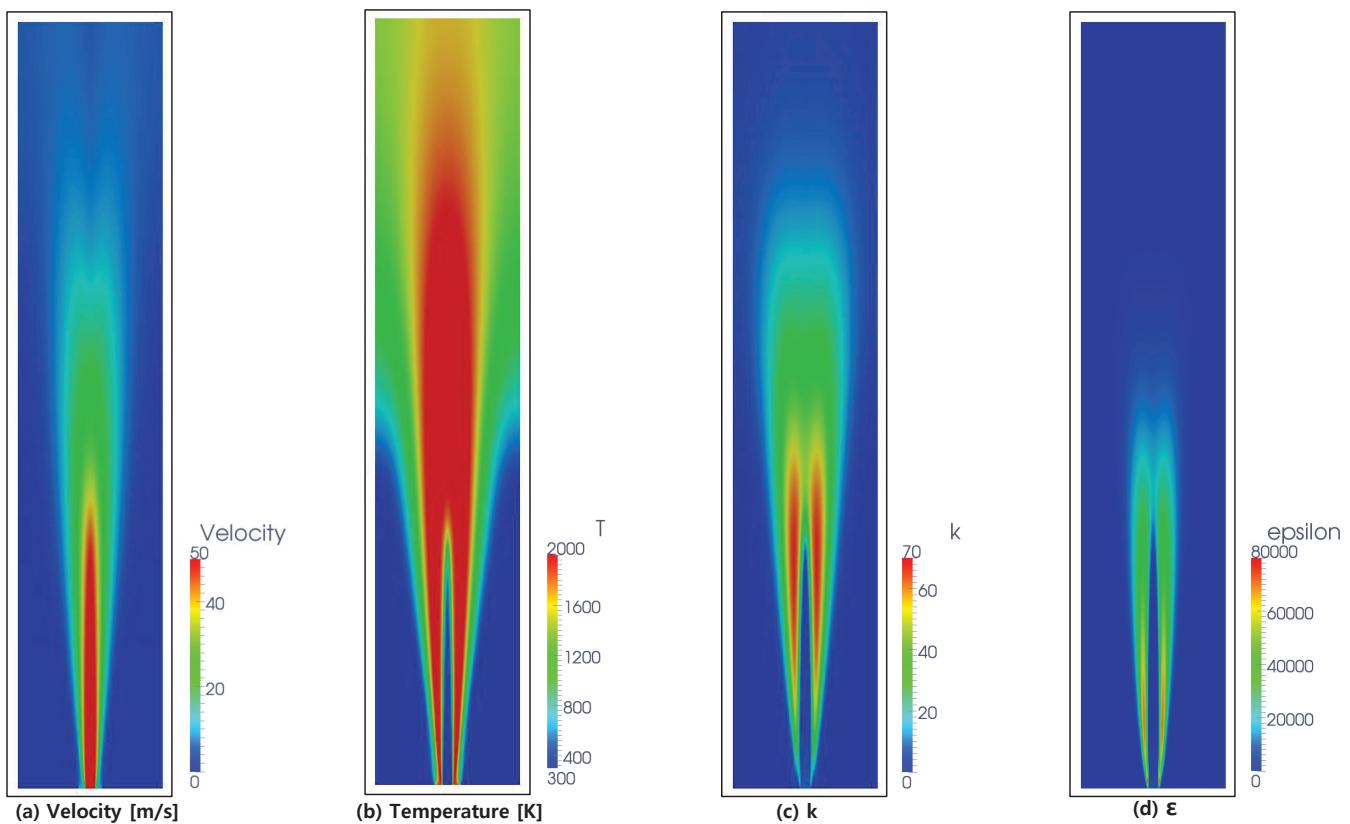
## Post Processing



Combustion Laboratory POSTECH

# simpleEdmFoam

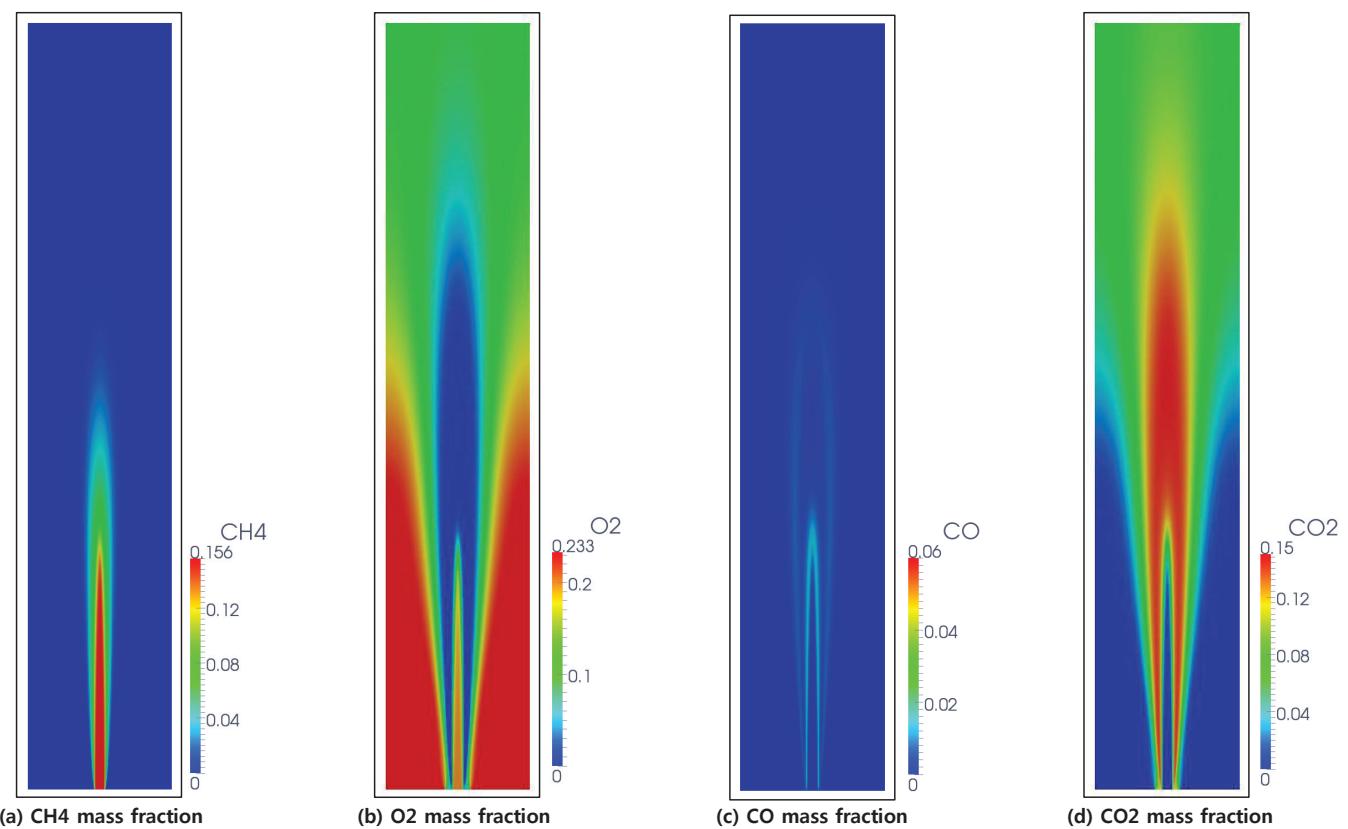
## Results



Combustion Laboratory POSTECH

# simpleEdmFoam

## Results



Combustion Laboratory POHANG UNIVERSITY OF SCIENCE AND TECHNOLOGY POSTECH

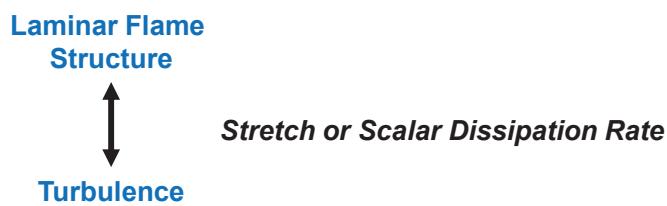
# SLFMFoam

## Numerical Combustion Model – Nonpremixed

- Equilibrium Assumption – **Infinitely fast chemistry**

- **Laminar Flamelet Model**

- Steady – **SLFM**(Stationary Laminar Flamelet Model)
- Transient – **RIF**(Representative Interactive Flamelet)



- **Conditional Averaging**

- CMC**(Conditional Moment Closure)

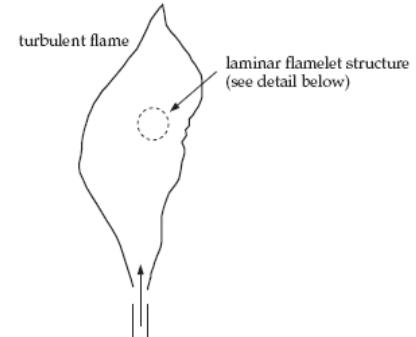
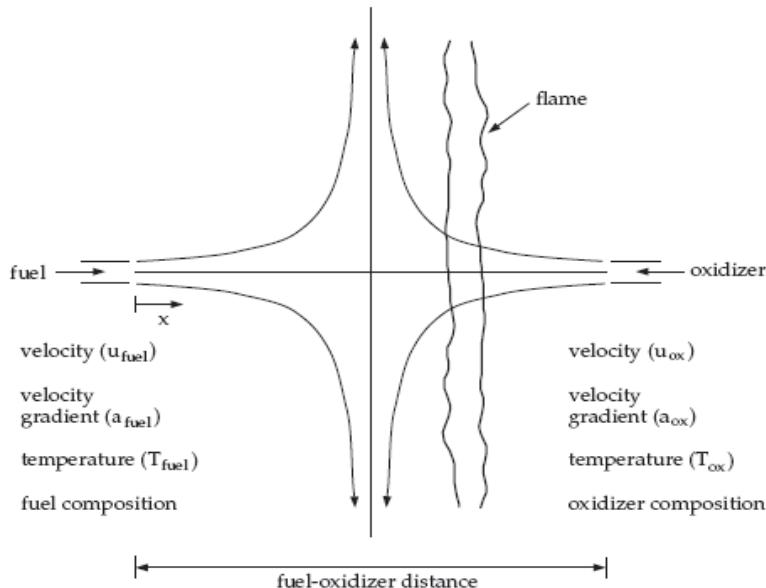
- Deterministic relationships between mixture fraction and all other reactive scalars.
- 1<sup>st</sup> order closure for chemical reaction rate.

Combustion Laboratory POHANG UNIVERSITY OF SCIENCE AND TECHNOLOGY POSTECH

# 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 ( $N_{st}$ )



$$\frac{\partial Y_i}{\partial t} = R_{i,kin} + D_i \chi \frac{\partial^2 Y_i}{\partial f^2}$$

$$\frac{\partial T}{\partial t} = - \sum_{i=1}^N \frac{h_i R_{i,kin}}{C_p} + \alpha \chi \frac{\partial^2 T}{\partial f^2}$$

Combustion Laboratory POHANG UNIVERSITY OF SCIENCE AND TECHNOLOGY  **POSTECH**

# 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

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

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

- Mixture fraction

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

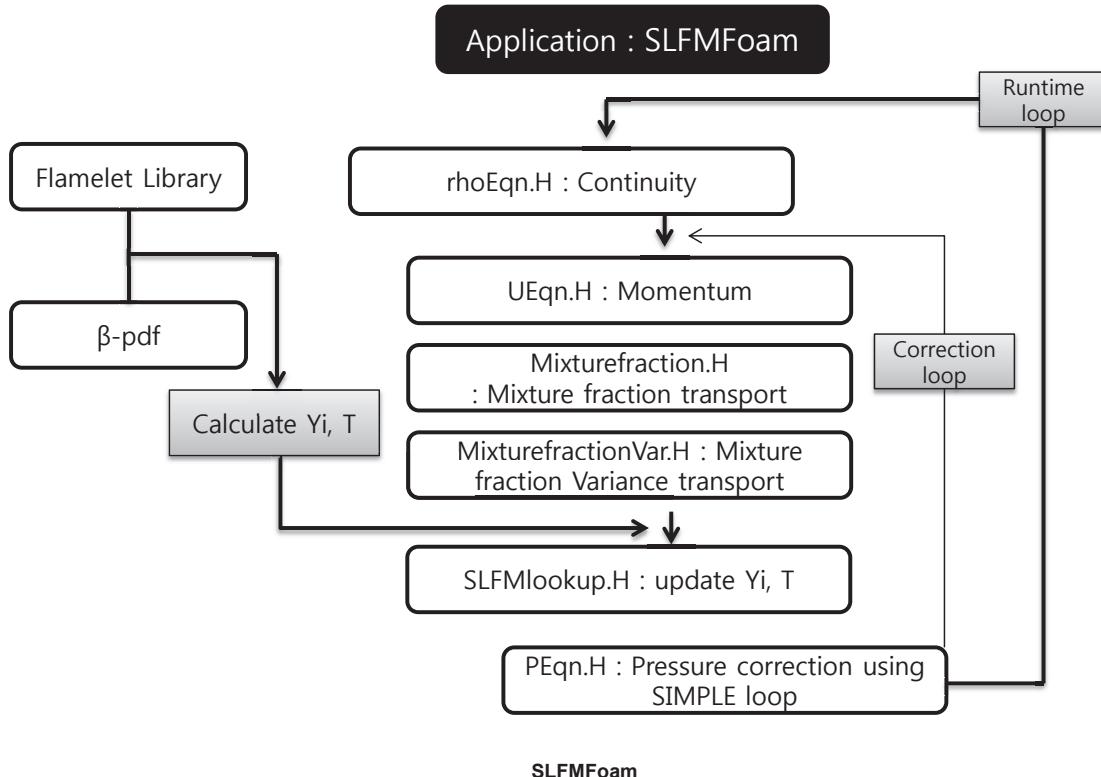
- Mixture fraction variance

$$\frac{\partial (\bar{\rho} \tilde{\xi''}^2)}{\partial t} + \nabla \cdot (\bar{\rho} \tilde{\mathbf{u}} \tilde{\xi''}^2) = \nabla \cdot \left[ \frac{\mu_t}{Sc_{\tilde{\xi''}^2}} \nabla \tilde{\xi''}^2 \right] + \frac{2\mu_t}{Sc_{\tilde{\xi''}^2}} (\nabla \tilde{\xi})^2 - \bar{\rho} \tilde{\chi}$$

Combustion Laboratory POHANG UNIVERSITY OF SCIENCE AND TECHNOLOGY  **POSTECH**

# SLFMFoam

## Code Structure



# 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 <iostream>
#include <iomanip>

#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
);

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

volScalarField SDR
(
    IOobject
    (
        "SDR",
        runTime.timeName(),
        mesh,
        IOobject::NO_READ,
        IOobject::AUTO_WRITE
    ),
    mesh,
    dimensionedScalar("SDR", dimEnergy/dimTime/dimEnergy, 0.0)
);

```

Mixture fraction

Mixture fraction variance

Scalar dissipation rate

## readCMCProperties.H

```

volScalarField deltaftn
(
    IOobject
    (
        "deltaftn",
        runTime.timeName(),
        mesh,
        IOobject::NO_READ,
        IOobject::NO_WRITE
    ),
    mesh,
    dimensionedScalar("deltaftn", dimless, -1.0)
);

IDictionary 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 data(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

```

// --- 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 "SLFMTlookup.H"
    #include "pEqn.H"
}

turbulence->correct();
runTime.write();

```

SLFMFoam.C

```

fvScalarMatrix mfEqn
(
    (
        mvConvection->fvmDiv(phi, mf)
        - fv::laplacian(1.47*turbulence->mut(), mf)
        + fvOptions(rho, mf)
    )
);

```

Mixturefraction.C

```

tmp<fvVectorMatrix> UEqn
(
    fvm::div(phi, U)
    + turbulence->divDevRhoReff(U)
    ==
    rho.dimensionedInternalField()*g
);

```

UEqn.C

```

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

fvScalarMatrix mfVarEqn
(
    (
        mvConvection->fvmDiv(phi, mfVar)
        - fv::laplacian(1.47*turbulence->mut(), mfVar)
        - 2*(1.47*turbulence->mut())*(Gradmf & Gradmf)
        + 2*rho*SDR
        + fvOptions(rho, mfVar)
    )
);

```

MixturefractionVar.C

# SLFMFoam

## Solver

### Flamelet library



```

for(label k=0; k<nny ; 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;
    }
}

makeSLFMylib.H
    
```

INPUT FILE FOR THE SR-CMC ROUTINES  
No.GRID POINTS No. SPECIES  
75 53  
PRESSURE(ATM)  
1.000000000000E+000  
MIXTURE FRACTION GRID  
0.000000000000E+000 1.250000000000E-002 2.500000000000E-002  
3.750000000000E-002 5.000000000000E-002 6.250000000000E-002  
7.500000000000E-002 8.750000000000E-002 0.100000000000  
0.11250000000000 0.12500000000000 0.13750000000000  
:  
H2  
0.000000000000E+000 1.199876917356024E-010 2.404599628458580E-010  
3.634374359506292E-010 4.923059051856218E-010 6.402815132399437E-010  
8.231616025136224E-010 1.106859158062047E-009 1.566269931137226E-009  
2.474411258063766E-009 4.221179938897097E-009 7.828444157308641E-009  
1.44660085194211E-008 2.504993582664863E-008 4.365857706813568E-008  
7.641434971437619E-008 1.437508917987143E-007 2.247941346143816E-007  
2.297697281978580E-007 7.3045549101008893E-007 1.227437577116229E-006  
:  
CO  
0.000000000000E+000 2.758643139581209E-008 5.546642769447130E-008  
8.403598134785700E-008 1.135376951029554E-007 1.449717185692140E-007  
1.789526711045E-007 2.184383358321492E-007 2.056790358058966E-007  
3.29829260745916E-007 4.2257805966158698E-007 5.744669341863719E-007  
8.1789884065641E-007 1.16528689987161E-006 1.738412965005708E-006  
2.702136085506771E-006 4.581708348802778E-006 6.82058793488328E-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 0.208017246870485 0.191771543120861  
0.183525817195215 0.175280038355829 0.167034166584222  
0.158788121161046 0.150541780465921 0.142294853615556  
0.134047082840520 0.127447653884125 0.128846788679216  
0.114243989786987 0.107637908827792 0.101030612932794  
9.441692407668151E-002 0.780100013979590E-002 8.118054334241118E-002  
7.455856389885659E-002 0.793579480660268E-002 6.132186250985606E-002  
5.472477529703249E-002 0.976905474115089E-002 4.484596939632648E-002  
3.99658079545329E-002 0.515712280589375E-002 3.043532021330975E-002  
2.585889376376704E-002 2.144854810691688E-002 1.731681682997562E-002

wylib.inp (OFstream wyFile)

Flamelet library

Combustion Laboratory POSTECH

# SLFMFoam

## Solver

```

private:
const scalar eta;
const scalar var_eta;
scalar gamma;
scalar alpha;
scalar beta;
scalar lndenum;
singularity s;

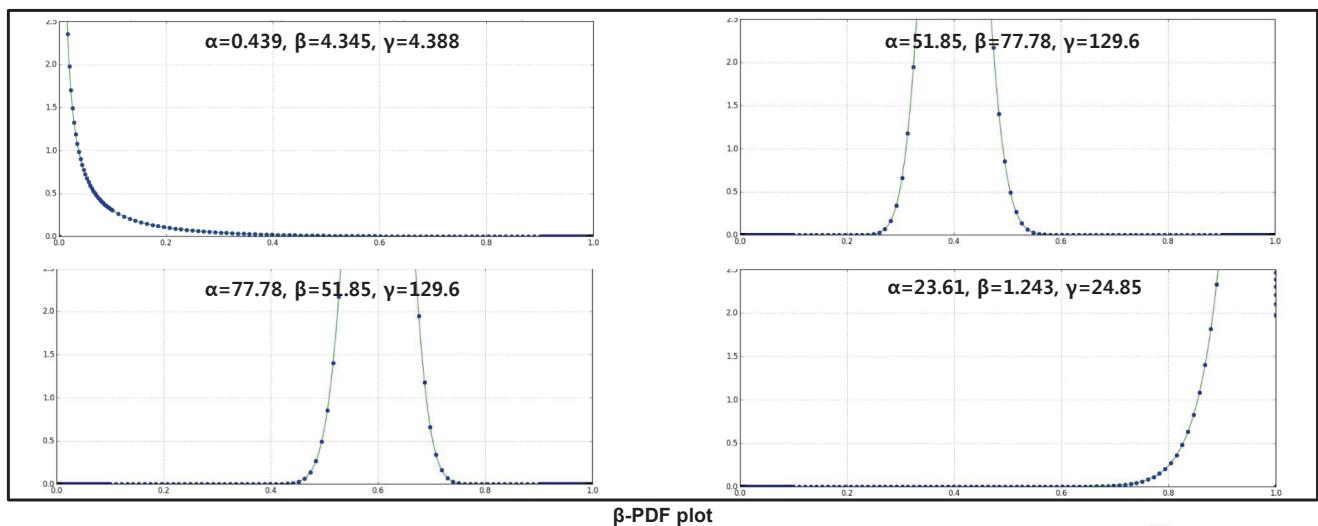
template <typename F, typename G>
inline scalar integrate_f_g (scalar low, scalar upper, label nInterval, F f, G g, bool debug=false)

template <typename G>
scalar beta_integration (scalar low, scalar upper, G g, Beta beta, bool debug=false)
    
```

$$\begin{aligned} \text{gamma} &= (\eta * (1.0 - \eta) / \text{var\_eta} - 1.0); \\ \alpha &= (\eta * \text{gamma}); \\ \beta &= ((1.0 - \eta) * \text{gamma}); \end{aligned}$$

BetaPDF.H

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

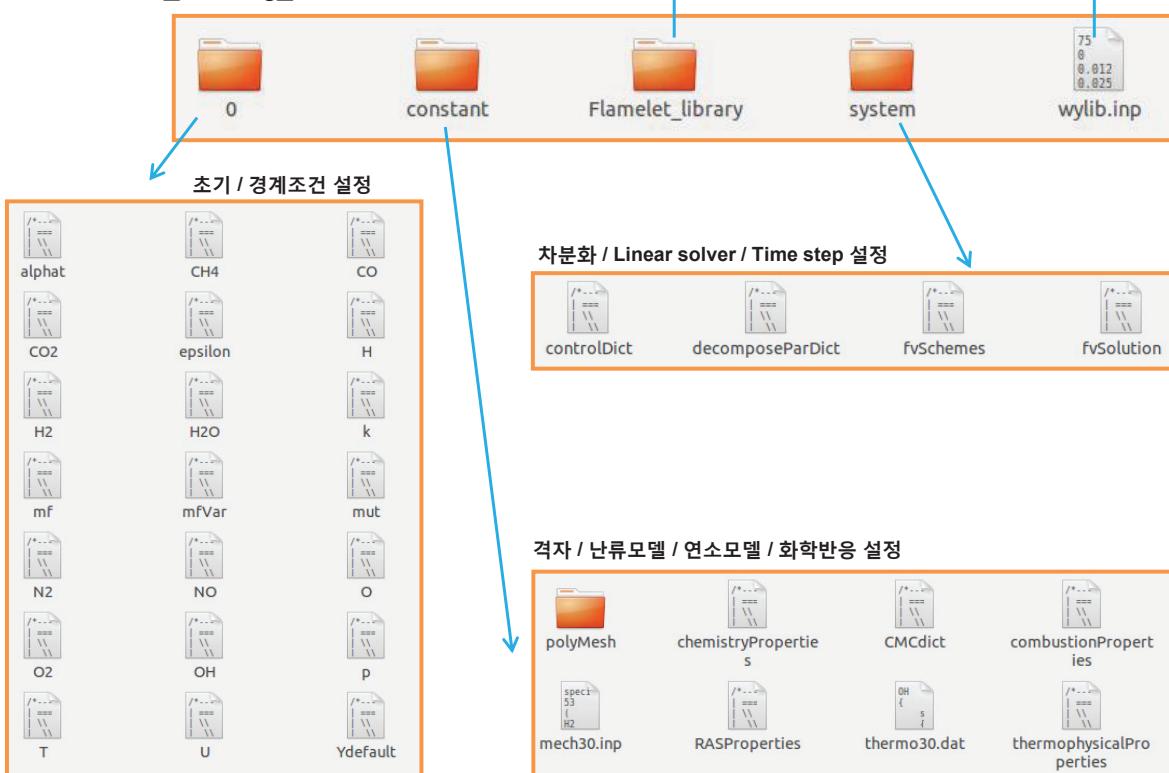


Combustion Laboratory POSTECH

# SLFMFoam

## Case Folder

/tutorials/SLFM\_reacting\_flow/



Combustion Laboratory POHANG UNIVERSITY OF SCIENCE AND TECHNOLOGY POSTECH

# 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];
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];
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;
    }
}
```

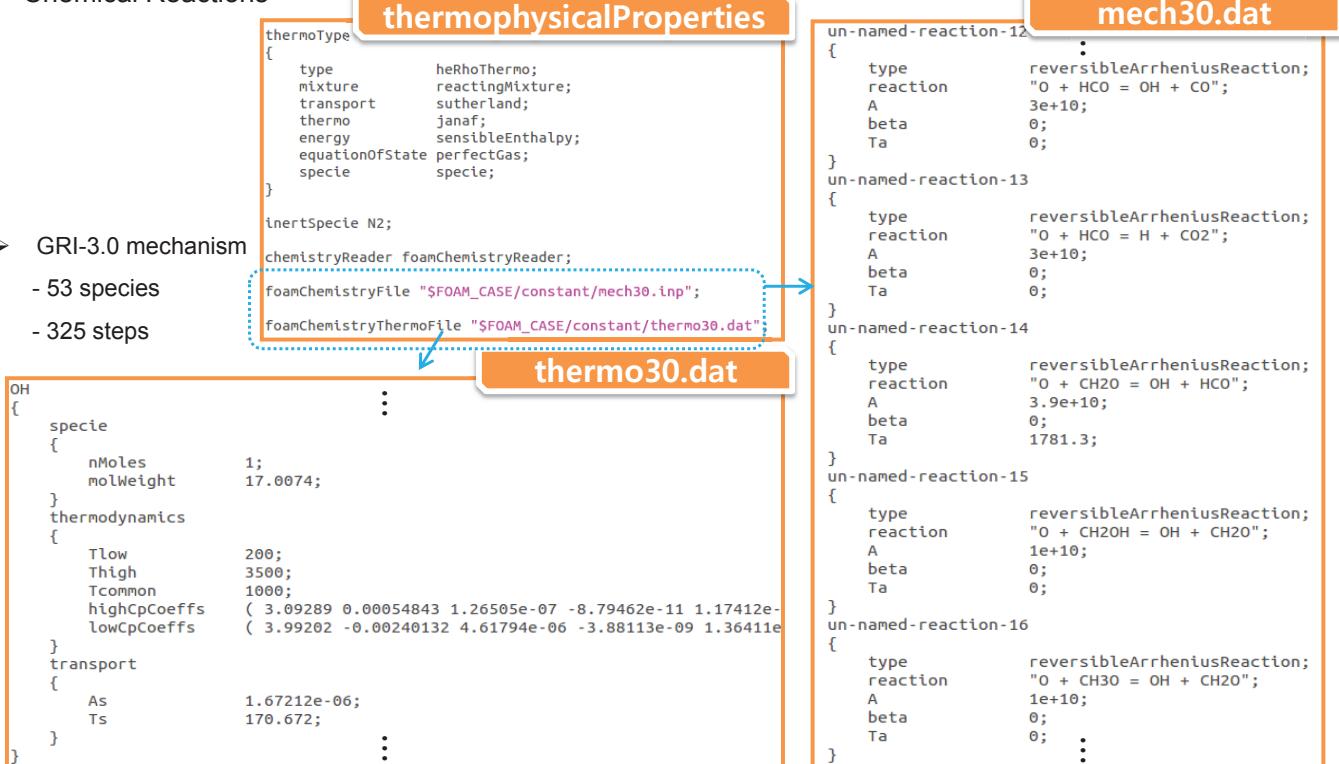
**mfVar**

Combustion Laboratory POHANG UNIVERSITY OF SCIENCE AND TECHNOLOGY POSTECH

# SLFMFoam

## 'constant' Folder

- Chemical Reactions



# SLFMFoam

## 'constant' Folder

- Mixture fraction space

```

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);
}

makeSLFMylib false; //true; ⑧

SLFM true; //false; ⑨

SLFM_dict
{
    SLFMylib_FileName wylib.inp;
    Maximum_Nst 300.1;
    Minimum_Nst 0.1;
    delta_Nst 3;
    eta_stoi 0.351;
}

```

CMCdict

- 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;
}
```

**fvSchemes**

```
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**

Combustion Laboratory  POSTECH

# 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  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

```
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;
}
```

```
Reading field mf
Reading field mfVar
Set eta space grid    false
```

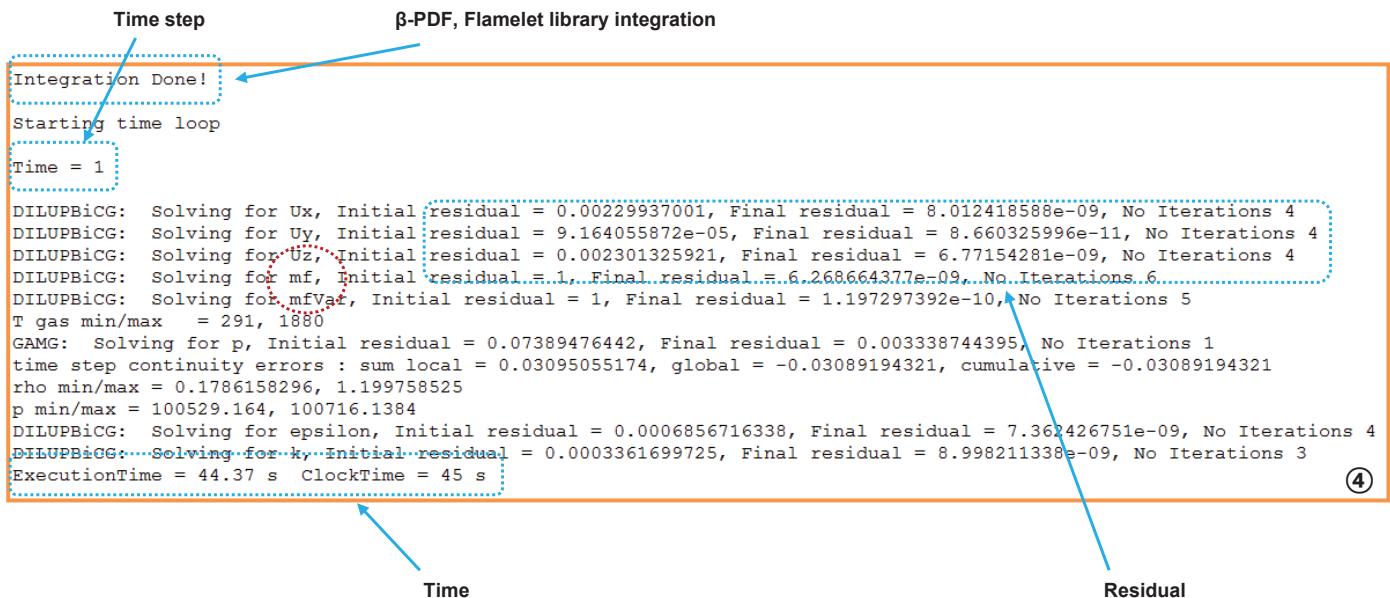
①

|              |                |   |             |             |
|--------------|----------------|---|-------------|-------------|
| 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 | 0.02110492262  |   | 1.868179825 | 5.835654505 |

# SLFMFoam

## Tutorial

- Calculating



Combustion Laboratory POHANG UNIVERSITY OF SCIENCE AND TECHNOLOGY POSTECH

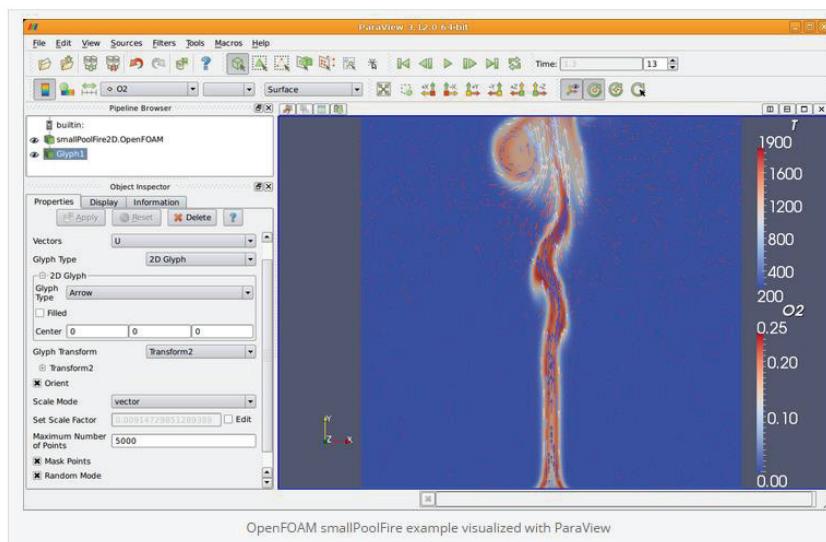
# SLFMFoam

## Post Processing

1. ~\$ reconstructPar      :~/tutorials/SLFM\_reacting\_flow\$ reconstructPar

2. ~\$ cd tutorials/EDM\_reacting\_flow/paraFoam

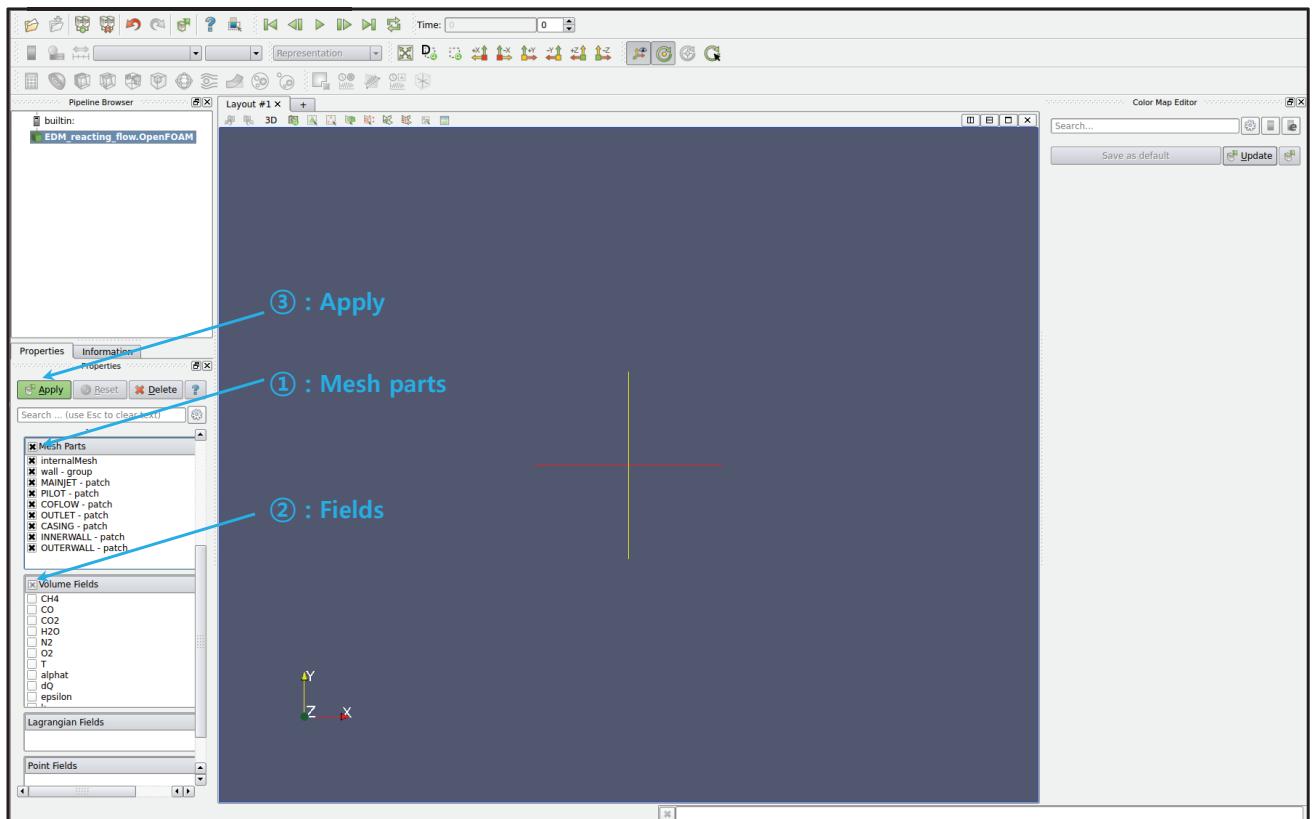
:~/tutorials/EDM\_reacting\_flow\$ paraFoam



Combustion Laboratory POHANG UNIVERSITY OF SCIENCE AND TECHNOLOGY POSTECH

# SLFMFoam

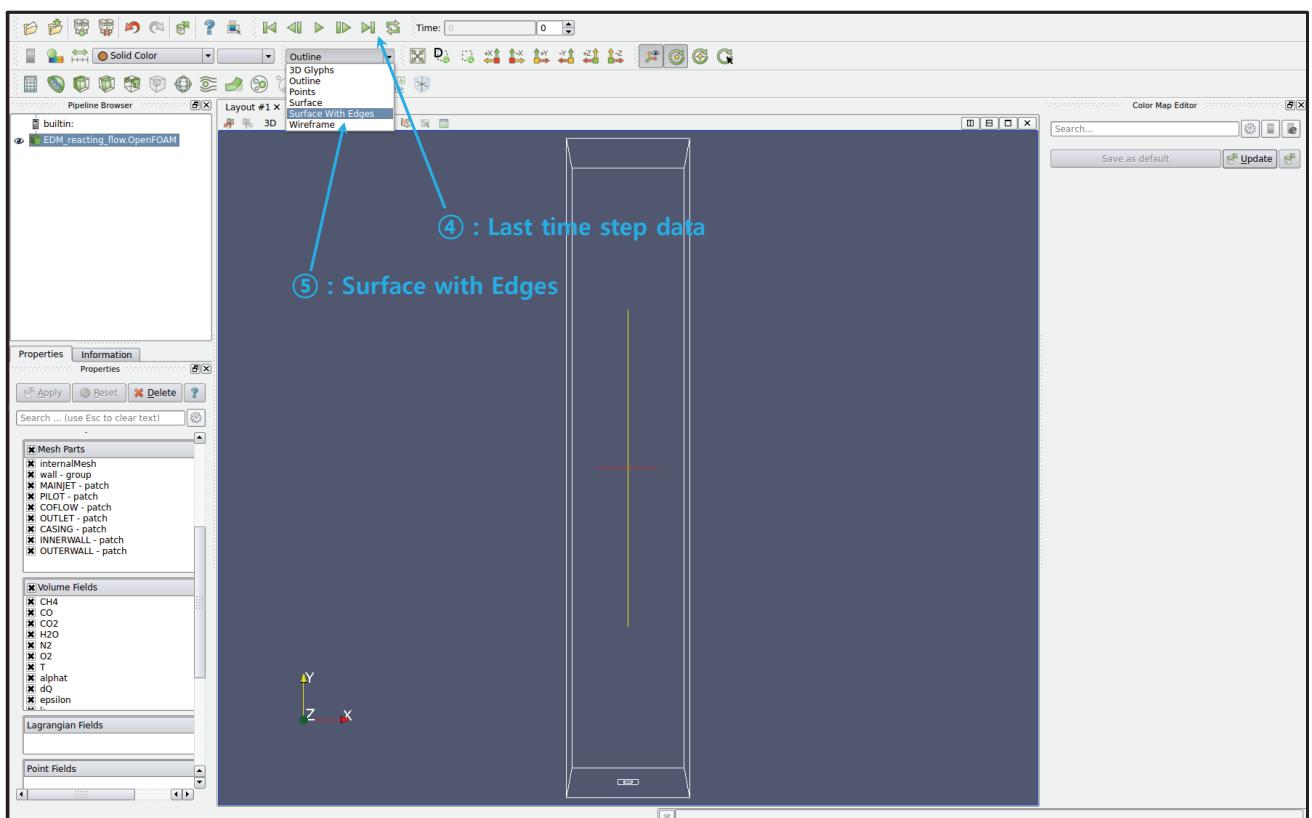
## Post Processing



Combustion Laboratory POHANG UNIVERSITY OF SCIENCE AND TECHNOLOGY POSTECH

# SLFMFoam

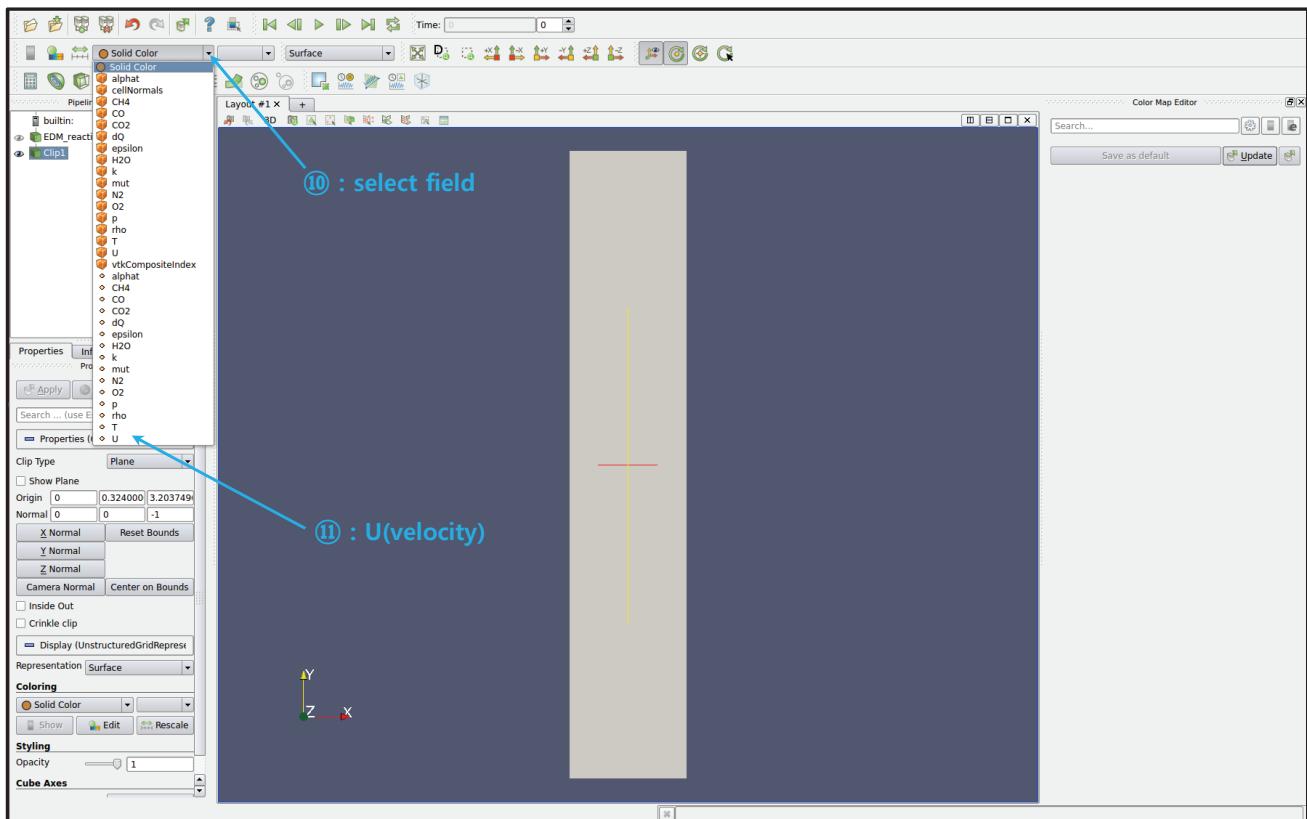
## Post Processing



Combustion Laboratory POHANG UNIVERSITY OF SCIENCE AND TECHNOLOGY POSTECH

# SLFMFoam

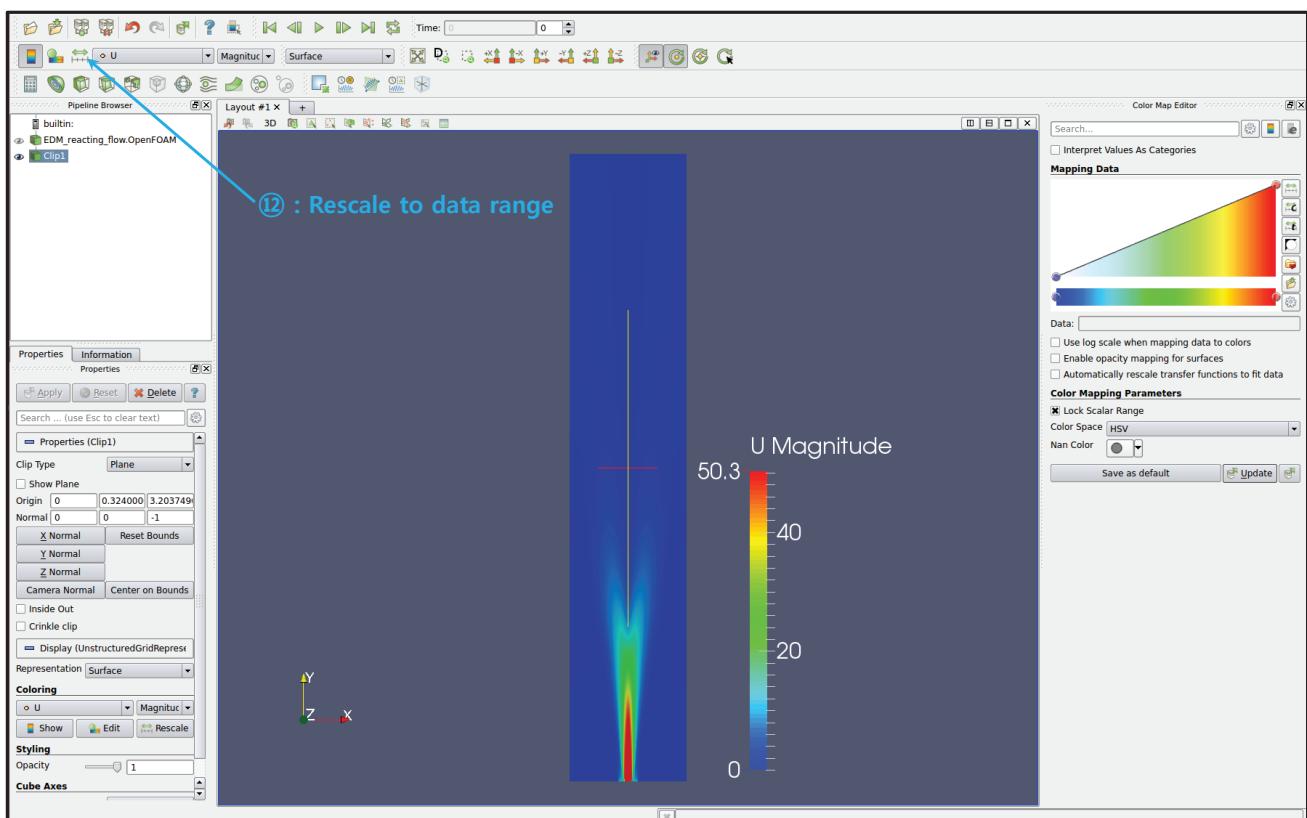
## Post Processing



Combustion Laboratory POHANG UNIVERSITY OF SCIENCE AND TECHNOLOGY POSTECH

# SLFMFoam

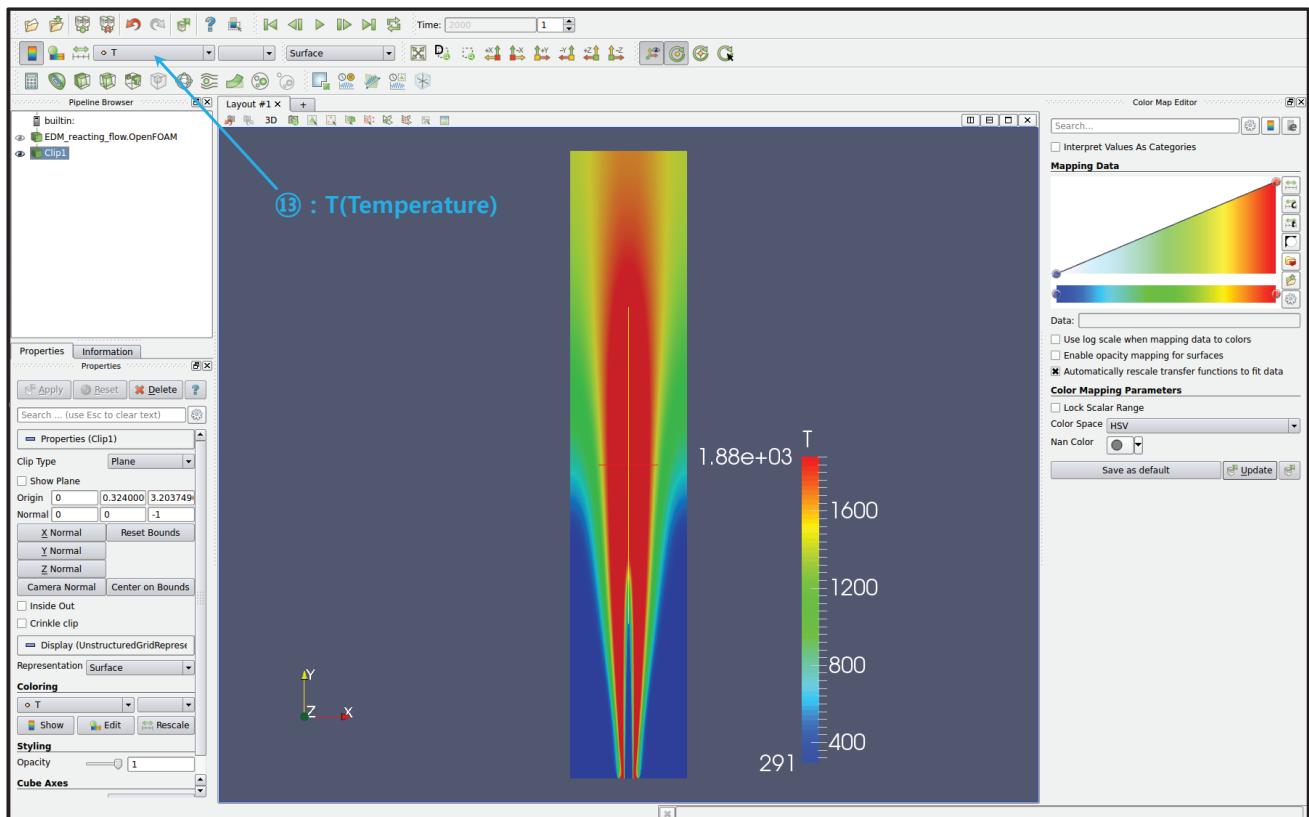
## Post Processing



Combustion Laboratory POHANG UNIVERSITY OF SCIENCE AND TECHNOLOGY POSTECH

# SLFMFoam

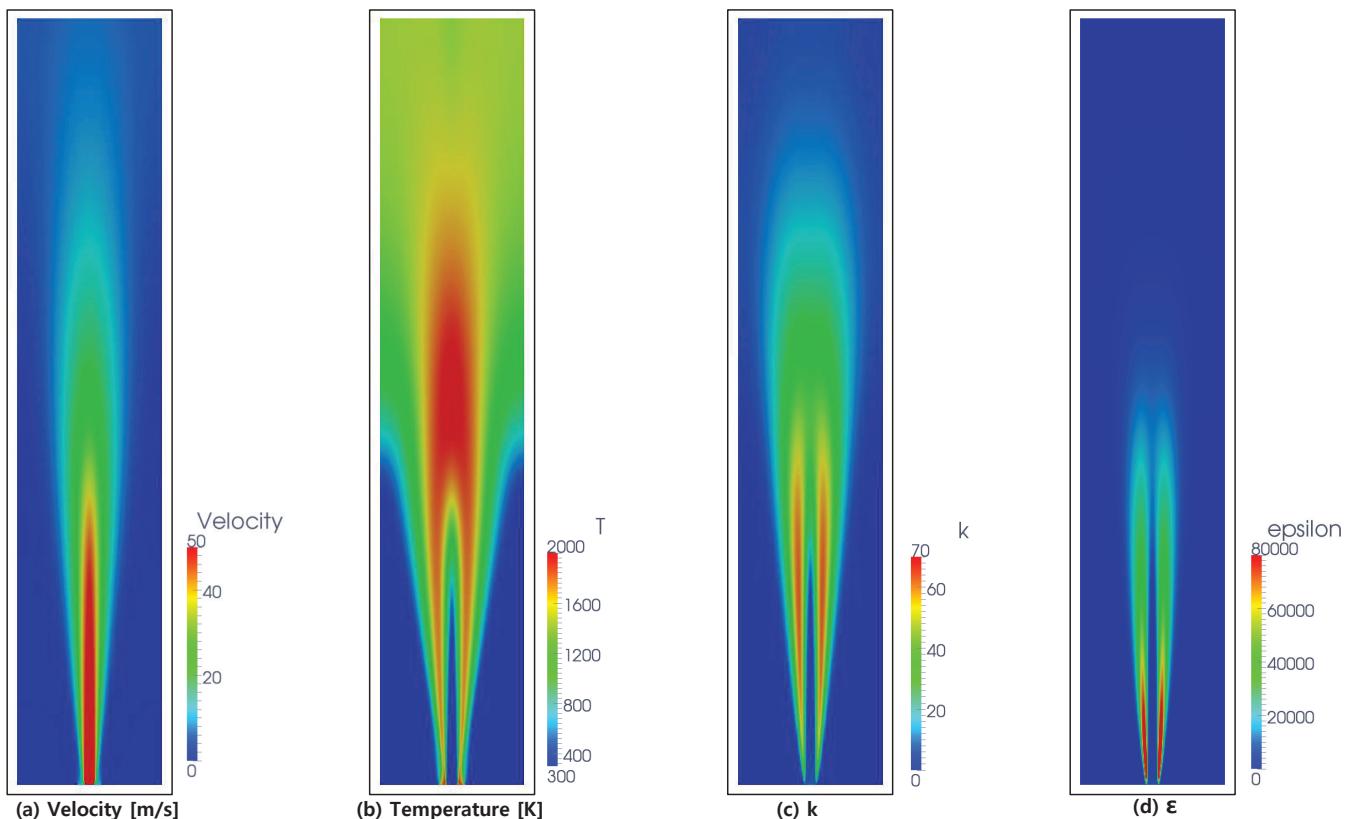
## Post Processing



Combustion Laboratory POHANG UNIVERSITY OF SCIENCE AND TECHNOLOGY POSTECH

# SLFMFoam

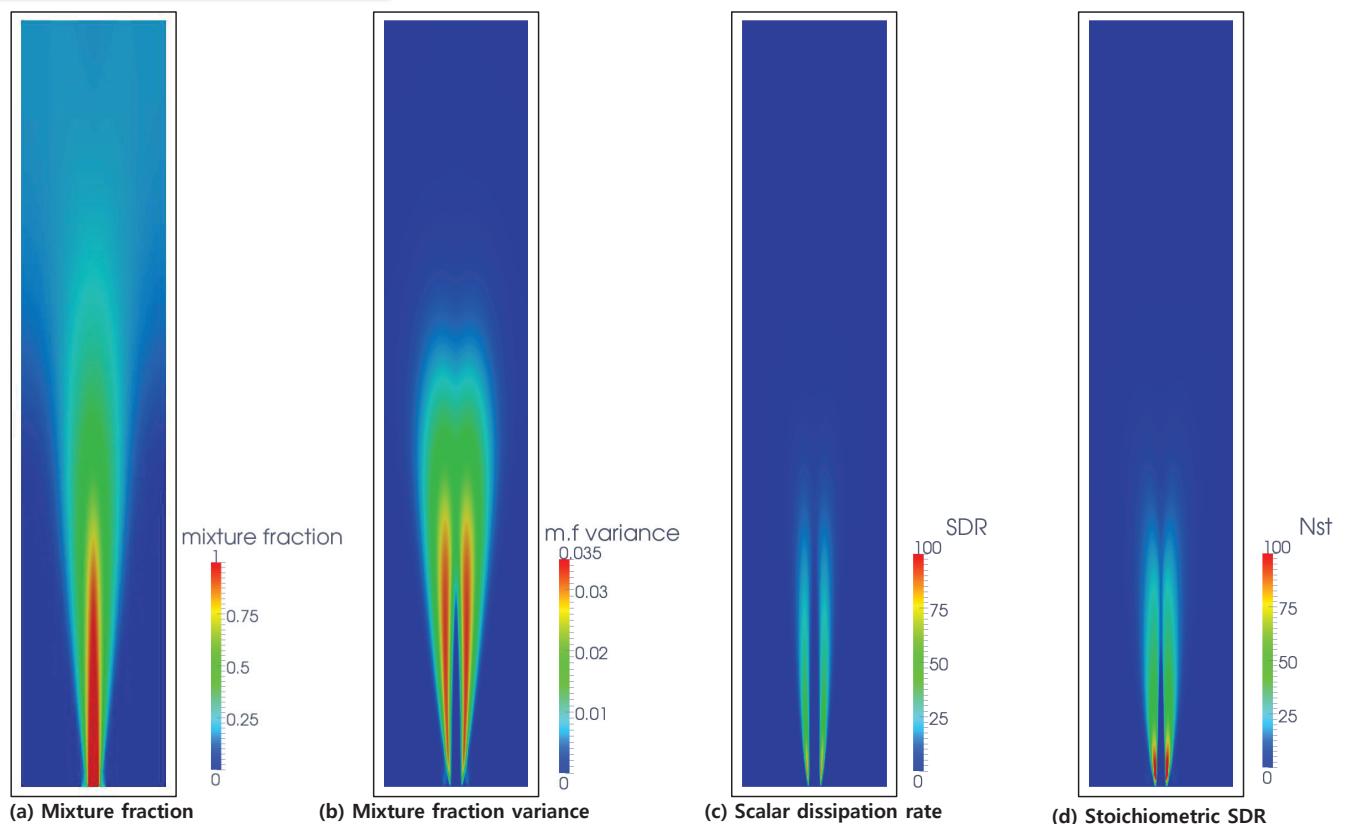
## Results



Combustion Laboratory POHANG UNIVERSITY OF SCIENCE AND TECHNOLOGY POSTECH

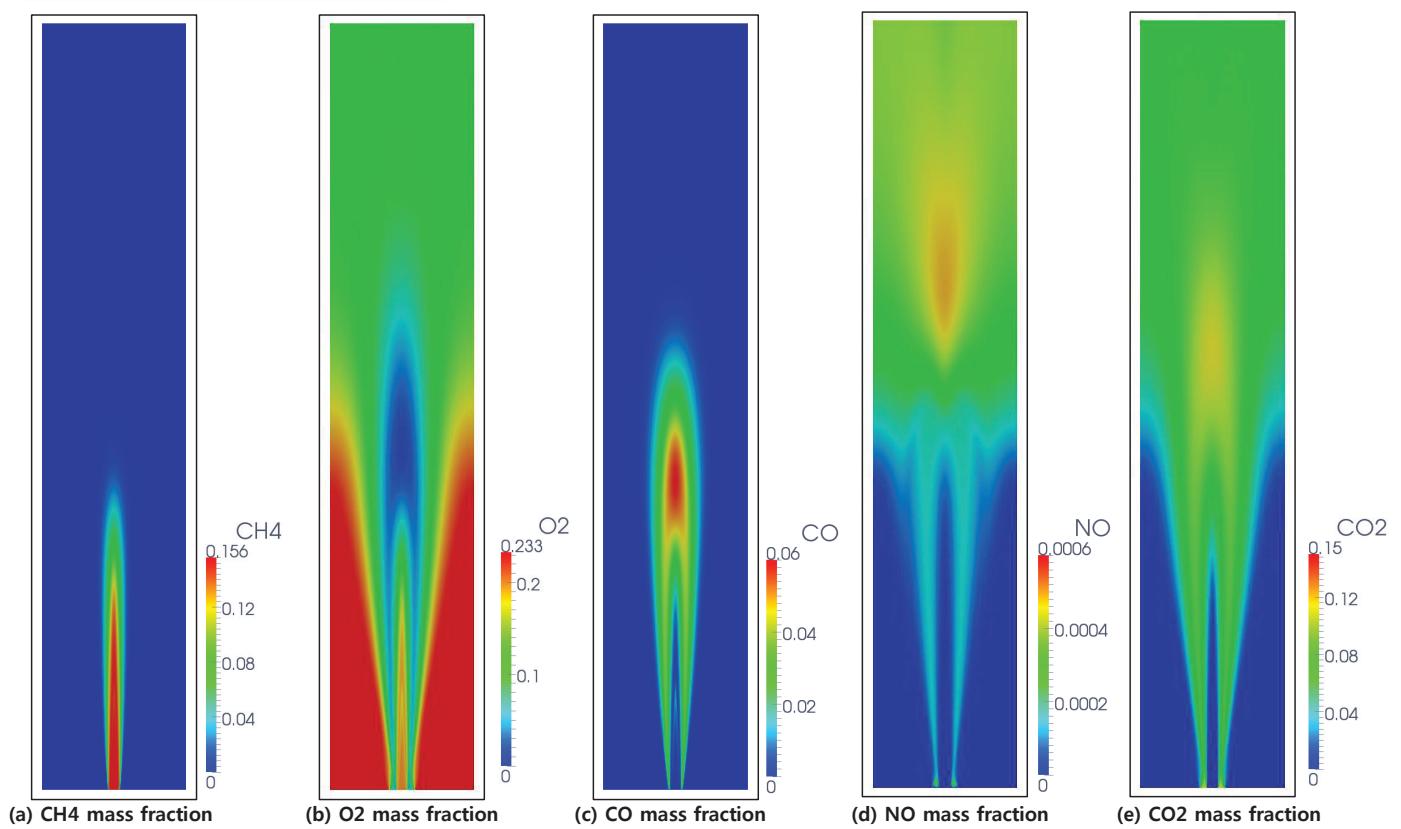
# SLFMFoam

## Results



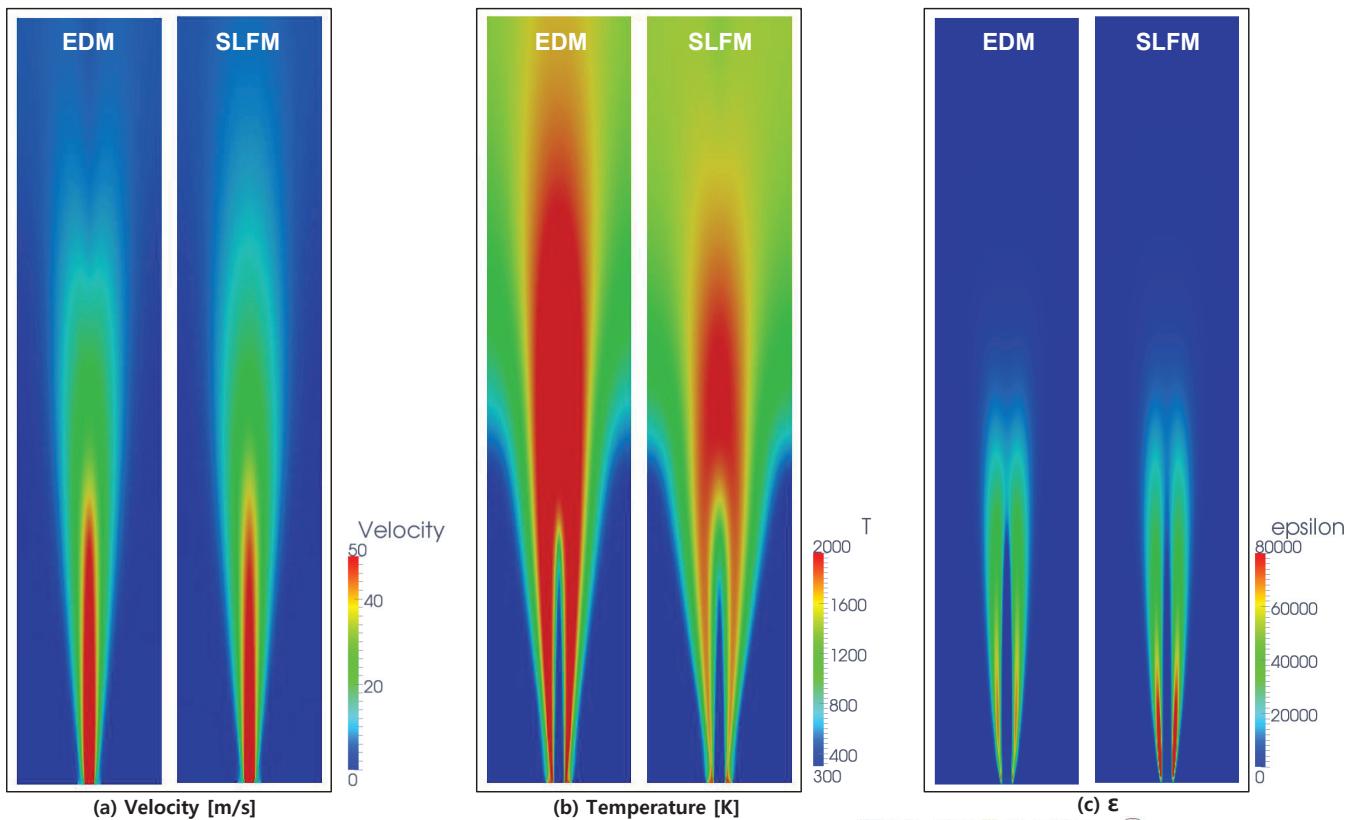
# SLFMFoam

## Results



# SLFMFoam

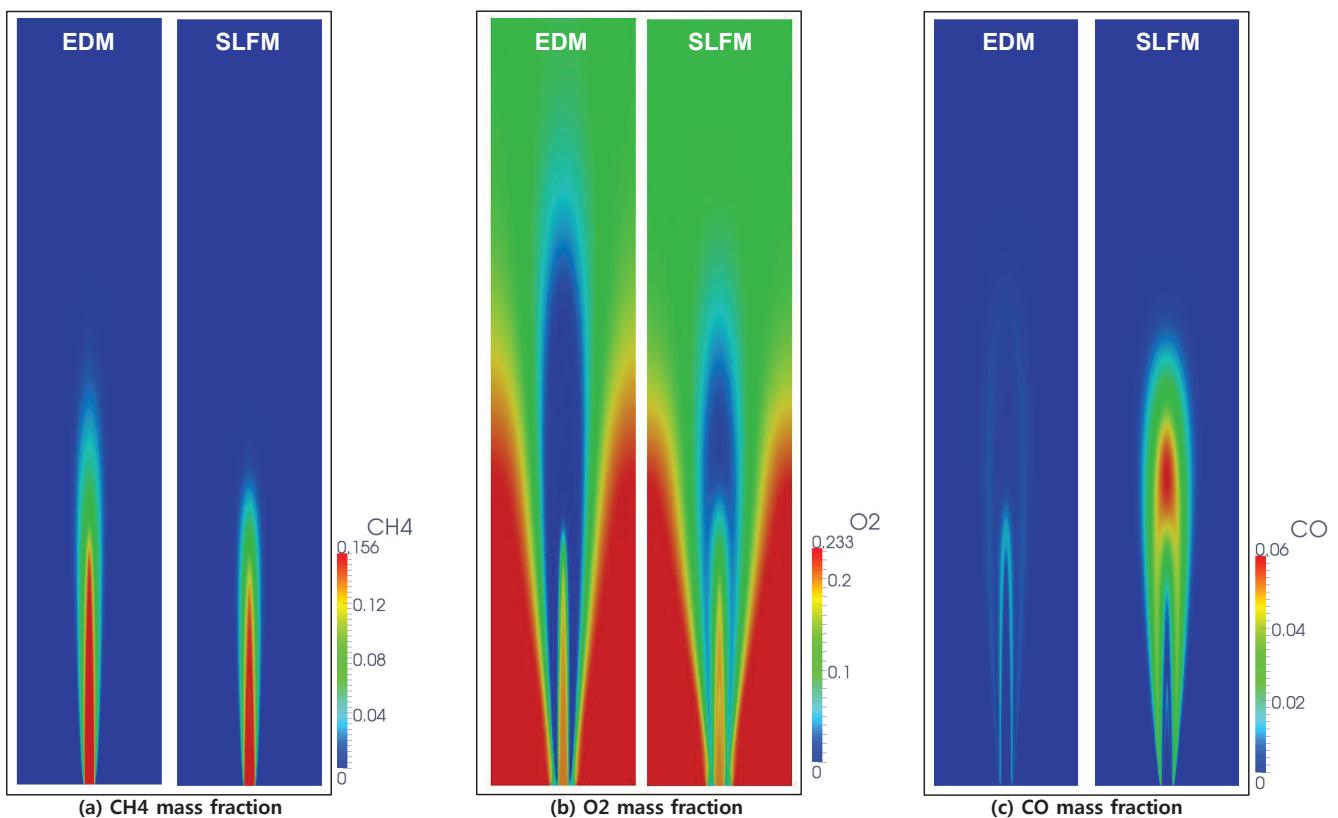
## Comparison



Combustion Laboratory POHANG UNIVERSITY OF SCIENCE AND TECHNOLOGY POSTECH

# SLFMFoam

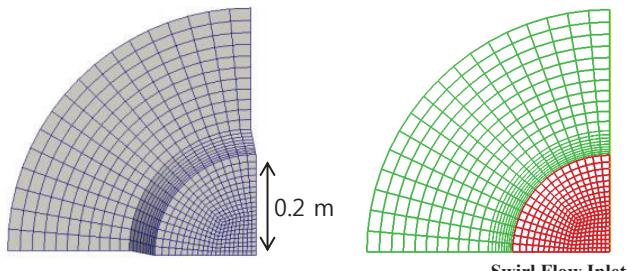
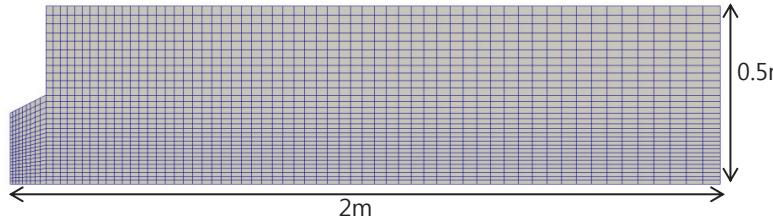
## Comparison



Combustion Laboratory POHANG UNIVERSITY OF SCIENCE AND TECHNOLOGY POSTECH

# Case description

## Computational grid



- ¼ quarter mesh
- Periodic boundary condition

### ● CheckMesh

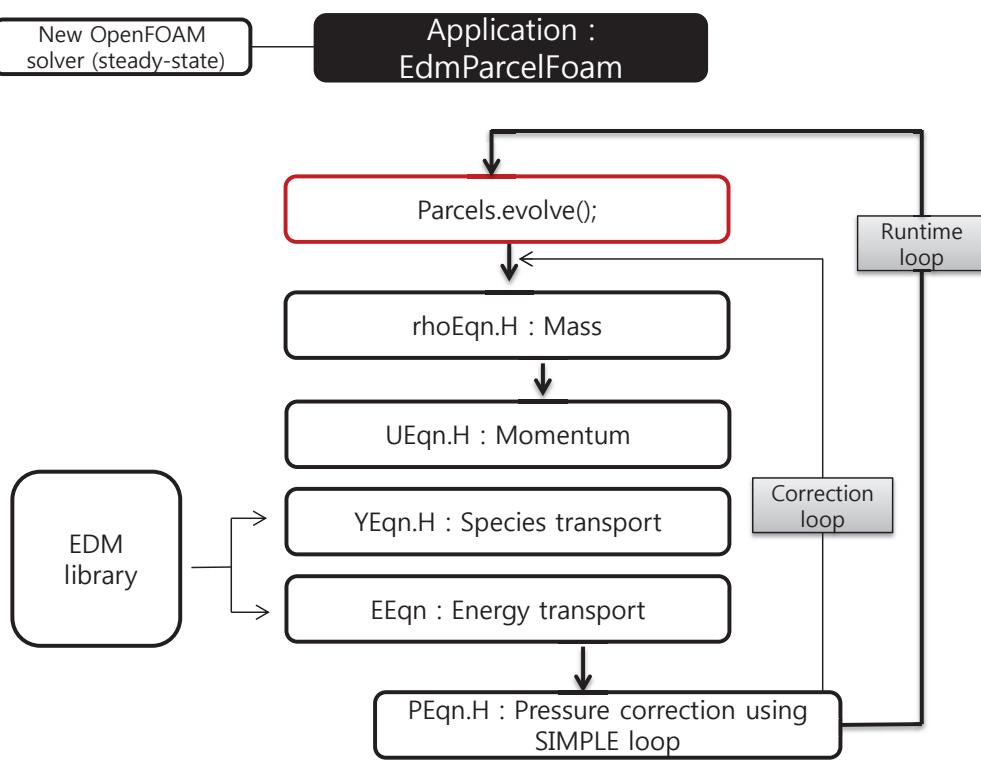
```
Mesh stats
points: 33861
faces: 95060
internal faces: 88864
cells: 30654
faces per cell: 6
boundary patches: 6
point zones: 0
face zones: 1
cell zones: 1

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 connected patches
Patch          Faces    Points
INLET          279      309
WALL           360      399
SLIPWALL       1062     1140
OUTLET          477      518
CYCLIC_half0   2009     2109
CYCLIC_half1   2009     2109
```

# EdmParcelFoam

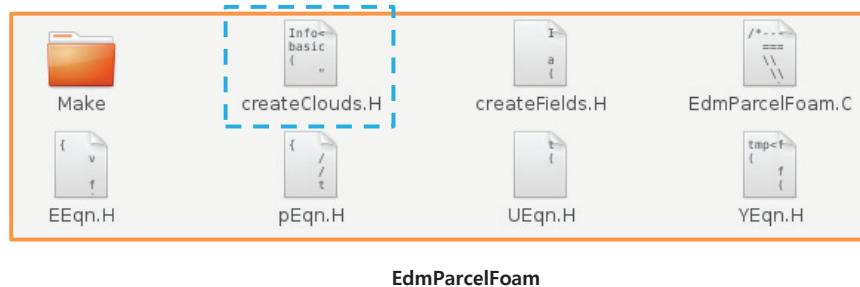
## Code Structure



# EdmParcelFoam

## Code Structure

/solvers/EdmParcelFoam



/libs/combustionModels\_POSTECH/EDM



# 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
basicReactingMultiphaseCloud.H
basicReactingMultiphaseParcel.H
Cloud.H
KinematicCloud.H
ThermoCloud.H
ReactingCloud.H
ReactingMultiphaseCloud.H
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$(FOAM_SOLVERS)/combustion/reactingFoam \
    -I$(DEV_PATH)/combustionModels_POSTECH/lnInclude \
    -I$(DEV_PATH)/chemistryModel_POSTECH/lnInclude

```

**options**

```

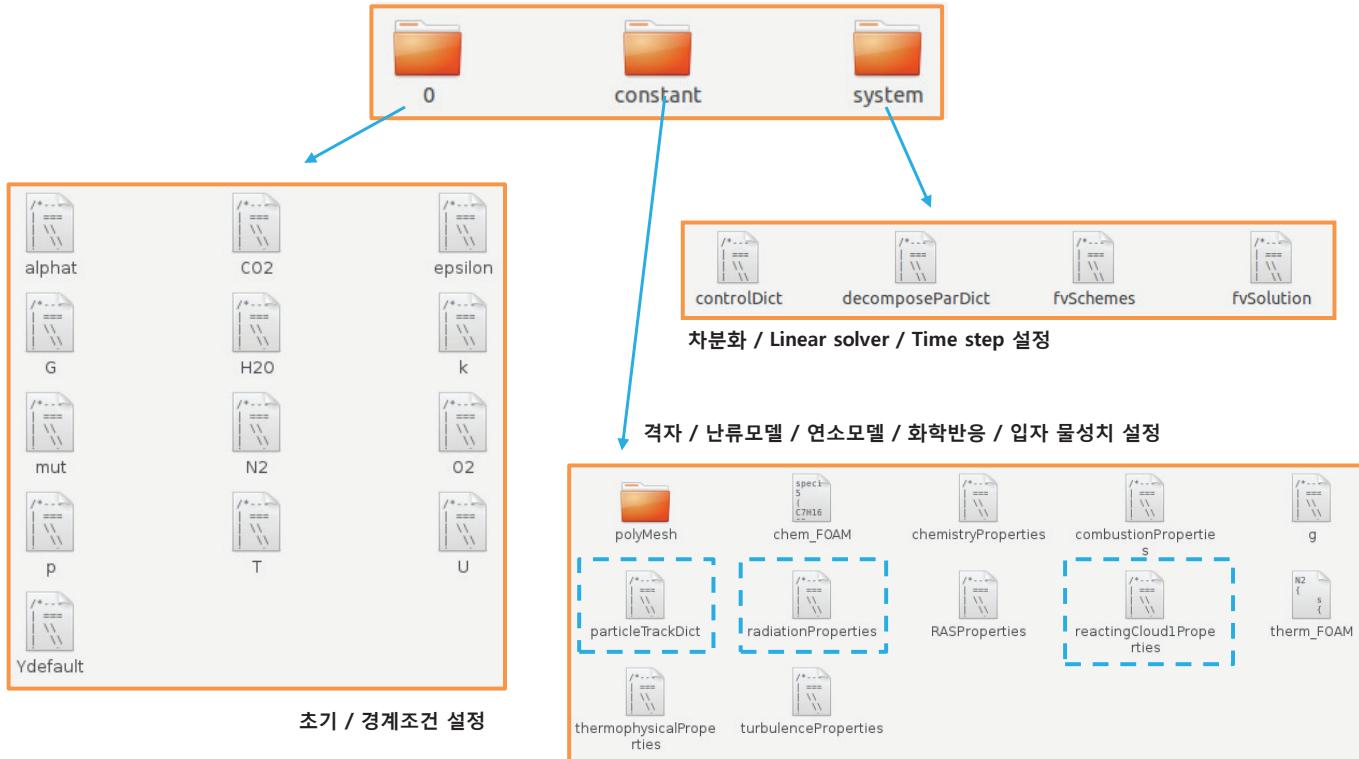
EXE_LIBS = \
    -L$(FOAM_USER_LIBBIN) \
    -lcombustionModels_POSTECH \
    -lchemistryModel_POSTECH \
    -lfiniteVolume \
    -lmeshTools \
    -lcompressibleTurbulenceModel \
    -lcompressibleRASModels \
    -lcompressibleLESModels \
    -llagrangian \
    -llagrangianIntermediate \
    -llagrangianTurbulence \
    -llagrangianSpray \
    -lspecie \
    -lfluidThermophysicalModels \
    -lliquidProperties \
    -lliquidMixtureProperties \
    -lsolidProperties \
    -lsolidMixtureProperties \
    -lthermophysicalFunctions \
    -lreactionThermophysicalModels \
    -lSLGThermo \
    -lradiationModels \
    -lODE \
    -lregionModels \
    -lsurfaceFilmModels \
    -lfvOptions \
    -lsampling

```

# EdmParcelFoam

## Case Folder

/tutorials/oilSpray-singleBurner/

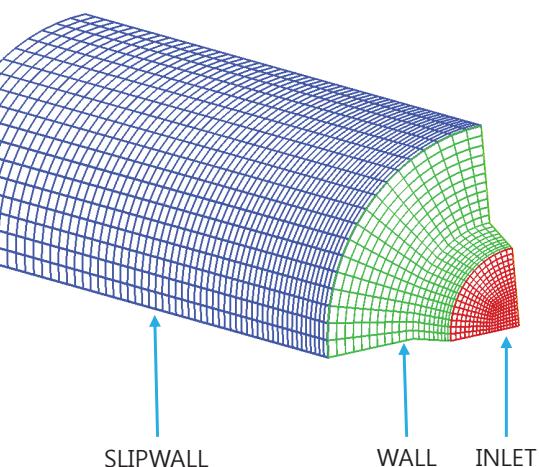
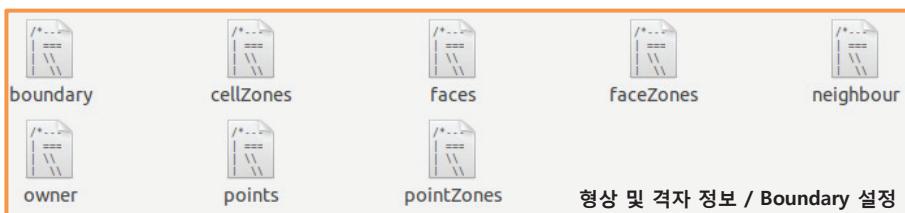


Combustion Laboratory POHANG UNIVERSITY OF SCIENCE AND TECHNOLOGY POSTECH

# EdmParcelFoam

## PolyMesh

/tutorials/oilSpray-singleBurner/constant/polyMesh



```

INLET
{
    type CYCLIC_half0
    nFaces 279;
    startFace 88864;
    patch;
    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);
}
WALL
{
    type CYCLIC_half1
    nFaces 360;
    startFace 89143;
    patch;
    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);
}
SLIPWALL
{
    type CYCLIC_half0
    nFaces 1062;
    startFace 89503;
    patch;
    type cyclic;
    inGroups 1(cyclic);
    nFaces 2009;
    startFace 93051;
    matchTolerance 0.0001;
    transform rotational;
    neighbourPatch CYCLIC_half1;
    rotationAxis (1 0 0);
    rotationCentre (0 0 0);
}
OUTLET
{
    type CYCLIC_half1
    nFaces 477;
    startFace 90565;
    patch;
    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

Combustion Laboratory POHANG UNIVERSITY OF SCIENCE AND TECHNOLOGY POSTECH

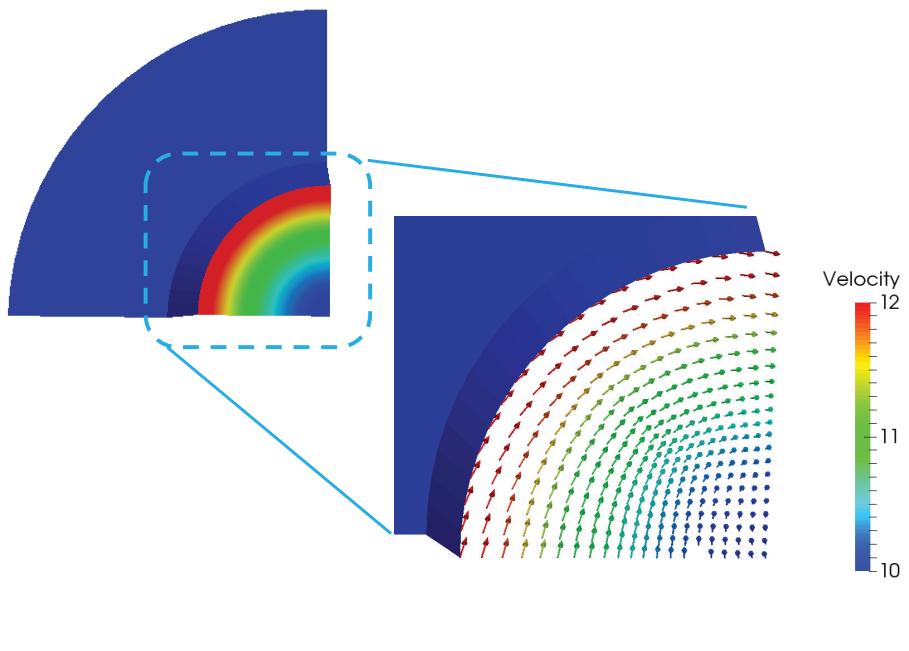
# EdmParcelFoam

## '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

```
particleForces
{
    sphereDrag;
}
```

```
dispersionModel none;
```

```
patchInteractionModel standardWallInteraction;
```

```
heatTransferModel RanzMarshall;
```

```
compositionModel singleMixtureFraction;
```

```
phaseChangeModel liquidEvaporation;
```

```
devolatilisationModel none;
```

```
surfaceReactionModel none;
```

```
stochasticCollisionModel none;
```

```
surfaceFilmModel none;
```

```
radiation off;
```

### Particle submodel

```
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   0.0004;
    d          0.0002;
    n          3.5;
}
```

# 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 + 11O2^1.5 = 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**

# 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 RosinRammel
Creating injector: model2
Selecting injection model coneNozzleInjection
    Constructing 3-D injection
Selecting distribution model RosinRammel
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

Combustion Laboratory POSTECH

# EdmParcelFoam

## Tutorial

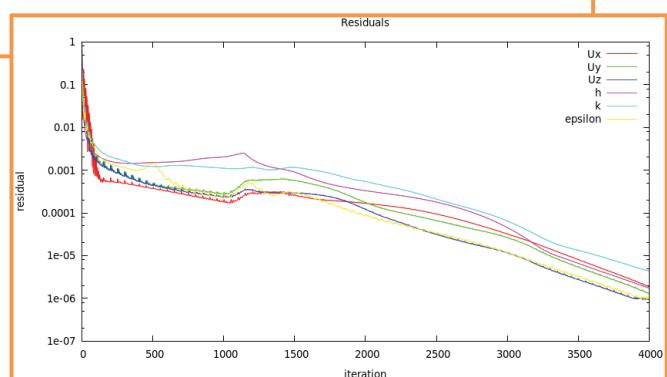
- Calculating

```
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 POSTECH

# EdmParcelFoam

## Post Processing

1. ~\$ reconstructPar

```
~/tutorials/oilSpray_singleBurner$ reconstructPar -latestTime
```

2. ~\$ cd tutorials/EDM\_reacting\_flow/paraFoam

```
:~/tutorials/oilSpray_singleBurner$ paraFoam
```

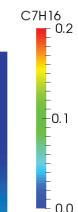
3. ~\$ steadyParticleTracks

```
:~/tutorials/oilSpray_singleBurner$ steadyParticleTracks
```

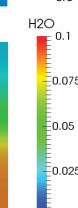
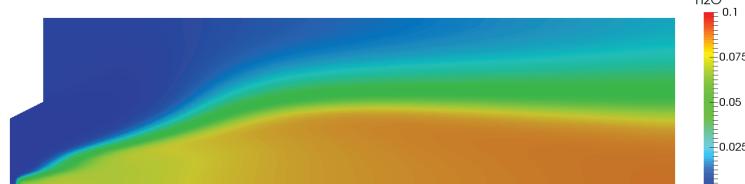
# EdmParcelFoam

## Results

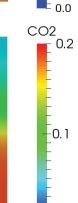
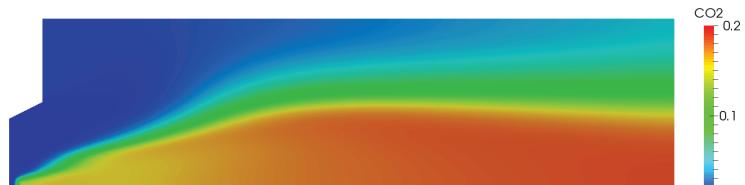
(a)  $C_7H_{16}$  mass fraction



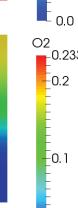
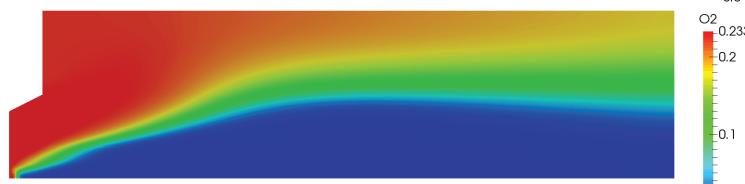
(b)  $H_2O$  mass fraction



(c)  $CO_2$  mass fraction



(d)  $O_2$  mass fraction



# EdmParcelFoam

## Results

(e) Turbulent dissipation rate ( $\text{m}^2/\text{s}^3$ )



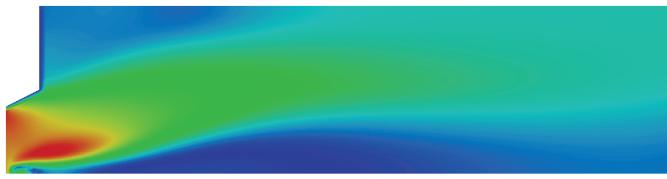
epsilon  
2000.  
1000  
0.0000

(f) Turbulent kinetic energy ( $\text{m}^2/\text{s}^2$ )



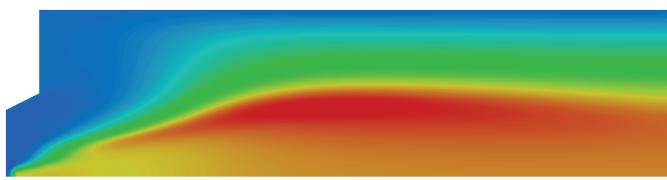
k  
10.0  
7.5  
5  
2.5  
0.0000

(g) Velocity (m/s)



Velocity  
12.  
10  
7.5  
5  
2.5  
0.00

(h) Temperature (K)



T  
2500.0  
2000  
1000  
300.00

Combustion Laboratory POHANG UNIVERSITY OF SCIENCE AND TECHNOLOGY POSTECH

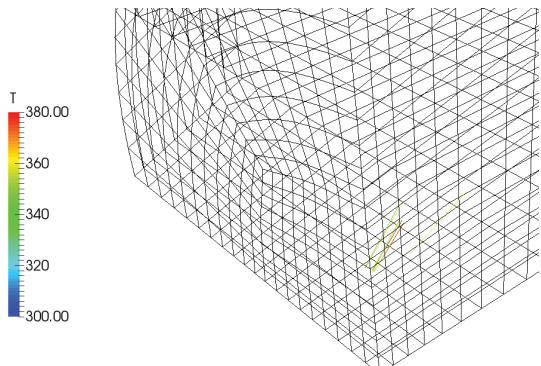
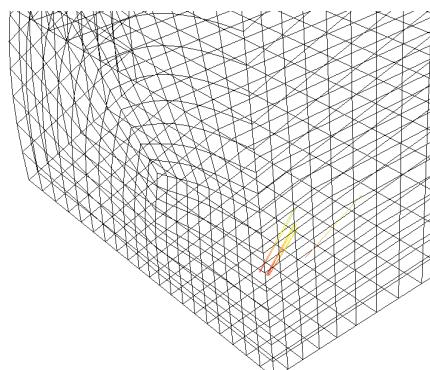
# EdmParcelFoam

## Results

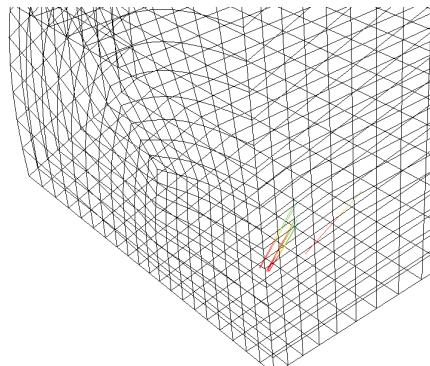
```
cloudName      reactingCloud1Tracks;  
  
fields  
(  
    d U T  
)
```

particleTrackDict

Velocity  
21.  
16  
12  
8  
4  
0.00



d  
2.0e-05  
1e-05  
1.0e-06



Combustion Laboratory POHANG UNIVERSITY OF SCIENCE AND TECHNOLOGY POSTECH