

POLITECNICO
MILANO 1863

OpenFOAM Training: Combustion
3-5 July 2017, Brussels

Use of external libraries for chemistry: Getting started with OpenSMOKE++

Alberto Cuoci

Download the training session material

The source code adopted in this Training Session (including this presentation) can be downloaded from the GitHub repository available at the following address:

<https://github.com/acuoci/OpenFOAMTrainingCombustion>

You can clone the repository (suggested):

```
git clone https://github.com/acuoci/OpenFOAMTrainingCombustion.git
```

or download the corresponding zip file:

```
https://github.com/acuoci/OpenFOAMTrainingCombustion/archive/master.zip
```

Outline

1. Presentation of OpenSMOKE++

1. Introduction
2. The OpenSMOKE++ Framework
3. The OpenSMOKE++ Suite
4. Coupling with OpenFOAM

2. Training

1. Introduction (environment preparation, organization, ...)
2. Preprocessing of thermodynamics, transport and kinetics
3. OpenSMOKE++ Maps for thermodynamics, transport and kinetics
4. OpenSMOKE++ for modeling a batch reactor
5. A multidimensional laminar solver for reacting flows

Outline

1. Presentation of OpenSMOKE++

1. Introduction

2. The OpenSMOKE++ Framework
3. The OpenSMOKE++ Suite
4. Coupling with OpenFOAM

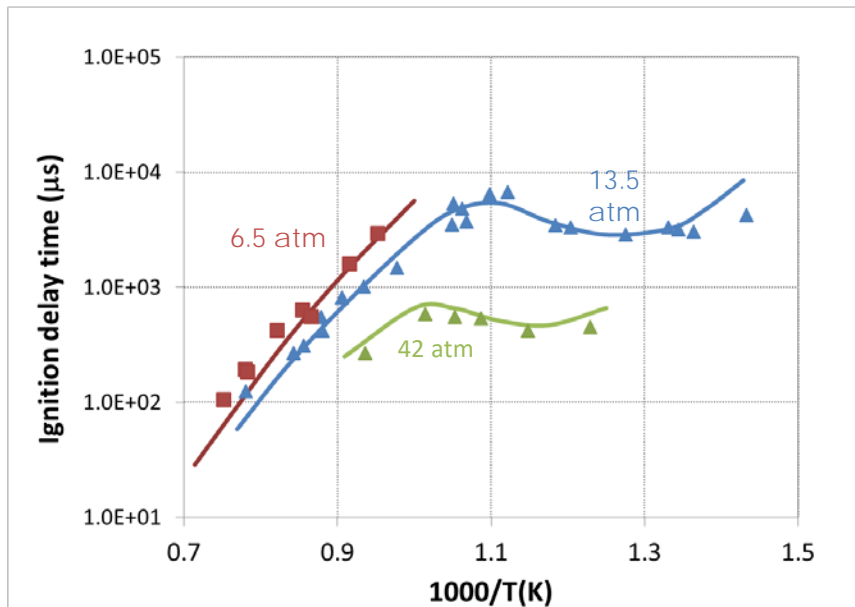
2. Training

1. Introduction (environment preparation, organization, ...)
2. Preprocessing of thermodynamics, transport and kinetics
3. OpenSMOKE++ Maps for thermodynamics, transport and kinetics
4. OpenSMOKE++ for modeling a batch reactor
5. A multidimensional laminar solver for reacting flows

Detailed combustion chemistry (I)

Detailed combustion chemistry is important for: ignition, extinction, instabilities ...

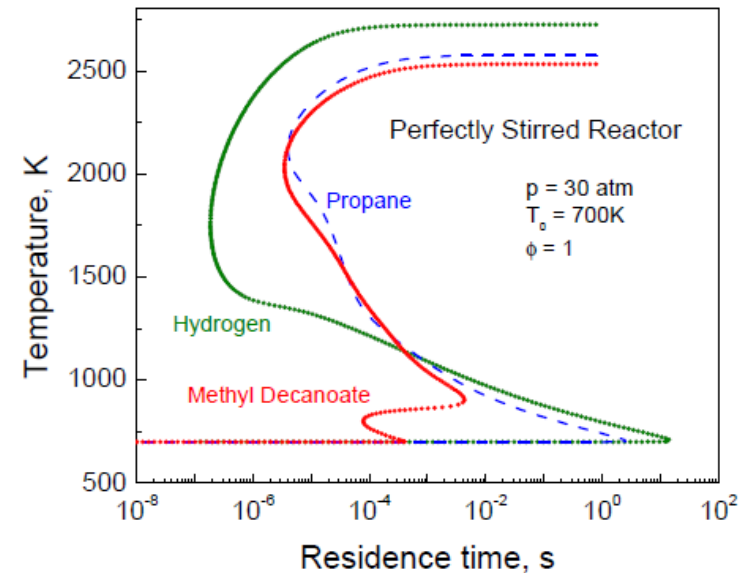
Negative temperature Coefficients (NTC)



Experimental data from:

Ciezeki H.K. and Adomeit G., *Shock-tube investigation of self-ignition of n-heptane-air mixtures under engine relevant conditions*, Combustion and Flame 93 p. 421–433 (1993)

Combustion “S”-curves



Plot from:

Lu T., *Computational Tools for Diagnostics and Reduction of Detailed Chemical Kinetics*, Princeton-CEFRS Summer School on Combustion (2012)

Detailed combustion chemistry (II)



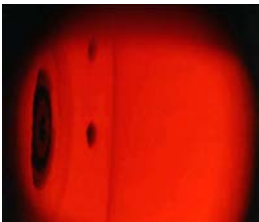
Real fuels and surrogate mixtures

need of modeling synergistic effects between the different components



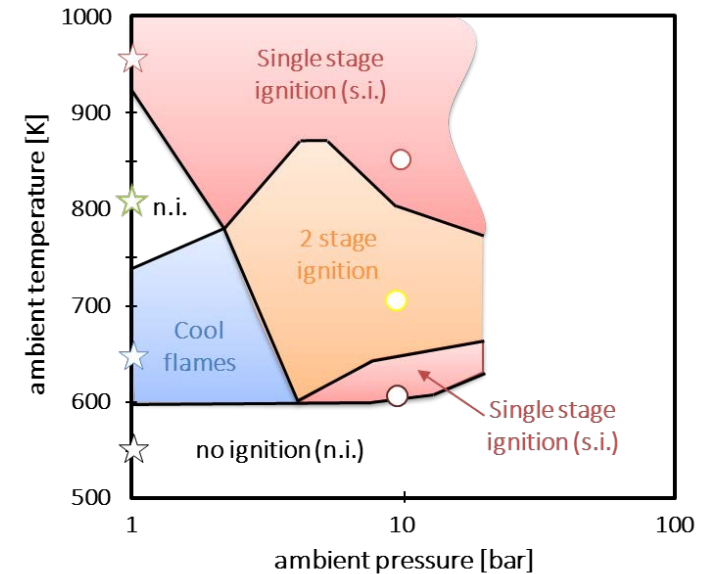
Biofuels

Bio-alcohols, biodiesel, green diesel, bio-ethers



Flameless combustion

(low Damkholer number, slow chemistry)



Auto-ignition regimes of n-alkanes droplets

Tanabe et al., 26th Symposium
(International) on Combustion, p.
1637-1643 (1996)

Cuoci A. et al., Proceedings of The
Combustion Institute, 2015

CFD and Combustion with detailed chemistry

1. Number of equations

Since detailed kinetic mechanisms involve hundreds or thousands of species, the number of coupled equations can be very large, especially when multidimensional geometries are simulated

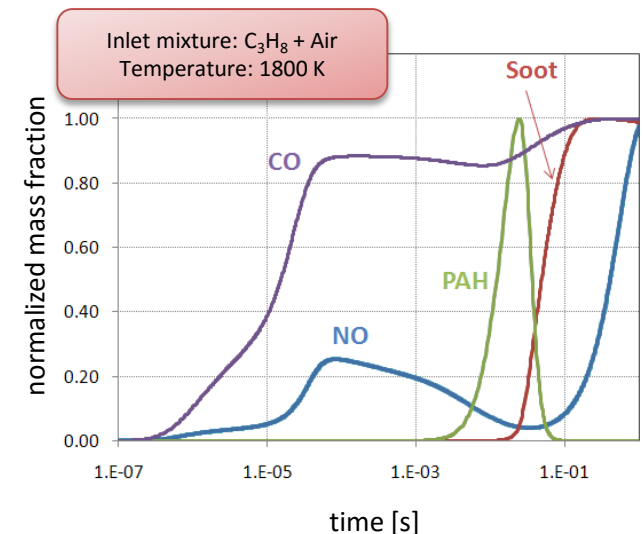
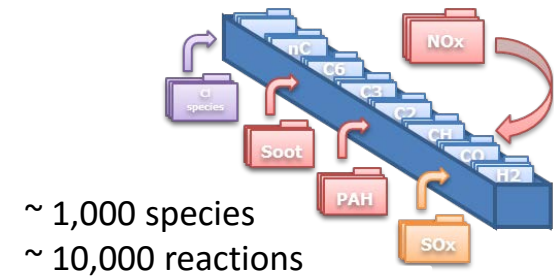
2. Non-linearity

The transport equations of species and energy are very non-linear, because of reaction rates expressions (power-law and exponential)

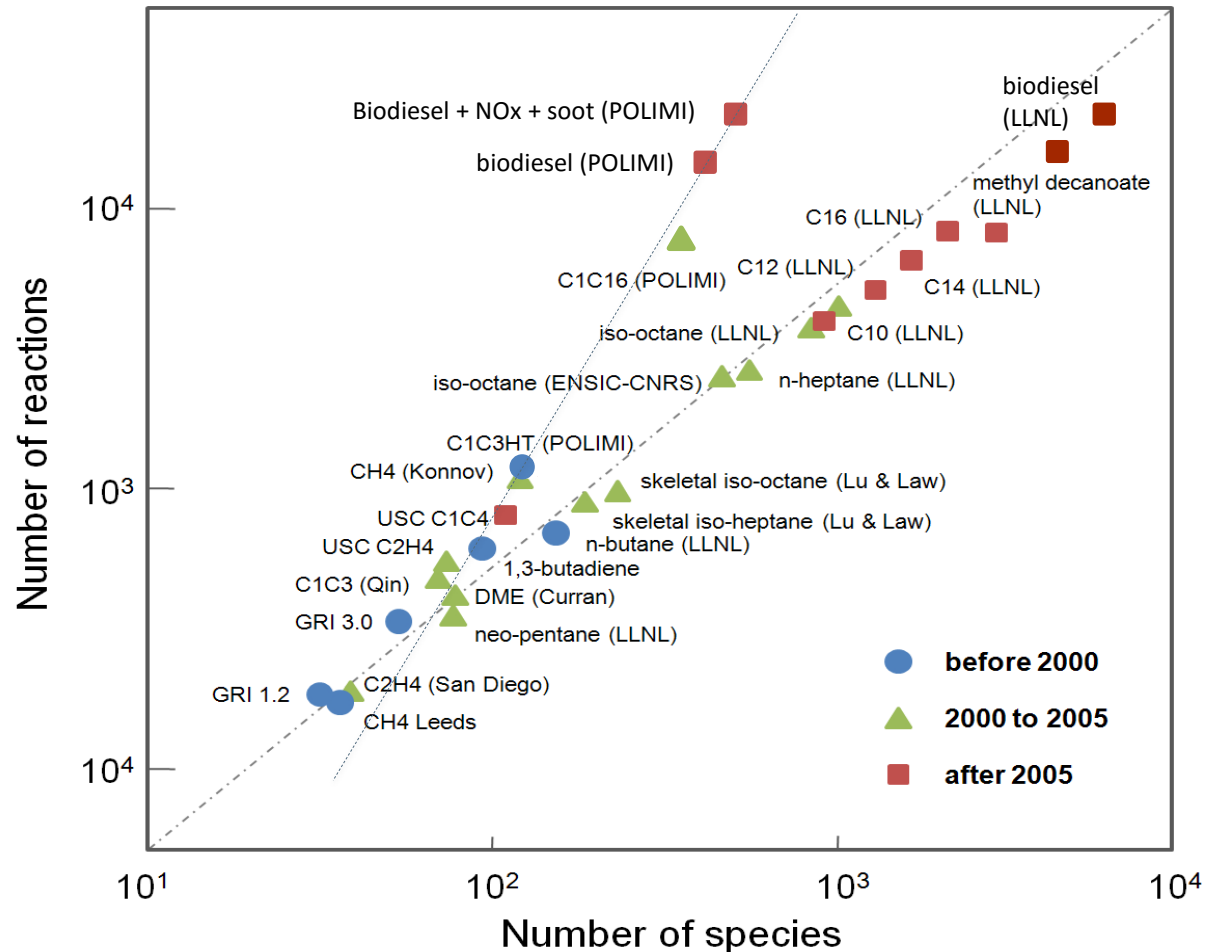
3. Stiffness

The characteristic times of species involved in a kinetic scheme cover several orders of magnitudes.

Detailed kinetic schemes

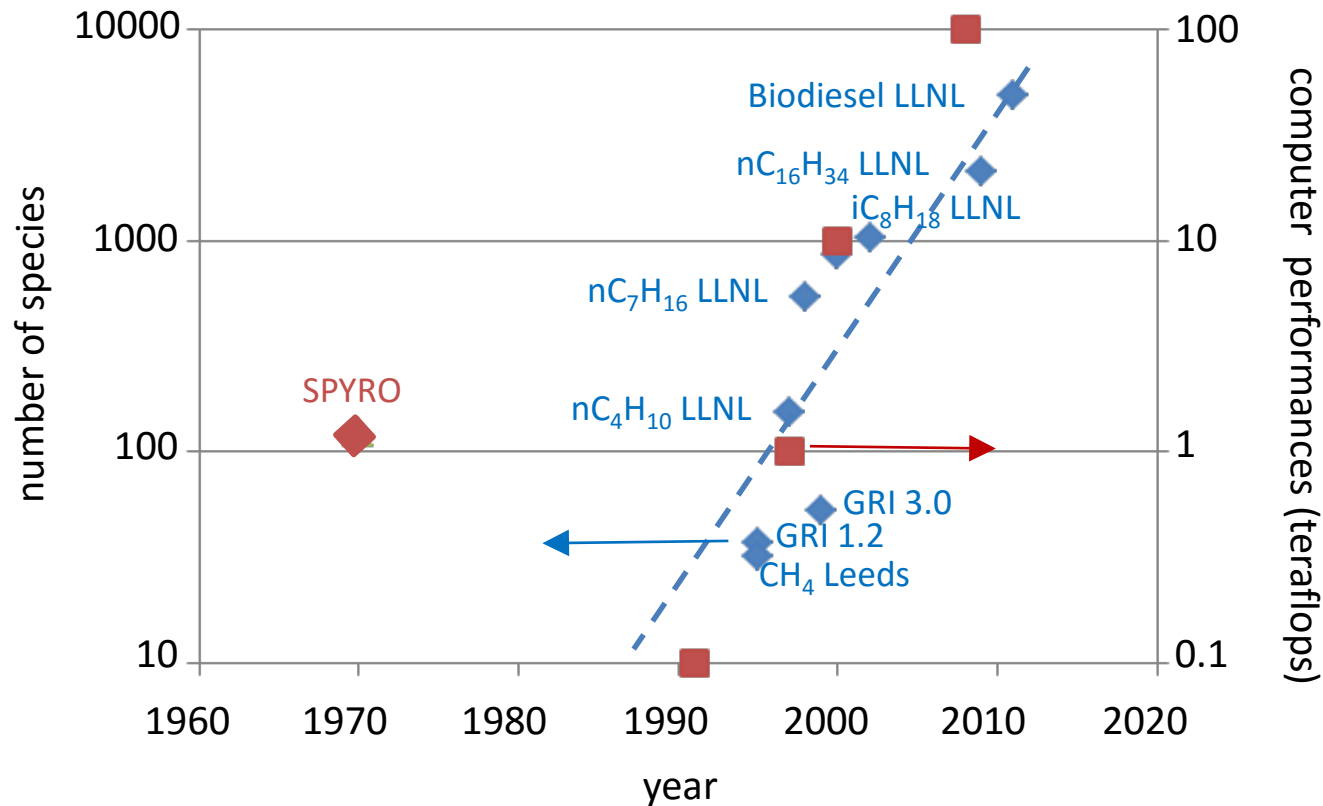


Combustion chemistry is very complex (I)



Adapted from: **T.F. Lu, C.K. Law**, *Toward accommodating realistic fuel chemistry in large-scale computations*, Progress in Energy and Combustion Science, 35, p. 192–215 (2009)

Combustion chemistry is very complex (II)

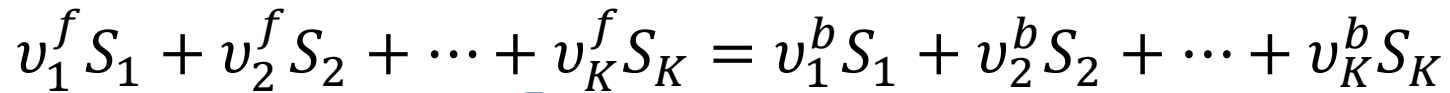


Adapted from:

T. Faravelli, *Numerical Modeling of Pollutant Emissions with Detailed Kinetics: from Ideal Reactors to Flames*, Invited Lecture at 14th ICNC 2013, San Antonio (TX)

Combustion chemistry is strongly non linear

A reaction in general form:



v_j^f forward stoichiometric
coefficients

v_j^b backward stoichiometric
coefficients

The reaction rates are strongly non-linear!

$$\dot{\Omega}_f = k_f(T) \prod_j C_j^{v_j^f}$$

$$\dot{\Omega}_r = k_r(T) \prod_j C_j^{v_j^b}$$

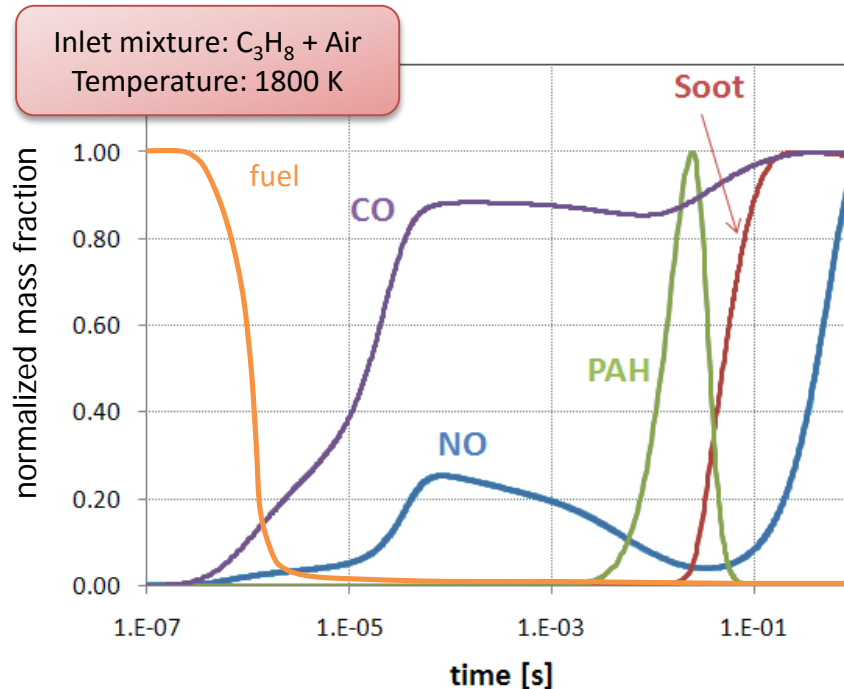
$$k_f(T) = AT^n \exp\left(-\frac{E}{RT}\right)$$

$$k_r(T) = \frac{k_f(T)}{K_{eq}(T)}$$

Adapted from:

Lu T., *Computational Tools for Diagnostics and Reduction of Detailed Chemical Kinetics*, Princeton-CEFRC Summer School on Combustion (2012)

Combustion chemistry is stiff



1. Slow reactions:

NO_x and soot formation

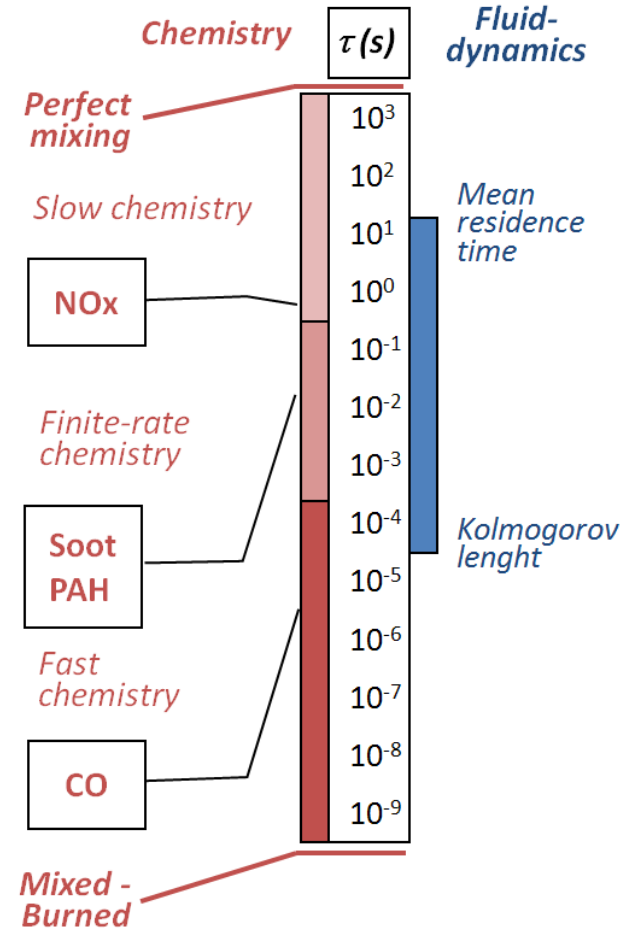
$CO \rightarrow CO_2$ (often rate limiting)

2. Fast reactions:

Reactions involving highly reactive radicals (H, O, OH, ...)

$HCO \rightarrow CO$

$CH_3O \rightarrow CH_2O$



Adapted from:
R. Fox, "Computational models for turbulent reacting flows", Cambridge University Press (2002)

Outline

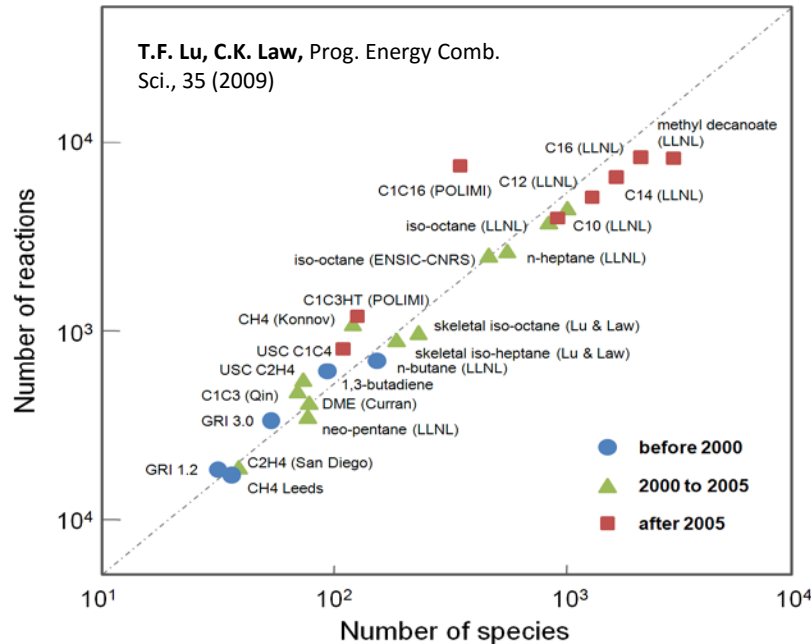
1. Presentation of OpenSMOKE++

1. Introduction
- 2. The OpenSMOKE++ Framework**
3. The OpenSMOKE++ Suite
4. Coupling with OpenFOAM

2. Training

1. Introduction (environment preparation, organization, ...)
2. Preprocessing of thermodynamics, transport and kinetics
3. OpenSMOKE++ Maps for thermodynamics, transport and kinetics
4. OpenSMOKE++ for modeling a batch reactor
5. A multidimensional laminar solver for reacting flows

Motivations



more complex reaction mechanisms in simulation of combustion processes

development of reaction mechanisms with different levels of **detail and comprehensiveness**

computational cost associated with such mechanisms is usually very high



OpenSMOKE++ Framework

Object oriented
User friendly
CPU efficient
Extensible



- need of **computational tools** to:
1. **manage** large kinetic mechanisms
 2. **integrate** them in new and/or existing numerical codes

The OpenSMOKE++ Framework

C++ Object-Oriented Programming

OOP produces code that is easier to write, validate, and maintain than procedural techniques. C++ is better suited for complex and highly dynamic data structures.

Templates and Policies

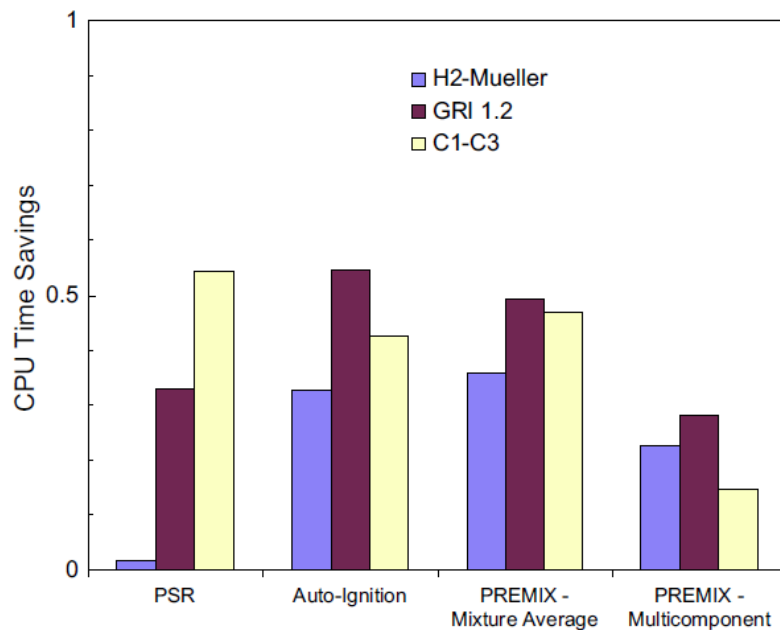
based on *template programming* and strongly relies on the concept of *policies* and *policy classes*, an important class design technique that enable the creation of flexible, highly reusable libraries.

Computation Cost Minimization (CCM)

- **code reformulation**: many parts of the numerical algorithms are reformulated in a less intuitive way in order to minimize the number of flops needed to perform some calculations
- **caching**: the code is written in order to cache as much as possible, which means storing items for future use in order to avoid retrieving or recalculating them
- **object pools**: they are a technique for avoiding the creation and deletion of a large number of objects during the code execution
- **optimized functions**: the numerical algorithms are often reformulated in order to exploit the Intel® MKL Vector Mathematical Functions Library (VML)

CCM: an example of code reformulation

Calculated savings in CPU time with CCM normalized by that of detailed mechanisms



Plot from: T.F. Lu, C.K. Law, Prog. Energy Comb. Sci., 35 (2009)

Natural implementation

$$k = AT^n \exp\left(-\frac{E}{RT}\right)$$

1 power: ~50 flops

1 exponentiation : ~50 flops

5 multiplications: ~5 flops

Total: ~105 flops

Smart implementation

$$k = \exp(\ln(A) + \alpha \ln(T) - E/RT)$$

1 exponentiation: ~50 flops

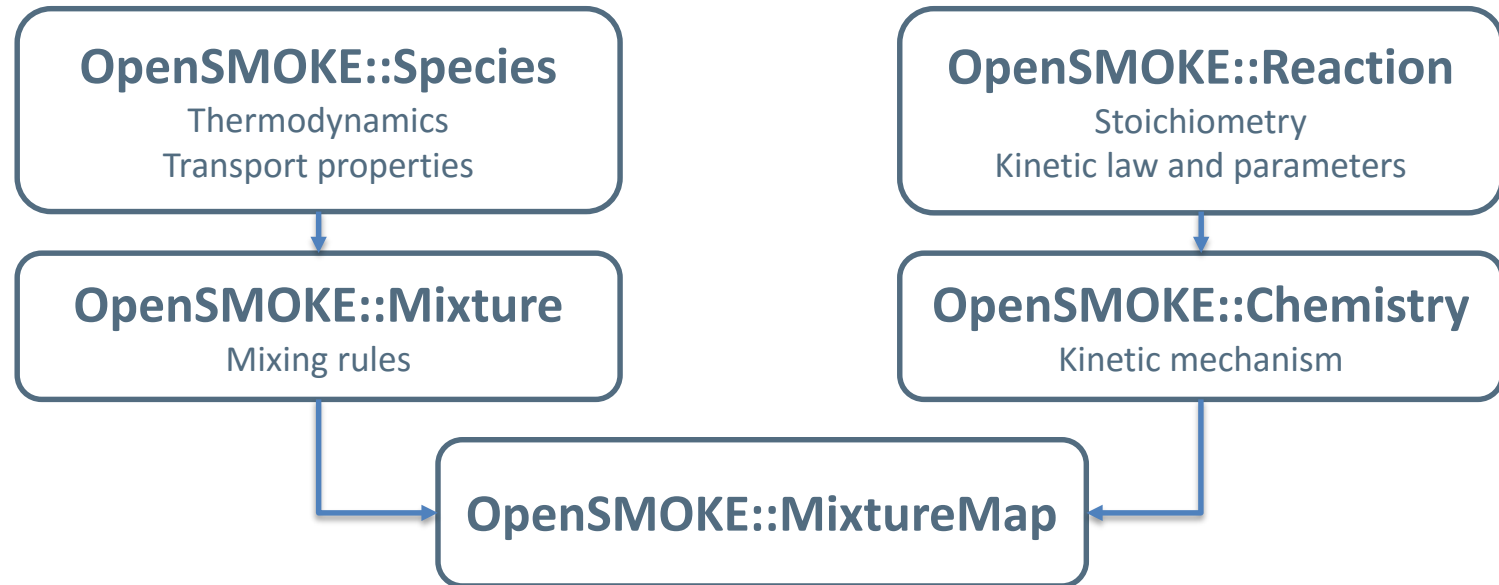
3 multiplications: ~5 flops

2 additions: ~2 flops

Total: ~57 flops

The $\ln(T)$ term above only has to be evaluated once for each call of the rate evaluation subroutine, and the $\ln(A)$ and E/R terms can be pre-evaluated.

OpenSMOKE++ Maps



```
1 OpenSMOKE::MixtureMap *map = new OpenSMOKE::MixtureMap(fileName);

2 map->updateStatus(T,P,y);           // update status
3 R = map->formationRates();           // formation rates
4 r = map->reactionRates();            // reaction rates
5 J = map->Jacobian();                 // Jacobian matrix
6 ropa = map->ROPA();                  // Rate of Production Analysis
```

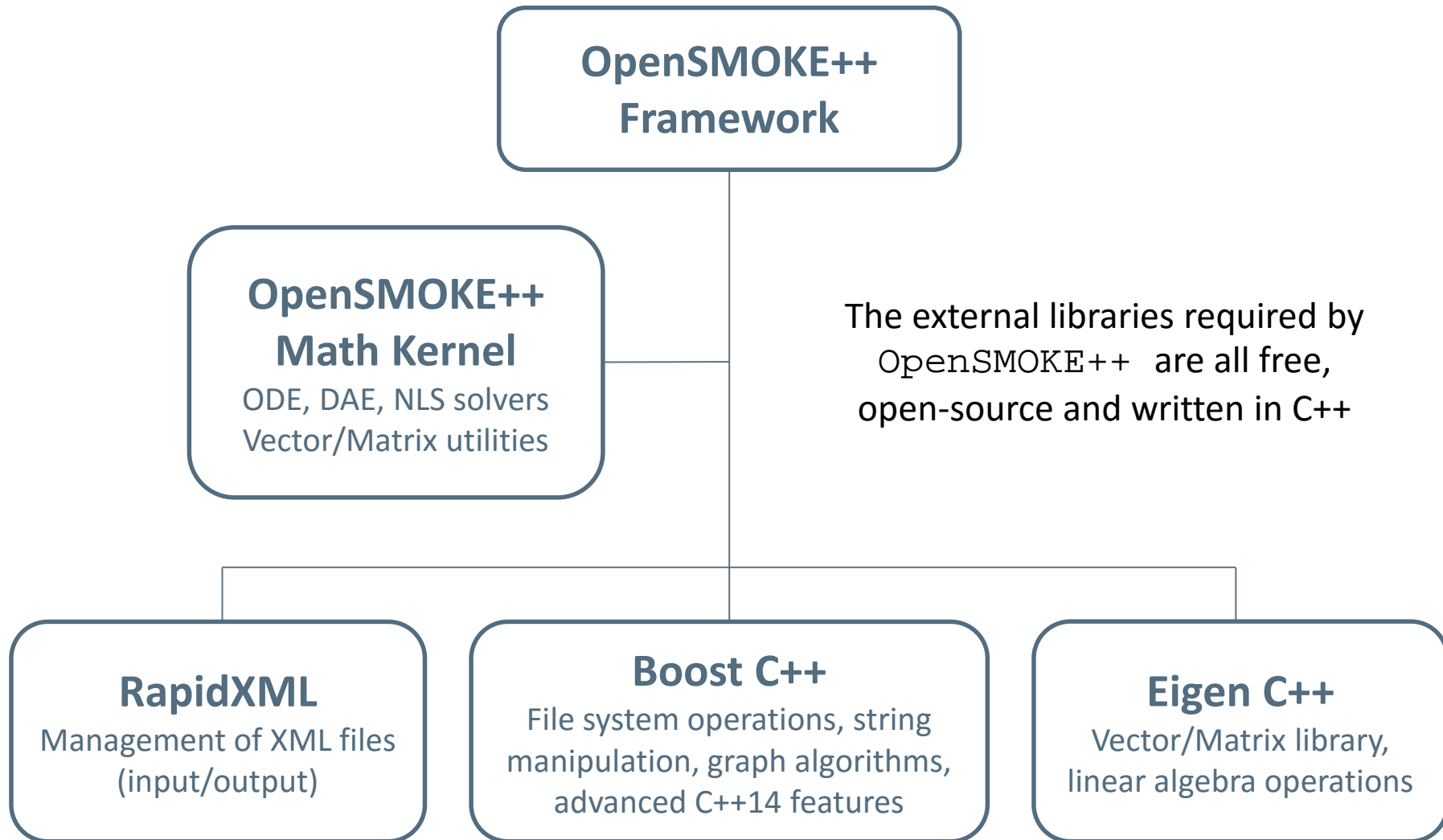
Relevant features of OpenSMOKE++

- Fully compatible with CHEMKIN format
- Heterogeneous catalytic reactions (CHEMKIN format)
- Detailed transport properties
- Species bundling (efficient calculations of diffusion coefficients)
- Semi-analytical Jacobian evaluation
- Dense and sparse (direct and iterative) linear solvers
- Coupling to a wide range of external ODE, DAE, and NLS solvers
- On-the-fly sensitivity and rate of production analysis

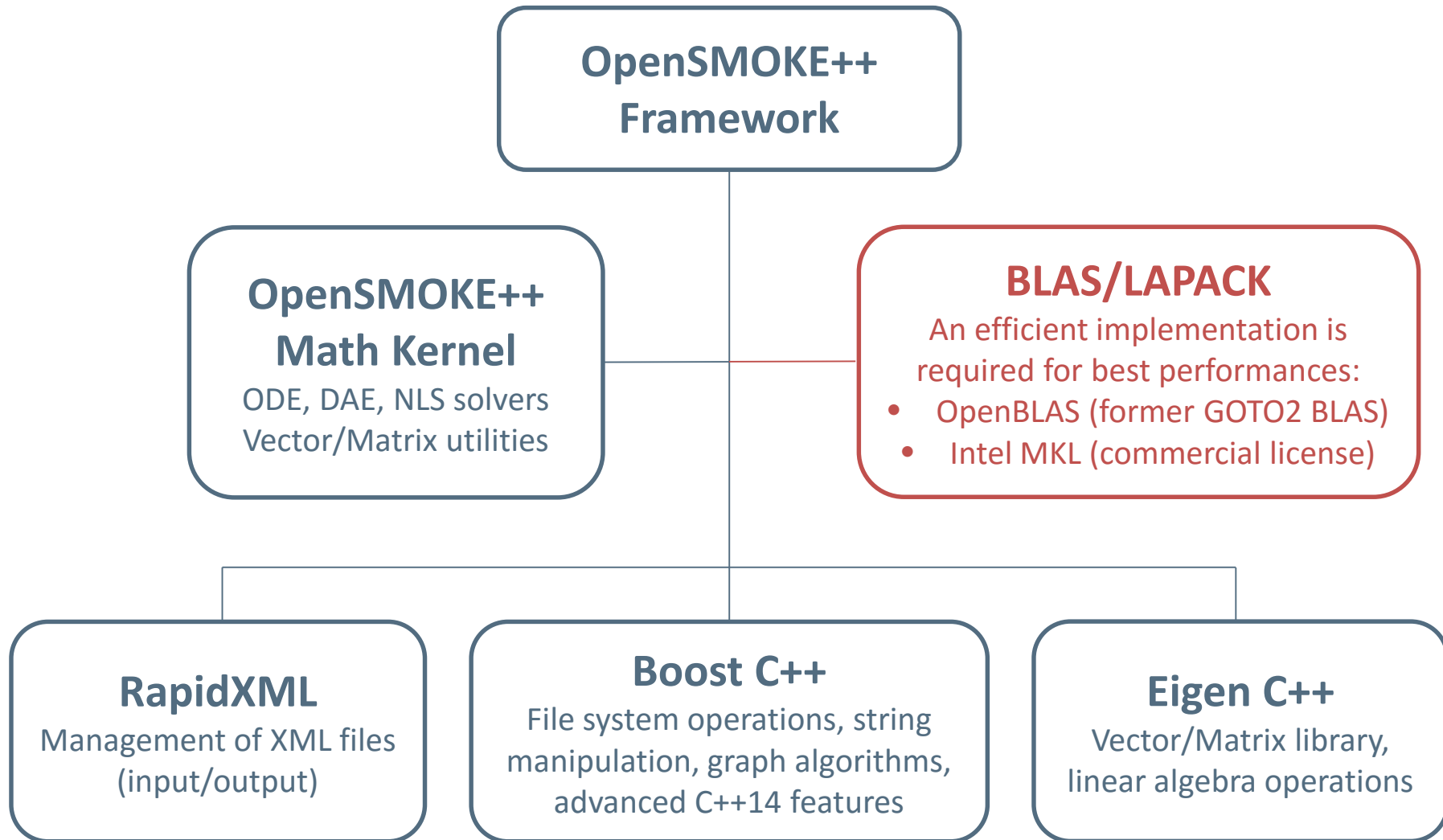
Work in progress

- Stefan-Maxwell approach for estimation of transport properties
- On-the-fly mechanism reduction (through DRG)
- On-the-fly stiffness removal
- Parallelization of ODE and DAE solvers (based on OpenMP®)

External dependencies



External dependencies



Additional external libraries (optional)

OpenSMOKE++ Framework

Sundials

Solvers for stiff ODE, DAE, and NL systems

ODEPACK

Solvers for stiff and non stiff ODE systems
(DLSODE, DLSODA, DVODE)

DASPK

Solver for DAE systems

RADAU

Solvers for stiff ODE systems

SuiteSparse

Solvers and utilities for linear sparse systems

SuperLU

Solvers and utilities for linear sparse systems

Outline

1. Presentation of OpenSMOKE++

1. Introduction
2. The OpenSMOKE++ Framework
- 3. The OpenSMOKE++ Suite**
4. Coupling with OpenFOAM

2. Training

1. Introduction (environment preparation, organization, ...)
2. Preprocessing of thermodynamics, transport and kinetics
3. OpenSMOKE++ Maps for thermodynamics, transport and kinetics
4. OpenSMOKE++ for modeling a batch reactor
5. A multidimensional laminar solver for reacting flows

Download the training session material

The source code adopted in this Training Session (including this presentation) can be downloaded from the GitHub repository available at the following address:

<https://github.com/acuoci/OpenFOAMTrainingCombustion>

You can clone the repository (suggested):

```
git clone https://github.com/acuoci/OpenFOAMTrainingCombustion.git
```

or download the corresponding zip file:

```
https://github.com/acuoci/OpenFOAMTrainingCombustion/archive/master.zip
```

The OpenSMOKE++ Suite (II)

1. Kinetic preprocessor

Fully compatible with CHEMKIN standard

2. Ideal reactors

Batch, plug flow, CSTR, shock-tube, rapid compression machine

3. Laminar flames

1D premixed flat flames, counterflow diffusion flames, burner stabilized stagnation flames

4. Laminar flamelets

Steady-state flamelet generator, look-up table generator

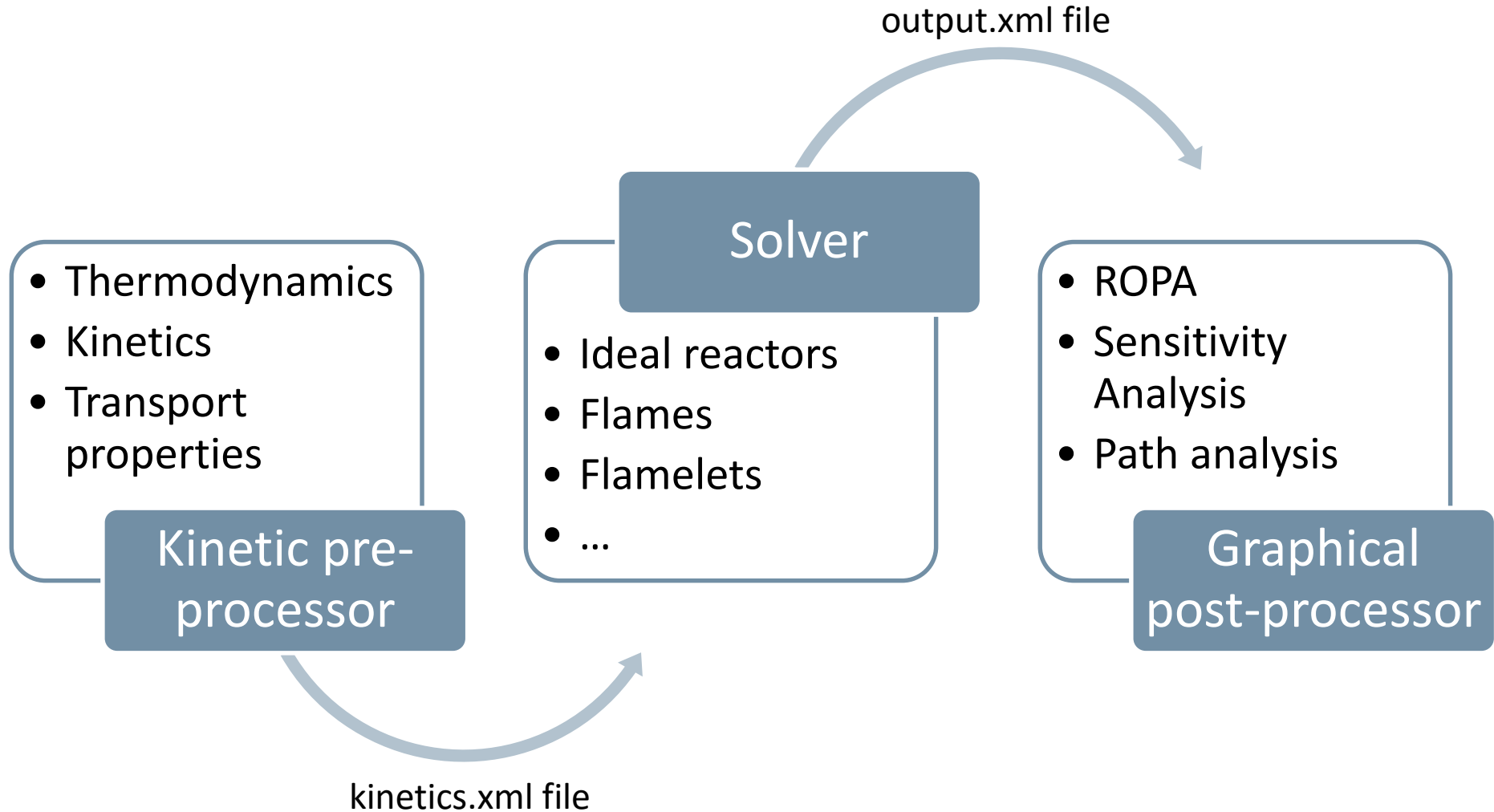
5. Heterogeneous catalytic reactors

Batch, plug-flow, honeycomb, CSTR

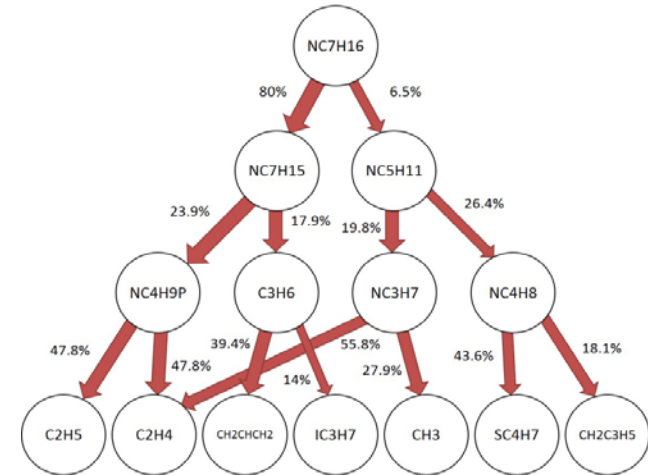
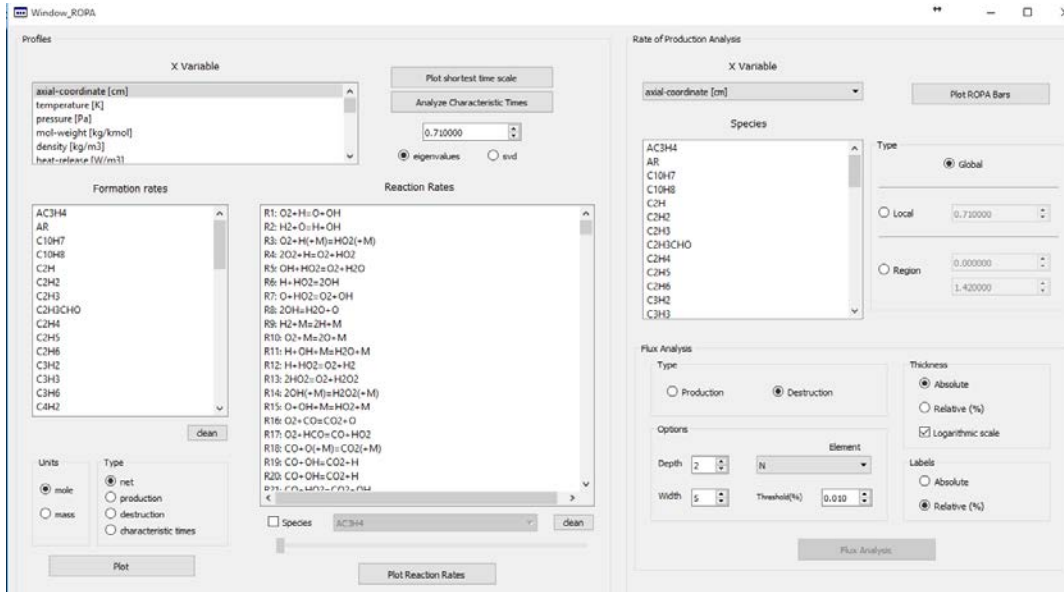
6. Reduction of kinetic mechanisms

DoctorSMOKE++ (flux-based (DRG with error propagation) and sensitivity analyses)

The OpenSMOKE++ Suite (III)



The Graphical Post-Processor (GPP)



adiabatic batch reactor (1 atm, 1200 K, $\Phi=1$) burning a mixture of nC7 and air (276 species, 8439 reactions)

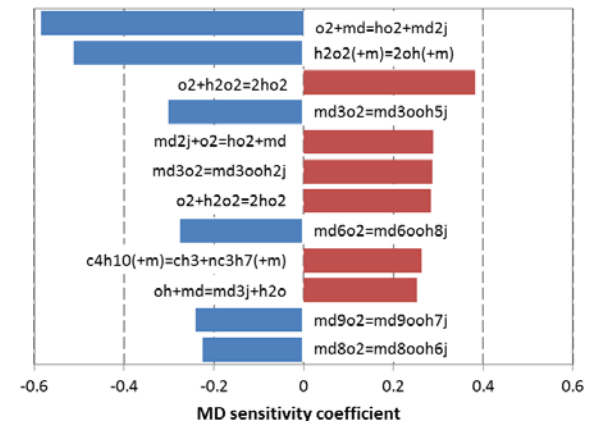
adiabatic batch reactor (2 atm, 750 K, $\Phi=1$) burning a mixture of MD and air (460 species, 16,000 reactions)

Automatic generation of bar charts and time/space profiles:

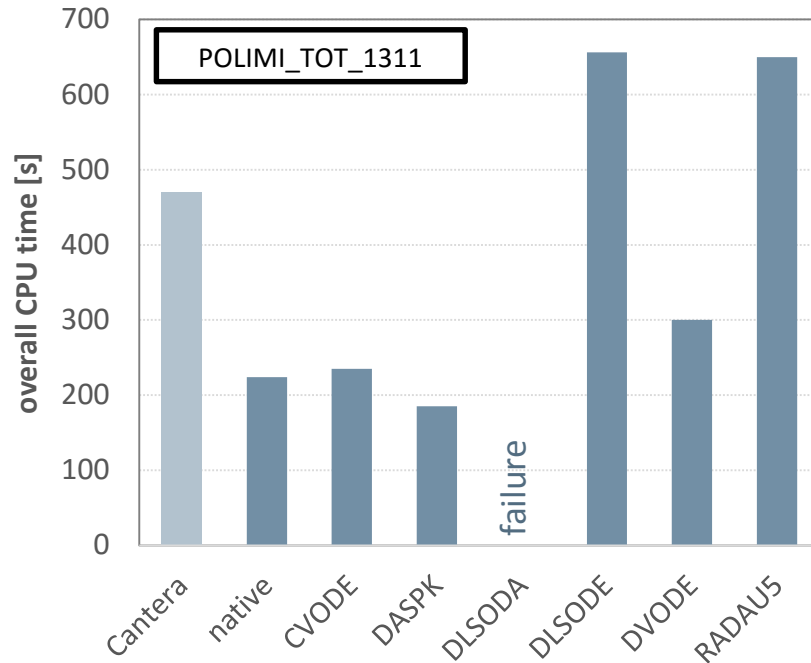
- Sensitivity analysis
- ROPA (Rate of Production Analysis)

Automatic generation of flux diagrams:

- Reaction Path Analysis

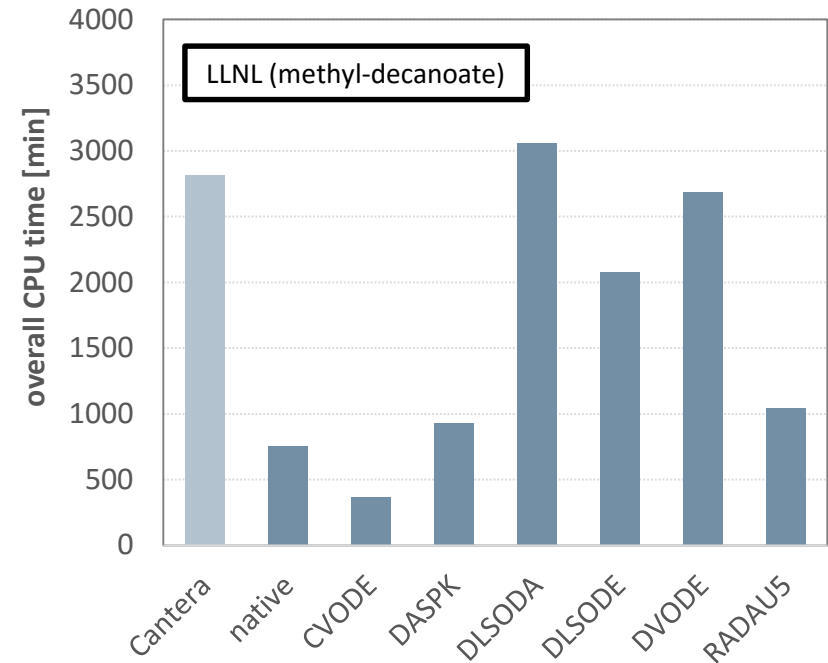


Performances



Constant volume, adiabatic batch reactors
burning a mixture of methyl-decanoate and air
(460 species, 16,000 reactions)

Pressures: 2-20 atm
Temperatures: 750-1500 K
Equivalence ratios: 0.5-2



Constant volume, adiabatic batch reactors
burning a mixture of methyl-decanoate and air
(2878 species, 8,855 reactions)

Pressures: 2-20 atm
Temperatures: 750-1500 K
Equivalence ratios: 0.5-2

Outline

1. Presentation of OpenSMOKE++

1. Introduction
2. The OpenSMOKE++ Framework
3. The OpenSMOKE++ Suite
- 4. Coupling with OpenFOAM**

2. Training

1. Introduction (environment preparation, organization, ...)
2. Preprocessing of thermodynamics, transport and kinetics
3. OpenSMOKE++ Maps for thermodynamics, transport and kinetics
4. OpenSMOKE++ for modeling a batch reactor
5. A multidimensional laminar solver for reacting flows

Coupling with OpenFOAM®

OpenFOAM®

Unstructured, multidimensional meshes
Spatial discretization of transport equations
Density/Pressure coupling algorithms
Input/Output management
MPI Parallelization
Turbulence models (RANS, LES)



OpenSMOKE++

Thermodynamics and detailed kinetics
Multicomponent transport properties
ODE solvers for stiff chemistry
Tools for kinetic analysis (ROPA)

laminarSMOKE

- Simulation of reacting flows in laminar conditions (coflow flames, burner stabilized stagnation flames, ...)

flameletSMOKE

- Simulation of turbulent flames based on the steady-state laminar flamelet

edcSMOKE

- Simulation of turbulent flames based on the Eddy Dissipation Concept (EDC) model

catalyticFOAM

- Simulation of catalytic heterogeneous (gas/solid) reactors (in cooperation with M. Maestri, Energy Dep. Polimi)

Coupling with OpenFOAM®

OpenFOAM®

Unstructured, multidimensional meshes
Spatial discretization of transport equations

Density/Pressure coupling algorithms
Input/Output management
MPI Parallelization
Turbulence models (RANS, LES)



OpenSMOKE++

Thermodynamics and detailed kinetics
Multicomponent transport properties
ODE solvers for stiff chemistry
Tools for kinetic analysis (ROPA)

laminarSMOKE

- Simulation of reacting flows in laminar conditions (coflow flames, burner stabilized stagnation flames, ...)

flameletSMOKE

- Simulation of turbulent flames based on the steady-state laminar flamelet

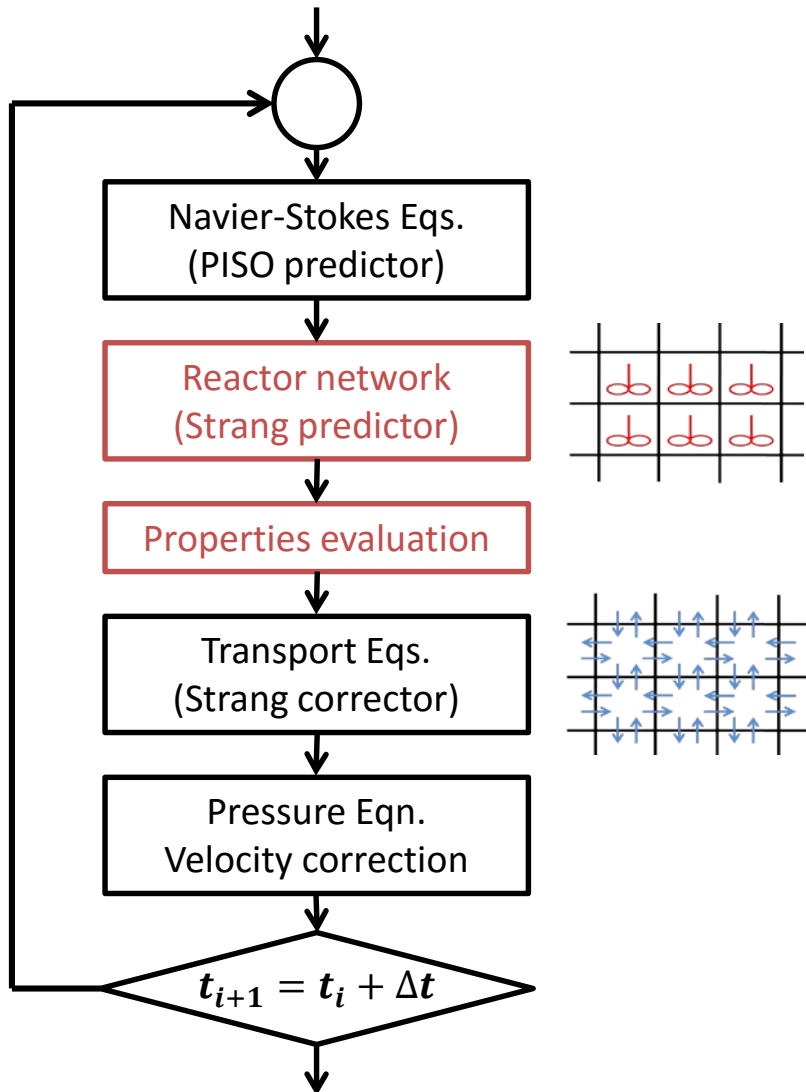
edcSMOKE

- Simulation of turbulent flames based on the Eddy Dissipation Concept (EDC) model

catalyticFOAM

- Simulation of catalytic heterogeneous (gas/solid) reactors (in cooperation with M. Maestri, Energy Dep. Polimi)

Operator-splitting algorithm



```
while (runTime.run())
{
    #include "readTimeControls.H"
    #include "readPISOControls.H"
    #include "compressibleCourantNo.H"
    #include "setDeltaT.H"

    runTime++;

    #include "rhoEqn.H"

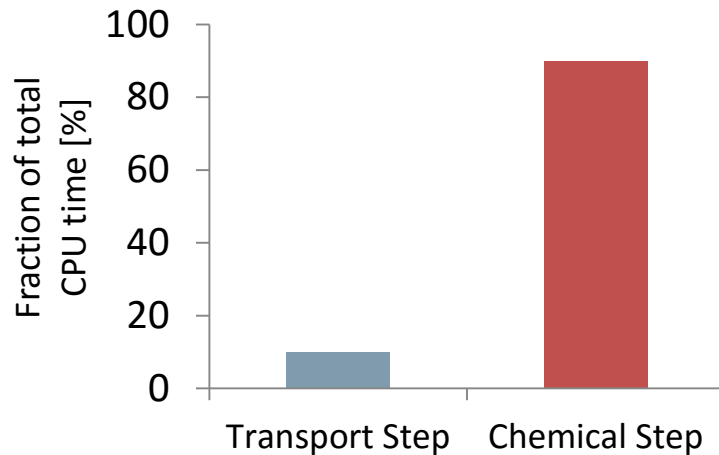
    for (label k=1;k<=nOuterCorr;k++)
    {
        #include "UEqn.H"
        #include "chemistry.H"
        #include "properties.H"
        #include "YEqn.H"
        #include "TEqn.H"

        for (int j=1;j<=nCorr;j++)
            #include "pEqn.H"
    }

    #include "write.H"
}
```

Performances

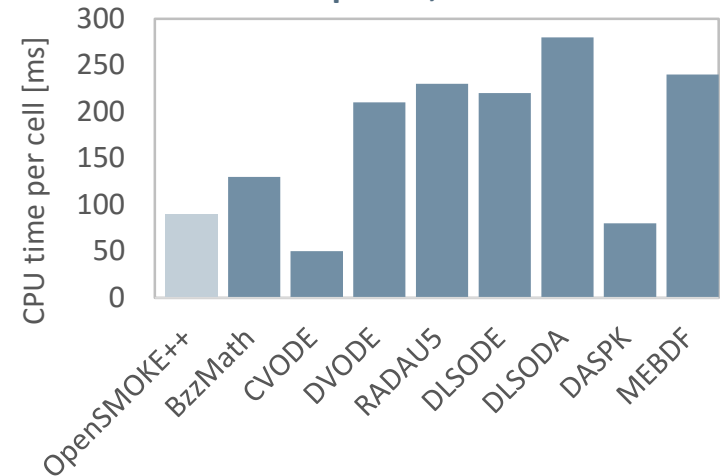
Most of the CPU Time (80-90%) is spent for the numerical integration of the ODE systems corresponding to the network of batch reactors



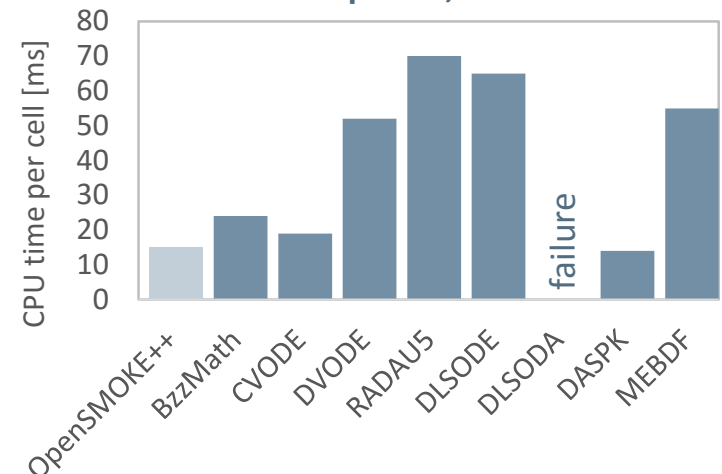
The best ODE solver depends on the features of the kinetic mechanism adopted:

- ✓ number of species
- ✓ species/reactions ratio
- ✓ lumped reactions

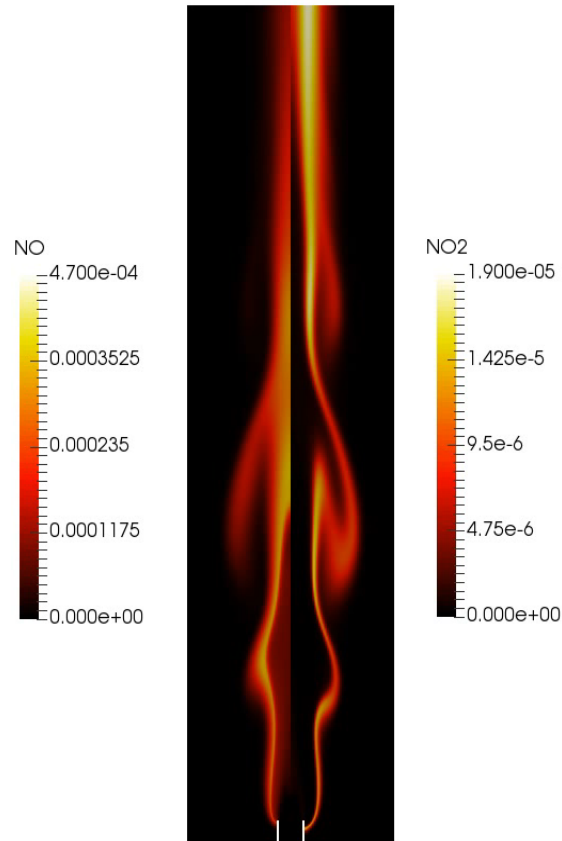
LLNL 680 species, 2400 reactions



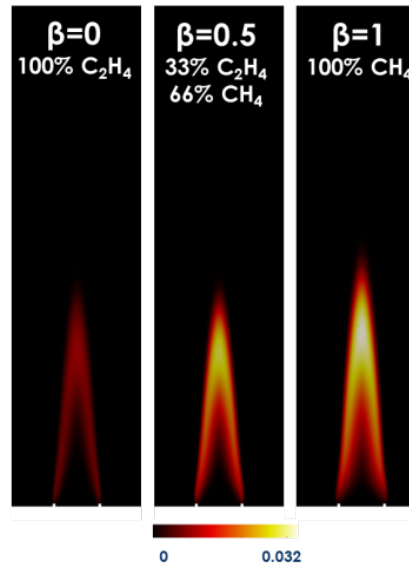
Polimi 156 species, 5400 reactions



Examples

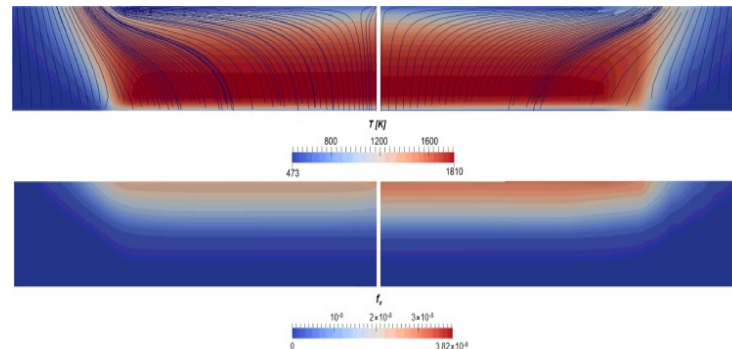
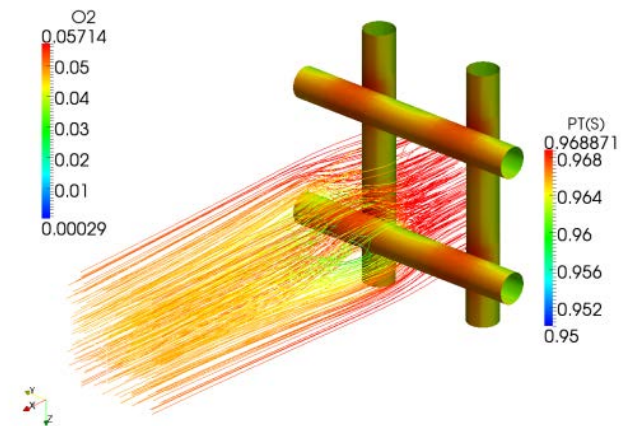


Naturally flickering coflow flame
fed with propane in air



Laminar coflow flames fed
with CH_4/C_2H_4 mixtures

Catalytic Partial Oxidation
of CH_4 over a Pt gauze



Soot formation in a
burner stabilized
stagnation flame

Naturally flickering flame (I)

Example of naturally flickering buoyancy-dominated diffusion flame



<https://www.youtube.com/watch?v=w5zWkSuYfIY>

Naturally flickering buoyancy-dominated diffusion flames exhibit natural flicker as a result of a **buoyancy-induced flow instability**, which leads to the formation of strong vortical motions that subsequently interact with the combusting regions of the flame.

Naturally occurring flickering flames are **difficult to investigate experimentally** because, even though the **flickering frequency** is well defined, there exist cycle-to-cycle variations. These variations lead to spatial and temporal averaging with a resulting loss in resolution.

Naturally flickering flame (II)

Chrystie & Chung (2014) Combustion Science and Technology, 186:4-5, 409-420, DOI: 10.1080/00102202.2014.883202

Fuel stream

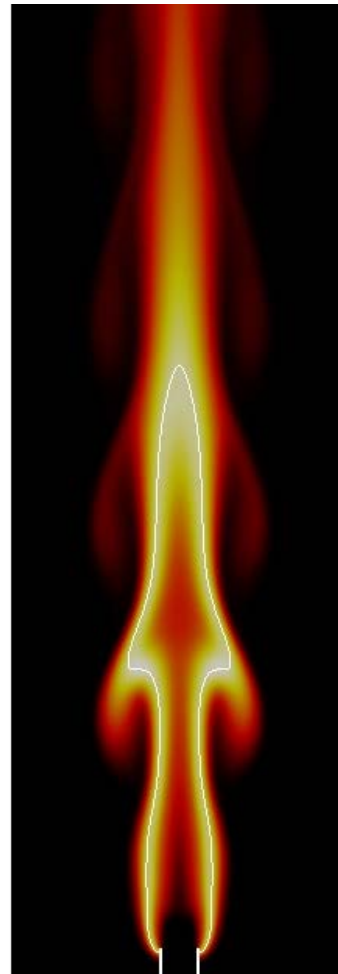
Composition: 100% C₃H₈
Temperature: 298 K
Velocity: 10 cm/s

Oxidizer stream

Composition: 21% O₂+79% N₂
Temperature: 298 K
Velocity: 7 cm/s

Burner

Diameter: 9 mm
Thickness: 0.8 mm

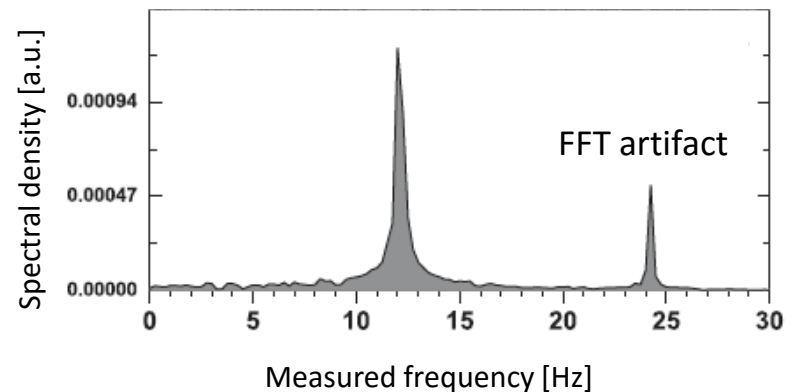


air fuel air

Under normal gravity conditions, laminar coflow diffusion flames have a well defined **oscillation frequency f** , which is inversely proportional to the square root of the burner diameter, D (in m):

$$f \sim \frac{1.5}{\sqrt{D}}$$

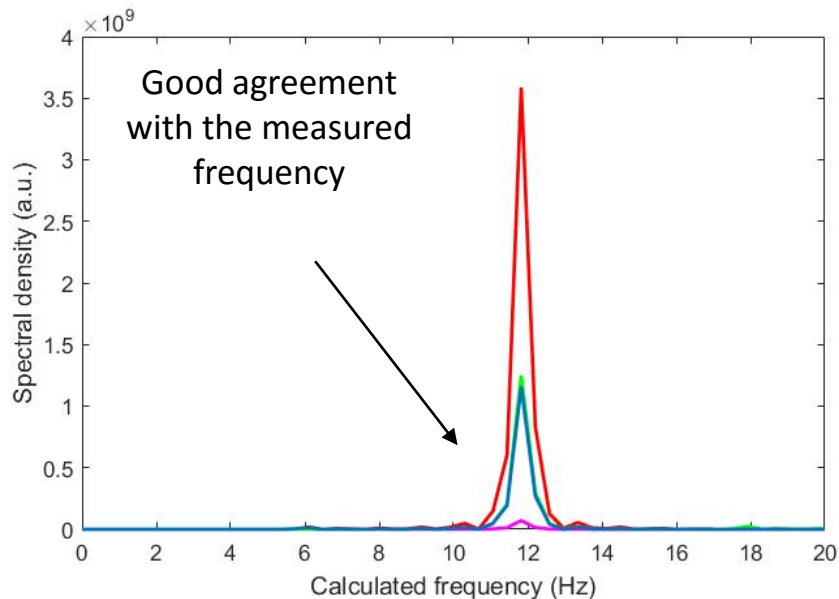
FFT decomposition of the frequencies (exp.)



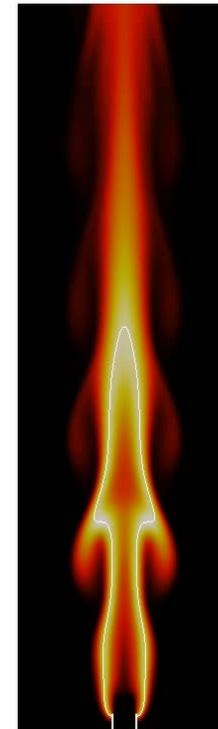
Naturally flickering flame (III)

2D computational domain: 75 x 300 mm²
Number of cells: 58,000
Spatial discretization: 2nd order centered
Time integration: Cranck-Nicolson
ODE solver: OpenSMOKE++
Absolute tolerance: 1e-12
Relative tolerance: 1e-7
Linear solver: Sparse, MKL Pardiso

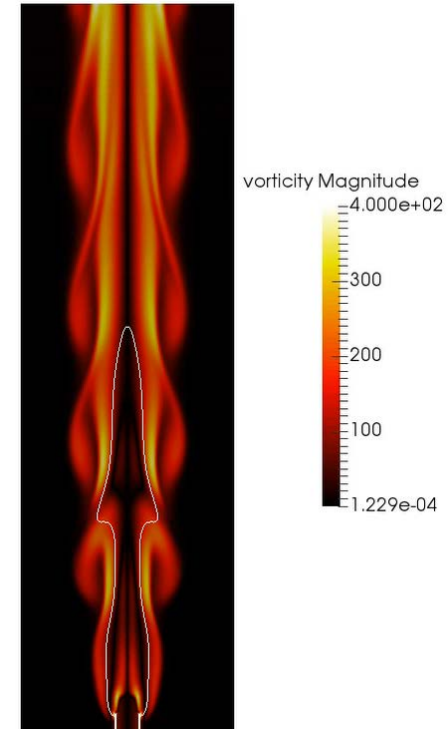
Kinetic mechanism: POLIMI_2015
Soot model: discrete sectional method
Number of species: 210
Number of reactions: 10,800



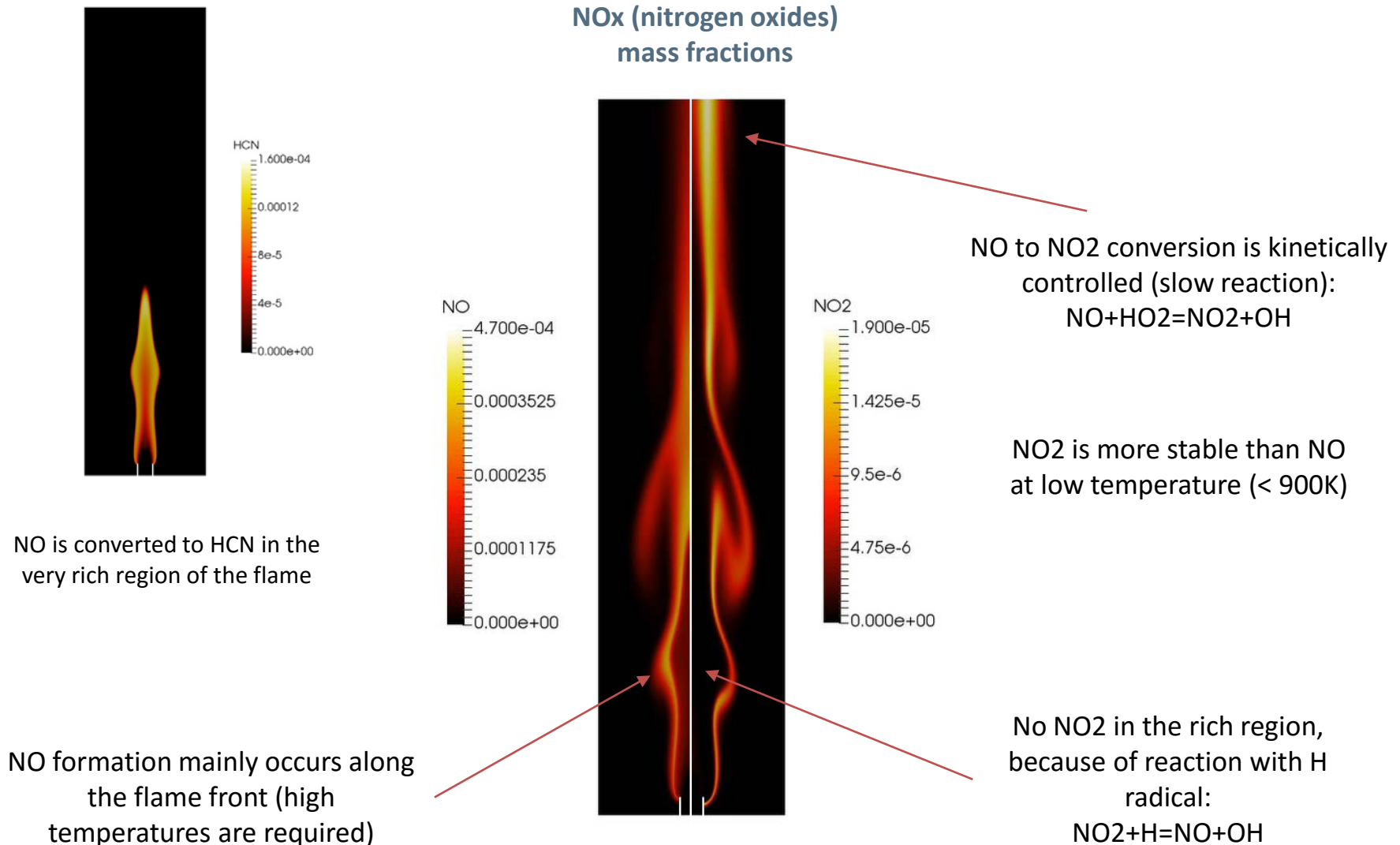
temperature



vorticity



Naturally flickering flame (IV)



Outline

1. Presentation of OpenSMOKE++

1. Introduction
2. The OpenSMOKE++ Framework
3. The OpenSMOKE++ Suite
4. Coupling with OpenFOAM

2. Training

- 1. Introduction (environment preparation, organization, ...)**
2. Preprocessing of thermodynamics, transport and kinetics
3. OpenSMOKE++ Maps for thermodynamics, transport and kinetics
4. OpenSMOKE++ for modeling a batch reactor
5. A multidimensional laminar solver for reacting flows

Download the training session material

The source code adopted in this Training Session can be downloaded from the GitHub repository available at the following address:

<https://github.com/acuoci/OpenFOAMTrainingCombustion>

You can download the corresponding zip file:

`https://github.com/acuoci/OpenFOAMTrainingCombustion/archive/master.zip`

or clone the repository:

```
git clone https://github.com/acuoci/OpenFOAMTrainingCombustion.git
```

Preparation of your environment

The OpenSMOKE++ libraries rely some third-party libraries for some important tasks:

- Eigen++ (for linear algebra operations)
- RapidXML (for XML file management)
- Boost C++ (for string manipulation, file system operations, etc.)

The Eigen++ and RapidXML libraries do not require precompilation, since they are entirely based on header files and they are provided together with the OpenSMOKE++ libraries. The Boost C++ libraries are not provided with OpenSMOKE++. Usually they are already available in your Linux distribution. If not, you need to install them. For Ubuntu and SuSE distribution you can use the following commands:

Ubuntu (versions 14.04 or above)

```
sudo apt-get install libboost-all-dev
```

SuSE (OpenSuSE/SLES v12 or above, or Tumbleweed)

```
sudo zypper install boost-devel
```

Compiling Boost C++ from source code

If you cannot install the Boost C++ libraries using the commands reported in the previous slide, You need to download the source code and compile it:

1. Download the source code from:

<https://dl.bintray.com/boostorg/release/1.64.0/source/>

2. Follow the compilation and installation instructions reported at:

http://www.boost.org/doc/libs/1_64_0/more/getting_started/unix-variants.html

After compilation and installation, set the environment variable pointing at the location where the compiled Boost C++ libraries have been installed (i.e. the folder containing the `include` and `lib` subfolders):

bash or ksh

```
export BOOST4OPENSMOKEPP=/path/to/boost
```

tcsh or csh

```
setenv BOOST4OPENSMOKEPP /path/to/boost
```


Checking if your environment is OK

In order to check if the environment is OK, we compile and run a simple application based on OpenSMOKE++ and OpenFOAM.

1. Go to the `Training/TestEnvironment` folder
2. Compile the source code by typing: `wmake`
3. If compilation successfully completed, run the solver by typing: `testEnvironment`

If everything was done properly, you should have the following output on the screen:

```
...
Selecting ODE solver seulex
0.000000e+00
1.000000e-06
2.000000e-06
3.000000e-06
4.000000e-06
5.000000e-06
6.000000e-06
7.000000e-06
8.000000e-06
9.000000e-06
```

Organization

OpenFOAMTrainingCombustion

- Docs
- KineticMechanisms
- Libraries
- PreProcessing
- Training

Docs

Documentation files (including this presentation)

KineticMechanisms

Collection of several detailed kinetic mechanisms in CHEMKIN format (thermodynamics, transport and kinetic files) adopted for running the simulations

Libraries

Source code of required libraries (OpenSMOKE++, Eigen++, RapidXML)

PreProcessing

Folder where pre-processing of kinetic mechanisms is carried out

Training

Source code of examples adopted in the current training session

KineticMechanisms folder

Global_H2_1step

Global 1-step mechanism for hydrogen combustion. Useful only for first-guess solutions. Here adopted only because its reduced computational time.

POLIMI_H2_1412

Detailed mechanism for hydrogen combustion: 9 species, 20 reactions

POLIMI_H2CO_1412

Detailed mechanism for syngas (H_2+CO) combustion: 14 species, 33 reactions

POLIMI_H2CO_NOX_1412

Detailed mechanism for syngas (H_2+CO) combustion with NOX (nitrogen oxides) chemistry (thermal, prompt, NNH, ...): 32 species, 173 reactions

GRI30

Detailed mechanism for methane combustion with NOX: 53 species, 325 reactions

GRI30noNOX

Detailed mechanism for methane combustion without NOX: 36 species, 219 reactions

Libraries folder

OpenSMOKEpp

This is the latest version of OpenSMOKE++ library. It is entirely based on header files, which means that no compilation is required.

eigen-3.3.3

Eigen is a C++ template library for linear algebra: matrices, vectors, numerical solvers, and related algorithms. It is entirely based on header files, which means that no compilation is required. New versions can be downloaded from:

http://eigen.tuxfamily.org/index.php?title=Main_Page

Rapidxml-1.13

RapidXML is a very fast XML parser, user-friendly and very portable, since entirely based on header files. New versions can be downloaded from:

<http://rapidxml.sourceforge.net/index.htm>

Training folder

TestEnvironment

Source code for testing the OpenFOAM/OpenSMOKE++ environment

OpenSMOKEpp_CHEMKIN_PreProcessor

Source code for OpenSMOKE++ kinetic preprocessor

Maps

Source code showing how to use the OpenSMOKE++ Maps for thermodynamics, transport properties, and detailed kinetics

BatchReactor

Source code showing how to implement a batch reactor solver coupling OpenSMOKE++ with OpenFOAM

Solvers

Source code for multidimensional solvers for laminar reacting flows with detailed kinetic mechanisms (steady-state and unsteady) based on OpenFOAM and coupled with OpenSMOKE++

Outline

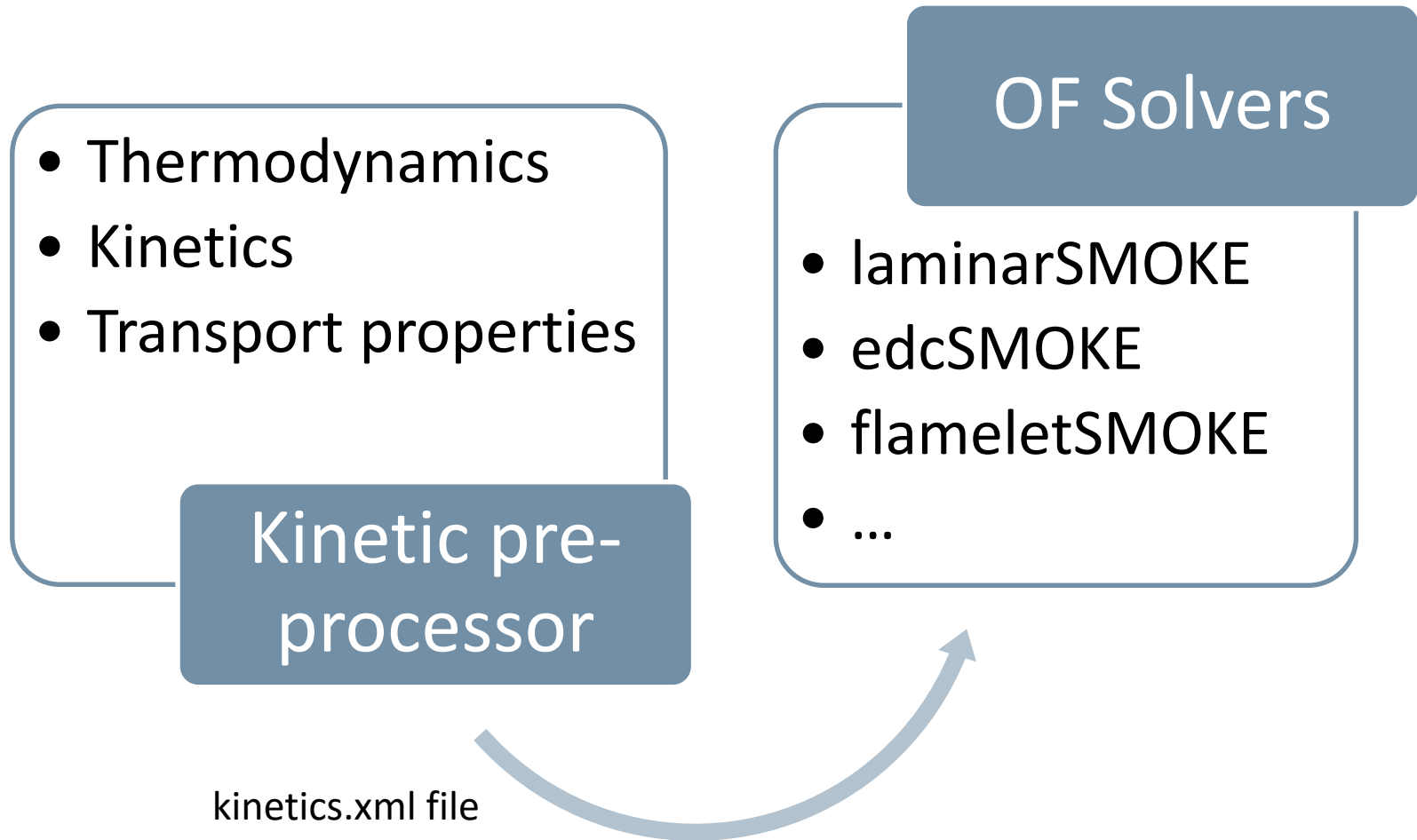
1. Presentation of OpenSMOKE++

1. Introduction
2. The OpenSMOKE++ Framework
3. The OpenSMOKE++ Suite
4. Coupling with OpenFOAM

2. Training

1. Introduction (environment preparation, organization, ...)
- 2. Preprocessing of thermodynamics, transport and kinetics**
3. OpenSMOKE++ Maps for thermodynamics, transport and kinetics
4. OpenSMOKE++ for modeling a batch reactor
5. A multidimensional laminar solver for reacting flows

Preprocessing of CHEMKIN mechanisms



Compiling the CHEMKIN PreProcessor

The `OpenSMOKEpp_CHEMKIN_PreProcessor` is a utility to pre-process thermodynamics, transport properties and chemical reactions available in CHEMKIN format.

1. The source code is available in the
 `Training/OpenSMOKEpp_CHEMKIN_PreProcessor` folder
2. Open the `Training/OpenSMOKEpp_CHEMKIN_PreProcessor` folder
3. Type `wmake`

Running the CHEMKIN PreProcessor (I)

In order to preprocess a kinetic mechanism, three files are needed (the names are arbitrary):

1. `Thermo.CKT` (thermodynamic properties)
2. `Transport.TRC` (transport properties)
3. `Kinetics.CKI` (reactions)

Examples of kinetic mechanisms are available in the `KineticMechanisms` folder

The CRECK Modeling group at Politecnico di Milano provides detailed kinetic mechanisms describing pyrolysis, combustion, and oxidation of several fuels (hydrogen, methane, propane, PRF, diesels, jet-fuels, biofuels, etc.):

<http://creckmodeling.chem.polimi.it/>

A **User's Guide** explaining the features and the complete list of options of OpenSMOKE++ Kinetic PreProcessor is available in the `Docs` folder (`CHEMKIN_PrePreprocessor.pdf` file).

Running the CHEMKIN PreProcessor (II)

As a first example, the POLIMI_H2_1412 kinetic mechanism will be pre-processed.

The required files (thermodynamics, transport, and kinetics) are available in the KineticMechanisms/POLIMI_H2_1412 folder, but we recommend to preprocess kinetic mechanisms from a different folder. In this training session, we will use the PreProcessing folder for preprocessing activities.

1. Go to the PreProcessing/POLIMI_H2_1412 folder
2. Run the OpenSMOKEpp_CHEMKIN_PreProcessor :
`OpenSMOKEpp_CHEMKIN_PreProcessor --input input.dic`
3. Results are available in the kinetics-POLIMI_H2_1412 folder

Running the CHEMKIN PreProcessor (III)

The `input.dic` file (the name is arbitrary) is a text file in which the user specifies the paths to the three files containing thermodynamics, transport properties and chemical reactions

```
Dictionary CHEMKIN_PreProcessor
{
    @Thermodynamics    POLIMI_TOT_NOX_1412.CKT;
    @Transport          POLIMI_TOT_NOX_1412.TRC;
    @Kinetics           POLIMI_H2_1412.CKI;

    @Output             kinetics-POLIMI_H2_1412;
}
```

`input.dic`

Analysis of thermodynamic properties (I)

It is important to check the consistency of thermodynamic data:

- continuity of thermodynamic functions
- continuity of first-order derivative (recommended)
- monotonic behavior of thermodynamic functions

```
Dictionary CHEMKIN_PreProcessor
{
    @Thermodynamics      POLIMI_TOT_NOX_1412.CKT;
    @Transport            POLIMI_TOT_NOX_1412.TRC;
    @Kinetics              POLIMI_H2_1412.CKI;
    @CheckThermodynamics  true;

    @Output                kinetics-POLIMI_H2_1412;
}
```

input.dic

Analysis of thermodynamic properties (II)

=====						
Status of specific heat						

Name	Tl	Tm	Th	Cp/R(-)	Cp/R(+)	e(%)
=====						
H2	200	750	3500	3.55E+00	3.55E+00	1.37E-07
O2	200	760	3500	4.02E+00	4.02E+00	2.84E-07
H2O	200	1590	3500	5.77E+00	5.77E+00	1.07E-07
H2O2	200	1180	3500	7.99E+00	7.99E+00	3.25E-08
O	200	950	3500	2.52E+00	2.52E+00	3.17E-08
H	200	1490	3500	2.50E+00	2.50E+00	1.71E-10
OH	200	880	3500	3.65E+00	3.65E+00	5.68E-08
HO2	200	1540	3500	6.32E+00	6.32E+00	1.77E-07
N2	200	1050	3500	3.96E+00	3.96E+00	1.80E-07

ThermodynamicsStatus.out

Analysis of thermodynamic properties (III)

```
=====
Anomalous behaviour of thermodynamic properties
=====
HT Cp for O in [950.00-3500.00] has local maxima at: 1926.35
LT Cp for H in [200.00-1490.00] has local maxima at: 605.76 1321.10
HT Cp for H in [1490.00-3500.00] has local maxima at: 2287.83 3148.97
LT Cp for N2 in [200.00-1050.00] has local maxima at: 330.66
```

ThermodynamicsStatus.out

Rewriting the thermodynamic properties (I)

When the @CheckThermodynamics option is turned on, the thermodynamic properties are corrected in order to ensure the continuity up to the 3rd order derivative and rewritten in new file

thermo.CHEMKIN.CKT

```
! This thermodynamic database was obtained by fitting the thermodynamic
! Properties extracted from the following file: POLIMI_TOT_NOX_1412.CKT
! The thermodynamic properties are fitted in order to preserve not only the
! continuity of each function at the intermediate temperature, but also the
! Continuity of the derivatives, from the 1st to the 3rd order. The intermediate
! temperatures are chosen in order to minimize the fitting error.
```

THERMO ALL

270. 1000. 3500.

H2	H	2	G	200.00	3500.00	750.00	1
3.73110903e+00	-8.86706232e-04	1.12286898e-06	-3.74349785e-10	4.17963677e-14			2
-1.08851547e+03	-5.35285859e+00	3.08866002e+00	2.53968851e-03	-5.72992050e-06			3
5.71701864e-09	-1.98865977e-12	-9.92148123e+02	-2.43823453e+00				4
O2	O	2	G	200.00	3500.00	760.00	1
2.81750647e+00	2.49838007e-03	-1.52493520e-06	4.50547600e-10	-4.87702781e-14			2
-9.31713391e+02	7.94729339e+00	3.46035082e+00	-8.85011230e-04	5.15281079e-06			3
-5.40712432e-09	1.87809548e-12	-1.02942573e+03	5.02236119e+00				4

Rewriting the thermodynamic properties (II)

In addition, the thermodynamic properties are also re-fitted and rewritten using exactly the same intermediate temperature for every species (default 1000 K). This could be required by some old applications.

thermo.CHEMKIN.fixedT.CKT

```
! This thermodynamic database was obtained by fitting the thermodynamic
! Properties extracted from the following file: POLIMI_TOT_NOX_1412.CKT
! The thermodynamic properties are fitted in order to preserve not only the
! continuity of each function at the intermediate temperature, but also the
! Continuity of the derivatives, from the 1st to the 3rd order. The
! intermediate temperatures are the same for all the species.
```

THERMO ALL

270. 1000. 3500.

H2	H	2	G	200.00	3500.00	1000.00	1
3.87655112e+00	-1.22795344e-03	1.39229943e-06	-4.61414188e-10	5.16896624e-14			2
-1.13335667e+03	-6.12445685e+00	3.23482792e+00	1.33893937e-03	-2.45803978e-06			3
2.10547862e-09	-5.90033540e-13	-1.00501203e+03	-3.02851335e+00				4
O2	O	2	G	200.00	3500.00	1000.00	1
2.67521006e+00	2.83203854e-03	-1.78825691e-06	5.35610921e-10	-5.84339610e-14			2
-8.87805402e+02	8.70230607e+00	3.32264375e+00	2.42303767e-04	2.09634526e-06			3
-2.05412386e-09	5.88999733e-13	-1.01729214e+03	5.57881275e+00				4

Analysis of transport properties (I)

Transport properties of individual species can be rewritten using a simple analytical expression based on a 3rd order polynomial function

```
Dictionary CHEMKIN_PreProcessor
{
    @Thermodynamics      POLIMI_TOT_NOX_1412.CKT;
    @Transport            POLIMI_TOT_NOX_1412.TRC;
    @Kinetics             POLIMI_H2_1412.CKI;
    @TransportFittingCoefficients true;

    @Output               kinetics-POLIMI_H2_1412;
}
```

input.dic

Analysis of transport properties (II)

VISCOSITY FITTING COEFFICIENTS

$$\mu = \exp(A + B \cdot \log T + C \cdot (\log T)^2 + D \cdot (\log T)^3) \quad [\text{kg/m/s}]$$

Species	A	B	C	D	$\mu(298\text{K})$	$\mu(1000\text{K})$
1. H2	-1.583982e+01	8.692958e-01	-3.054617e-02	1.384164e-03	8.958993e-06	1.967286e-05
2. O2	-1.906650e+01	2.494518e+00	-2.440586e-01	1.072526e-02	2.053924e-05	4.794243e-05
3. H2O	-2.028994e+01	2.046198e+00	-7.669556e-02	-7.595355e-04	1.284832e-05	4.252813e-05
4. H2O2	-1.903595e+01	2.494518e+00	-2.440586e-01	1.072526e-02	2.117638e-05	4.942964e-05
5. O	-1.702676e+01	1.755265e+00	-1.474358e-01	6.515978e-03	2.473511e-05	5.605138e-05
6. H	-2.213944e+01	3.389886e+00	-3.580432e-01	1.556519e-02	9.447645e-06	2.306615e-05
7. OH	-1.699621e+01	1.755265e+00	-1.474358e-01	6.515978e-03	2.550241e-05	5.779014e-05
8. HO2	-1.905099e+01	2.494518e+00	-2.440586e-01	1.072526e-02	2.086024e-05	4.869171e-05
9. N2	-1.853769e+01	2.232971e+00	-2.102014e-01	9.264622e-03	1.799150e-05	4.152188e-05

TransportProperties_Coefficients.out

Reverse reaction rates (I)

For each reversible reaction the reverse kinetic constants are estimated assuming the Arrhenius' law.

```
Dictionary CHEMKIN_PreProcessor
{
    @Thermodynamics      POLIMI_TOT_NOX_1412.CKT;
    @Transport            POLIMI_TOT_NOX_1412.TRC;
    @Kinetics              POLIMI_H2_1412.CKI;
    @ReverseFitting       true;

    @Output                kinetics-POLIMI_H2_1412;
}
```

input.dic

Reverse reaction rates (II)

The kinetic parameters of the reverse reactions are then written on the `Reaction_FittedKinetics.out` file.

Reaction index	A [kmol,m3,s]	Beta	E [cal/mol]	Reaction
8	1.2072e+10	-1.327	-7393.17	H2+M=2H+M
9	9.8066e+07	-0.922	-3665.47	O2+M=2O+M
1	2.9746e+09	0.183	-92.14	O2+H=O+OH
15	6.4850e+06	1.060	4188.86	H2O+H=H2+OH
2	2.2959e+10	-0.022	8594.16	H2+O=H+OH
7	4.9835e+02	2.362	14106.98	2OH=H2O+O
17	1.0893e+08	0.690	23969.46	H2O2+H=H2+HO2
5	1.0054e+08	0.677	38520.04	H+HO2=2OH
12	2.8566e+11	-0.374	38723.57	2HO2=O2+H2O2
3	2.9177e+16	-1.779	48821.40	O2+H(+M)=HO2(+M)
13	2.9572e+23	-3.406	50244.70	2OH(+M)=H2O2(+M)
6	4.2181e+09	0.294	53337.18	O+HO2=O2+OH
...				

Reaction tables (I)

For each reaction it is possible to write details about the reaction rate, the reaction enthalpy and entropy, equilibrium constant, etc.

```
Dictionary CHEMKIN_PreProcessor
{
    @Thermodynamics      POLIMI_TOT_NOX_1412.CKT;
    @Transport            POLIMI_TOT_NOX_1412.TRC;
    @Kinetics             POLIMI_H2_1412.CKI;
    @ReactionTables       true;

    @Output               kinetics-POLIMI_H2_1412;
}
```

input.dic

Reaction tables (II)

The reaction tables are then written on the `Reaction_FittedKinetics.out` file.

```
=====
KINETIC DATA - REACTION  1 Simple
  O2 + H  = O + OH
=====
Change in moles in the reaction = 0.000
Reaction order (forward)  = 2.000
Reaction order (backward) = 2.000
k = 9.600000e+11 T^-0.200 exp(-6.956615e+07/RT)  [m3/kmol/s] and [J/kmol]
k = 9.600000e+14 T^-0.200 exp(-16625.00/RT)  [cm3/mol/s] and [cal/mol]
Reverse reaction units: [m3/kmol/s] or [cm3/mol/s]
-----
```

Temp. [K]	kF [kmol,m3,s]	Keq [-]	kR [kmol,m3,s]	DG [kcal/mol]	DH [kcal/mol]
300	2.369e-01	2.418e-11	9.796e+09	14.572	16.375
1000	5.605e+07	5.119e-03	1.095e+10	10.481	16.011
1500	8.406e+08	7.248e-02	1.160e+10	7.822	15.521
2000	3.201e+09	2.620e-01	1.222e+10	5.324	15.086
2500	7.067e+09	5.547e-01	1.274e+10	2.928	14.700

```
-----
```

Outline

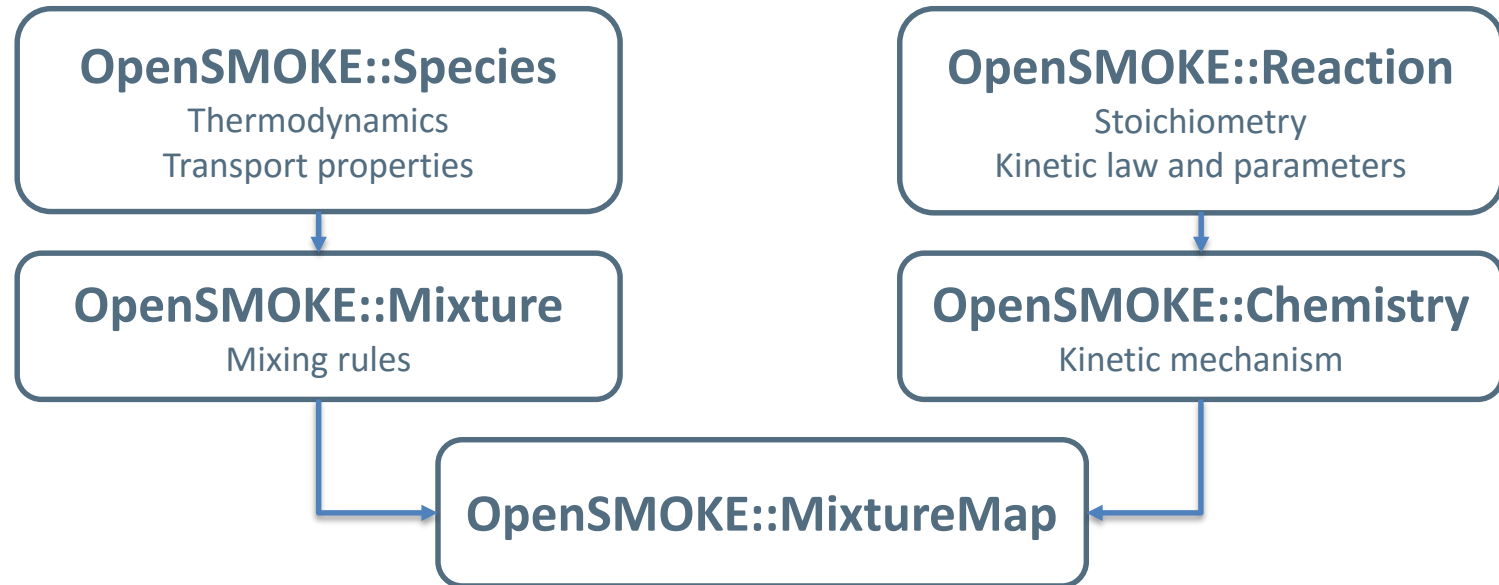
1. Presentation of OpenSMOKE++

1. Introduction
2. The OpenSMOKE++ Framework
3. The OpenSMOKE++ Suite
4. Coupling with OpenFOAM

2. Training

1. Introduction (environment preparation, organization, ...)
2. Preprocessing of thermodynamics, transport and kinetics
- 3. OpenSMOKE++ Maps for thermodynamics, transport and kinetics**
4. OpenSMOKE++ for modeling a batch reactor
5. A multidimensional laminar solver for reacting flows

OpenSMOKE++ Maps



```
1 OpenSMOKE::MixtureMap *map = new OpenSMOKE::MixtureMap(fileName);

2 map->updateStatus(T,P,y);           // update status
3 R = map->formationRates();           // formation rates
4 r = map->reactionRates();            // reaction rates
5 J = map->Jacobian();                 // Jacobian matrix
5 ropa = map->ROPA();                  // Rate of Production Analysis
```


OpenSMOKE++ Maps (II)

OpenSMOKE++ is based on the concept of Maps, i.e. classes containing the data (imported from the xml files resulting from the preprocessing operations) and the functions to manage thermodynamic and transport properties and chemical reactions.

In particular in this training session we will focus the attention on maps based on the CHEMKIN standard:

- `OpenSMOKE::ThermodynamicsMap_CHEMKIN`
- `OpenSMOKE::TransportPropertiesMap`
- `OpenSMOKE::KineticsMap_CHEMKIN`

The source code is available in the Training/Maps folder:

Training/Maps

- 01-thermodynamics
- 02-transport
- 03-kinetics

Including OpenSMOKE++ definitions

1. Go to the Training/Maps/01-thermodynamics/ folder
2. Open the trainingMapsThermodynamics.C file

In order to use the OpenSMOKE++ Maps (thermodynamics, transport and kinetics), proper definitions must be included

```
// OpenSMOKE++ Definitions
#include "OpenSMOKEpp"

// CHEMKIN maps
#include "maps/Maps_CHEMKIN"
```

Importing the thermodynamic map

```
// OpenSMOKE++ Thermodynamic Map
OpenSMOKE::ThermodynamicsMap_CHEMKIN* thermoMap;

// Import map from preprocessed XML file
{
    boost::filesystem::path file_path =
        "kinetics-POLIMI_H2_1412/kinetics.xml";

    // Open XML file containing the thermodynamic data
    rapidxml::xml_document<> doc;
    std::vector<char> xml_string;
    OpenSMOKE::OpenInputFileXML(doc, xml_string, file_path);

    // Import
    thermoMap = new OpenSMOKE::ThermodynamicsMap_CHEMKIN(doc);
}
```

Basic public functions

```
// Number of species
thermoMap->NumberOfSpecies();

// List of species
for (unsigned int i=0;i<thermoMap->NumberOfSpecies();i++)
    thermoMap->NamesOfSpecies()[i];

// Molecular weights [kg/kmol]
for (unsigned int i=0;i<thermoMap->NumberOfSpecies();i++)
    thermoMap->MW(i);
```

The complete list of public functions for ThermodynamicsMap_CHEMKIN class is available here:

`Libraries/OpenSMOKEpp/maps/ThermodynamicsMap_CHEMKIN.h`

Mixture averaged properties

```
// Set thermodynamic map
thermoMap->SetTemperature(T);
thermoMap->SetPressure(P);

// Molecular weight (in kg/kmol)
const double mw = thermoMap->MolecularWeight_From_MoleFractions(x.data());

// Constant pressure specific heat (in J/kmol/K)
const double cp = thermoMap->cpMolar_Mixture_From_MoleFractions(x.data());

// Enthalpy (in J/kmol)
const double h = thermoMap->hMolar_Mixture_From_MoleFractions(x.data());

// Entropy (in J/kmol/K)
const double s = thermoMap->sMolar_Mixture_From_MoleFractions(x.data());

// Internal energy (in J/kmol)
const double u = thermoMap->uMolar_Mixture_From_MoleFractions(x.data());

// Gibb's free energy (in J/kmol)
const double g = thermoMap->gMolar_Mixture_From_MoleFractions(x.data());

// Helmotz's free energy (in J/kmol)
const double a = thermoMap->aMolar_Mixture_From_MoleFractions(x.data());
```

Species properties

```
// Constant pressure specific heat (in J/kmol/K)
Eigen::VectorXd cp_species(thermoMap->NumberOfSpecies());
thermoMap->cpMolar_Species(cp_species.data());

// Enthalpy (in J/kmol)
Eigen::VectorXd h_species(thermoMap->NumberOfSpecies());
thermoMap->hMolar_Species(h_species.data());

// Entropy (in J/kmol/K)
Eigen::VectorXd s_species(thermoMap->NumberOfSpecies());
thermoMap->sMolar_Species(s_species.data());

// Internal energy (in J/kmol)
Eigen::VectorXd u_species(thermoMap->NumberOfSpecies());
thermoMap->uMolar_Species(u_species.data());

// Gibb's free energy (in J/kmol)
Eigen::VectorXd g_species(thermoMap->NumberOfSpecies());
thermoMap->gMolar_Species(g_species.data());

// Helmotz's free energy (in J/kmol)
Eigen::VectorXd a_species(thermoMap->NumberOfSpecies());
thermoMap->aMolar_Species(a_species.data());
```

Temperature from enthalpy

```
// Enthalpy (in J/kmol)
double h = 1e6;

// Temperature from enthalpy
const double Tguess = 300.;
const double T = thermoMap->GetTemperatureFromEnthalpyAndMoleFractions
                    (h, P, x.data(), Tguess);
```

Using different types of vectors

```
{  
    // STL vectors (from 0 to N-1)  
    std::vector<double> x(thermoMap->NumberOfSpecies());  
  
    // Molecular weight (in kg/kmol)  
    const double mw = thermoMap->MolecularWeight_From_MoleFractions(x.data());  
}  
  
{  
    // Native OpenSMOKE++ Vectors (from 1 to N)  
    OpenSMOKE::OpenSMOKEVectorDouble x(thermoMap->NumberOfSpecies());  
  
    // Molecular weight (in kg/kmol)  
    const double mw =  
        thermoMap->MolecularWeight_From_MoleFractions(x.GetHandle());  
}  
  
{  
    // Eigen C++ vectors (from 0 to N-1)  
    Eigen::VectorXd x(thermoMap->NumberOfSpecies());  
  
    // Molecular weight (in kg/kmol)  
    const double mw = thermoMap->MolecularWeight_From_MoleFractions(x.data());  
}
```


Importing the transport map

```
// OpenSMOKE++ Maps
OpenSMOKE::ThermodynamicsMap_CHEMKIN*      thermoMap;
OpenSMOKE::TransportPropertiesMap_CHEMKIN*   tranMap;

// Import map from preprocessed XML file
{
    boost::filesystem::path file_path =
        "kinetics-POLIMI_H2_1412/kinetics.xml";

    // Open XML file containing the thermodynamic data
    rapidxml::xml_document<> doc;
    std::vector<char> xml_string;
    OpenSMOKE::OpenInputFileXML(doc, xml_string, file_path);

    // Import
    thermoMap = new OpenSMOKE::ThermodynamicsMap_CHEMKIN(doc);
    tranMap = new OpenSMOKE::TransportPropertiesMap_CHEMKIN(doc);
}
```

The complete list of public functions is available here:

`Libraries/OpenSMOKEpp/maps/TransportPropertiesMap.h`

Mixture averaged transport properties

```
// Set maps
tranMap->SetTemperature(T);
tranMap->SetPressure(P);

// Dynamic viscosity (in kg/m/s)
const double eta = tranMap->DynamicViscosity(x.data());

// Thermal conductivity (in W/m/K)
const double lambda = tranMap->ThermalConductivity(x.data());

// Planck mean absorption coefficient (in 1/m)
const double kPlanck = tranMap->kPlanckMix(x.data());
```

Diffusion coefficients

```
// Molecular diffusion coefficients (in m2/s)
Eigen::VectorXd GammaMix(thermoMap->NumberOfSpecies());
tranMap->MassDiffusionCoefficients(GammaMix.data(), x.data());

// Thermal diffusion ratios (i.e. Soret effect)
Eigen::VectorXd TetaMix(thermoMap->NumberOfSpecies());
tranMap->ThermalDiffusionRatios(TetaMix.data(), x.data());

// Thermal diffusion coefficients (i.e. Soret effect) (in m2/s)
Eigen::VectorXd GammaSoretMix(thermoMap->NumberOfSpecies());
for (unsigned int i=0; i<thermoMap->NumberOfSpecies(); i++)
    GammaSoretMix(i) = GammaMix(i)*TetaMix(i)*thermoMap->MW(i)/mw;
```

Importing the kinetic map

```
// OpenSMOKE++ Maps
OpenSMOKE::ThermodynamicsMap_CHEMKIN*    thermoMap;
OpenSMOKE::KineticsMap_CHEMKIN*          kineticsMap;

// Import map from preprocessed XML file
{
    boost::filesystem::path file_path =
        "kinetics-POLIMI_H2_1412/kinetics.xml";

    // Open XML file containing the thermodynamic data
    rapidxml::xml_document<> doc;
    std::vector<char> xml_string;
    OpenSMOKE::OpenInputFileXML(doc, xml_string, file_path);

    // Import
    thermoMap = new OpenSMOKE::ThermodynamicsMap_CHEMKIN(doc);
    kineticsMap = new OpenSMOKE::KineticsMap_CHEMKIN(*thermoMap, doc);
}
```

The complete list of public functions is available here:

`Libraries/OpenSMOKEpp/maps/TransportPropertiesMap.h`

Reaction rates (net, forward, backward)

```
// Set maps
thermoMap->SetTemperature(T);
thermoMap->SetPressure(P);
kineticsMap->SetTemperature(T);
kineticsMap->SetPressure(P);

// Concentrations (in kmol/m3)
const double cTot = P/(PhysicalConstants::R_J_kmol*T);
Eigen::VectorXd c(thermoMap->NumberOfSpecies());
c = cTot*x;

// Reaction rates (in kmol/m3/s)
Eigen::VectorXd r(kineticsMap->NumberOfReactions());
kineticsMap->ReactionRates(c.data());
kineticsMap->GetReactionRates(r.data());

// Forward and backward reaction rates (in kmol/m3/s)
Eigen::VectorXd rf(kineticsMap->NumberOfReactions());
kineticsMap->GetForwardReactionRates(rf.data());
Eigen::VectorXd rb(kineticsMap->NumberOfReactions());
kineticsMap->GetBackwardReactionRates(rb.data());
```

Formation rates (net, production, destruction)

```
// Formation rates (in kmol/m3/s)
Eigen::VectorXd R(thermoMap->NumberOfSpecies());
kineticsMap->FormationRates(R.data());

// Production and destruction rates (in kmol/m3/s)
Eigen::VectorXd RP(thermoMap->NumberOfSpecies());
Eigen::VectorXd RD(thermoMap->NumberOfSpecies());
kineticsMap->ProductionAndDestructionRates(RP.data(), RD.data());

// Heat release (in J/m3/s)
const double Q = kineticsMap->HeatRelease(R.data());
```

Derivatives of formation rates

```
// Derivative of formation rates with respect to conc. (in 1/s)
const int NC = thermoMap->NumberOfSpecies();
Eigen::MatrixXd dR_over_dC(NC, NC);
kineticsMap->DerivativesOfFormationRates(c.data(), &dR_over_dC);
```

Outline

1. Presentation of OpenSMOKE++

1. Introduction
2. The OpenSMOKE++ Framework
3. The OpenSMOKE++ Suite
4. Coupling with OpenFOAM

2. Training

1. Introduction (environment preparation, organization, ...)
2. Preprocessing of thermodynamics, transport and kinetics
3. OpenSMOKE++ Maps for thermodynamics, transport and kinetics
- 4. OpenSMOKE++ for modeling a batch reactor**
5. A multidimensional laminar solver for reacting flows

Isothermal, constant volume batch reactor (I)

We want to write a solver for a single **isothermal, constant volume batch reactor**, by combining OpenSMOKE++ with OpenFOAM

$$\text{Governing equations} \quad \left\{ \begin{array}{l} \frac{dN_i}{dt} = R_i V \\ N_i(t = 0) = N_i^0 \end{array} \right.$$

N_i : number of moles of species i
 R_i : formation rate of species i
 V : reactor volume

Since the reactor volume is constant, the ODE system reported above can be rewritten in terms of concentrations ($N_i = C_i V$)

$$\left\{ \begin{array}{l} \frac{dC_i}{dt} = R_i \\ C_i(t = 0) = C_i^0 \end{array} \right.$$

Governing equations for a constant volume batch reactor

Isothermal, constant volume batch reactor (II)

Unless reactions occur without change of moles, total concentration of species and pressure changes in time. This can be described through the equation of state of ideal gases:

$$P = \frac{nRT}{V} = C_{tot}RT = RT \sum_{i=1}^N C_i$$

Or equivalently, in a differential form:

$$\left\{ \begin{array}{l} \frac{dP}{dt} = RT \sum_{i=1}^N \frac{dC_i}{dt} = RT \sum_{i=1}^N R_i \\ P(t = 0) = P_0 \end{array} \right.$$

Isothermal, constant volume batch reactor (III)

Thus, the isothermal, constant volume batch reactor is described by a system of non-linear ODEs with initial conditions + an algebraic (explicit) equation for pressure:

$$\left\{ \begin{array}{l} \frac{dC_i}{dt} = R_i \\ C_i(t = 0) = C_i^0 \end{array} \right. \quad + \quad P(t) = RT \sum_{i=1}^N C_i(t)$$

OpenFOAM

solution of ODE system by using the `ODESolver` and `ODESystem` classes

OpenSMOKE++

Thermodynamic and kinetic maps for calculating the formation rates for arbitrarily complex kinetic mechanism

Basic structure

1. Definition of a class corresponding to the equations describing the batch reactor (derived from `ODESystem` and based on `OpenSMOKE++ Maps`)

```
class batchIsothermalOdeSystem : public ODESystem
```

2. Creation of a `batchIsothermalOdeSystem` object

```
batchIsothermalOdeSystem batch(thermoMap, kineticsMap, T);
```

3. Creation of a `ODESolver` object

```
autoPtr<ODESolver> odeSolver = ODESolver::New(batch, dict);
```

4. Numerical solution of ODE system

```
odeSolver->solve(tStart, tStart + dt, cStart, dtStart);
```

ODESystem for isothermal batch reactor (I)

```
class batchIsothermalOdeSystem : public ODESystem
{

public:

    batchIsothermalOdeSystem(OpenSMOKE::ThermodynamicsMap_CHEMKIN& thermoMap,
                             OpenSMOKE::KineticsMap_CHEMKIN& kineticsMap, const double T);

    label nEqns() const;

    void derivatives( const scalar t, const scalarField& c, scalarField& dcdt )
    const;

    void jacobian( const scalar t, const scalarField& c, scalarField& dfdt,
                   scalarSquareMatrix& dfdc ) const;

private:

    OpenSMOKE::ThermodynamicsMap_CHEMKIN& thermoMap_;
    OpenSMOKE::KineticsMap_CHEMKIN& kineticsMap_;

    double T_;

};
```

Equations (isothermal conditions)

```
void batchIsothermalOdeSystem::derivatives
( const scalar t, const scalarField& c, scalarField& dcdt ) const
{
    // Calculate pressure
    const double cTot = std::accumulate(c.begin(), c.end(), 0.);
    const double P = cTot*(PhysicalConstants::R_J_kmol*T_);

    // Set maps
    thermoMap_.SetTemperature(T_);
    thermoMap_.SetPressure(P);
    kineticsMap_.SetTemperature(T_);
    kineticsMap_.SetPressure(P);

    // Calculates kinetics
    Eigen::VectorXd R(thermoMap_.NumberOfSpecies());
    kineticsMap_.ReactionRates(c.cdata());
    kineticsMap_.FormationRates(R.data());

    // Species equations
    for (unsigned int i=0; i<thermoMap_.NumberOfSpecies(); i++)
        dcdt[i] = R(i);
}
```

Jacobian matrix (isothermal conditions)

```
void batchIsothermalOdeSystem::jacobian( const scalar t, const scalarField&
c, scalarField& dfdt, scalarSquareMatrix& dfdc ) const
{
    // Calculate pressure
    const double cTot = std::accumulate(c.begin(), c.end(), 0.);
    const double P = cTot*(PhysicalConstants::R_J_kmol*T_);

    // Set maps
    thermoMap_.SetTemperature(T_);
    thermoMap_.SetPressure(P);
    kineticsMap_.SetTemperature(T_);
    kineticsMap_.SetPressure(P);

    // Derivative of formation rates with respect to conc (in 1/s)
    Eigen::MatrixXd dR_over_dC(NC, NC);
    kineticsMap_.DerivativesOfFormationRates(c.cdata(), &dR_over_dC);

    for (unsigned int i=0; i<thermoMap_.NumberOfSpecies(); i++)
        for (unsigned int j=0; j<thermoMap_.NumberOfSpecies(); j++)
            dfdc[i][j] = dR_over_dC(i,j);
}
```

Creating the ODE solver

```
// Create the ODE system as object of type batchOdeSystem
batchIsothermalOdeSystem batch(thermoMap, kineticsMap, T);

// Create dictionary and add the odeSolver name (from command line)
dictionary dict;
dict.add("solver", args[1]);

// Create the selected ODE system solver
autoPtr<ODESolver> odeSolver = ODESolver::New(batch, dict);
```


Solving the ODE system

```
// Integration loop
for (label i=0; i<n; i++)
{
    batch.derivatives(tStart, c, dc);
    odeSolver->solve(tStart, tStart + dt, c, dtStart);
    tStart += dt;

    // Reconstruct pressure
    const double cTot = std::accumulate(c.begin(), c.end(), 0.);
    P = cTot*(PhysicalConstants::R_J_kmol*T);
}
```

Compiling and running the batch reactor

Compilation

1. Go to the `Training/BatchReactor/01-isothermal/` folder
2. Type `wmake`

Execution

1. Type `batchReactorIsothermal <ODESolver>`
2. Look at the solution in the `Solution.out` file

A stiff ODE solver (such as `seulex` or `SIBS`) is recommended, even if the kinetic mechanism is small:

```
batchReactorIsothermal seulex
```

<ODESolver> options

```
Euler
EulerSI
RKCK45
RKDP45
RKF45
Rosenbrock12
Rosenbrock23
Rosenbrock34
SIBS
Trapezoid
rodas23
rodas34
seulex
```

Example: syngas

Initial conditions

$$T = 1000K$$

$$P = 101325 Pa$$

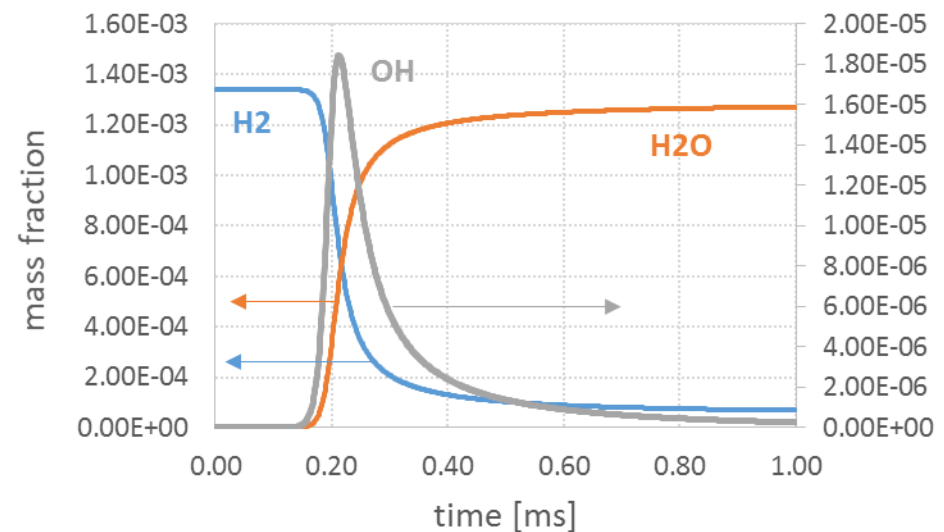
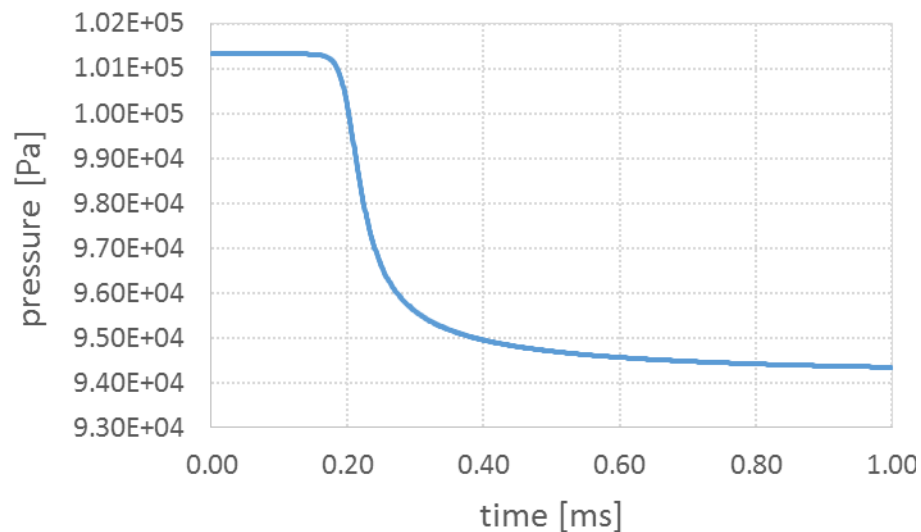
11% H₂ + 11% CO + 17% O₂ + 61% N₂

Kinetic mechanism

POLIMI_H2CO_1412

Species: 14

Reactions: 33



Example: methane

Initial conditions

$$T = 1200K$$

$$P = 101325 Pa$$

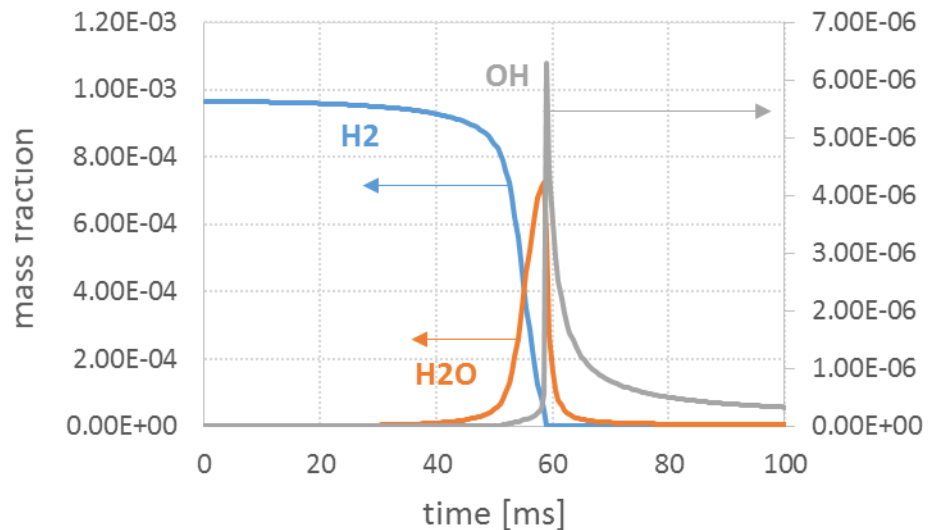
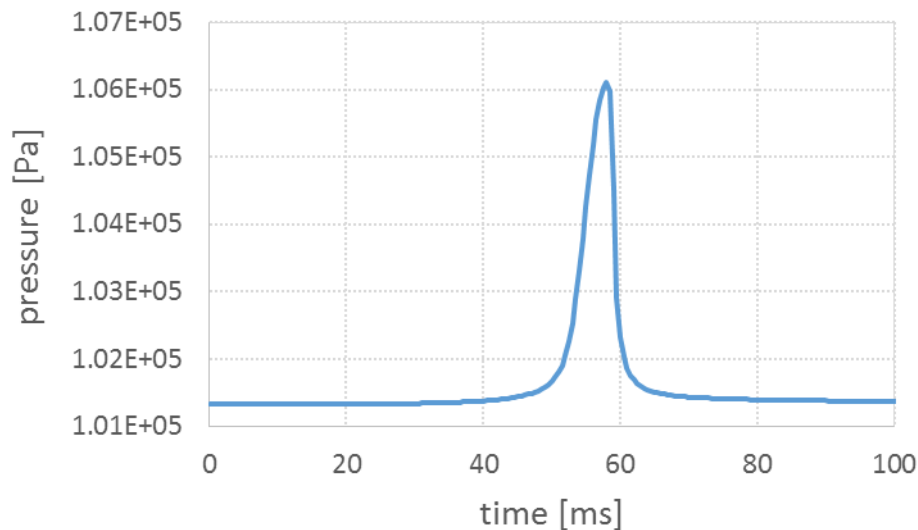
9.5% CH₄ + 19% O₂ + 71.5% N₂

Kinetic mechanism

GRI30

Species: 53

Reactions: 325



Adiabatic, constant volume batch reactor

In case of **adiabatic, constant volume batch reactor**, the equations governing the evolution of concentrations of species and pressure are the same we already developed in isothermal conditions:

$$\left\{ \begin{array}{l} \frac{dC_i}{dt} = R_i \\ C_i(t=0) = C_i^0 \end{array} \right. \quad + \quad P(t) = RT(t) \sum_{i=1}^N C_i(t)$$

The difference is now that temperature changes in time. Instead of writing an additional differential equation describing the evolution of temperature, we can, equivalently, impose that the total internal energy is constant (definition of adiabatic reactor):

$$U(t) = U_0$$

From the internal energy we can estimate the enthalpy and then, from pressure and composition we can always calculate the temperature (see the `GetTemperatureFromEnthalpyAndMoleFractions` function)

Equations (adiabatic conditions) (I)

```
void batchAdiabaticOdeSystem::derivatives
( const scalar t, const scalarField& cc, scalarField& dcdt ) const
{
    // Reconstruct concentrations and mole fractions
    Eigen::VectorXd c(thermoMap_.NumberOfSpecies());
    for (unsigned int i=0;i<thermoMap_.NumberOfSpecies();i++)
        c(i) = std::max(cc[i],0.);
    Eigen::VectorXd x(thermoMap_.NumberOfSpecies());
    const double cTot = c.sum();
    for (unsigned int i=0;i<thermoMap_.NumberOfSpecies();i++)
        x(i) = c[i]/cTot;
    const double mw = thermoMap_.MolecularWeight_From_MoleFractions(x.data());

    ...
}
```

It is extremely important to clean the current concentrations, by removing negative values

Equations (adiabatic conditions) (II)

```
void batchAdiabaticOdeSystem::derivatives
( const scalar t, const scalarField& cc, scalarField& dcdt ) const
{
    ...

    // Calculate temperature and pressure (by successive substitutions)
    double P = PInitial_;
    double T = TInitial_;
    for(int i=0;i<maxIterations_;i++)
    {
        const double Pold = P;
        const double H = U_+P/(cTot*mw);
        T = thermoMap_.GetTemperatureFromEnthalpyAndMoleFractions
            (H*mw, P, x.data(), T);
        P = cTot*(PhysicalConstants::R_J_kmol*T);
        if (std::fabs(P-Pold)/P<1e-4) break;
    }
    ...
}
```

Internal energy is constant (by definition of adiabatic constant volume batch reactor). In order to get the temperature, an **iterative procedure** is needed, since the reactor pressure is unknown

Equations (adiabatic conditions) (III)

```
void batchAdiabaticOdeSystem::derivatives
( const scalar t, const scalarField& cc, scalarField& dcdt ) const
{
    ...

    // Set maps
    thermoMap_.SetTemperature(T);
    thermoMap_.SetPressure(P);
    kineticsMap_.SetTemperature(T);
    kineticsMap_.SetPressure(P);

    // Calculates kinetics
    Eigen::VectorXd R(thermoMap_.NumberOfSpecies());
    kineticsMap_.ReactionRates(c.data());
    kineticsMap_.FormationRates(R.data());

    // Species equations
    for (unsigned int i=0;i<thermoMap_.NumberOfSpecies();i++)
        dcdt[i] = R(i);
}
```


Jacobian matrix (adiabatic conditions)

```
void batchAdiabaticOdeSystem::jacobian( const scalar t, const scalarField& c,
scalarField& dfdt, scalarSquareMatrix& dfdc ) const
{
    // Reconstruct concentrations and mole fractions
    [Same code: derivatives]

    // Calculate temperature and pressure (by successive substitutions)
    [Same code: derivatives]

    // Set maps
    thermoMap_.SetTemperature(T);
    thermoMap_.SetPressure(P);
    kineticsMap_.SetTemperature(T);
    kineticsMap_.SetPressure(P);

    // Derivative of formation rates with respect to conc, (in 1/s)
    Eigen::MatrixXd dR_over_dC(NC, NC);
    kineticsMap_.DerivativesOfFormationRates(c.data(), &dR_over_dC);

    for (unsigned int i=0; i<thermoMap_.NumberOfSpecies(); i++)
        for (unsigned int j=0; j<thermoMap_.NumberOfSpecies(); j++)
            dfdc[i][j] = dR_over_dC(i,j);
}
```

Compiling and running the batch reactor

Compilation

1. Go to the Training/BatchReactor/02-adiabatic/ folder
2. Type `wmake`

Execution

1. Type `batchReactorAdiabatic <ODESolver>`
2. Look at the solution in the `Solution.out` file

A stiff ODE solver (such as `seulex` or `SIBS`) is recommended, even if the kinetic mechanism is small:

```
batchReactorAdiabatic seulex
```

<ODESolver> options

```
Euler
EulerSI
RKCK45
RKDP45
RKF45
Rosenbrock12
Rosenbrock23
Rosenbrock34
SIBS
Trapezoid
rodas23
rodas34
seulex
```

Example: syngas

Initial conditions

$$T = 1000K$$

$$P = 101325 Pa$$

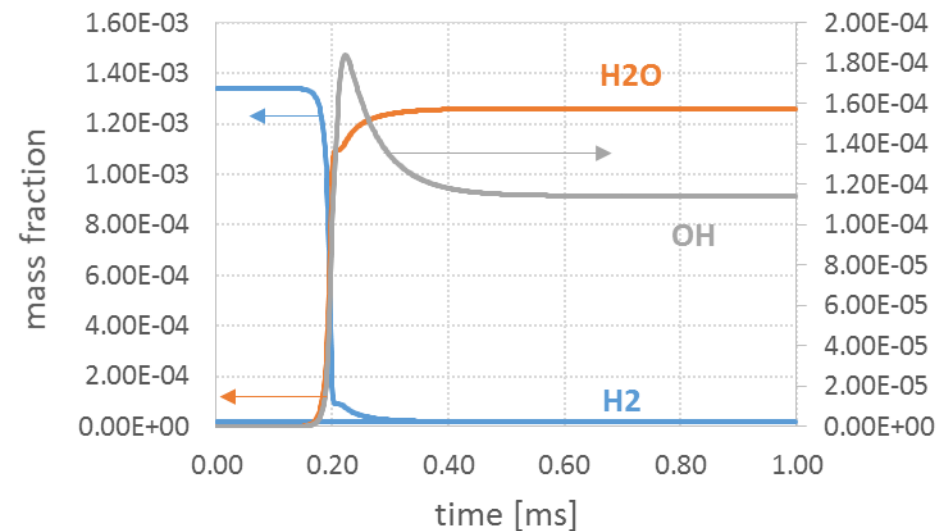
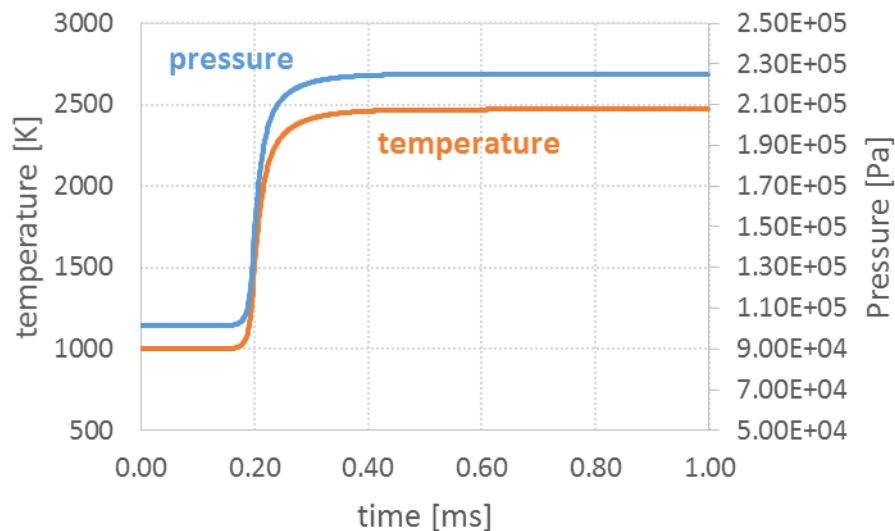
11% H₂ + 11% CO + 17% O₂ + 61% N₂

Kinetic mechanism

POLIMI_H2CO_1412

Species: 14

Reactions: 33



Example: methane

Initial conditions

$$T = 1400K$$

$$P = 101325 Pa$$

$$9.5\% CH_4 + 19\% O_2 + 71.5\% N_2$$

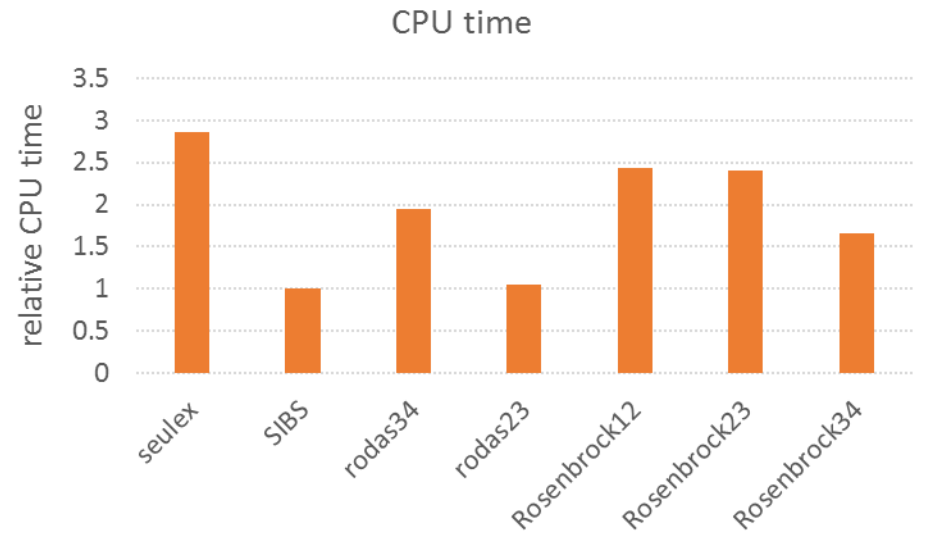
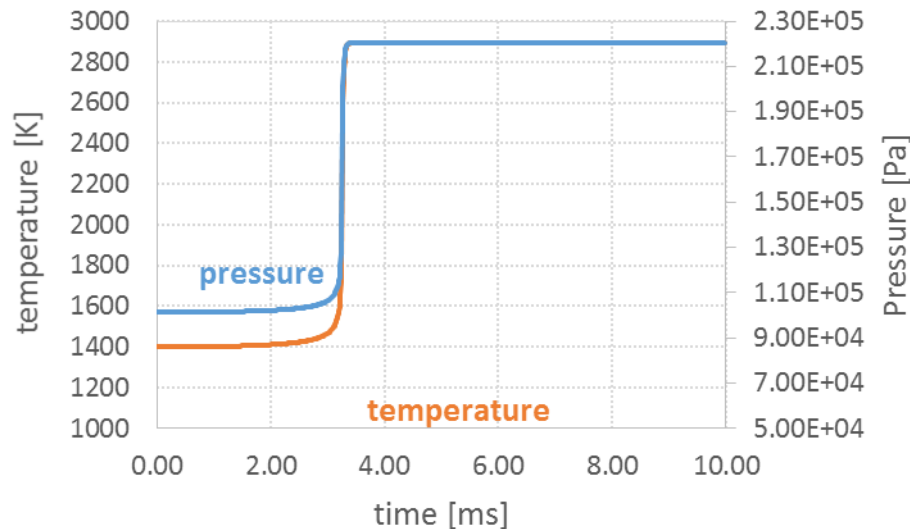
$$t=0.01 s$$

Kinetic mechanism

GRI30 (without NOX)

Species: 36

Reactions: 219



Outline

1. Presentation of OpenSMOKE++

1. Introduction
2. The OpenSMOKE++ Framework
3. The OpenSMOKE++ Suite
4. Coupling with OpenFOAM

2. Training

1. Introduction (environment preparation, organization, ...)
2. Preprocessing of thermodynamics, transport and kinetics
3. OpenSMOKE++ Maps for thermodynamics, transport and kinetics
4. OpenSMOKE++ for modeling a batch reactor
- 5. A multidimensional laminar solver for reacting flows**

Introduction

We want to build a couple of solvers (steady-state and unsteady) for homogeneous laminar reacting flows described by arbitrarily complex kinetic mechanisms

OpenFOAM

Input/Output, mesh, spatial and temporal discretization of transport equations, solution of linear systems, solution of ODE systems

OpenSMOKE++

Thermodynamic and transport properties, chemical reactions

Outline

1. Unsteady solver based on the operator splitting algorithm

1. Introduction and theory
2. Implementation
3. Examples

2. Steady state solver based on the linearization of source terms

1. Introduction and theory
2. Implementation
3. Examples

Outline

1. Unsteady solver based on the operator splitting algorithm

1. Introduction and theory
2. Implementation
3. Examples

2. Steady state solver based on the linearization of source terms

1. Introduction and theory
2. Implementation
3. Examples

Unsteady solver: equations

Continuity equation

$$\frac{\partial \rho}{\partial t} + \nabla(\rho \mathbf{v}) = 0$$

Momentum equations

$$\frac{\partial}{\partial t}(\rho \mathbf{v}) + \nabla(\rho \mathbf{v} \mathbf{v}) = -\nabla p + \nabla \boldsymbol{\tau} + \rho \mathbf{g}$$

Energy equation

$$\rho C_P \frac{\partial T}{\partial t} + \rho C_P \mathbf{v} \nabla T = \lambda \nabla^2 T + Q_R$$

Species equations

$$\frac{\partial}{\partial t}(\rho Y_k) + \nabla(\rho \mathbf{v} Y_k) = \nabla(\rho \Gamma_k \nabla Y_k) + R_k \quad k = 1, \dots, N$$

Simplifying hypotheses

- No radiative heat transfer
- No enthalpy fluxes due to preferential mass diffusion
- No thermodiffusion (Soret effect)
- No correction velocity on mass diffusion fluxes

Unsteady solver: algorithm

Continuity
equation

$$\frac{\partial \rho}{\partial t} + \nabla(\rho \mathbf{v}) = 0$$

Momentum
equations

$$\frac{\partial}{\partial t}(\rho \mathbf{v}) + \nabla(\rho \mathbf{v} \mathbf{v}) = -\nabla p + \nabla \boldsymbol{\tau} + \rho \mathbf{g}$$

**PIMPLE
Algorithm**

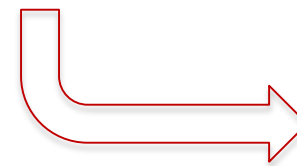
Energy equation

$$\rho C_P \frac{\partial T}{\partial t} + \rho C_P \mathbf{v} \nabla T = \lambda \nabla^2 T + Q_R$$

Strongly non
linear terms

Species equations

$$\frac{\partial}{\partial t}(\rho Y_k) + \nabla(\rho \mathbf{v} Y_k) = \nabla(\rho \Gamma_k \nabla Y_k) + R_k \quad k = 1, \dots, N$$



Operator-splitting
approach

Operator-splitting (I)

Energy equation $\rho C_P \frac{\partial T}{\partial t} = -\rho C_P \mathbf{v} \nabla T + \lambda \nabla^2 T + Q_R$

Species equations $\frac{\partial}{\partial t} (\rho Y_k) = -\nabla (\rho \mathbf{v} Y_k) + \nabla (\rho \Gamma_k \nabla Y_k) + R_k \quad k = 1, \dots, N$

T = transport processes (convection and diffusion), **weakly non linear**, non-local

S = non linear, stiff processes (homogeneous reactions), **local**

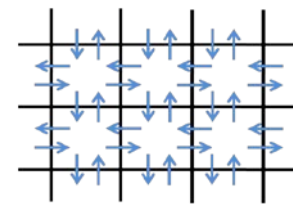
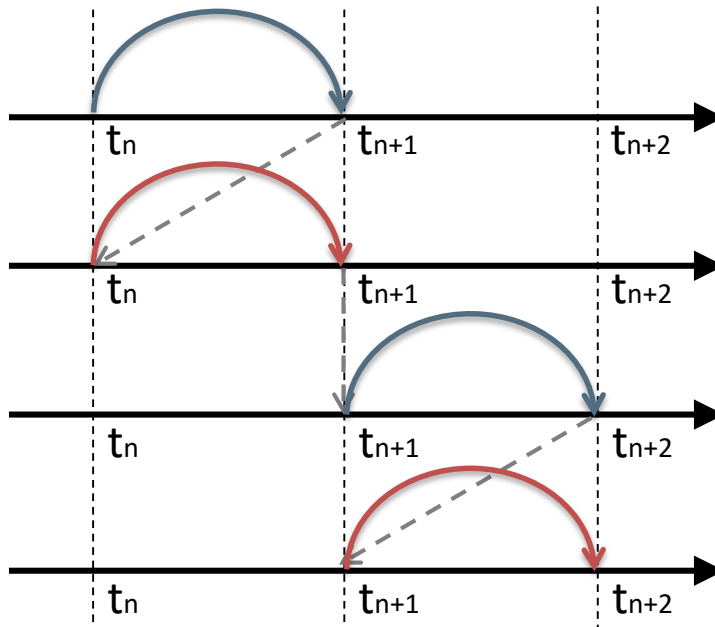
Energy equation $\rho C_P \frac{\partial T}{\partial t} = T_T + S_T$

Species equations $\frac{\partial}{\partial t} (\rho Y_k) = T_k + S_k \quad k = 1, \dots, N$

Operator-splitting (II)

Non linear transport equations for species and energy

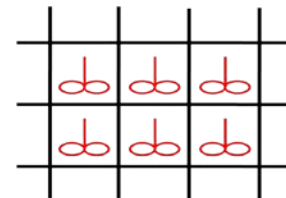
$$\frac{\partial \phi}{\partial t} = T + S$$



Transport

$$\frac{\partial \phi}{\partial t} = T$$

Chemistry



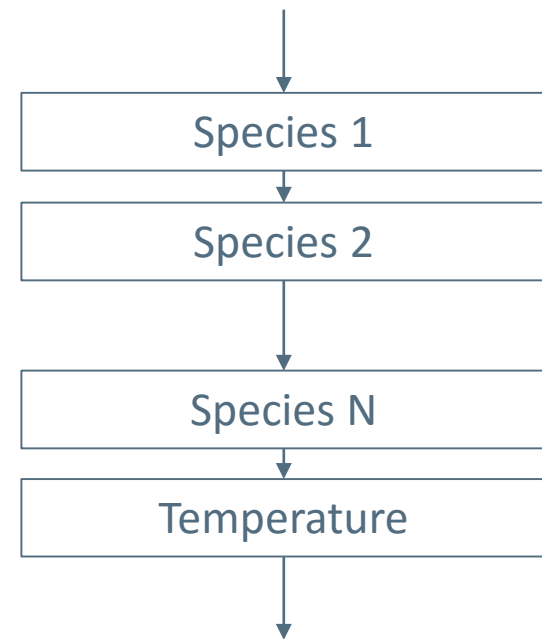
$$\frac{\partial \phi}{\partial t} = S$$

Transport step

Species equations
$$\frac{\partial}{\partial t}(\rho Y_k) + \nabla(\rho \mathbf{v} Y_k) = \nabla(\rho \Gamma_k \nabla Y_k) \quad k = 1, \dots, N$$

Energy equation
$$\rho C_P \frac{\partial T}{\partial t} + \rho C_P \mathbf{v} \nabla T = \lambda \nabla^2 T$$

- The equations are weakly non linear
- The only source of non-linearity is represented by transport and thermodynamic properties, which are function of temperature and composition
- A segregated approach is in general feasible



Chemical step

Species equations $\frac{\partial}{\partial t}(\rho Y_k) = R_k \quad k = 1, \dots, N$

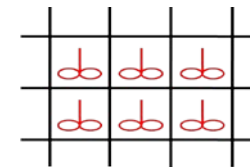
Energy equation $\rho C_P \frac{\partial T}{\partial t} = Q_R$

These equations are **LOCAL**,
i.e. no spatial discretization
operators are present

- Since there are no transport, the total mass remains constant
- The volume of a cell is constant in time
- Thus, the density is constant during the chemical step

Chemical step
equations
$$\begin{cases} \frac{\partial Y_k}{\partial t} = \frac{R_k}{\rho} \\ \frac{\partial T}{\partial t} = \frac{Q_R}{\rho C_P} \end{cases} \quad k = 1, \dots, N$$

Each cell behaves like a
batch reactor



Outline

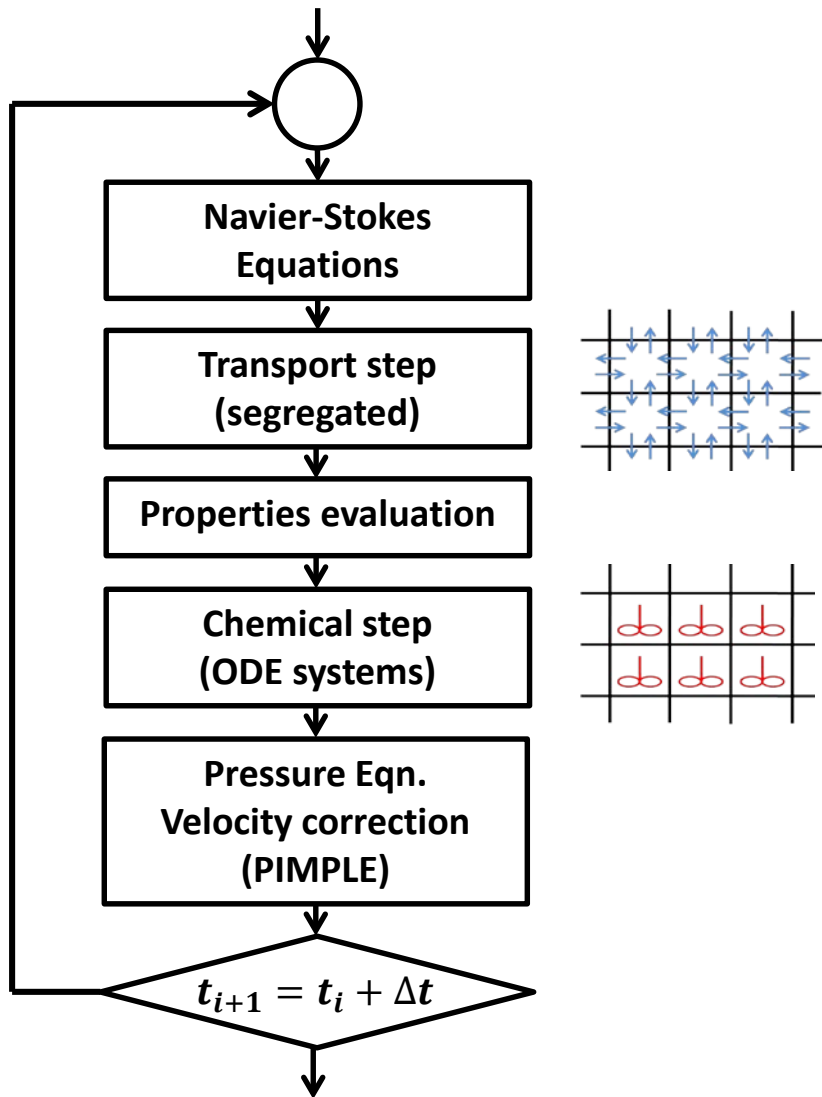
1. Unsteady solver based on the operator splitting algorithm

1. Introduction and theory
- 2. Implementation**
3. Examples

2. Steady state solver based on the linearization of source terms

1. Introduction and theory
2. Implementation
3. Examples

Unsteady solver (I)



$$\frac{\partial \varphi}{\partial t} = T(\varphi) + S(\varphi)$$

φ : Dependent variables (Y_k and T)

$T(\varphi)$ is the rate of change of φ due to transport
 $S(\varphi)$ is the rate of change of φ due to reactions

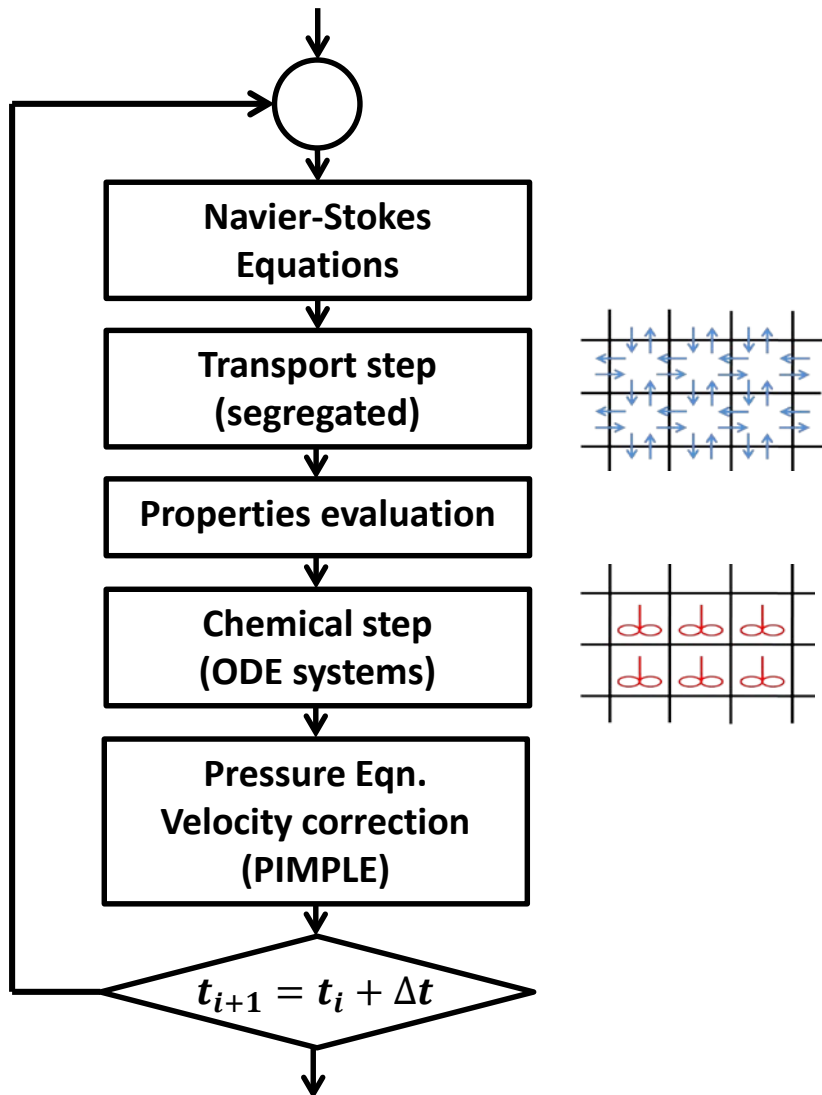
Sub-step 1. The transport terms (convection and diffusion) are integrated over Δt by solving:

$$\frac{\partial \varphi^{(a)}}{\partial t} = T(\varphi^{(a)})$$

Sub-step 2. The reaction terms are integrated over Δt through the solution of N_{cells} independent stiff ODE systems:

$$\frac{\partial \varphi^{(b)}}{\partial t} = S(\varphi^{(b)})$$

Unsteady solver (II)



```
while (pimple.loop())
{
    // Transport
    #include "UEqn.H"
    #include "YEqn.H"
    #include "TEqn.H"

    // Chemistry
    #include "chemistry.H"

    // Update transport properties
    #include "transportProperties.H"

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

Unsteady solver initialization

```
int main(int argc, char *argv[])
{
    // OpenFOAM stuff
    #include "setRootCase.H"
    #include "createTime.H"
    #include "createMesh.H"
    #include "createControl.H"
    #include "createTimeControls.H"
    #include "initContinuityErrs.H"
    #include "createMRF.H"
    #include "createFvOptions.H"
    #include "createBasicFields.H"

    // OpenSMOKE++
    #include "createChemicalFields.H"
    #include "transportProperties.H"

    // OpenFOAM stuff
    #include "createAdditionalFields.H"
    #include "compressibleCourantNo.H"
    #include "setInitialDeltaT.H"
    ...
}
```

Importing OpenSMOKE++ Maps

```
const dictionary& kineticsDictionary = solverOptions.subDict("Kinetics");
Foam::string kinetics_folder = kineticsDictionary.lookup("folder");

// Open XML file containing the thermodynamic data
rapidxml::xml_document<> doc;
std::vector<char> xml_string;
OpenSMOKE::OpenInputFileXML(doc, xml_string, kinetics_folder + "/kinetics.xml");

OpenSMOKE::ThermodynamicsMap_CHEMKIN      thermoMap(doc);
OpenSMOKE::KineticsMap_CHEMKIN            kineticsMap(thermoMap, doc);
OpenSMOKE::TransportPropertiesMap_CHEMKIN  transportMap(doc);

// Inert species
word inertSpecies(kineticsDictionary.lookup("inertSpecies"));
label inertIndex = thermoMap.IndexOfSpecies(inertSpecies)-1;

// ODE solver
word odeSolver(kineticsDictionary.lookup("ODESolver"));
```

Transport properties

```
forAll(TC, celli)
{
    // Set maps
    thermoMap.SetPressure(pC[celli]);
    thermoMap.SetTemperature(TC[celli]);
    transportMap.SetPressure(pC[celli]);
    transportMap.SetTemperature(TC[celli]);

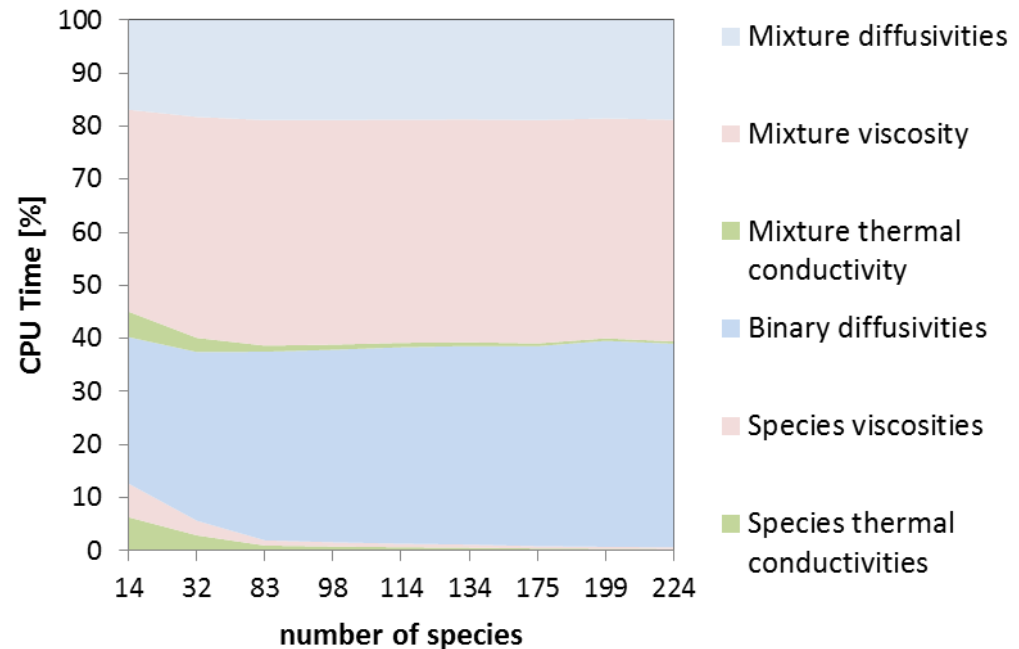
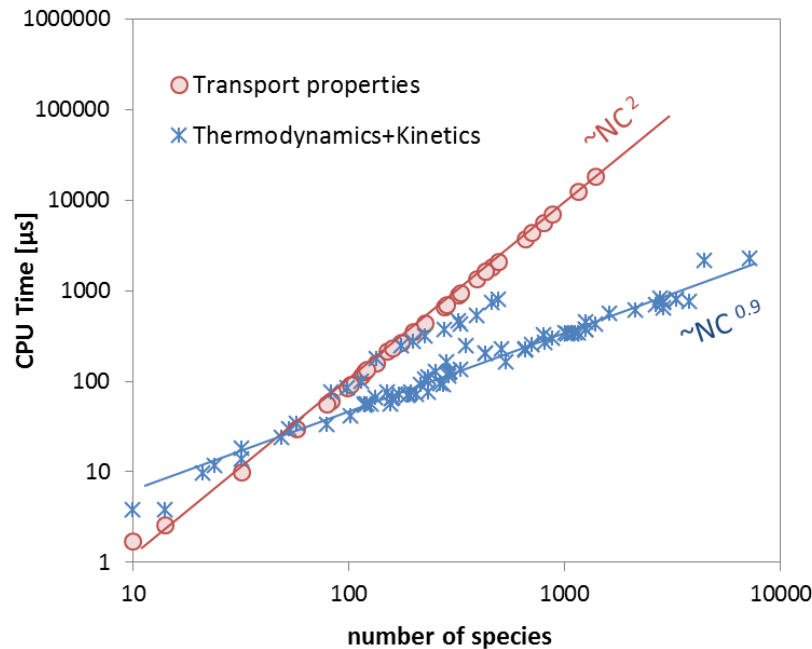
    // Constant pressure specific heat [J/kg/K]
    cpC[celli] = thermoMap.cpMolar_Mixture_From_MoleFractions(x.data())/mw;

    // Dynamic viscosity [kg/m/s]
    etaCells[celli] = transportMap.DynamicViscosity(x.data());

    // Thermal conductivity [W/m/K]
    lambdaCells[celli] = transportMap.ThermalConductivity(x.data());

    // Diffusion coefficients [m2/s]
    transportMap.MassDiffusionCoefficients(GammaMixVector.data(), x.data());
    for(int i=0;i<thermoMap.NumberOfSpecies();i++)
        GammaMix[i].ref()[celli] = GammaMixVector(i);
}
```

Computational cost of transport properties



- The cost of evaluation of transport properties (in particular diffusion) increases **quadratically** with the number of species
- For large mechanisms (>100 species) the computational cost of transport properties is not negligible
- In fully-coupled methods proper techniques must be applied to **reduce the computational cost of transport properties**

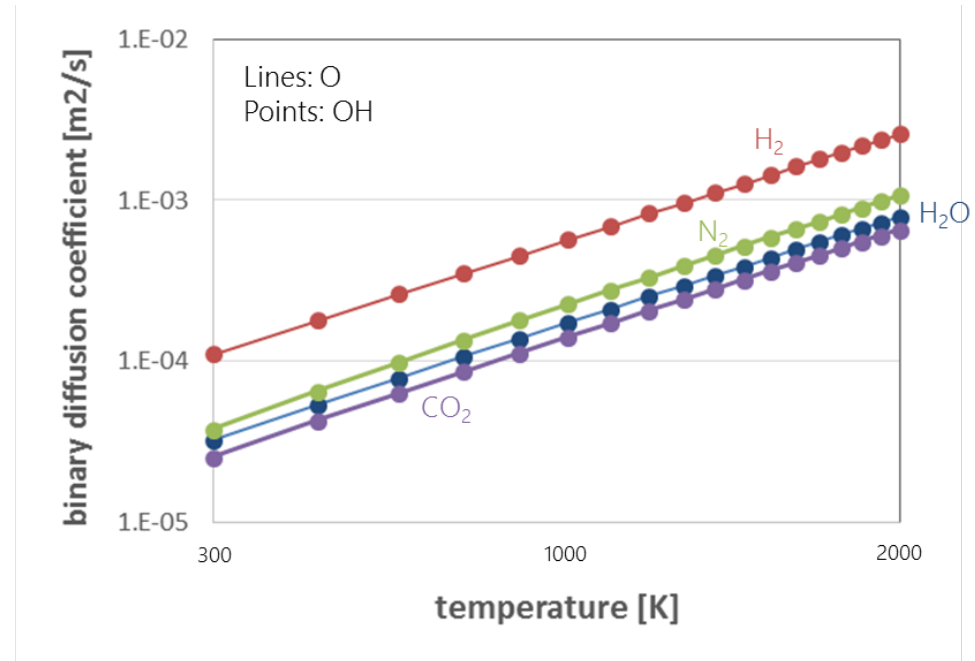
Species bundling

Many species possess similar diffusivities because of similar molecular properties (molecular weight, structure, collision cross section, etc.)

Such species are expected to behave similarly in terms of diffusive transport



Species with similar diffusivities can be bundled in a same group with a representative species



The diffusivities of O and OH with other species are almost identical

Lu, Law, *Diffusion coefficient reduction through species bundling*,
Combustion and Flame, 148, p. 117-126 (2007)

Species bundling in OpenSMOKE++

Step 1: pre-processing with species bundling

```
Dictionary CHEMKIN_PreProcessor
{
    @Thermodynamics      POLIMI_TOT_NOX_1412.CKT;
    @Transport            POLIMI_TOT_NOX_1412.TRC;
    @Kinetics             POLIMI_PRF_PAH_HT_1412.CKI;
    @SpeciesBundling      true;
    @Output               kinetics-POLIMI_PRF_PAH_HT_1412;
}
```

Step 2: importing the transport map with species bundling

```
tranMap = new OpenSMOKE::TransportPropertiesMap_CHEMKIN(doc);
tranMap->ImportSpeciesBundlingFromXMLFile(doc, bundling_eps);
```

Step 3: evaluation of diffusion coefficients through species bundling

```
tranMap.MassDiffusionCoefficients(GammaMixVector.data(), x.data(), true);
for(int i=0;i<thermoMap.NumberOfSpecies();i++)
    GammaMix[i].ref()[celli] = GammaMixVector(i);
```

chemistry.H (I)

- Each cell behaves like a constant volume batch reactor, which is described by a set of ODE with initial conditions
- There is no need to create a ODESystem object and a ODESolver object for each cell. We can reuse the same for all the cells

```
// Create dictionary and add the odeSolver name
dictionary dict;
dict.add("solver", odeSolver);

// Create the ODE system as object of type batchOdeSystem
batchAdiabaticOdeSystem batch(thermoMap, kineticsMap);

// Create the selected ODE system solver
autoPtr<ODESolver> odeSolver = ODESolver::New(batch, dict);
```


chemistry.H (II)

```
forAll(TC, celli)
{
    ...
```

We imagine that enthalpy is kept constant during the chemical step

```
// Enthalpy
thermoMap.SetTemperature(TC[celli]); thermoMap.SetPressure(pC[celli]);
const double H = thermoMap.hMolar_Mixture_From_MoleFractions(x.data())/mw;
batch.setEnthalpy(H);
```

```
// Solve ODE system
batch.setTemperature(TC[celli]); batch.setPressure(pC[celli]);
batch.derivatives(tStart, c, dcStart);
odeSolver->solve(tStart, tEnd, c, dtStart);
```

```
// From concentrations to mass fractions
cTot = std::accumulate(c.begin(), c.end(), 0.);
for(unsigned int i=0;i<NC;i++) x(i) = c[i]/cTot;
thermoMap.MassFractions_From_MoleFractions(x.data(), mw, y.data());
for(unsigned int i=0;i<NC;i++)
    Y[i].ref()[celli] = massFractions(i);
```

```
// Temperature
TC[celli] = thermoMap.GetTemperatureFromEnthalpyAndMoleFractions
    (H*mw, pC[celli], x.data(), TC[celli]);
```

```
}
```

Batch reactor (adiabatic) (I)

```
void batchAdiabaticOdeSystem::derivatives( const scalar t, const
                                           scalarField& cc, scalarField& dcdt ) const
{
    // Reconstruct concentrations and mole fractions
    Eigen::VectorXd c(thermoMap_.NumberOfSpecies());
    for (unsigned int i=0;i<thermoMap_.NumberOfSpecies();i++)
        c(i) = std::max(cc[i],0.);

    Eigen::VectorXd x(thermoMap_.NumberOfSpecies());
    const double cTot = c.sum();
    for (unsigned int i=0;i<thermoMap_.NumberOfSpecies();i++)
        x(i) = c[i]/cTot;
    const double mw = thermoMap_.MolecularWeight_From_MoleFractions(x.data());

    // Calculate temperature (pressure is assumed constant)
    const double T_ = thermoMap_.GetTemperatureFromEnthalpyAndMoleFractions
        (Hfixed_*mw, P0_, x.data(), TStart_);

    ...
}
```

Batch reactor (adiabatic)

```
void batchAdiabaticOdeSystem::derivatives( const scalar t, const
                                           scalarField& cc, scalarField& dcdt ) const
{
    ...

    // Calculates thermodynamic properties
    thermoMap_.SetTemperature(T_);
    thermoMap_.SetPressure(P0_);

    // Calculates kinetics
    Eigen::VectorXd R(thermoMap_.NumberOfSpecies());
    kineticsMap_.SetTemperature(T_);
    kineticsMap_.SetPressure(P0_);
    kineticsMap_.ReactionRates(c.data());
    kineticsMap_.FormationRates(R.data());

    // Species equations
    for (unsigned int i=0;i<thermoMap_.NumberOfSpecies();i++)
        dcdt[i] = R(i);
}
```

Compiling the unsteady solver

Compilation

1. Go to the `Training/Solvers/laminarSolverUnsteady/src` folder
2. Type `wmake`

Example of a coflow flame

1. Go to the following folder:
`Training/Solvers/laminarSolverUnsteady/run/coflowFlame/01-global-1step`
2. Type `blockMesh`
3. Type `laminarSolverUnsteady`

Outline

1. Unsteady solver based on the operator splitting algorithm

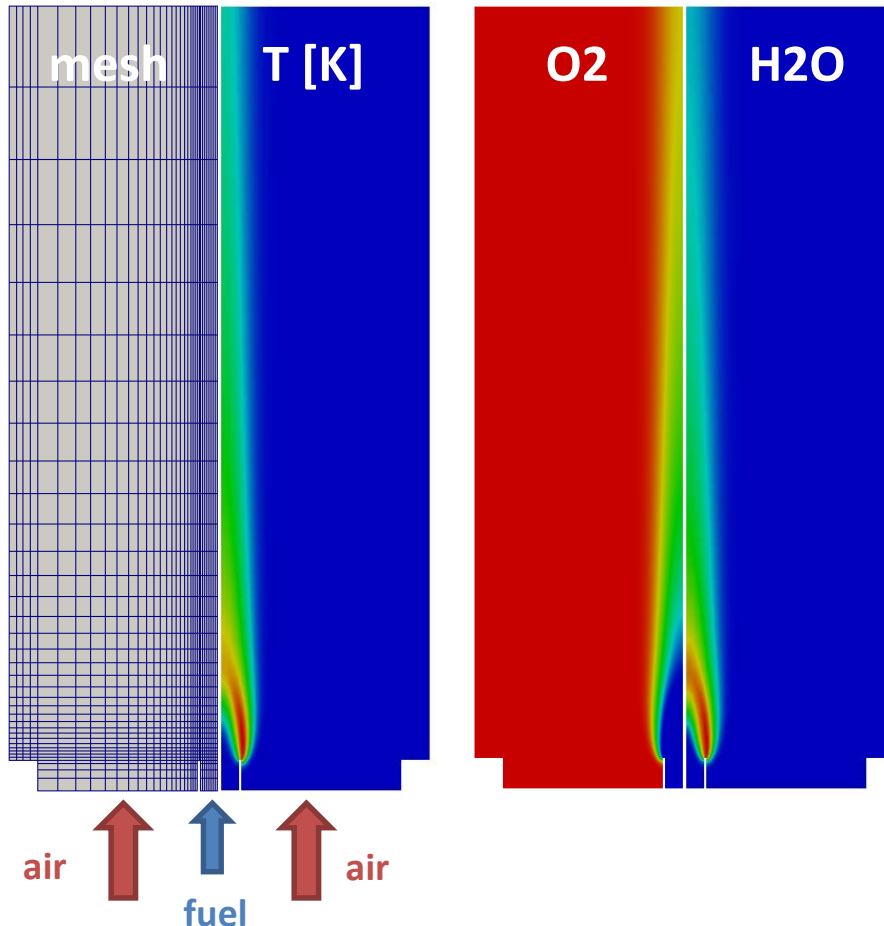
1. Introduction and theory
2. Implementation
- 3. Examples**

2. Steady state solver based on the linearization of source terms

1. Introduction and theory
2. Implementation
3. Examples

Example: laminar coflow flame

V. V. Toro, A. V. Mokhov, H. B. Levinsky, M. D. Smooke, Proceedings of the Combustion Institute, 30 (2005), 485-492

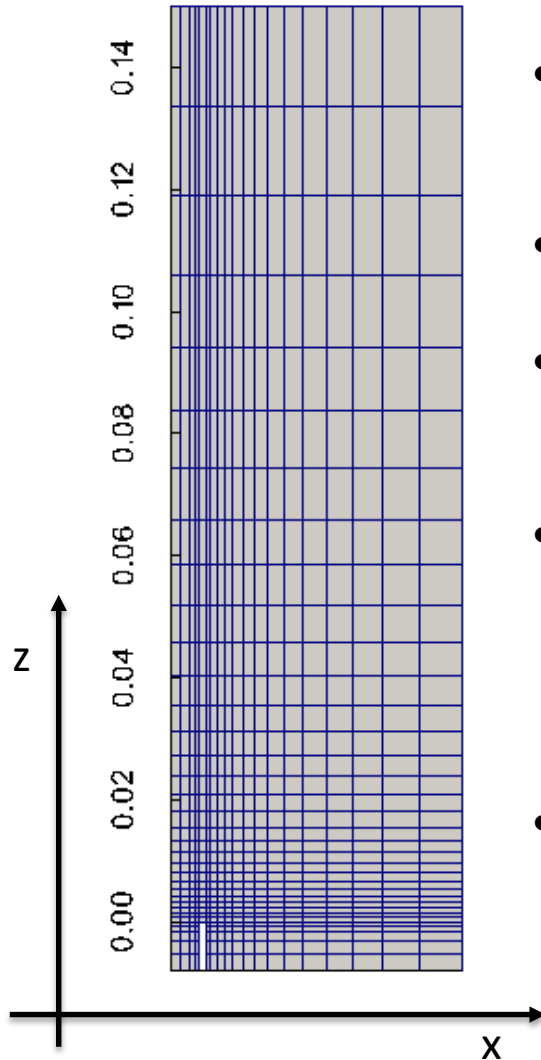


The fuel stream (**50% H₂ and 50% N₂ by volume**) is injected at ambient temperature through a circular nozzle (**i.d. 9 mm**), surrounded by an air-coflow annulus (**i.d. 95 mm**).

The fuel and coflow inlet velocities are assumed equal to **50 cm/s** (**Flame F3**).

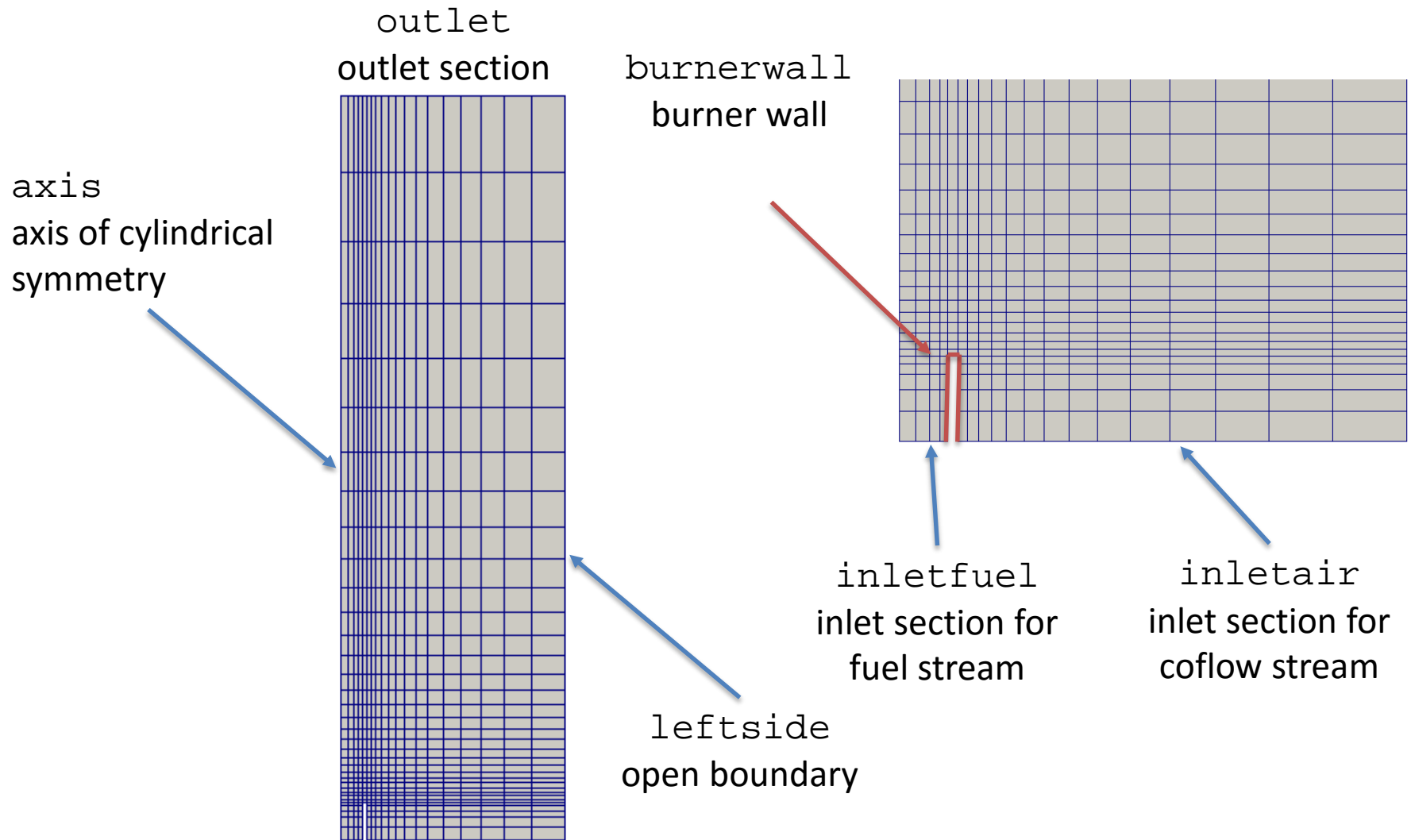
A **2D rectangular domain** meshed with a structured grid is considered.

Computational mesh



- A axisymmetric mesh is chosen, due to the cylindrical symmetry of the system
- The mesh is built using the `blockMeshDict`
- The length is 150 mm (z direction) and the width is 47.5 mm (x direction)
- A very coarse mesh was chosen (with 35 cells along the axis and 19 cells along the radial direction) in order to reduce the computational time as much as possible during the training session
- You should not use this mesh for your production simulations because is too coarse

Boundaries



Global kinetic mechanism

Our preliminary calculations are carried out using a global 1 step mechanism, available in the PreProcessing/Global_H2_1step folder

```
ELEMENTS
```

```
H O N
```

```
END
```

```
SPECIES
```

```
H2 O2 H2O N2
```

```
END
```

```
REACTIONS
```

```
H2 + 0.5O2 => H2O 1e14 0 20000
```

```
FORD / O2 1.00 /
```

```
END
```

Warning!

This global 1-step mechanism is used in the training session simply to reduce the computational time. Do not use it for your production simulations

The kinetic mechanism has to be pre-processed (see previous sections) using the OpenSMOKEpp_CHEMKIN_PreProcessor

1. Go to the PreProcessing/Global_H2_1step folder
2. Type `OpenSMOKEpp_CHEMKIN_PreProcessor --input input.dic`

Setup the case

The kinetic mechanism to be used is specified in the `constant/solverOptions` file

```
Kinetics
{
    folder            "../.../PreProcessing/Global_H2_1step/";
    inertSpecies      N2;
    ODESolver         seulex;
}
```

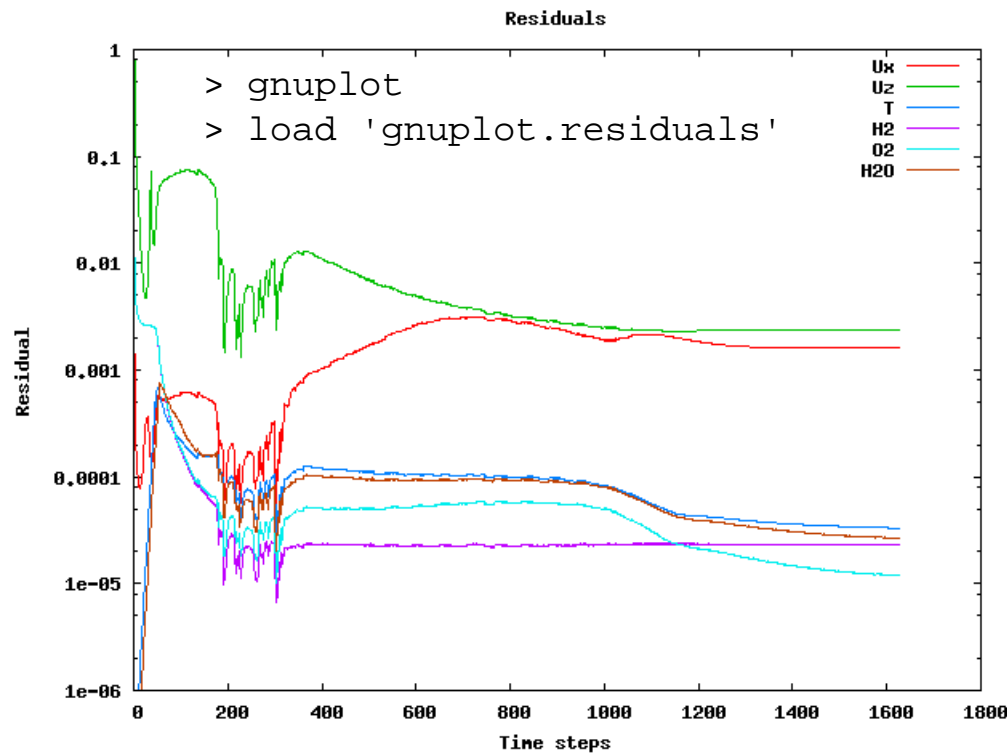
Since both the fuel and oxidizer streams are fed at ambient temperature, it is necessary to introduce a **spark** in order to ignite the mixture.

```
Spark
{
    spark             on;
    position           (5.95e-3  0.0  1.5e-3);
    time              0.;
    temperature        2200;
    duration           0.025;
    diameter           1.5e-3;
}
```

In our solver the spark is nothing but a (circular) region in which the temperature is kept fixed at a high value for a certain amount of time

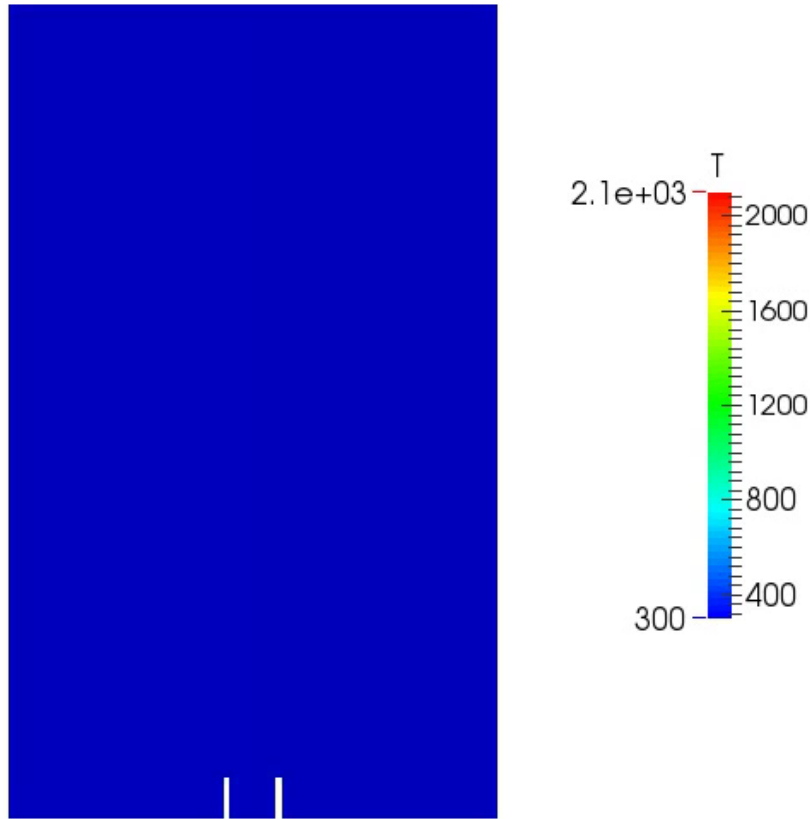
Running the case

1. Go to the following folder:
Training/Solvers/laminarSolverUnsteady/run/coflowFlame/01-global-1step
2. Type `blockMesh`
3. Type `laminarSolverUnsteady > log &`
4. The simulation takes about 5 min on a single core

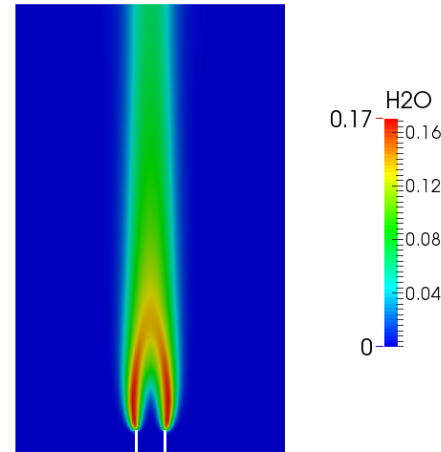


Results

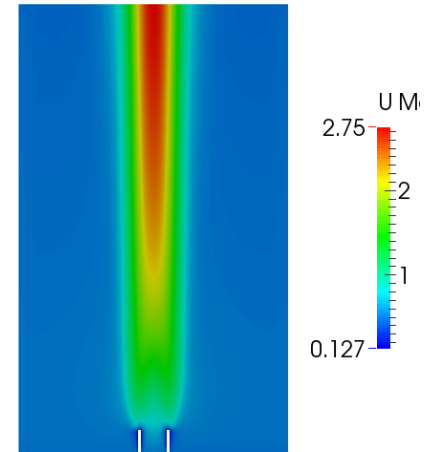
Temperature field
(evolution from 0 to 0.25 s)



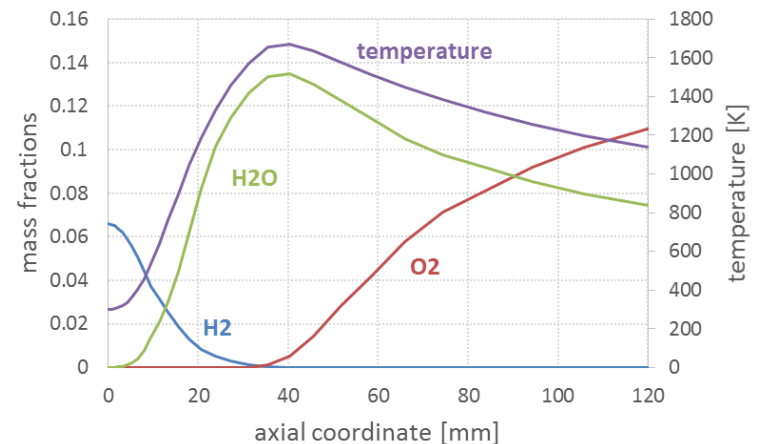
H2O mass fraction



U magnitude



> postProcess -func sample



Refining the solution using a detailed mech

Mechanism preprocessing

We want now refine the solution by using the detailed kinetic mechanism contained in the PreProcessing/POLIMI_H2_1412:

1. Go to the PreProcessing/POLIMI_H2_1412 folder
2. Type `OpenSMOKEpp_CHEMKIN_PreProcessor --input input.dic`

Restarting from previous solution

We restart from the last solution (0.25 s) corresponding to the global, 1-step mechanism

1. Go to the following folder:
`Training/Solvers/laminarSolverUnsteady/run/coflowFlame/02-detailed-polimi`
2. Copy the last solution:
`cp -r ../01-global-1step/0.25/ .`
`cp ../01-global-1step/0/Ydefault 0.25/`
`rm -r 0.25/uniform`

Running the simulation

1. Go to the following folder:

`Training/Solvers/laminarSolverUnsteady/run/coflowFlame/02-detailed-polimi`

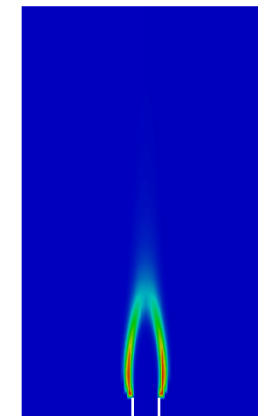
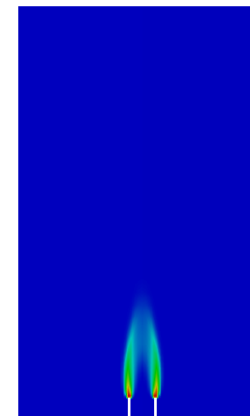
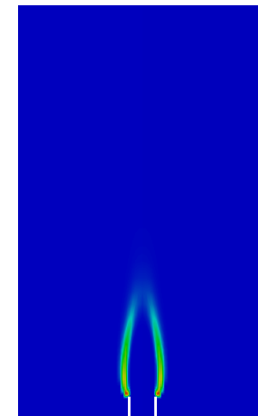
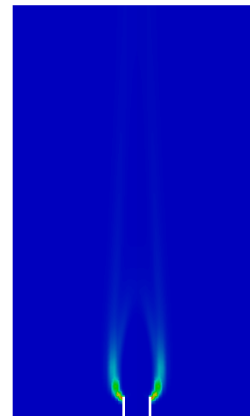
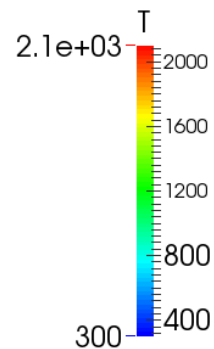
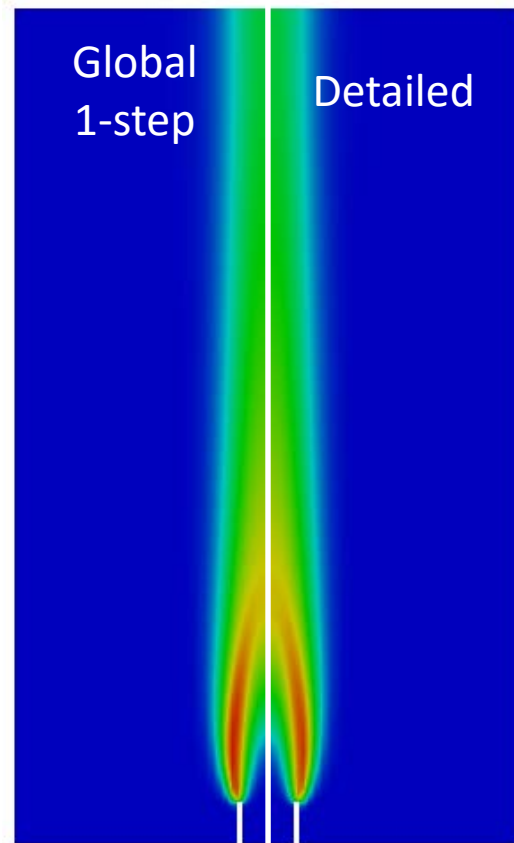
2. Type `blockMesh`

3. Type `laminarSolverUnsteady > log &`

4. The simulation takes about 10 min on a single core (from 0.25 s to 0.30 s)

Results

Mass fraction of radical species



NOx predictions

Mechanism preprocessing

We want now add NOX predictions by using the detailed kinetic mechanism contained in the PreProcessing/POLIMI_H2_NOX_1412:

1. Go to the PreProcessing/POLIMI_H2_NOX_1412 folder
2. Type `OpenSMOKEpp_CHEMKIN_PreProcessor --input input.dic`

Restarting from previous solution

We restart from the last solution obtained with the POLIMI_H2_NOX_1412 mechanism

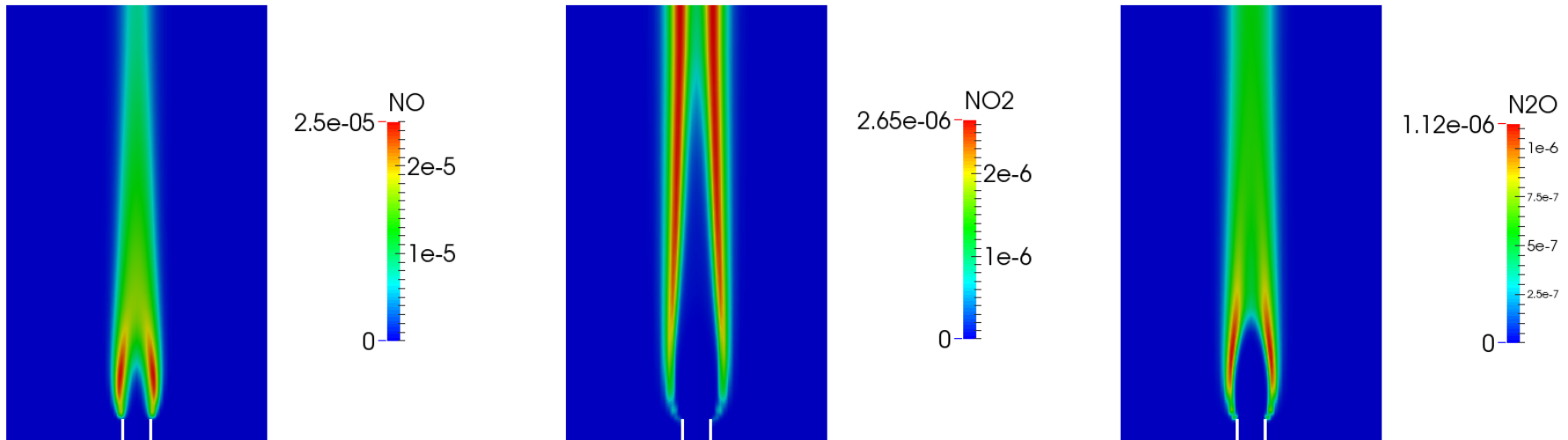
1. Go to the following folder:
`Training/Solvers/laminarSolverUnsteady/run/coflowFlame/03-detailed-polimi-nox`
2. Copy the last solution:

```
cp -r ../02-detailed-polimi/0.3/ .
cp ../01-global-1step/0/Ydefault 0.3/
rm -r 0.3/uniform
```


Running the simulation

1. Go to the following folder:
Training/Solvers/laminarSolverUnsteady/run/coflowFlame/02-detailed-polimi-nox
2. Type `blockMesh`
3. Type `laminarSolverUnsteady > log &`
4. The simulation takes more than 1 hour on a single core

Solutions @ 0.5 s

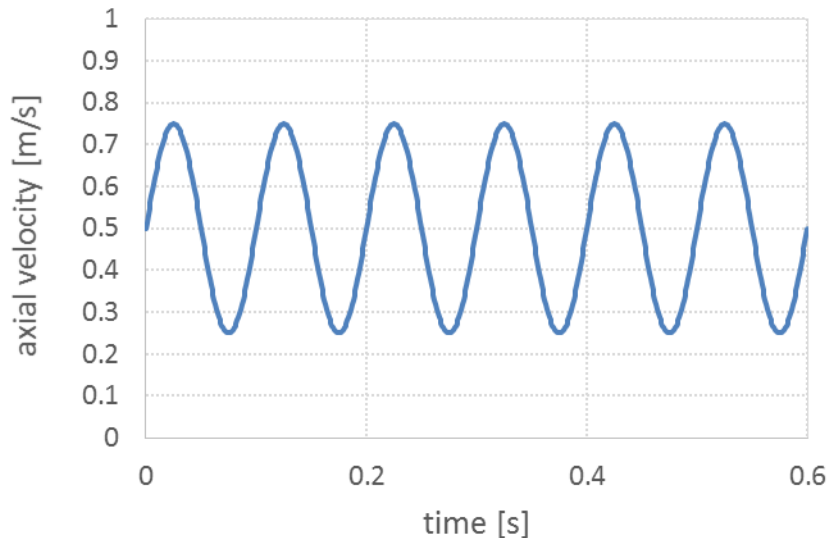


Pulsating flame (I)

We want now to simulate the same flame under **pulsating conditions**.

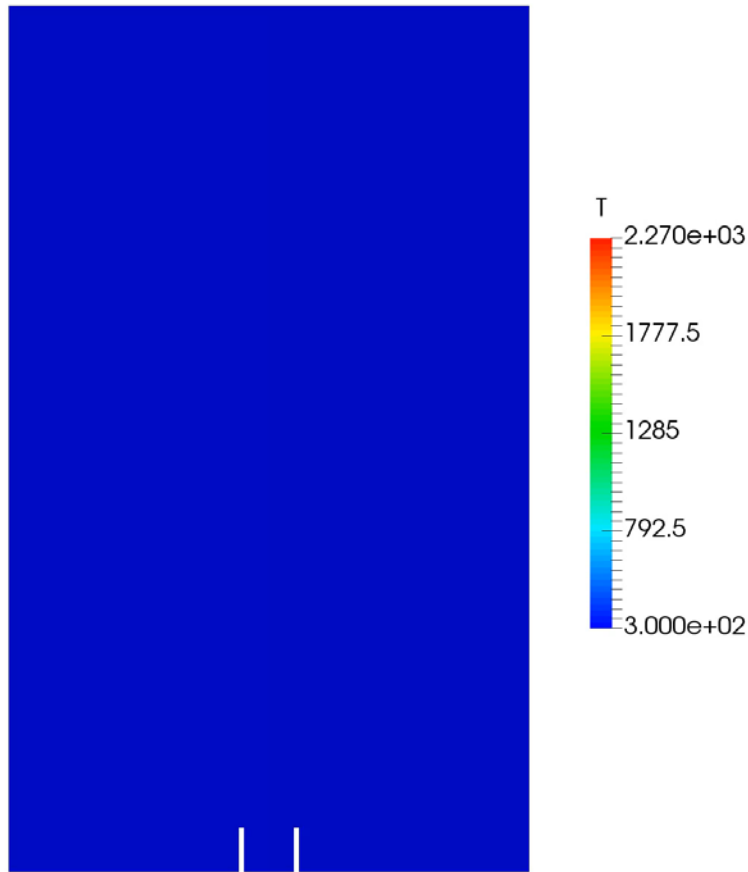
A sinusoidal velocity profile is imposed to the fuel stream, with amplitude **A=50%** and frequency **f=10Hz**

```
inletfuel
{
    type            uniformFixedValue;
    uniformValue    tableFile;
    file            "$FOAM_CASE/constant/myVelocityProfile";
}
```



```
(
    ( 0      ( 0 0 0.500000000 ) )
    ( 0.001  ( 0 0 0.515697630 ) )
    ( 0.002  ( 0 0 0.531333308 ) )
    ( 0.003  ( 0 0 0.546845329 ) )
    ( 0.004  ( 0 0 0.562172472 ) )
    ( 0.005  ( 0 0 0.577254249 ) )
    ( 0.006  ( 0 0 0.592031138 ) )
    ( 0.007  ( 0 0 0.606444823 ) )
    ...
)
```

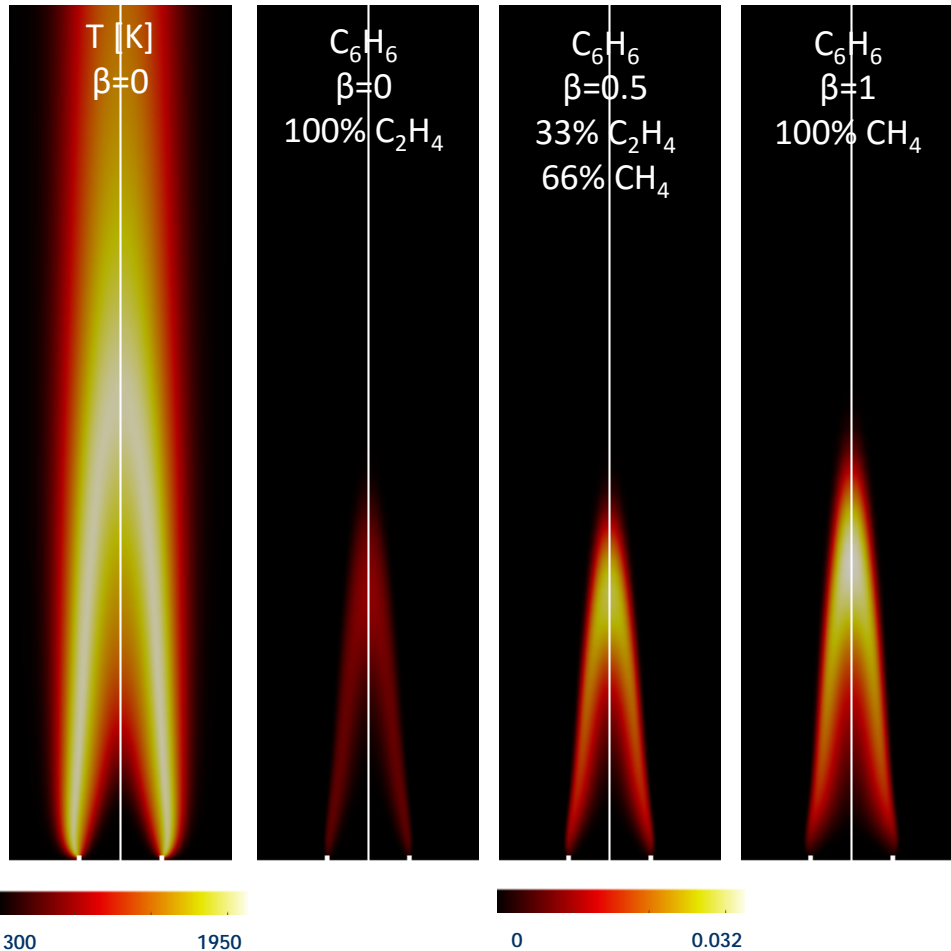
Pulsating flame (II)



In order to reduce the computational cost, the simulation is carried using the global 1-step mechanism

1. Go to the following folder:
Training/Solvers/laminarSolverUnsteady
/run/coflowFlame/04-global-1step-
pulsating
2. Type `blockMesh`
3. Type `laminarSolverUnsteady > log`
&
4. The whole simulation takes about 30 min on a single core. A single, complete cycle of oscillation about 15 min.

An example: C₂H₄/CH₄/N₂ coflow flames (I)



Flame details

Fuel: CH₄/C₂H₄

Air: O₂/N₂ (23.2%, 76.8% mass)

V_{fuel} : 12.52 cm/s

V_{air} : 10.50 cm/s

Fuel nozzle diameter: 11.1 mm

Chamber diameter: 110 mm

Computational details

Domain: 2D axisymmetric (55 x 200 mm)

Computational grid: ~25,000 cells

Discretization: second order centered

Kinetic scheme

POLIMI_HT1212:

198 species, 6307 reactions

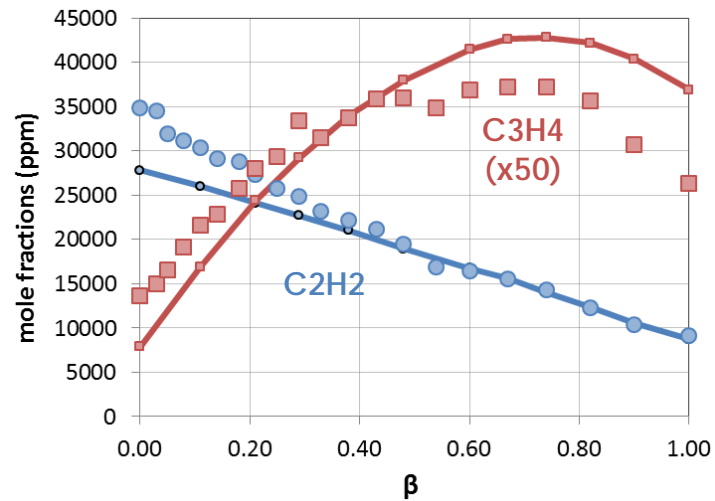
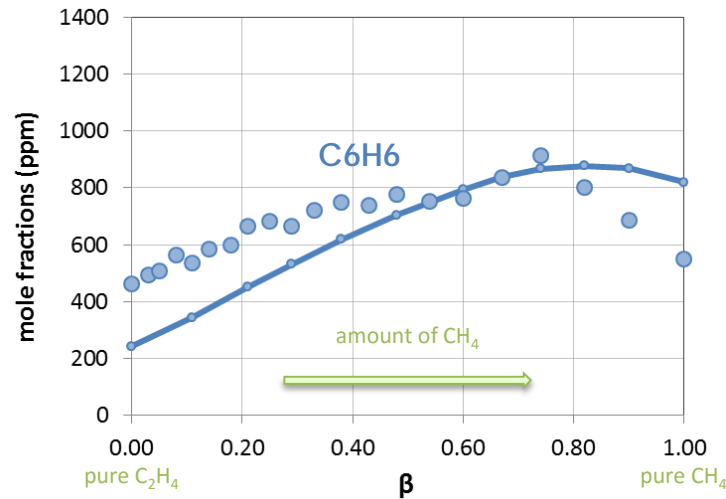
The concentrations of C₂H₄ and CH₄ are identified by the mixture parameter β :

$$\beta = \frac{X_{CH_4}}{X_{CH_4} + 2X_{C_2H_4}}$$

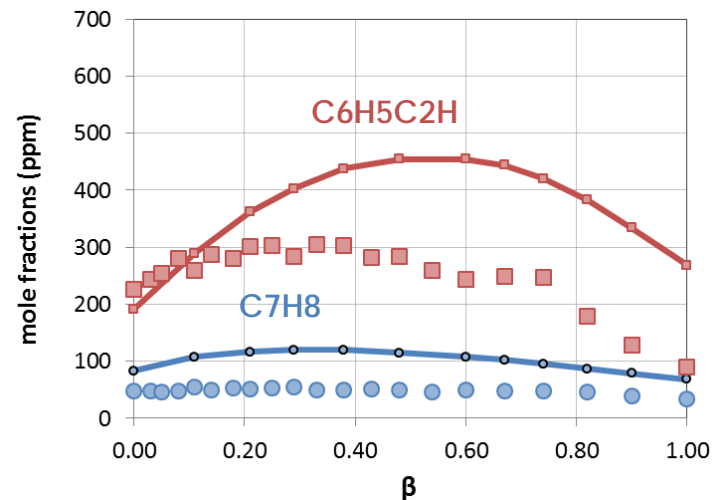
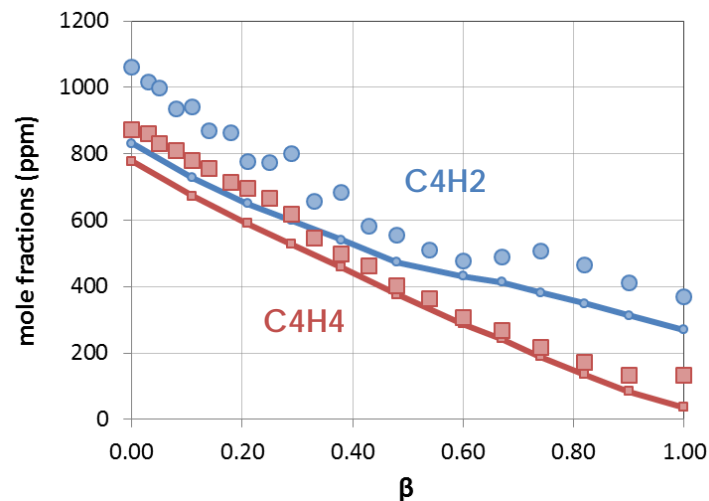
Cuoci A., Frassoldati A., Faravelli T., Ranzi E., Combustion and Flame, 160(5), p. 870-886 (2013)

An example: $C_2H_4/CH_4/N_2$ coflow flames (II)

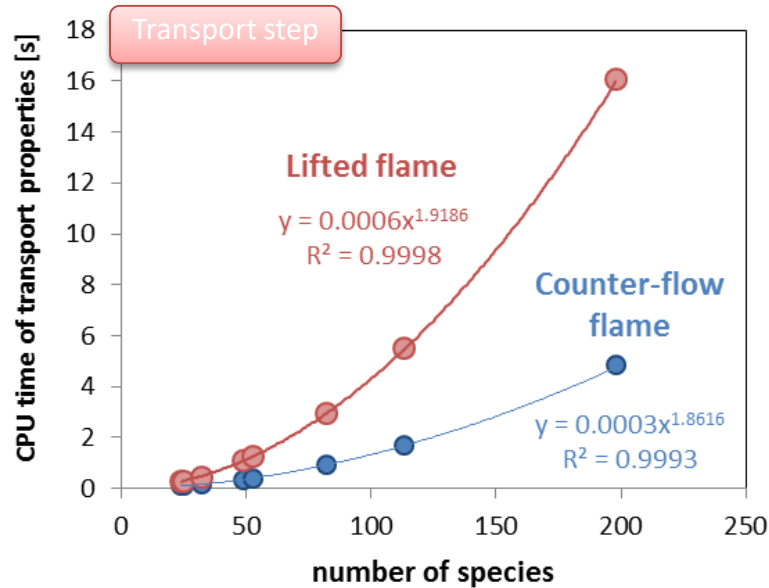
Peak values (along the center-line) of mole fractions



Experimental data from:
J.F. Roesler, M. Martinot, C.S. McEnally, L.D. Pfefferle, J.L. Delfau, C. Vovelle, *Combustion and Flame*, 134 (2003) 249-260.



Computational cost

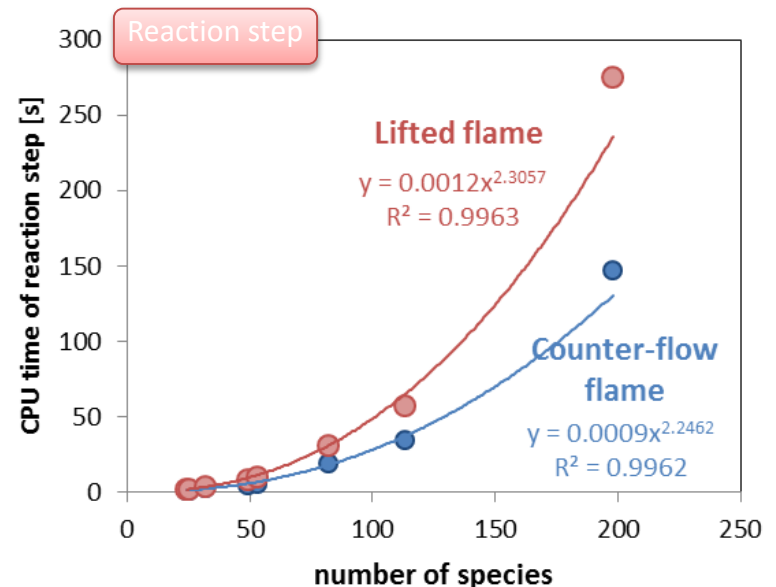


The CPU time of the reaction steps **increases more than quadratically (~2.3)** with the number of species, while the transport properties with a power of ~1.8

Increasing the number of species, the relative weight of the reaction step increases.

The reaction step results to be the most consuming part of the code, requiring more than **80-85% of the total computational time**.

The evaluation of the transport properties and the transport step cover the 5-7% and the 10% of the total time, respectively.



Outline

1. Unsteady solver based on the operator splitting algorithm

1. Introduction and theory
2. Implementation
3. Examples

2. Steady state solver based on the linearization of source terms

- 1. Introduction and theory**
2. Implementation
3. Examples

Steady-state solver: equations

Continuity
equation

$$\nabla(\rho \mathbf{v}) = 0$$

Momentum
equations

$$\nabla(\rho \mathbf{v} \mathbf{v}) = -\nabla p + \nabla \boldsymbol{\tau} + \rho \mathbf{g}$$

Energy equation

$$\rho C_p \mathbf{v} \nabla T = \lambda \nabla^2 T + Q_R$$

Species equations

$$\nabla(\rho \mathbf{v} Y_k) = \nabla(\rho \Gamma_k \nabla Y_k) + R_k \quad k = 1, \dots, N$$

Simplifying hypotheses

- No radiative heat transfer
- No enthalpy fluxes due to preferential mass diffusion
- No thermodiffusion (Soret effect)
- No correction velocity on mass diffusion fluxes

Steady-state solver: algorithm

Continuity
equation

$$\nabla(\rho \mathbf{v}) = 0$$

Momentum
equations

$$\nabla(\rho \mathbf{v} \mathbf{v}) = -\nabla p + \nabla \boldsymbol{\tau} + \rho \mathbf{g}$$

SIMPLE
Algorithm

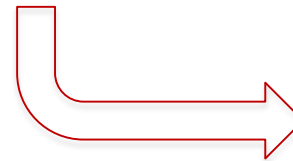
Energy equation

$$\rho C_p \mathbf{v} \nabla T = \lambda \nabla^2 T + Q_R$$

Strongly non
linear terms

Species equations

$$\nabla(\rho \mathbf{v} Y_k) = \nabla(\rho \Gamma_k \nabla Y_k) + R_k \quad k = 1, \dots, N$$



LINEARIZATION

Steady-state solver: chemistry (I)

$$R_k^{(n+1)} \approx R_k^{(n)} + \sum_{j=1}^N \left. \frac{\partial R_k}{\partial Y_j} \right|^{(n)} (Y_j^{(n+1)} - Y_j^{(n)}) = R_k^{(n)} + \sum_{j=1}^N J_{kj}^{(n)} (Y_j^{(n+1)} - Y_j^{(n)})$$

Jacobian matrix $J_{kj} = \frac{\partial R_k}{\partial Y_j}$

$$R_k^{(n+1)} \approx R_k^{(n)} + J_{kk}^{(n)} (Y_k^{(n+1)} - Y_k^{(n)}) + \sum_{\substack{j=1 \\ j \neq k}}^N J_{kj}^{(n)} (Y_j^{(n+1)} - Y_j^{(n)})$$

$$R_k^{(n+1)} \approx \underbrace{\left[J_{kk}^{(n)} Y_k^{(n+1)} \right]}_{\text{implicit (linear)}} + \underbrace{\left[R_k^{(n)} - J_{kk}^{(n)} Y_k^{(n)} \right]}_{\text{explicit}} + \underbrace{\left[\sum_{\substack{j=1 \\ j \neq k}}^N J_{kj}^{(n)} (Y_j^{(n+1)} - Y_j^{(n)}) \right]}_{\text{off-diagonal}}$$

Steady-state solver: chemistry (II)

$$R_k^{(n+1)} \approx \underbrace{\left[J_{kk}^{(n)} Y_k^{(n+1)} \right]}_{\text{implicit (linear)}} + \underbrace{\left[R_k^{(n)} - J_{kk}^{(n)} Y_k^{(n)} \right]}_{\text{explicit}} + \underbrace{\left[\sum_{\substack{j=1 \\ j \neq k}}^N J_{kj}^{(n)} \left(Y_j^{(n+1)} - Y_j^{(n)} \right) \right]}_{\text{off-diagonal}}$$

Option 1

$$R_k^{(n+1)} \approx \left[J_{kk}^{(n)} Y_k^{(n+1)} \right] + \left[R_k^{(n)} - J_{kk}^{(n)} Y_k^{(n)} \right]$$

Option 2

$$R_k^{(n+1)} \approx \left[J_{kk}^{(n)} Y_k^{(n+1)} \right] + \left[R_k^{(n)} - J_{kk}^{(n)} Y_k^{(n)} \right] + \left[\sum_{\substack{j=1 \\ j \neq k}}^N J_{kj}^{(n)} \left(Y_{j,pred}^{(n+1)} - Y_j^{(n)} \right) \right]$$

Predictor-corrector approach

Steady-state solver: source terms

$$\nabla(\rho \mathbf{v} Y_k) - \nabla(\rho \Gamma_k \nabla Y_k) = R_k$$

$$R_k^{(n+1)} \approx \left[J_{kk}^{(n)} Y_k^{(n+1)} \right] + \left[R_k^{(n)} - J_{kk}^{(n)} Y_k^{(n)} \right]$$

```
fvScalarMatrix YiEqn
(
    mvConvection->fvmDiv(phi, Yi)
  - fvm::laplacian(rho*GammaMixi, Yi)
  ==
    sourceExplicit[i]
  + fvm::Sp(sourceImplicit[i], Yi)
  + fvOptions(rho, Yi)
);
```

Outline

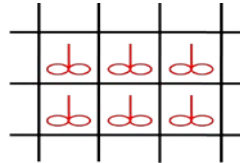
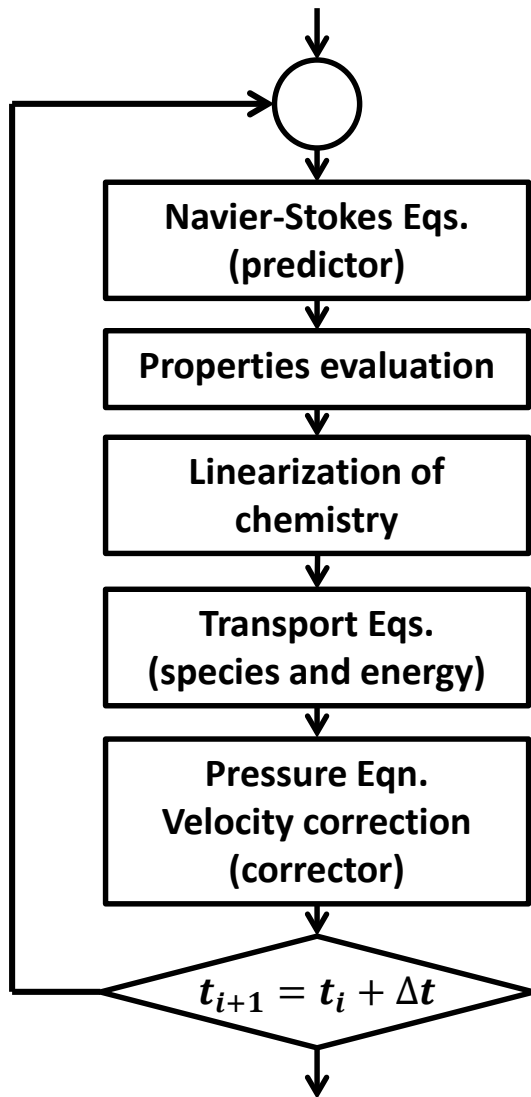
1. Unsteady solver based on the operator splitting algorithm

1. Introduction and theory
2. Implementation
3. Examples

2. Steady state solver based on the linearization of source terms

1. Introduction and theory
- 2. Implementation**
3. Examples

Steady-state solver



```
while (simple.loop())
{
    #include "UEqn.H"

    #include "transport.H"
    #include "chemistry.H"

    #include "YEqn.H"
    #include "TEqn.H"

    if (simple.consistent())
        #include "pcEqn.H"
    else
        #include "pEqn.H"

    runTime.write();
}
```

Steady-state solver: initialization

```
int main(int argc, char *argv[])
{
    // OpenFOAM stuff
    #include "setRootCase.H"
    #include "createTime.H"
    #include "createMesh.H"
    #include "createControl.H"
    #include "readGravitationalAcceleration.H"
    #include "createBasicFields.H"

    // OpenSMOKE++
    #include "createChemicalFields.H"
    #include "createSourceFields.H"
    #include "transportProperties.H"
    #include "createAdditionalFields.H"

    // Linear model for reacting source term
    linearModelChemistry chemistry(thermoMap, kineticsMap);

    ...
}
```

Fields for source terms: implicit

```
for (int i=0;i<thermoMap.NumberOfSpecies();i++)
{
    sourceImplicit.set
    (
        i,
        new volScalarField
        (
            IOobject
            (
                "SI_" + thermoMap.NamesOfSpecies()[i],
                mesh.time().timeName(),
                mesh,
                IOobject::NO_READ,
                IOobject::NO_WRITE
            ),
            mesh,
            dimensionedScalar("SI", dimensionSet(1, -3, -1, 0, 0), 0.0)
        )
    );
}
```


Fields for source terms: explicit

```
for (int i=0;i<thermoMap.NumberOfSpecies();i++)
{
    sourceExplicit.set
    (
        i,
        new volScalarField
        (
            IOobject
            (
                "SE_" + thermoMap.NamesOfSpecies()[i],
                mesh.time().timeName(),
                mesh,
                IOobject::NO_READ,
                IOobject::NO_WRITE
            ),
            mesh,
            dimensionedScalar("SE", dimensionSet(1, -3, -1, 0, 0), 0.0)
        )
    );
}
```

Evaluating chemical source terms

```
{
    const scalarField& TC = T.internalField();
    const scalarField& pC = p.internalField();

    Eigen::VectorXd J(NC+1), Source(NC+1), y(NC+1);

    forAll(TC, celli)
    {
        for(int i=0;i<NC;i++)
            y(i) = Y[i].internalField()[celli];
        y(NC) = TC[celli];

        chemistry.reactionSourceTerms(thermoMap, kineticsMap, y, pC[celli], Source);
        chemistry.reactionJacobian( thermoMap, kineticsMap, y, pC[celli], J );

        for(int i=0;i<NC+1;i++)
        {
            sourceImplicit[i].ref()[celli] = J(i);
            sourceExplicit[i].ref()[celli] = Source(i) - J(i)*y(i);
        }
    }
}
```

linearModelChemistry (I)

```
class linearModelChemistry
{
public:

    linearModelChemistry( OpenSMOKE::ThermodynamicsMap_CHEMKIN& thermoMap,
                          OpenSMOKE::KineticsMap_CHEMKIN& kineticsMap);

    void reactionSourceTerms( OpenSMOKE::ThermodynamicsMap_CHEMKIN& thermoMap,
                              OpenSMOKE::KineticsMap_CHEMKIN& kineticsMap,
                              const Eigen::VectorXd& y, const double P0,
                              Eigen::VectorXd& S);

    void reactionJacobian(   OpenSMOKE::ThermodynamicsMap_CHEMKIN& thermoMap,
                              OpenSMOKE::KineticsMap_CHEMKIN& kineticsMap,
                              const Eigen::VectorXd& y, const double P0,
                              Eigen::VectorXd &J );

private:

    ...
};
```

linearModelChemistry (II)

```
void linearModelChemistry::reactionSourceTerms
(
    OpenSMOKE::ThermodynamicsMap_CHEMKIN& thermoMap_,
    OpenSMOKE::KineticsMap_CHEMKIN& kineticsMap_,
    const Eigen::VectorXd& y, const double P0, Eigen::VectorXd& S)
{
    ...

    // Calculates thermodynamic properties
    thermoMap_.SetTemperature(T); thermoMap_.SetPressure(P0);
    kineticsMap_.SetTemperature(T); kineticsMap_.SetPressure(P0);

    // Calculates kinetics
    kineticsMap_.KineticConstants();
    kineticsMap_.ReactionRates(c_.data());
    kineticsMap_.FormationRates(R_.data());

    // Species
    for (unsigned int i=0;i<NC_;++i)
        S(i) = R_(i)*thermoMap_.MW(i);

    // Energy
    const double QR_ = kineticsMap_.HeatRelease(R_.data());
    S(NC_) = QR_;
}
```

linearModelChemistry (III)

```
void linearModelChemistry::reactionJacobian(
    OpenSMOKE::ThermodynamicsMap_CHEMKIN& thermoMap,
    OpenSMOKE::KineticsMap_CHEMKIN& kineticsMap,
    const Eigen::VectorXd& y, const double P0, Eigen::VectorXd &J )
{
    ...

    // Call equations
    reactionSourceTerms(thermoMap, kineticsMap, y, P0, dy_original_);

    // Derivatives with respect to y(kd)
    for(int kd=0;kd<NE_;kd++)
    {
        ...

        double dy = std::min(hJ, 1.e-3 + 1e-3*fabs(y(kd)));
        double udy = 1. / dy;
        y_plus_(kd) += dy;
        reactionSourceTerms(thermoMap, kineticsMap, y_plus_, P0, dy_plus_);

        J(kd) = (dy_plus_(kd)-dy_original_(kd)) * udy;
        y_plus_(kd) = y(kd);
    }
}
```

Numerical Jacobian: finite differences

Definition of Jacobian matrix

$$J_{kj} = \frac{\partial R_k}{\partial Y_j}$$

Forward finite-difference

$$J_{kj} \approx \frac{R_k(Y_j + \Delta Y_j) - R_k(Y_j)}{\Delta Y_j} \quad \Delta Y_j \text{ is a small increment}$$

$$\Delta Y_j = \epsilon_a + \epsilon_r |Y_j| \quad \text{where } \epsilon_a \text{ and } \epsilon_r \text{ are sufficiently small numbers}$$

Decreasing the increment ΔY_j will reduce the truncation error. Unfortunately a smaller increment has the opposite effect on the cancellation error. Selecting the optimal step size for a certain problem is not trivial and may be computationally very expensive. A more accurate option is the following (ε is the machine precision):

$$\Delta Y_j = \min(h, \epsilon_a + \epsilon_r |Y_j|) \quad h = \varepsilon \max \left(|Y_j|, \frac{1}{tol_{abs} + tol_{rel} |Y_j|} \right)$$

Outline

1. Unsteady solver based on the operator splitting algorithm

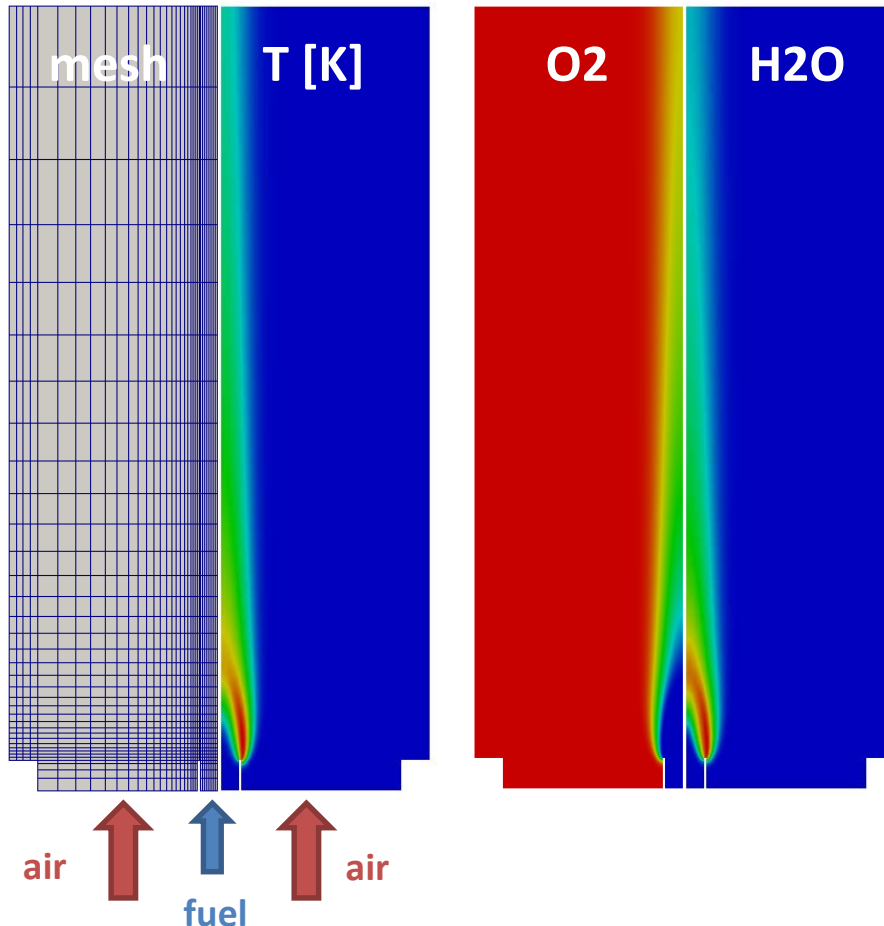
1. Introduction and theory
2. Implementation
3. Examples

2. Steady state solver based on the linearization of source terms

1. Introduction and theory
2. Implementation
- 3. Examples**

Example: laminar coflow flame

V. V. Toro, A. V. Mokhov, H. B. Levinsky, M. D. Smooke, Proceedings of the Combustion Institute, 30 (2005), 485-492



The fuel stream (**50% H₂ and 50% N₂ by volume**) is injected at ambient temperature through a circular nozzle (**i.d. 9 mm**), surrounded by an air-coflow annulus (**i.d. 95 mm**).

The fuel and coflow inlet velocities are assumed equal to **50 cm/s** (**Flame F3**).

A **2D rectangular domain** meshed with a structured grid is considered.

Global kinetic mechanism

Our preliminary calculations are carried out using a global 1 step mechanism, available in the PreProcessing/Global_H2_1step folder

```
ELEMENTS
```

```
H O N
```

```
END
```

```
SPECIES
```

```
H2 O2 H2O N2
```

```
END
```

```
REACTIONS
```

```
H2 + 0.5O2 => H2O 1e14 0 20000
```

```
FORD / O2 1.00 /
```

```
END
```

The kinetic mechanism was already pre-processed (see previous sections) using the OpenSMOKEpp_CHEMKIN_PreProcessor.

Setup the case

The kinetic mechanism to be used is specified in the `constant/solverOptions` file

```
Kinetics
{
    folder            ".../.../.../.../.../PreProcessing/Global_H2_1step/";
    inertSpecies      N2;
}
```

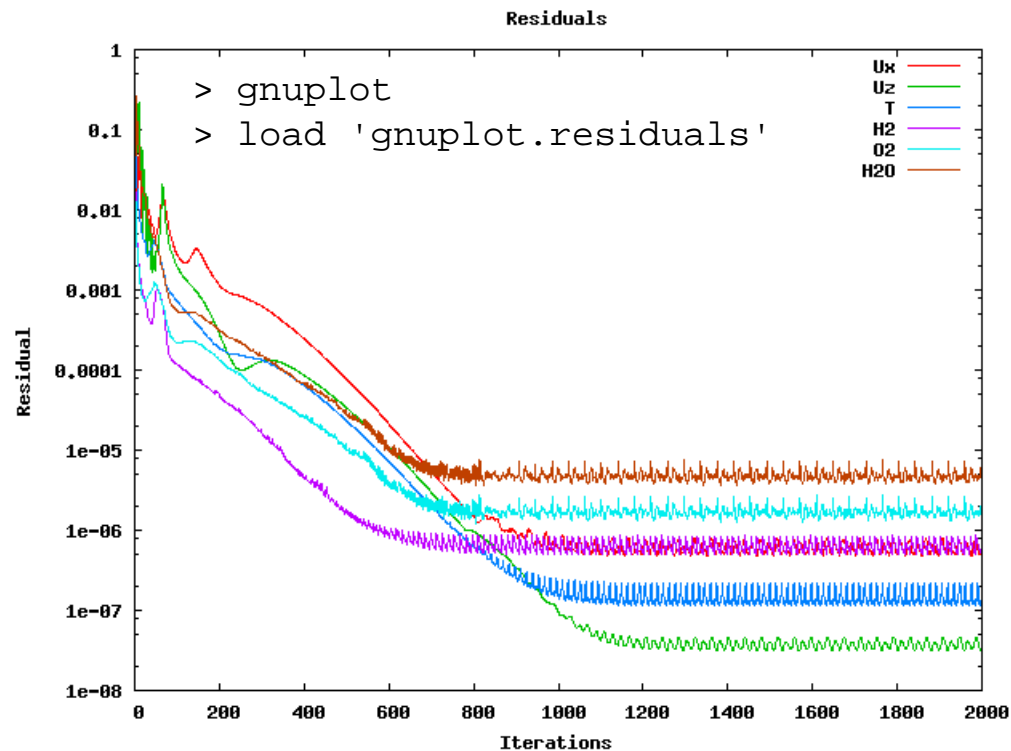
Since both the fuel and oxidizer streams are fed at ambient temperature, it is necessary to introduce a spark in order to ignite the mixture.

```
Spark
{
    spark            on;
    position          (5.95e-3  0.0  1.5e-3);
    time             0.;
    temperature       2200;
    duration         25;
    diameter          1.5e-3;
}
```

The only difference with respect to the unsteady case is the duration (which in steady state conditions means number of iterations)

Running the case

1. Go to the following folder:
Training/Solvers/laminarSolverSteady/run/coflowFlame/01-global-1step
2. Type `blockMesh`
3. Type `laminarSolverSteady > log &`
4. The simulation takes less than 1 min on a single core

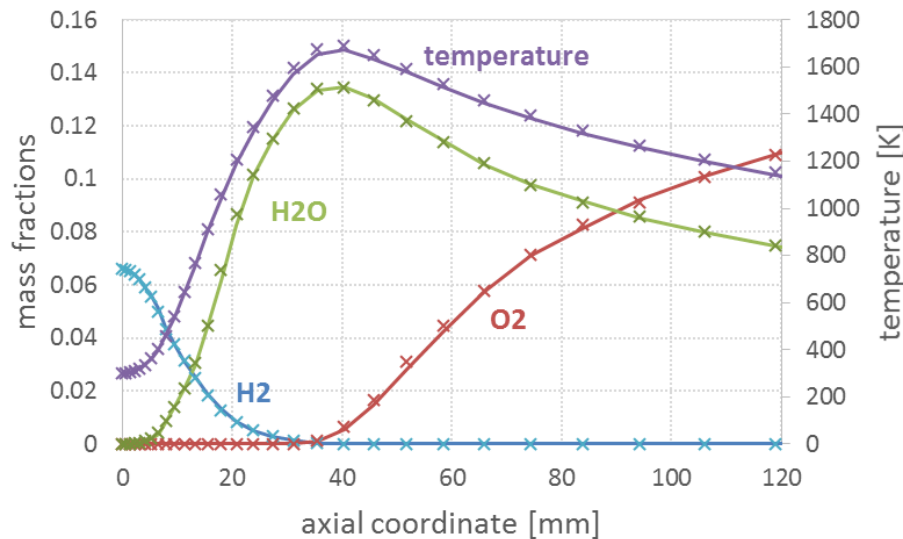


Results

Solution along the axis

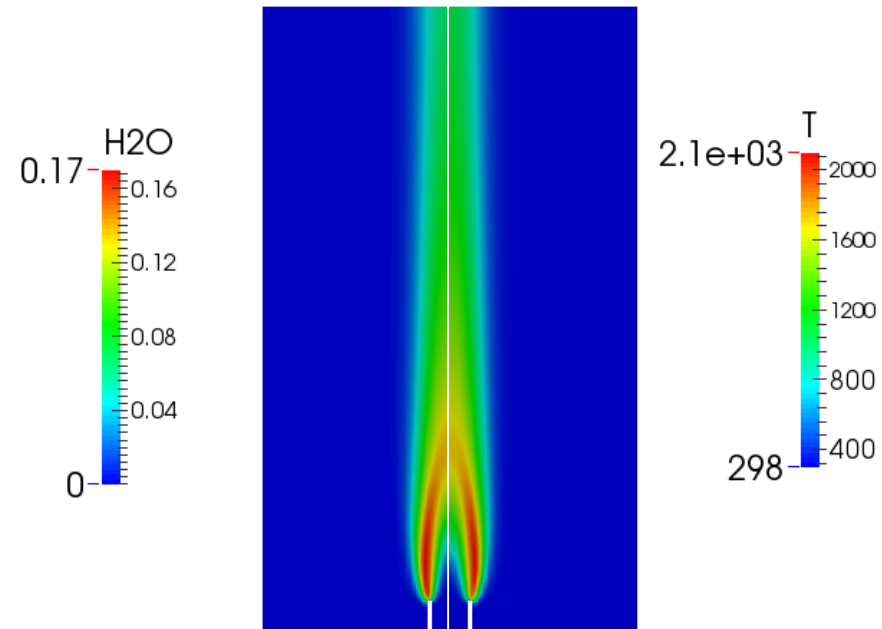
Symbols: steady state solver (2000 iterations)

Lines: unsteady solver (@ 0.3 s)



```
> postProcess -func sample
```

Solution after 2000 iterations



Refining the solution using a detailed mech

Mechanism preprocessing

We want now refine the solution by using the detailed kinetic mechanism contained in the `PreProcessing/POLIMI_H2_1412`, which has been already preprocessed.

Restarting from previous solution

We restart from the last solution (2000 iterations) corresponding to the global, 1-step mechanism

1. Go to the following folder:

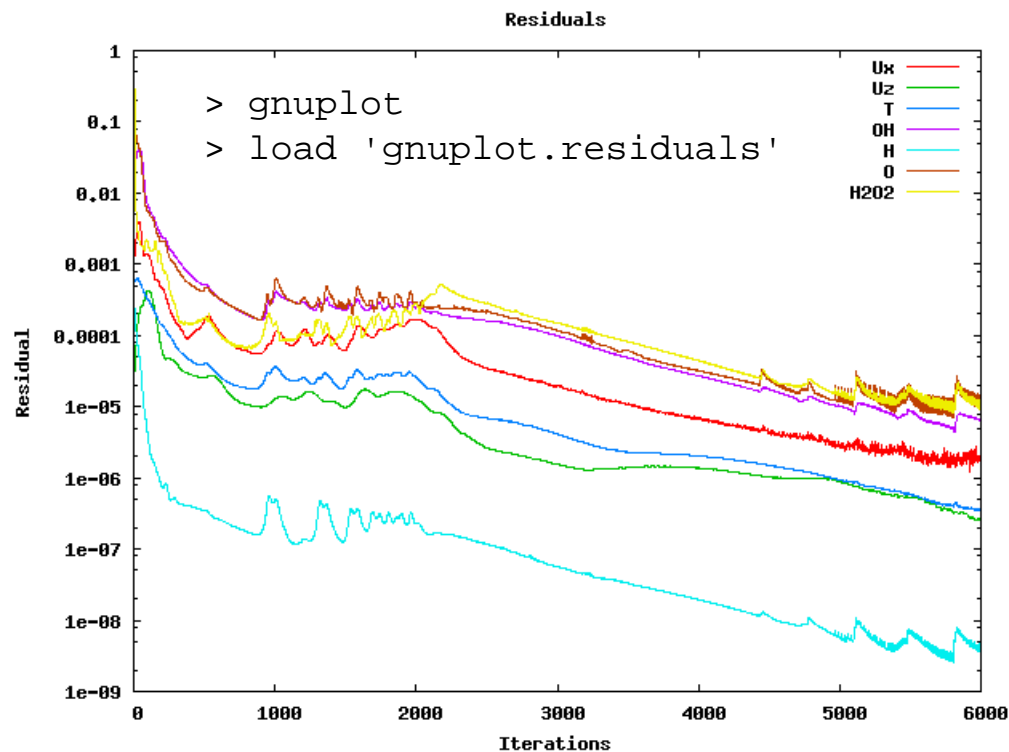
`Training/Solvers/laminarSolverSteady/run/coflowFlame/02-detailed-polimi`

2. Copy the last solution:

```
cp -r ../01-global-1step/2000/ 0/  
cp ../01-global-1step/0/Ydefault 0/  
rm -r 0/uniform
```

Running the case

1. Go to the following folder:
Training/Solvers/laminarSolverSteady/run/coflowFlame/02-polimi-detailed
2. Type `blockMesh`
3. Type `laminarSolverSteady > log &`
4. The simulation takes less than 5 min on a single core



NOx predictions

Mechanism preprocessing

We want now add NOX predictions by using the detailed kinetic mechanism contained in the PreProcessing/POLIMI_H2_NOX_1412 (which was already preprocessed)

Restarting from previous solution

We restart from the last solution obtained with the POLIMI_H2_NOX_1412 mechanism

1. Go to the following folder:

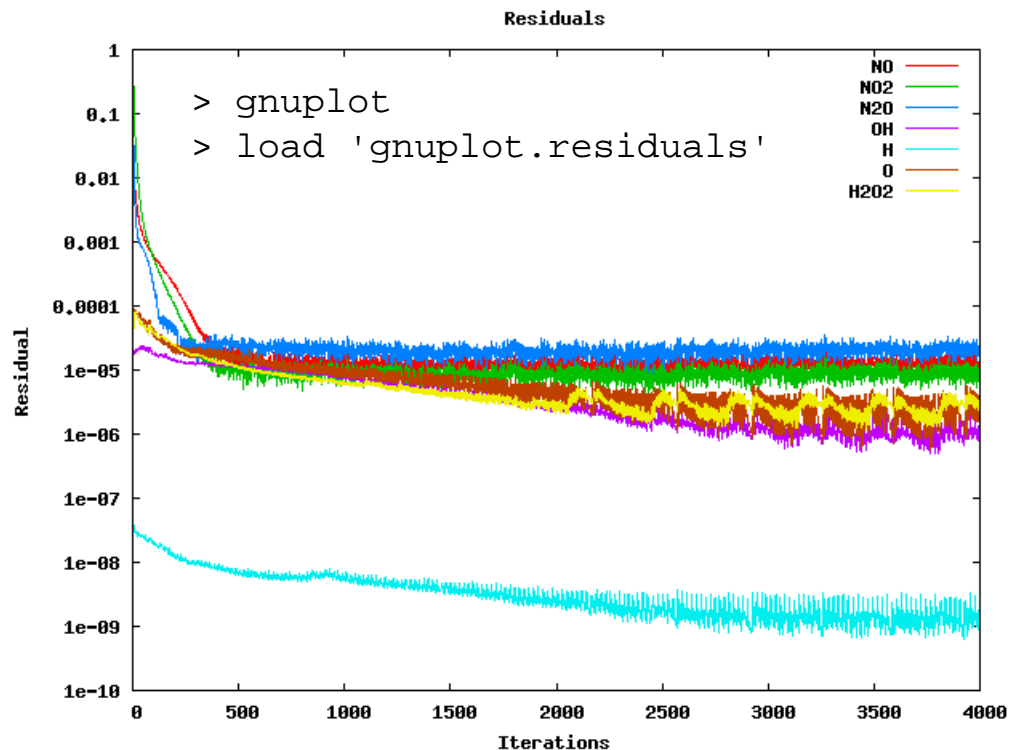
Training/Solvers/laminarSolverSteady/run/coflowFlame/03-detailed-polimi-nox

2. Copy the last solution:

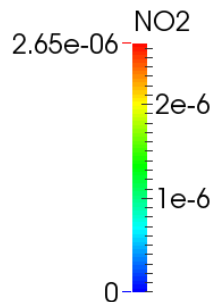
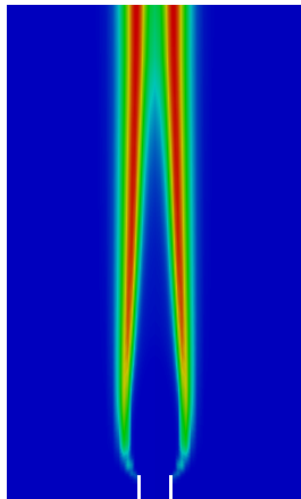
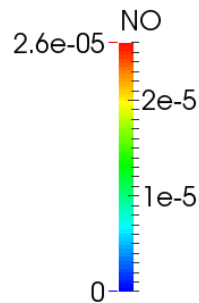
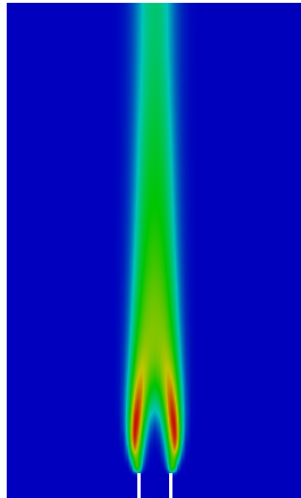
```
cp -r ../02-detailed-polimi/6000/ 0/  
cp ../01-global-1step/0/Ydefault 0/  
rm -r 0/uniform
```

Running the case

1. Go to the following folder:
Training/Solvers/laminarSolverSteady/run/coflowFlame/03-polimi-detailed-nox
2. Type `blockMesh`
3. Type `laminarSolverSteady > log &`
4. The simulation takes about 5 min on a single core



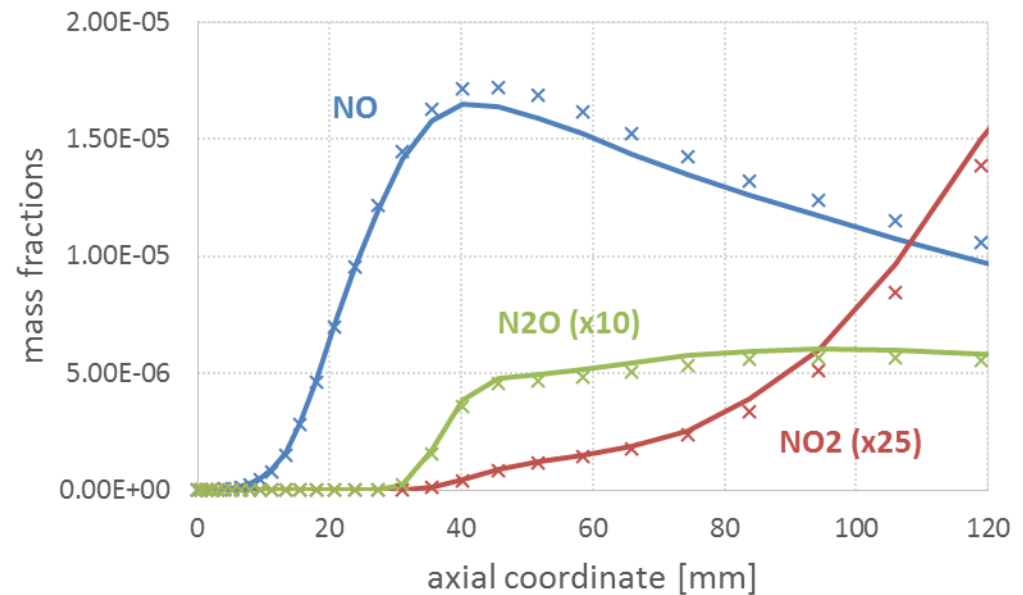
Results



Solution along the axis

Symbols: steady state solver (5000 iterations)

Lines: unsteady solver (@ 0.5 s)



Next steps

1. Chemical step

- Energy equation as differential equation
- External ODE solvers (CVODE, DASPK, etc.)
- Native OpenSMOKE++ solvers
- Intel MKL Libraries

2. Transport step

- Inclusion of thermodiffusion (i.e. Soret effect)
- Inclusion of enthalpy fluxes due to preferential mass diffusion
- Inclusion of correction diffusion of mass diffusion fluxes

3. Numerical Algorithm

- Strang splitting variants
- Consistent Strang splitting
- Predictor/Corrector policy for steady state solver

Acknowledgements

The CRECK Modeling Group



Tiziano Faravelli



Eliseo Ranzi



Alessio
Frassoldati



Alessandro
Stagni



Matteo Pelucchi

PhD Students



Ghobad Bagheri



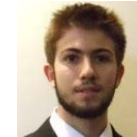
Agn s Bodor



Giancarlo Gentile



Abd Essamade
Saufi



Paulo Debiagi



Kik
Pejpichestakul



Ali Shamooni

References (I)

OpenSMOKE++ Development

Cuoci, A., Frassoldati, A., Faravelli, T., Ranzi, E., *OpenSMOKE++: An object-oriented framework for the numerical modeling of reactive systems with detailed kinetic mechanisms* (2015) Computer Physics Communications, 192, pp. 237-264, DOI: [10.1016/j.cpc.2015.02.014](https://doi.org/10.1016/j.cpc.2015.02.014)

Cuoci, A., Frassoldati, A., Faravelli, T., Ranzi, E. *Numerical modeling of laminar flames with detailed kinetics based on the operator-splitting method* (2013) Energy and Fuels, 27 (12), pp. 7730-7753, DOI: [10.1021/ef4016334](https://doi.org/10.1021/ef4016334)

Cuoci, A., Frassoldati, A., Faravelli, T., Ranzi, E. *A computational tool for the detailed kinetic modeling of laminar flames: Application to C₂H₄/CH₄ coflow flames* (2013) Combustion and Flame, 160 (5), pp. 870-886, DOI: [10.1016/j.combustflame.2013.01.011](https://doi.org/10.1016/j.combustflame.2013.01.011)

M.Maestri, A.Cuoci, *Coupling CFD with detailed microkinetic modeling in heterogeneous catalysis*, Chemical Engineering Science 96(7), pp. 106-117 (2013) DOI: [10.1016/j.ces.2013.03.048](https://doi.org/10.1016/j.ces.2013.03.048)

References (II)

Weller H. G., Tabor G., Jasak H., Fureby C., *Computers in Physics* 12: 620-631 (1998)

Stroustrup B., *The C++ Programming Language, 3rd Edition*, Addison–Wesley, Reading (MA), 1997

Cary J. R., et al., *Computational Physics Communications* 105: 20-36 (1997)

Alexandrescu A., *Modern C++ Design: Generic Programming and Design Patterns Applied*, Addison-Wesley, (2001)

Buzzi-Ferraris G., *BzzMath 6.0 Numerical Libraries*, 2011

OpenFOAM, *OpenFOAM®*, <http://www.openfoam.org/>