CANTERA Tutorials

A series of tutorials to get started with the python interface

of Cantera version 2.3

Anne Felden

Updated : Quentin Cazeres

November 2018

Introduction

Cantera is a suite of object-oriented software tools for problems involving chemical kinetics, thermodynamics, and/or transport processes. Cantera provides types (or classes) of objects representing phases of matter, interfaces between these phases, reaction managers, time-dependent reactor networks, and steady one-dimensional reacting flows. Cantera is currently used for applications including combustion, detonations, electrochemical energy conversion and storage, fuel cells, batteries, aqueous electrolyte solutions, plasmas, and thin film deposition.

The version of Cantera that we will present and use in this tutorial is the version 2.3. As we will work with a precompiled version, the installation requirements and guidelines will not be recalled hereafter, but can be found on the official website :

https://cantera.org/install/index.html

Cantera can be used from both Python and Matlab interfaces, or in applications written in C++ and Fortran 90. There are several advantages in choosing the Python interface. First of all, it offers most of the features of the C++ core in a much more flexible environment. Its most obvious advantage over the Matlab toolbox is that it can be downloaded free of charge. Furthermore, Python assistance can be easily found online thanks to a broad community of users. As such, the Python module is nowadays the most commonly used, so that online help and scripts can be easily found, adapted to your needs and discussed with experimented users. To be consistent with this trend, the tutorials will focus on this user-friendly interface as well. Specifically, Python 3.X will be used. If you are more familiar with the Python 2.X syntax don't worry as there are no huge differences. The key ones are reviewed on this article : *https://sebastianraschka.com/Articles/2014_python_2_3_key_diff.html*

To get started, you only need to open a terminal and type :

```
module load cantera
```

Contents

List of Figures

1. *Tutorial 1 : An example to start with ...*

1.1 Introduction

This first tutorial's main objective is to help you get started with the Python interface of Cantera. The goal here is to get acquainted with the notion of objects and functions that we will encounter in all Python scripts. We will start by defining a simple gaseous object directly in a 'script', before introducing the Python terminal and its possibilities. Next, we will see how to access and modify the state of this newly instantiated "gas" object through adequate functions; before concluding with a general discussion on the specificities of gaseous objects.

The second objective of this first tutorial is to introduce the 'data file' format of Cantera, which will be the subject of the second tutorial.

1.2 Exercises

1.2.1 Define a single gas object through a script

Go into the folder 'Tuto1' and start by creating a new Python script -filename ending with .py, say 'gas_simple.py':

```
$ touch gas_simple.py
```

Open it in the text editor of your choice :

```
$ vi gas_simple.py
```

Put the following statement at the top of your script to import the Cantera Python module :

```
import cantera as ct
```

Now, when using Cantera, the first thing you usually require is an object representing some phase of matter. Let us begin by defining a gaseous mixture, with the attributes of the GRIMech3.0 mechanism, that we will label "gas". This is done by adding the following command line on your script:

```
gas = ct.Solution('gri30.xml')
```

Here, we imported information from an external file, 'gri30.xml'. This file should already be present in your working directory. It contains all information necessary to instantiate your "gas" object. We will see what this file contains in more details in the next tutorial. For now, notice that the "Solution" object is imported from the Cantera package, and as such, a "ct." has to be added at the beginning of its name. See B.1.3 for more information on how to load packages in a Python environment. All functionalities from the Cantera package have to be called this way.

As we are in an "object-oriented" environment, this "Solution" object is a class instance; and as such, has many attributes and methods associated to it. Let us begin by printing a few through this script. To see, for example, the current state of your "gas" object, you could print it on your terminal by adding the next line on your script:

```
print(gas())
```

Next, we could print a list of the species present in this gaseous mixture. Add the following line to your script :

```
print(gas.species_names)
```

Don't forget to save your work. You are now ready to launch this first script ! To do so, simply type the following command from your working directory, on your terminal:

```
$ python gas_simple.py
```

Results

The summary of the state of that "gas" object that you will see on your screen, shows that new objects created from the 'gri30.xml' input file start out with a temperature of 300 K, a pressure of 1 atm, and have a composition that consists of only one species, in this case hydrogen. There is nothing special about H2 - it just happens to be the first species listed in the input '.xml' file. In general, whichever species is listed first will initially have a mole fraction of 1.0, and all of the others will be zero.

Notice that the name of the species involved are printed in an array.

1.2.2 Define a single gas object through the Python terminal

You could also work directly from the terminal, by launching a python compiler in your terminal with the command :

```
$ python
```

If you are already familiar with Python, you can see that this way of proceeding offers many advantages for very simple scripts, or to test certain functionalities. Your terminal will open a Python environment, recognizable at the symbols that appear on the left of your terminal:

```
>>>
```

Simply type individually each line of the script you designed in the previous section, and spot errors/results directly. You could define another gas object, and continue to test objects and functions for as long as you wish.

A very interesting feature that can be accessed through this more flexible environment is the 'help' function. This will make it possible for you to access all methods and attributes associated to a particular object. If you remember from the previous section, we printed, for example, the list of species in the gas with the command:

```
print(gas.species_names)
```

But many more functionalities are available, and you will probably forget them as you go ... To remember what they are, from the Python environment, we would type the following commands:

```
>>> import cantera as ct
>>> gas = ct.Solution('gri30.xml')
>>> help(gas)
```

As you will see, a list of functions will appear on your terminal screen. Now, as a gas representative, this particular "gas" object has properties that you would expect for a gas mixture : a temperature, a pressure, species mole and mass fractions, etc. Continue by calling a few functions of your choice from this list on your "gas" object :

```
>>> gas.species_names
>>> gas.T
>>> gas.Y
...
```

Once you are done with the Python environment, close it with the command:

```
>>> exit()
```

1.2.3 Continue your simple script: modify the state of your "gas" object

Open again the script generated in section 1.2.1:

```
$ vi gas_simple.py
```

We will now modify the state of the newly created object. For example, adding the line:

```
gas.TP = 800, 202650
```

will set its temperature to 800 K and pressure to 202650 Pa (Cantera always uses SI units). Print the state of the gas again, right after this command, by adding the line :

```
print(gas())
```

to see the difference. Save your modifications, and launch your script again, from your terminal:

```
$ python gas_simple.py
```

NOTE : you could also work directly from the Python terminal if you find it more handy ! If you decide to do so, remember that every command that start with a 'print', such as:

```
print(gas())
```

could simply be replaced by the same command, without stating the 'print' explicitly:

```
>>> gas()
```

to print the "gas" object state.

Results

Notice the differences from the two 'gas()' printout: the temperature and pressure have changed, along with the density.

1.2.4 Conclusion and discussion about gaseous phases

Thermodynamics generally requires that two properties in addition to composition information be specified to fix the intensive state of a substance (or mixture). The state of the mixture can be set using several combinations of two properties. The following are all equivalent:

```
gas.TP = 800, 202650            # temperature, pressure
gas.TD = 800, 0.0614168         # temperature, density
gas.HP = 7.29094e6, 202650      # specific enthalpy, pressure
gas.UV = 3.99135e6, 1/0.0614168 # specific internal energy, specific volume
gas.SP = 76290.8, 202650        # specific entropy, pressure
gas.SV = 76290.8, 1/0.0614168   # specific entropy, specific volume
```

Try adding these line to your 'gas_simple.py', and calling the gas() function in between each of them to convince yourself. Note that the values of the extensive properties must be entered per unit mass.

Fortunately, properties may be read independently or together. For example, from the Python terminal, the following commands on your previous gaseous object would produce:

```
>>> gas.T
800.0
>>> gas.TP
(800.0, 202650.0)
```

Notice the format of each printout.

To modify the composition of the gas, use 'X' or 'Y' in order to specify the mole or mass fractions, respectively. For example :

```
gas.X = 800, 202650, 'CH4:1, O2:2, N2:7.52' # temperature, pressure, composition
```

Will instantiate the "gas" object by specifying a temperature, a pressure and mole fractions. The composition, in this example, was specified using a string (it can also be done with a dictionary). The format is composed of a separated list of <species name>:<relative mole numbers> pairs. Note that the mole or mass numbers will automatically be normalized to produce the mole or mass fractions, respectively. The composition can also be set using an array, which must then have the same size as the number of species. For example with the GRIMech3.0 mechanism, as there are 53 species, the mass fraction composition could be set using:

```
gas.Y = np.ones(53) # NumPy array of 53 ones
```

Where the 'np' here refers to the numpy python package, see B.1.3. When the composition alone is changed, Cantera helds the temperature and density constant. To change that, use the keyword 'None' to specify which properties are to stay constant. For example, if you want to change the mixture's mole fractions while keeping the temperature and pressure constant:

```
gas.X = None, None, 'CH4:1.0, O2:0.5'
```

Another possibility is to directly specify the equivalence ratio ϕ of your mixture defined as :

$$\phi = \frac{m_{fuel}/m_{ox}}{\left(m_{fuel}/m_{ox}\right)_{st}} = \frac{n_{fuel}/n_{ox}}{\left(n_{fuel}/n_{ox}\right)_{st}}$$

For that you can use a dedicated method of the Solution object, specifying the equivalence ratio you want, fuel composition and oxidizer composition

```
gas.set_equivalence_ratio(1, 'CH4: 1', 'O2: 1, N2: 3.76')
```

This can be useful if you have complex fuel or oxidizer composition and don't want to make mistakes computing stoichiometric variables and equivalence ratio

1.3 To go further ...

1.3.1 Cantera's 'CTI' format for mechanism files

The information required to compute the previous quantities were specified in the '.xml' file, also labeled "mechanism file" or "data file". Cantera also supports a data file format that is easier to write than the '.xml' format, with the extension '.cti'. The next tutorial will cover in more details what needs to be specified in this file; but for now, several reaction mechanism files in this format are included your working directory, 'Tuto1'. You can peruse them to get familiar with the syntax. Open, for example, the 'gri.cti' and the 'gri.xml' files to see what is in them and notice the differences.

1.3.2 Exercise : Generating a 'CTI' input file from CHEMKIN input files

Many existing reaction mechanism files are already available in a "CHEMKIN format", by which we mean the input file format developed for use with the Chemkin-II software package. Cantera comes with a useful converter utility program ck2cti (or ck2cti.py), that converts CHEMKIN format into Cantera format. This program is run from the terminal directly, to convert any CK files you plan to use with Cantera into the right '.cti' format:

```
$ ck2cti
ck2cti.py: Convert Chemkin-format mechanisms to Cantera input files (.cti)


Usage:
    ck2cti --input=<filename>
           [--thermo=<filename>]
           [--transport=<filename>]
           [--id=<phase-id>]
           [--output=<filename>]
           [--permissive]
           [-d | --debug]


Example:
    ck2cti --input=chem.inp --thermo=therm.dat --transport=tran.dat


If the output file name is not given, an output file with the same name as the
input file, with the extension changed to '.cti'.
```

```
The '--permissive' option allows certain recoverable parsing errors (e.g.
duplicate transport data) to be ignored.
```

As explained, it will produce the file chem.cti in the current directory. We will try using the ck2cti program to transform the mechanism for aromatics formation and growth in laminar premixed acetylene and ethylene flames, by Frenklach and Wang [Combustion and Flame vol. 110, pp. 173-221 (1997)]; that can be found on the folder CK2CTI in your working directory; or alternatively on the web site *http://ignis.usc.edu/Mechanisms/PAH/pah.html*. Simply type from this directory on your terminal :

```
$ ck2cti --input=mech.inp --transport=tran.dat -- thermo=therm.dat
```

Results

A new file of the right format, 'mech.cti' should have been created in your working directory. If errors occur, look at the log printed on your screen and try to modify the files accordingly. Notice that those error messages are usually very self-explanatory. You should get a printout :

```
Wrote CTI mechanism file to 'mech.cti'.
Mechanism contains 99 species and 533 reactions.
```

Remarks

The errors that you will encounter at this step will usually be syntaxing errors. The parser might have trouble with particular symbols, names or unexpected blank spaces. Be careful especially whenever you are using an "old" mechanism of yours, that might contain comments in French !

It might also happen that the names in your CHEMKIN mechanism file do not directly match those of your CHEMKIN transport or thermo data files...

2. *Tutorial 2 : All there is to know about input files*

2.1 Introduction

Cantera simulations will always involve one or more phases of matter. Depending on the calculation being performed, it may be necessary to evaluate thermodynamic properties, but also transport properties, and/or homogeneous reaction rates for the phase(s) present. In problems with multiple phases, the properties of the interfaces between phases, and the heterogeneous reaction rates at these interfaces, may also be required.

Before the properties can be evaluated, each phase must be defined, meaning that the models used to compute its properties and reaction rates must be specified, along with any parameters the models require. For example, a solid phase might be defined as being incompressible, with a specified density and composition. A gaseous phase for a combustion simulation might be defined as an ideal gas consisting of a mixture of many species that react with one another via a specified set of reactions.

If phases contain multiple species and reactions, as it is often the case in combustion application, a large amount of data is required to define it, since the contribution of each species to the thermodynamic and transport properties must be specified, and rate information must be given for each reaction. Rather than defining this information via an application program, the Cantera approach is to put the phase and interface definitions in a text file that can be called from and read by an application program - or a script.

In this tutorial, we will review what must be included in such 'data files', or 'mechanism files', and provide guidelines on how to write them to define phases and interfaces for use in Cantera simulations. We will start by a review of some basic writing rules, followed by a discussion on how they are processed, and of how errors are handled. We will work with examples in the last sections.

2.2 Data files syntax

2.2.1 Entries and directives

A data file consists of **entries** and **directives**, both of which have a syntax much like functions. An **entry** defines an object, for example, a reaction or a species. Entries have fields, that can be assigned values. Take the definition of the argon species :

```
species(name = "AR",
    atoms = " Ar:1 ",
    thermo = (
       NASA( [  300.00,  1000.00], [  2.500000000E+00,   0.000000000E+00,
                0.000000000E+00,   0.000000000E+00,   0.000000000E+00,
               -7.453750000E+02,   4.366000000E+00] ),
       NASA( [ 1000.00,  5000.00], [  2.500000000E+00,   0.000000000E+00,
                0.000000000E+00,   0.000000000E+00,   0.000000000E+00,
               -7.453750000E+02,   4.366000000E+00] )
             ),
    transport = gas_transport(
                    geom = "atom",
                    diam =     3.33,
                    well_depth =   136.50),
       )
```

Its fields are its name, atoms and thermodynamic and transport properties. The syntax is $< field\_name >=< value >$, and the fields can be specified on one line or extended across several to be read more easily, as it was the case in the previous example. Some fields are required, otherwise processing the file will abort and an error message will be printed.

As can be seen on this example, the transport field for instance is defined via another entry, "gas_transport", which in turn has several fields ('geom', 'diam', 'well_depth'). These types of entries are *embedded entries*, whereas the leftmost column entries are labeled *top-level entries*. Embedded entries often specify a model or a large group of parameters.

A **directive** will tell the code how the entry parameters are to be interpreted, such as what is the default unit system, or how certain errors should be handled. For example :

```
units(length = "cm", time = "s", quantity = "mol", act_energy = "cal/mol")
```

Specifies the unit system.

2.2.2 Cantera specific syntax rules

Units syntax rules

Many fields have numerical values that represent dimensional quantities. If somehow the units are different for some, they can always be specified :

```
pressure = 1.0e5       # default is Pascals
pressure = (1.0, 'bar') # but this is equivalent
```

as long as a few syntax rules are followed for the unit string compound:

1. Units in the denominator follow '/'
2. Units in the numerator follow '-' except the first one
3. Numerical exponents follow the unit string and must be in the range 2-6

Example :

```
A = (1.0e20, 'cm6/mol2/s') # OK


A = (1.0e20, 'cm^6/mol2/s') # error (^)
A = (1.0e20, 'cm6/mol2-s') # error (put nominator units first)
```

Fields syntax rules

Fields names may be omitted if the values are entered in the order specified in the entry declaration. It is also possible to omit only some of the fields names, as long as these fields are listed first and in order, before any named field. However, this is not advised because it can lead to errors that will be difficult to spot. For example, the first four entries below are equivalent, while the last two are incorrect :

```
element(symbol="Ar", atomic_mass=39.948) # OK
element(atomic_mass=39.948, symbol="Ar") # OK
element("Ar", atomic_mass=39.948)        # OK
element("Ar", 39.948)                    # OK


element(39.948, "Ar")                    # error
element(symbol="Ar", 39.948)             # error
```

Whenever multiple items need to be specified, for instance in a field, they must be separated with a comma and enclosed in parenthesis or square brackets.

```
s0 = (3.5, 'J/mol/K') # these are
s0 = [3.5, 'J/mol/K'] # equivalent
```

2.2.3 Supported units

The unit directive must be present to define the default unit system. Note that unit conversions are not done until the entire file has been read and that only one units directive should be present in a file. The defaults it specifies apply to the entire file. If the file does not contain a units directive, **the default units are meters, kilograms, kmoles, seconds, Joules, Kelvins and Pascals**. See for example the two following ways of defining the same interface :

```
units(length = 'cm', quantity = 'molec')
interface(name = 'Si-100',
          site_density = 1.0e15, # molecules/cm2 (default units)
          ...)


units(length = 'cm', quantity = 'mol')
interface(name = 'Si-100',
          site_density = (1.0e15, 'molec/cm2') # override default units
          ...)
```

The second version uses a different default unit system, but overrides the default units by specifying an explicit units string for the site density field. Note that since activation energies are often specified in units other than those used for thermodynamic properties, a separate field is devoted to the default units for activation energies, 'act_energy'.

```
units(length = "cm", time = "s", quantity = "mol", act_energy = "cal/mol")
 ...
#  Reaction
reaction( "CH4+1.5 O2 => CO+2 H2O", [4.9E+09, 0.0, 35500], order="CH4:0.5 O2:0.65")
```

Notice in that last example, that the only explicitly stated field name of the reaction entry is the last one, consistently with what was discussed in the section 2.2.2.

2.3 Summary: General structure of a data file

Now that we have covered how to write syntactically-correct entries, directives, units and other fields composing a data file, we can focus on the content of a physically-correct data file. Hereafter is provided the general skeleton of a data file :

field	Supported Units
length	'cm', 'm', 'mm'
mass	'kg'
quantity	'mol', 'kmol', 'molec'
time	's', 'min', 'hr', 'ms'
energy	'J', 'kJ', 'cal', 'kcal'
act_energy	'kJ/mol', 'J/mol', 'J/kmol', 'kcal/mol', 'cal/mol', 'eV', 'K'
pressure	'Pa', 'atm', 'bar'

Table 2.1: Allowed units in Cantera

```
#
# This is a mechanism.cti file to be used in Cantera applications

UNITS DIRECTIVE

#-------------------------------------------------------------------------------
#  Phase & Interface data
#-------------------------------------------------------------------------------

PHASE ENTRIES(name, elements, species, reactions,
             kinetics, transport, initial_state*, options)

INTERFACE ENTRIES(name, elements, species, reactions,
                 phases, site_density, initial_state*)

#-------------------------------------------------------------------------------
#  Species & Elements data
#-------------------------------------------------------------------------------

ELEMENT ENTRIES(symbol, atomic_mass)

SPECIES ENTRIES(name, atoms, thermo ,transport,
               note, size, charge)
```

```
#-------------------------------------------------------------------------------
#  Reaction data
#-------------------------------------------------------------------------------

REACTION ENTRIES(equation, rate_coeff, id, options)
THREE BODY REACTIONS ENTRIES(equation, rate_coeff,
                             efficiencies, id, options)
FALLOFF REACTIONS ENTRIES(equation, rate_coeff_inf, rate_coeff_0,
                          efficiencies, falloff, id, options)
OTHER TYPES OF ENTRIES ...
```

Of course, each entry has several different possible types and fields associated, a summary of which is provided in Appendix A.

2.4 Exercises

The goal of the first two exercises of this section is to get acquainted with the syntax of the data files, and to be able to read and understand error logs so as to modify the '.cti' in consequence. The last exercise is optional, and will guide you through the generation of a special kind of data file; specifically, 'sub' data files of existing data files.

2.4.1 Modify the CTI file and run your first script !

Go into the subdirectory 'SANDIEGO' of the directory 'Tuto2', where you will find a python script, 'h2_flame.py'. *This script will compute several standard H_2/Air flames in a row, so as to obtain the evolution of the laminar flame speed with the equivalence ratio. Now, you do not need to be concerned with what is in the script at this point, just know that it calls for the file 'h2_sandiego.cti', also provided in this directory.* Try to run the python script from there on your terminal, with the command previously introduced :

```
$ python h2_flame.py
```

A bunch of errors should appear on your screen. Try to understand them, and modify your '.cti' file accordingly, by looking at the information about *entries* and *fields* provided in Appendix A if necessary.

Results

You will find the list of errors to modify in this '.cti' file in Appendix C.1.1. Note that if you don't find all the errors in the mechanism file, you can also generate it with the procedure 'CK2CTI' explained in the first tutorial, section 1.3.2, and the files that can be found in the current subdirectory 'H2_sanDiego'.

In this exercice, the '.cti' was not viable because of syntax errors, that Cantera helped you spot in the file through a specific error printout :

```
get_CTML_Tree: caught an exception:

***************************************************************************
CanteraError thrown by ct2ctml:
...
```

Once you have a viable '.cti', the python script should generate several '.csv' files and a '.png' in your working directory. The image displays the evolution of the H_2/Air laminar flame speed with the equivalence ratio.

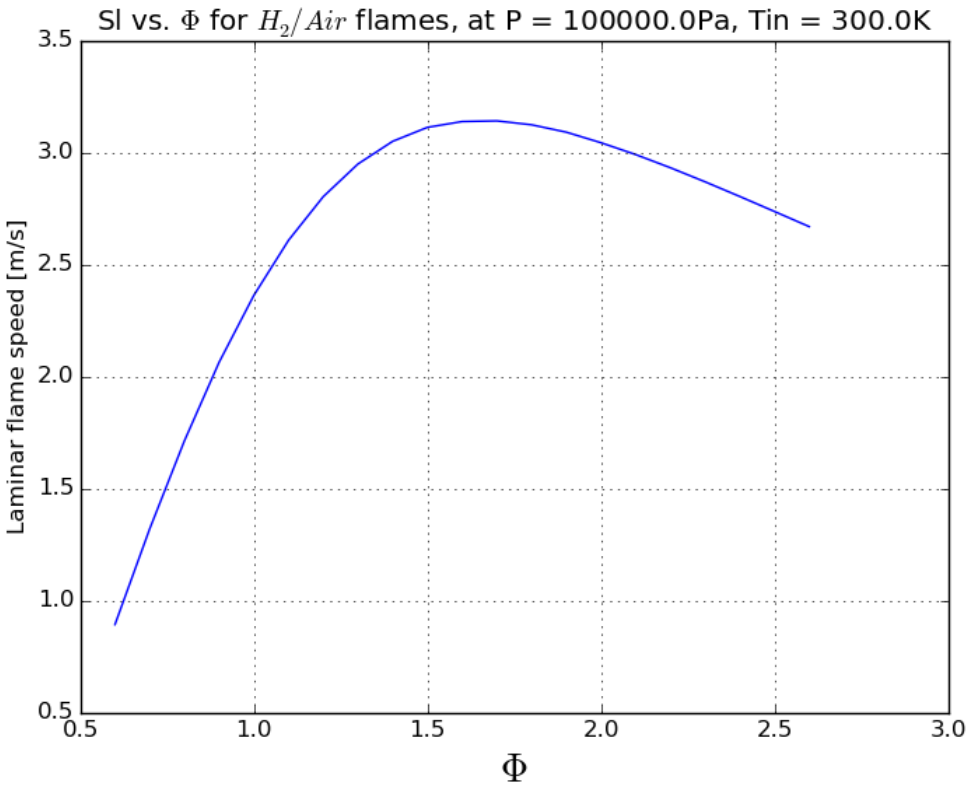

Figure 2.1: Adiabatic flame temperature for different equivalence ratios of H_2/Air mixture, computed with the mechanism of San Diego, version of 2011 and shortened as in Petrova and Williams, C&F 2005.

2.4.2 A second example

Now, go into the subdirectory 'SANDIEGO2' of 'Tuto2', and try to run the python script that you will find there. The simulation is exactly the same as in the previous exercise. Can you explain what went wrong ?

Results

Here, the data file was in fact viable, and the parser did not have any trouble generating the '.xml' file Cantera requires. However, the data file was generated from CHEMKIN's mechanisms and thermo data files only... no transport data was provided ! It is perfectly fine to generate that kind of data files, if you only need to perform 0D or pseudo 0D simulations; however, the script 'h2_flame.py' simulated a series of 1D freely propagating premixed laminar flames.

2.4.3 Generate submechanisms with a special processing option (OPTIONAL)

Go into the subdirectory 'submechanisms' of 'Tuto2'. *An interesting feature of the options field -see the paragraph pertaining to phases and interfaces in Appendix A, is that it is possible to readily extract a sub-mechanism from a larger reaction mechanism.* Using the special 'option' field of a phase entry (see A.2), and the mechanism file 'gri30.cti' that you will find there, try to :

1. import an H_2/Air oxidation submechanism
2. generate a NOx-less submechanism for CH_4/Air oxidation

Do do so, the best way to proceed is to create a new 'submechanisms.cti' file :

```
$ touch submechanisms.cti
```

in the same folder than the 'gri30.cti' file. Next, open it in your favorite text editor and create two new phase entries, of type 'ideal_gas'. You will need to fill out the various fields, using Appendix A, again. Remember to use species and reactions information from the 'gri30.cti' file !

Next, you will need to import your new mechanisms via a script, like in section 1.2.1 of the first tutorial or directly in your terminal via the Python interface, like in section 1.2.2 of the first tutorial... it's up to you ! Try printing the state of the gas, the number of species, specific reactions equations ... This can be done via a script, with the lines:

```
gas()
gas.n_species
gas.reaction_equation(indexOfReaction)
...
```

You can go online at *http://www.cantera.org/docs/sphinx/html/index.html* to search for more attributes to look at.

Results

The 'submechanisms.cti' data file is provided in section C.1.2, along with the gas printout that each phase will produce, when imported in the same fashion as in section 1.2.1 of the first tutorial.

3. *Tutorial 3 : Equilibrium calculations*

3.1 Equilibrium calculations

3.1.1 Introduction

In this tutorial, we will tackle equilibrium calculations. Equilibrium calculations are usually performed to obtain the adiabatic flame temperature, equilibrium composition, and thermodynamic state of a specific mixture under given conditions; but are performed in virtually every simulation. For example, Cantera will call its equilibrium solver to initialize the gas state before trying to obtain a solution to the equations for a free flame. As such, it is interesting to understand how Cantera proceeds.

There a 2 different types of solver currently implemented for equilibrium calculation in Cantera that deserves our attention.

The element potential 'ChemEquil' solver

The class ChemEquil implements a chemical equilibrium solver for single-phase solutions. It is a "non-stoichiometric" solver in the terminology of Smith and Missen, meaning that every intermediate state is a valid chemical equilibrium state, but does not necessarily satisfy the element constraints. Non-stoichiometric methods are faster when they converge, but stoichiometric ones tend to be more robust.

The 'VCS' chemical equilibrium solver

The other type of solver is designed to be used to set a mixture containing one or more phases to a state of chemical equilibrium. It uses a "stoichiometric" algorithm, in which each intermediate state satisfies the element constraints but is not a state of chemical equilibrium. More specifically, it implements the VCS algorithm, described in Smith and Missen, "Chemical Reaction Equilibrium". It finds a set of component species and a complete set of formation reactions for the non-components in terms of the components.

As expected, the ChemEquil solver is the fastest of the Cantera equilibrium solvers for many single-phase equilibrium problems (particularly if there are only a few elements but very many species), but can be less stable. Problem situations include low temperatures where only a few species have non-zero mole fractions, precisely stoichiometric compositions (we will see an example shortly). In general, if speed is important, this solver should always be tried first before falling back to another one in case of failure. The default setting in Cantera, when launching an equilibrium calculation without specifying the solver, is to try the 'element_potential' before falling back to another vcs solver labelled 'gibbs' :

```
gas.equilibrate('TP')
```

Now, to compute an equilibrium state, two independant intensive properties of the system have to be specified. These solver can only handle chemical equilibrium at a specified temperature and pressure. To compute equilibrium holding other properties fixed, it is necessary to iterate on T and P in an "outer" loop, until the specified properties have the desired values. Don't worry, it is done internally; although there might come a time, if you use the 'gibbs' solver, where you would need to increase the number of T loops for convergence. This is done through a specific field, 'maxiter' :

```
gas.equilibrate('HP', maxiter = 1500)
```

3.1.2 The equilibrate function

The equilibrate function can be applied on a single phase or on a mixture. Here, we recall its definition:

```
equilibrate(self, XY, solver, double rtol, int maxsteps, int maxiter, int loglevel)

Parameters:
XY      - A two-letter string, which must be one of the set: ['TP','TV','HP','SP','SV','UV']
solver  - Specifies the equilibrium solver to use. May be one of the following:
element_potential - A fast solver using the element potential method
gibbs - A slower but more robust Gibbs minimization solver
vcs   - The VCS non-ideal equilibrium solver
auto  - The element potential solver will be tried first, then if it fails
the gibbs solver will be tried
rtol        - The relative error tolerance.
maxsteps    - Maximum number of steps in composition to take to find a converged solution.
```

```
maxiter     - This specifies the number of outer iterations on T or P when some
property pair other than TP is specified (only for 'gibbs')
loglevel    - Is currently deprecated
```

3.2 Exercises

3.2.1 Generate your first equilibrium script !

Go into the directory 'Tuto3' and recall the procedure from the first tutorial: start by creating a new Python script 'equil_simple.py' in the text editor of your choice:

```
$ touch equil_simple.py
$ gedit equil_simple.py
```

Import the cantera python module at the top of your script :

```
import cantera as ct
```

Recall that with this method, a 'ct.' will have to be added at the beginning of each cantera function and objects. Try to design a script that will calculate the equilibrium composition of a methane/air mixture, with the GRIMech3.0 mechanism we used previously, under atmospheric condition (1 bar, 300 K) and at stoechiometry ($CH_4 + 2O_2 <=> 2H_2O + CO_2$). Everything you need was discussed in the previous tutorials. You will need to do the following :

- Import your gas and set its initial state. Note that to set the gaseous composition, you could pass a string of mole fractions, for the fuel and the oxidizer. This should be of the form 'species1:mole fraction1, species2:mole fraction2, ...'.

  ```
  gas = ...
  gas.TPX = 300.0, ..., 'CH4:0.5,...'
  ```

- Call the equilibrate function on your gas object, specifying that the temperature and pressure stay constant for now. See section 3.1.2 to get acquainted with the equilibrate function.

  ```
  gas.equilibrate(...)
  ```

- Check your gas state, as we did several times in the first tutorial.

```
print gas()
```

Once your script is ready, launch it with the python command :

```
$ python equil_simple.py
```

Results

The script 'equil_simple.py' is provided in section C.2.1. You should get a gas printout that looks like this:

```
  gri30:

      temperature             300  K
         pressure          100000  Pa
          density         1.10784  kg/m^3
 mean mol. weight         27.6332  amu

                       1 kg            1 kmol
                  -----------      ------------
         enthalpy    -3.01529e+06       -8.332e+07     J
  internal energy    -3.10555e+06       -8.582e+07     J
          entropy         7233.97        1.999e+05     J/K
   Gibbs function    -5.18548e+06       -1.433e+08     J
 heat capacity c_p          1111.3        3.071e+04     J/K
 heat capacity c_v         810.419        2.239e+04     J/K

                        X                Y          Chem. Pot. / RT
                  -------------     ------------     ------------
                H2    6.28376e-32      4.58408e-33         -87.5752
                 H    1.05969e-51      3.86527e-53         -43.7876
                 O    2.36189e-50      1.36751e-50         -33.7583
                O2    2.50759e-19      2.90375e-19         -67.5167
                OH    1.18723e-31      7.30699e-32         -77.5459
               H2O       0.190114         0.123943         -121.334
               HO2    2.80815e-39      3.35421e-39         -111.304
```

H2O2	3.61855e-32	4.45419e-32	-155.092
C	1.24469e-168	5.41014e-169	-118.323
CH	1.44411e-165	6.8037e-166	-162.11
CH2	2.69558e-148	1.3683e-148	-205.898
CH2(S)	4.16871e-155	2.11607e-155	-205.898
CH3	1.33507e-124	7.26389e-125	-249.686
CH4	1.8856e-105	1.0947e-105	-293.473
CO	3.36172e-37	3.40761e-37	-152.081
CO2	0.095057	0.151392	-185.839
HCO	2.21913e-81	2.33037e-81	-195.869
CH2O	1.80343e-74	1.9596e-74	-239.656
CH2OH	1.5675e-108	1.76042e-108	-283.444
CH3O	1.00808e-114	1.13215e-114	-283.444
CH3OH	2.52861e-95	2.93205e-95	-327.232
C2H	5.96825e-210	5.40598e-210	-280.433
C2H2	9.13867e-171	8.61107e-171	-324.221
C2H3	1.62444e-200	1.58991e-200	-368.008
C2H4	3.02119e-177	3.06717e-177	-411.796
C2H5	2.49148e-206	2.62027e-206	-455.584
C2H6	5.05956e-191	5.50565e-191	-499.371
HCCO	3.55626e-155	5.28027e-155	-314.191
CH2CO	3.00039e-135	4.56437e-135	-357.979
HCCOH	1.60847e-157	2.44689e-157	-357.979
N	4.33568e-80	2.19767e-80	-11.691
NH	1.71484e-77	9.3177e-78	-55.4786
NH2	4.38732e-67	2.54391e-67	-99.2662
NH3	8.68854e-45	5.35481e-45	-143.054
NNH	1.31024e-61	1.37606e-61	-67.1696
NO	2.4268e-25	2.63519e-25	-45.4493
NO2	1.57692e-28	2.62536e-28	-79.2077
N2O	2.9812e-28	4.74831e-28	-57.1403
HNO	1.94556e-46	2.18359e-46	-89.2369
CN	5.57971e-123	5.25351e-123	-130.014
HCN	1.95092e-88	1.90802e-88	-173.801

H2CN	1.4383e-126	1.45914e-126	-217.589
HCNN	1.12485e-148	1.67029e-148	-185.492
HCNO	5.77247e-108	8.98775e-108	-207.56
HOCN	3.67232e-76	5.71781e-76	-207.56
HNCO	9.92424e-58	1.54521e-57	-207.56
NCO	1.17871e-82	1.79226e-82	-163.772
N2	0.714829	0.724665	-23.382
AR	0	0	
C3H7	0	0	
C3H8	0	0	
CH2CHO	1.37634e-165	2.14397e-165	-401.767
CH3CHO	1.68509e-151	2.68638e-151	-445.554

3.2.2 The 'element_potential' solver

Go into the 'Tuto3' subdirectory 'ChemEquil', and try to launch the python script it contains :

```
$ python ChemEquil.py
```

You will notice a warning on your screen :

```
ChemEquil solver failed! Try the vcs solver...
```

Here, the ChemEquil solver previously introduced, that uses the chemical potentials, fails. This often arises when only a few species have non-zero mole fractions, as it is the case in this example. Open the file with your favorite text editor :

```
$ gedit ChemEquil.py
```

uncomment the line :

```
#gas.equilibrate('TP', solver = 'vcs', maxsteps = 1500)
```

by removing the # at the beginning, and comment the line :

```
exit()
```

by adding a # at the beginning (see Appendix B). Save your file, and try re-running the script with the previous python command... Now it works !

Results

In this script, we forced Cantera to only try the 'element_potential' solver, which failed. Usually, as said previously, if you do not specify which solver you want to use, Cantera will try two different methods. The cases when both those method fail are relatively rare; this is discussed in more depth in section 3.3.

The "Comparison between Chem potentials and element potentials" that appears on screen is a good method to check whether the equilibrium state is reached. Indeed, recall from this morning that when in equilibrium, the chemical potentials for each species should be equal to the sum of the element potentials of the elements it contains; so for example, the chemical potential of H_2 should equal twice the chemical potential of H. Notice that the two ways of calculating the chemical potentials for the major species involved coincides, when we re-launch the script using the 'vcs' solver :

```
 Comparison between Chem potentials and element potentials:


mu_H2   :  -2.2012e+08 ,     -2.2012e+08
mu_O2   :  -2.0667e+08 ,     -2.0667e+08
mu_OH   :  -2.1339e+08 ,     -2.1339e+08
mu_H2O  :  -3.2345e+08 ,     -3.2345e+08
```

You will also notice that a '.csv' has been created in your working directory. This file contains all equilibrium mole fractions. The mole fractions where saved in the script by invoking the gas attribute 'X' on line 113 of your script (see Appendix B for more information) :

```
xeq = np.zeros(gas.n_species)
xeq[:] = gas.X
```

3.2.3 Compare different solvers (OPTIONAL)

Now you can go into the folder 'SolverComparison', where you will find another python script, 'SolverComparison.py'. Open it, and notice that you are given the choice of which solver to use :

```
#Equilibrium:
#gas.equilibrate('TP', solver = 'element_potential', maxsteps = 1500, loglevel = 50)
gas.equilibrate('TP', solver = 'vcs', maxsteps = 1500)
#gas.equilibrate('TP', solver = 'gibbs', maxsteps = 1500)


...
```

```
# Save the state after the simulation
#csv_file = 'all_mole_fractions_ChemEquil.csv'
csv_file = 'all_mole_fractions_VCS.csv'
#csv_file = 'all_mole_fractions_Gibbs.csv'
```

Uncomment whichever you want to use, and run the script addind the keyword 'time' in front of your command :

```
$ time python equil_phi_cte.py
```

Try it for all the solvers.

You should notice that the 'element_potential' method is the fastest one. Notice also that the chemical potential balance is verified each time, but that their values can slightly differ from one method to the other. You could compare the '.csv' files, and see that the mole fractions are almost similar in all cases.

3.2.4 Perform adiabatic flame calculations

Introduction

We will now perform several constant pressure equilibrium calculations of an ethylene/air mixture, at an initial temperature of 300K and under atmospheric pressure, in order to obtain the adiabatic flame temperature and burnt gas state for several equivalence ratios. The goal of this exercise is to find a way to loop through several gaseous composition, in order to perform several computations in a single script; and to learn how to properly store the results in a 'csv' file.

Generate the adiabatic flame temperature of an ethylene/air mixture

We will use the scheme from the first tutorial, that can be found in the subdirectory 'CK2CTI' of 'Tuto1'; so you should already have a viable 'mech.cti'. Go into the folder 'AdiabaticFlameT' and copy your mechanism file here. In this folder, you will find a python script, 'AdiabaticFlameT.py'. Open it, it should look like this :

```
"""
Equilibrium calculation : application to the computation of adiabatic flame temperature
and equilibrium composition for a fuel/air mixture as a function of equivalence ratio.
```

```
"""

import cantera as ct
import numpy as np
import sys
import csv

#############################################################################
# Edit these parameters to change the initial temperature, the pressure, and
# the phases in the mixture.

# Import the gas :
gas = ct.Solution('mech.cti')

# Choose the equivalence ratio range range :
phi_min = ...
phi_max = ...
npoints = ...

# Set the gas composition :
T = 300.0
P = 101325.0

# find fuel, nitrogen, and oxygen indices
# to set the composition of the gas
fuel_species = 'C2H4'
ifuel = gas.species_index(fuel_species)
io2 = gas.species_index('O2')
in2 = gas.species_index('N2')

#############################################################################

# Create some arrays to hold the data with numpy
#1D arrays : phi, t_adiabatic
```

```
phi = np.zeros(npoints)
tad = ...

#2D arrays :
xeq = np.zeros((gas.n_species,npoints))

# Start the loop on Phi :
for i in range(npoints):
        # Start by computing the composition of the gas
        #whenever you are working with mole fractions, recall that
        #phi=(X_fuel/X_oxi)*(\nu_oxi/\nu_fuel)
        #and that the air_N2_O2_molar_ratio = 3.76,
        air_N2_O2_molar_ratio = 3.76
        #this helps you set the gas state
        phi[i] = phi_min + (phi_max - phi_min)*i/(npoints - 1)
        #dont forget to create an array to hold the composition
        X = np.zeros(gas.n_species)
        #then to inizialise it
        X[ifuel] = ...
        X[io2]   =  ...
        X[in2]   = X[io2]*air_N2_O2_ratio
        gas.TPX = T, P, X

        # Equilibrate the mixture adiabatically at constant P
        #with the solver vcs
        gas.equilibrate(...)

        # Save the adiabatic temperature each time
        tad[i] = gas.T
        #you can even print it on screen
        #print "At phi = ","%10.4f"% (phi[i])+ "  Tad = ","%10.4f"% (tad[i])

        # You could also save the equilibrium mass fractions at each
        #phi, as long as you initialized an array properly:
```

```
        xeq[:,i] = gas.X



# Save your results in a CSV file
csv_file = 'yourfile.csv'
with open(csv_file, 'w') as outfile:
     writer = csv.writer(outfile)
     writer.writerow(['Phi','T (K)'] + gas.species_names)
     for i in range(npoints):
             writer.writerow([phi[i], tad[i]] + list(xeq[:,i]))
print "Output written to", "%s"%(csv_file)
```

Modify it so that it runs a series of constant pressure equilibrium calculation for 50 equivalence ratios, in between 0.3 and 3.5, for example. This is done by changing the settings in the 'Choose the equivalence ratio range' part of this script, by setting the gaseous composition of each loop, and by calling the 'equilibrate' function with the right settings.

Next, don't forget to initialize arrays to store your data and results ! This is done in the 'Create some arrays to hold the data with numpy' part of this script.

Uncomment the line :

```
#print "At phi = ","%10.4f"% (phi[i])+ "  Tad = ","%10.4f"% (tad[i])
```

by removing the #, so that the adiabatic temperature for each gaseous composition gets printed on your screen.

Notice that the part of the script that will write your results in a 'csv' file is already provided, at the end of the script, and that it makes use of a 'csv.writer' function. This function is included in the 'csv' package you imported at the beginning of your file (see B.1.3 for more information about packages).

Results

The complete script is provided in section C.2.2. You should see something like this appear on your terminal screen :

```
**** WARNING ****
For species C2H3O, discontinuity in cp/R detected at Tmid = 1000
Value computed using low-temperature polynomial:  26.0651.
```

```
Value computed using high-temperature polynomial: 11.7479.


**** WARNING ****
For species C2H3O, discontinuity in s/R detected at Tmid = 1000
Value computed using low-temperature polynomial:  47.9078.
Value computed using high-temperature polynomial: 43.1354.


**** WARNING ****
For species C4H612, discontinuity in cp/R detected at Tmid = 1000
Value computed using low-temperature polynomial:  20.128.
Value computed using high-temperature polynomial: 20.1809.
At phi =      0.3000  Tad =   1142.8444
At phi =      0.3653  Tad =   1300.2147
At phi =      0.4306  Tad =   1450.9491
At phi =      0.4959  Tad =   1595.5514
At phi =      0.5612  Tad =   1734.1500
At phi =      0.6265  Tad =   1866.3163
At phi =      0.6918  Tad =   1990.7248
At phi =      0.7571  Tad =   2104.7918
At phi =      0.8224  Tad =   2204.7840
At phi =      0.8878  Tad =   2286.8441
At phi =      0.9531  Tad =   2348.1574
At phi =      1.0184  Tad =   2386.8245
At phi =      1.0837  Tad =   2401.1648
At phi =      1.1490  Tad =   2391.6955
At phi =      1.2143  Tad =   2364.6792
At phi =      1.2796  Tad =   2328.6934
At phi =      1.3449  Tad =   2289.2477
At phi =      1.4102  Tad =   2248.8200
At phi =      1.4755  Tad =   2208.4206
At phi =      1.5408  Tad =   2168.4585
At phi =      1.6061  Tad =   2129.0954
```

```
At phi =      1.6714  Tad =   2090.3865
At phi =      1.7367  Tad =   2052.3407
At phi =      1.8020  Tad =   2014.9470
At phi =      1.8673  Tad =   1978.1863
At phi =      1.9327  Tad =   1942.0373
At phi =      1.9980  Tad =   1906.4786
At phi =      2.0633  Tad =   1871.4897
At phi =      2.1286  Tad =   1837.0511
At phi =      2.1939  Tad =   1803.1442
At phi =      2.2592  Tad =   1769.7512
At phi =      2.3245  Tad =   1736.8547
At phi =      2.3898  Tad =   1704.4377
At phi =      2.4551  Tad =   1672.4831
At phi =      2.5204  Tad =   1640.9741
At phi =      2.5857  Tad =   1609.8934
At phi =      2.6510  Tad =   1579.2237
At phi =      2.7163  Tad =   1548.9477
At phi =      2.7816  Tad =   1519.0479
At phi =      2.8469  Tad =   1489.5080
At phi =      2.9122  Tad =   1460.3204
At phi =      2.9776  Tad =   1431.6155
At phi =      3.0429  Tad =   1426.1341
At phi =      3.1082  Tad =   1427.0860
At phi =      3.1735  Tad =   1424.6954
At phi =      3.2388  Tad =   1420.8400
At phi =      3.3041  Tad =   1416.7760
At phi =      3.3694  Tad =   1412.7750
At phi =      3.4347  Tad =   1408.8670
At phi =      3.5000  Tad =   1405.0536
Output written to adiabatic.csv
```

That last line was generated by the last python command :

```
print "Output written to", "%s"%(csv_file)
```

and it is nice to add it so that it reminds you of the name of the '.csv' file that should have been generated in your working directory. Check to see if it was the case ! This '.csv' file should contain

the list of phi's, temperatures, and species moles fractions after each equilibrium calculation :

```
phi,T (K),H,OH,HO2,...,P2-,P2-H
0.3,1142,1.28270997e-12,7.7304907e-07,3.33743472e-09,...,0.0,0.0
...
3.5,1405,2.62368651e-06,8.24390443e-13,3.97557527e-26,...,3.41478641e-13,1.37145940e-16
```

To go further ...

You could generate plots to see the evolution of the temperature, with the package 'matplotlib'. Simply add a dedicated subsection at the end of your script :

```
import matplotlib.pyplot as plt
plt.plot(phi, tad)
plt.xlabel('Equivalence ratio')
plt.ylabel('Adiabatic flame temperature [K]')
plt.show()
```

From the way that matplotlib was imported, the 'plt' in front of each function tells Python that the functions come from the matplotlib package. See B.1.3 for further information on the meaning of each line. You should obtain fig. 3.1.

Now, if you saved the mole or mass fractions as well, you could also plot the mole or mass fractions evolutions of the most important species :

```
# Add a plot option
if '--plot' in sys.argv:
    import matplotlib.pyplot as plt
    for i, cas in enumerate(gas.species_names):
        if cas in ['O2','CO2','CO']:
        plt.plot(phi,xeq[i,:], label = cas)
    plt.xlabel('Equivalence ratio')
    plt.ylabel('Mole fractions')
    plt.legend(loc='best')
    plt.show()
```

Now, this more elaborated plot will be drawn only if the python script is launched with the '–plot' option, as such:

```
$ python AdiabaticFlameT.py --plot
```

Here, the mass fraction evolution of species 'O_2', 'CO_2' and 'CO' will be plotted against the equivalence ratio. The 'label' option in line :

```
plt.plot(phi,xeq[i,:], label = cas)
```

Will generate a legend for the different curves, here taken to be the name (iteration with 'cas' on species_names) of each species plotted. The legend localization will be chosen by Python, as specified by the line:

```
plt.legend(loc='best')
```

You should obtain the plot of fig. 3.2.

Conclusion

We have just generated the skeleton of a script to perform a series of common equilibrium calculations; to obtain the constant pressure equilibrium composition of a fuel/air mixture. Starting from there, you could modify your initial conditions, plot the mole/mass fractions of other species, change the solver or even try another fuel (methane, acetylene) without changing your mechanism.

Technically, adiabatic flame calculations could also be performed at constant volume: simply invoke the good equilibrate option of your equilibrate function, 'UV' (see 3.1.2), in your script.

3.3 About convergence, failures of equilibrium calculations

Convergence-wise, you will not encounter many issues, as the equilibrium solvers are quite robust. It is worth noting however, that a number of bugs have been reported on the Cantera community about the 'element_potential' method. Sometimes, it can fail when there is no reason, and changing the order of definition of the species will solve the problem. Also, the fallback in 'auto' mode can crash and Cantera can tell you that no solution was found when in fact it did not even try another solver besides the 'element_potential'. In those cases, simply switch to the 'vcs' solver and usually, it will work just fine. For an exemple of such a fallback issue, you could modify the first script we will use in this tutorial, 'ChemEquil.py' in folder 'ChemEquil'. Simply replace line :

```
gas.equilibrate('TP', solver = 'vcs')
```

by :

```
gas.equilibrate('TP')
```

And see what happens when you run it. Problems have a tendency to arise at stoichimetry and low temperature. Cantera is not perfect, yet !

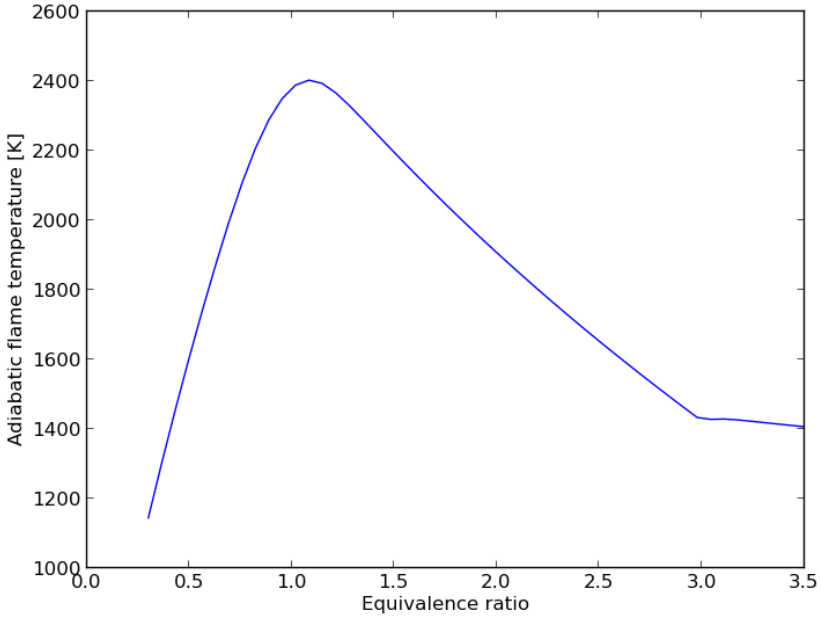

Figure 3.1: C_2H_4/Air adiabatic flame temperature for different equivalence ratios, computed with the mechanism of aromatics formation and growth in laminar premixed acetylene and ethylene flames, by Wang and Frenklach.

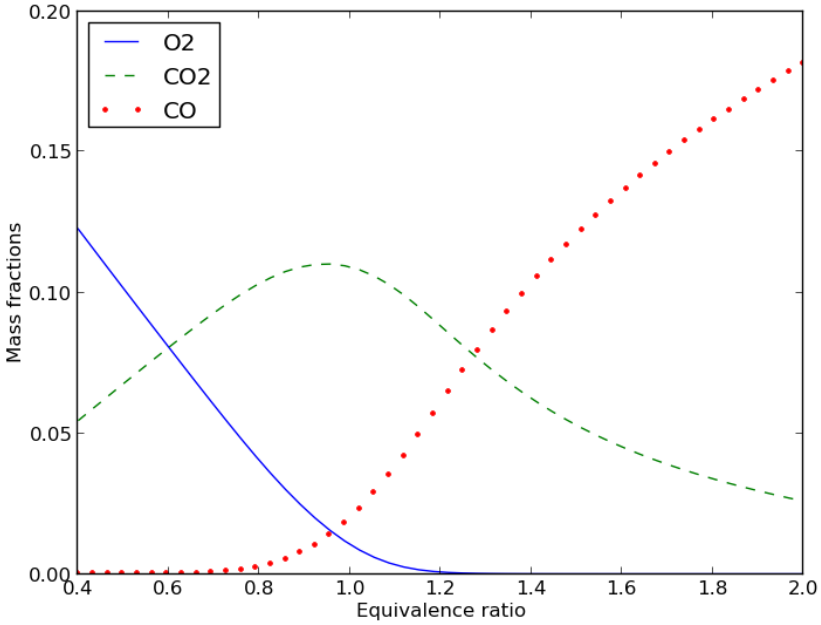

Figure 3.2: Equilibrium composition of major species for different equivalence ratios of C_2H_4/Air mixture, computed with the mechanism of aromatics formation and growth in laminar premixed acetylene and ethylene flames, by Wang and Frenklach.

4. *Tutorial 4 : Reactor simulations*

4.1 The Reactor object with Cantera

4.1.1 Introduction

This tutorial is dedicated to the simulation of various homogeneous zero-dimensional reactors. A Cantera reactor can be viewed as box reduced to a single point, such that there is no spatial evolution but only a temporal evolution of the quantities it contains. Such reactors can be useful, for example, to determine the auto-ignition timing of a mixture, or can be viewed as building blocks of more complex simulations as we will see shortly in the exercises section.

Basically, in Cantera, a reactor is composed of an object from the **"ReactorBase" class**, that describes the vessel, and an object from the **"FlowDevice" class** to describe the flow across or in between several reactors. By default, reactors are closed (no inlets or outlets), have fixed volume, and have adiabatic, chemically-inert walls. These properties may all be changed by adding appropriate components, as we will see shortly. FlowDevice objects are assumed to be adiabatic, non-reactive, and have negligible internal volume, so that they are internally always in steady-state even if the upstream and downstream reactors are not. The fluid enthalpy, chemical composition, and mass flow rate are constant everywhere, and the pressure difference equals the difference in pressure between the upstream and downstream reactors.

4.1.2 The ReactorBase class

Here are the properties of the common base class for reactors and reservoirs "ReactorBase" :

```
ReactorBase(ThermoPhase contents, name=None, arguments)
```

The ThermoPhase contents should be the gaseous object that fills the ReactorBase object. The name is optional, it could be useful in case several reactors are needed in a same simulation. The arguments depend upon the type of reactor considered. The different types are :

1. *Reservoir* : reactors with a constant state. The temperature, pressure, and chemical composition in a reservoir never change from their initial values.

2. *Reactor*

3. *IdealGasReactor* : constant volume, zero-dimensional reactor for ideal gas mixtures.

4. *ConstPressureReactor* : homogeneous, constant pressure, zero-dimensional reactor. The volume of the reactor changes as a function of time in order to keep the pressure constant.

5. *IdealGasConstPressureReactor* : homogeneous, constant pressure, zero-dimensional reactor for ideal gas mixtures. The volume of the reactor changes as a function of time in order to keep the pressure constant.

6. *FlowReactor* : a steady-state plug flow reactor with constant cross sectional area. Time integration follows a fluid element along the length of the reactor. The reactor is assumed to be frictionless and adiabatic.

4.1.3 The FlowDevice class

As previously stated, reactors can be added inlets and outlets, to be able to loop or chain several reactors, control mass flow rates, volume or pressure. Here are the properties of the common base class "FlowDevice" for flow controllers between reactors and reservoirs :

```
FlowDevice(upstream, downstream, name=None, arguments)
```

The FlowDevice controls the fluids passage between an upstream and a downstream object, which should be specified. The arguments depend upon the different types of flow controllers available in Cantera :

1. *MassFlowController* : mass flow controllers maintain a specified mass flow rate independent of upstream and downstream conditions. Unlike a real mass flow controller, a MassFlowController object will maintain the flow even if the downstream pressure is greater than the upstream pressure.

2. *Valve* : valves are flow devices with mass flow rate that is a function of the pressure drop across them. Valve objects are often used between an upstream reactor and a downstream reactor or reservoir to maintain them both at nearly the same pressure. By setting the constant Kv to a sufficiently large value, very small pressure differences will result in flow between the reactors that counteracts the pressure difference.

3. *PressureController* : a PressureController is designed to be used in conjunction, typically, with a MassFlowController. The master flow controller is installed on the inlet of the reactor, and the corresponding PressureController is installed on the outlet of the reactor. The PressureController

mass flow rate is equal to the master mass flow rate, plus a small correction dependent on the pressure difference.

4.1.4 Walls

Reactors can also be added heat transfer and heterogeneous reactions at the walls, through a special object "wall". Walls separate two reactors, or a reactor and a reservoir. A wall has a finite area, may conduct or radiate heat between the two reactors on either side, and may move like a piston. They are stateless objects in Cantera, meaning that no differential equation is integrated to determine any wall property. A heterogeneous reaction mechanism may be specified for one or both of the wall surfaces.

4.1.5 Time evolution inside a Reactor

Now, the evolution of a reactor, or a network of reactors -which are 0-D objects- with time, is performed through an integrator object "ReactorNet" :

```
... (define gases) ...
r1              = Reactor(gas1)
r2              = Reactor(gas2)
... (install walls, inlets, outlets, etc)...
reactor_network = ReactorNet([r1, r2])
time            = 1 #s
reactor_network.advance(time)
```

There are two possibilities to advance a reactor simulation in time : the 'advance(self, double t)' option (shown in the example above) or the 'step(self, double t)' option. The first one will advance the state of the reactor network in time from the current time to the specified 't' time, taking as many integrator timesteps as necessary; whereas the second one will take a single internal time step toward the specified time 't' [s]... The use of 'advance' is recommended, unless you need to elucidate a bug.

4.2 Exercises

4.2.1 A simple closed vessel

In this section, we will start by designing a very simple constant volume reactor, to try to catch the autoignition time of a methane/air mixture, with the GRIMech3.0 mechanism. Go into the subdirectory 'ClosedVessel' in the directory 'Tuto4', with your terminal. You will find a script 'reactorUV.py', partially filled :

```
"""
Constant-volume, adiabatic kinetics simulation.
"""
import sys
import numpy as np
import cantera as ct

#############################################################################

#Mechanism used for the process
gas = ct.Solution('gri30.xml')

#Gas state
gas.TPX = 1000.0, ct.one_atm, 'CH4:0.5,O2:1,N2:3.76'

#Create an ideal gas Reactor and fill with gas
#r = ...

#Prepare the simulation with a ReactorNet object
sim = ct.ReactorNet([r])
time = 4.e-1

#Arrays to hold the datas
times = np.zeros(200)
data = np.zeros((200,4))

#Advance the simulation in time
#and print the internal evolution of temperature, volume and internal energy
print('%10s %10s %10s %14s' % ('t [s]','T [K]','vol [m3]','u [J/kg]'))
for n in range(200):
    time += 5.e-3
    sim.advance(time)
    times[n] = time  # time in s
    data[n,0] = r.T
```

```
    data[n,1:] = r.thermo['O2','H','CH4'].X
    print('%10.3e %10.3f %10.3f %14.6e' % (sim.time, r.T,
                                           r.thermo.v, r.thermo.u))

# Plot the results if matplotlib is installed.
# ... plot options ...
```

Complete it to get a constant volume ideal gas reactor (see section 4.1.2) and run it with the plot option :

```
$ python reactorUV.py --plot
```

Results

You should get a gas printout that looks like this :

```
     t [s]      T [K]   vol [m3]       u [J/kg]
 4.050e-01   1001.112      2.970   2.870655e+05
 4.100e-01   1001.140      2.970   2.870655e+05
 4.150e-01   1001.168      2.970   2.870655e+05
....
 1.395e+00   2768.160      2.970   2.870655e+05
 1.400e+00   2768.160      2.970   2.870655e+05
```

Notice that the internal volume and energy stay constant, as required by the method used. The evolution of the temperature and of the most important species' mole fractions are displayed in fig. 4.1. As the reaction proceeds, we see the temperature's sharp increase and the fuel mass fraction decay. Usually, you will need to refine the time steps, to try and catch the autoignition timing more precisely.

To go further

Next, you could try to modify the gas initial composition and temperature in the 'Gas state' section of your script. Note that you will also need to adjust the initial 'time' parameter under the 'Prepare the simulation with a ReactorNet object' section of your script, as well as the subsequent 'time' parameters under the 'Advance the simulation in time' section, to account for the different autoignition timing. You should see the same type of evolution than that of fig. 4.1.

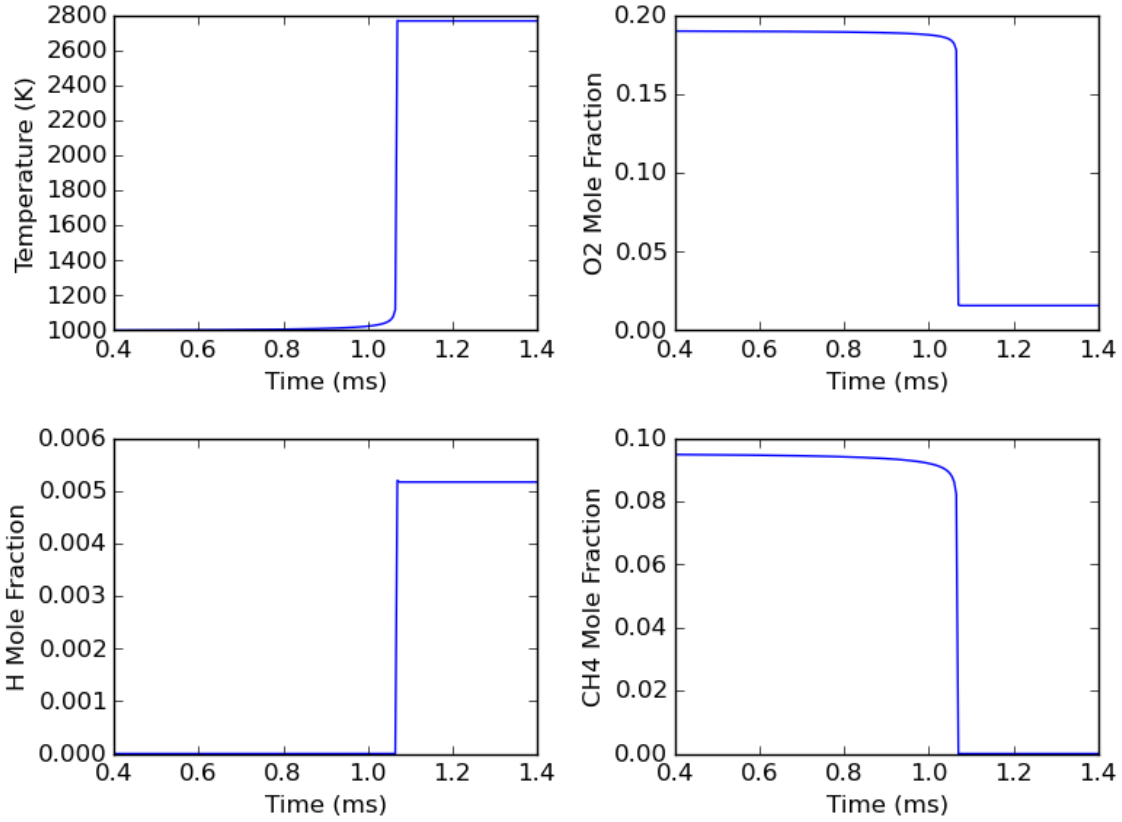

Figure 4.1: Autoignition of a methane/air mixture at stoichiometry and 1000K, with the GRIMech3.0 mechanism.

4.2.2 A simple constant pressure reactor

We turn our attention on another type of reactor, the constant pressure reactor. Here, the vessel can change volume, through moveable walls for example, so as to maintain the pressure constant throughout the simulation. A first script for this type of simulation is provided in the 'ConstantPressureReactor' folder, 'reactorHP.py', with the exact same initial gas state as in the previous exercise:

```
"""
Constant-pressure, adiabatic kinetics simulation.
"""

import sys
import numpy as np
import cantera as ct

#############################################################################

#Mechanisms used for the process
gri3 = ct.Solution('gri30.xml')
air = ct.Solution('air.xml')

#Gas state
gri3.TPX = 1000.0, ct.one_atm, 'CH4:0.5,O2:1,N2:3.76'

#Create Reactors used for the process and initialize them
r = ct.IdealGasReactor(gri3)
env = ct.Reservoir(air)

# Define a wall between the reactor and the environment, and
# make it flexible, so that the pressure in the reactor is held
# at the environment pressure.
w = ct.Wall(r, env)
w.expansion_rate_coeff = 1.0e6  # set expansion parameter. dV/dt = KA(P_1 - P_2)
w.area = 1.0

#Prepare the simulation with a ReactorNet object
```

```
sim = ct.ReactorNet([r])
time = 4.e-1

#Arrays to hold the datas
times = np.zeros(200)
data = np.zeros((200,4))

#Advance the simulation in time
print('%10s %10s %10s %14s' % ('t [s]','T [K]','P [Pa]','h [J/kg]'))
for n in range(200):
    time += 5.e-3
    sim.advance(time)
    times[n] = time  # time in s
    data[n,0] = r.T
    data[n,1:] = r.thermo['O2','H','CH4'].X
    print('%10.3e %10.3f %10.3f %14.6e' % (sim.time, r.T,
                                           r.thermo.P, r.thermo.h))

# Plot the results if matplotlib is installed.
# ... plot options ...
```

Note that whenever a Reactor type is used, Cantera will assume its volume constant. To modify this, a reservoir with a constant state -thus pressure- is defined, and a moveable wall is set to separate it from the reactor where the reaction occurs. This wall moves with the pressure difference between the reservoir and the reactor. If the movement is fast enough, the pressure remains constant in the reactor throughout the reaction. Run the script with the plot option :

```
$ python reactorHP.py --plot
```

Results

You should get a gas printout that looks like this :

```
     t [s]      T [K]     P [Pa]       h [J/kg]
 4.050e-01   1000.859 101325.000   5.879517e+05
 4.100e-01   1000.880 101325.000   5.879517e+05
 4.150e-01   1000.902 101325.000   5.879517e+05
```

```
...
 1.395e+00   2541.146 101325.000   5.879515e+05
 1.400e+00   2541.146 101325.000   5.879515e+05
```

Notice this time that it is the pressure and enthalpy stay constant, consistent with what we wanted to obtain. The evolution of the temperature and of the most important species' mole fractions are displayed in fig. 4.2. As the reaction proceeds, we see the temperature's sharp increase and the fuel mass fraction decay. Usually, you will need to refine the time steps, to try and catch the autoignition timing more precisely. Compare with the constant volume autoignition results of the previous exercise, fig. 4.1.

To go further

Try simplifying this script with the use of the "IdealGasConstPressureReactor" object. This object allows you to instantiate a reactor with moveable walls included, so that you don't need the "walls trick" we used in our previous script. Run both scripts and compare results. Note : you can redirect the screen printing with > :

```
$ python reactorHP.py > outfile.dat
```

You can find this simplified script in Appendix C.3.2.

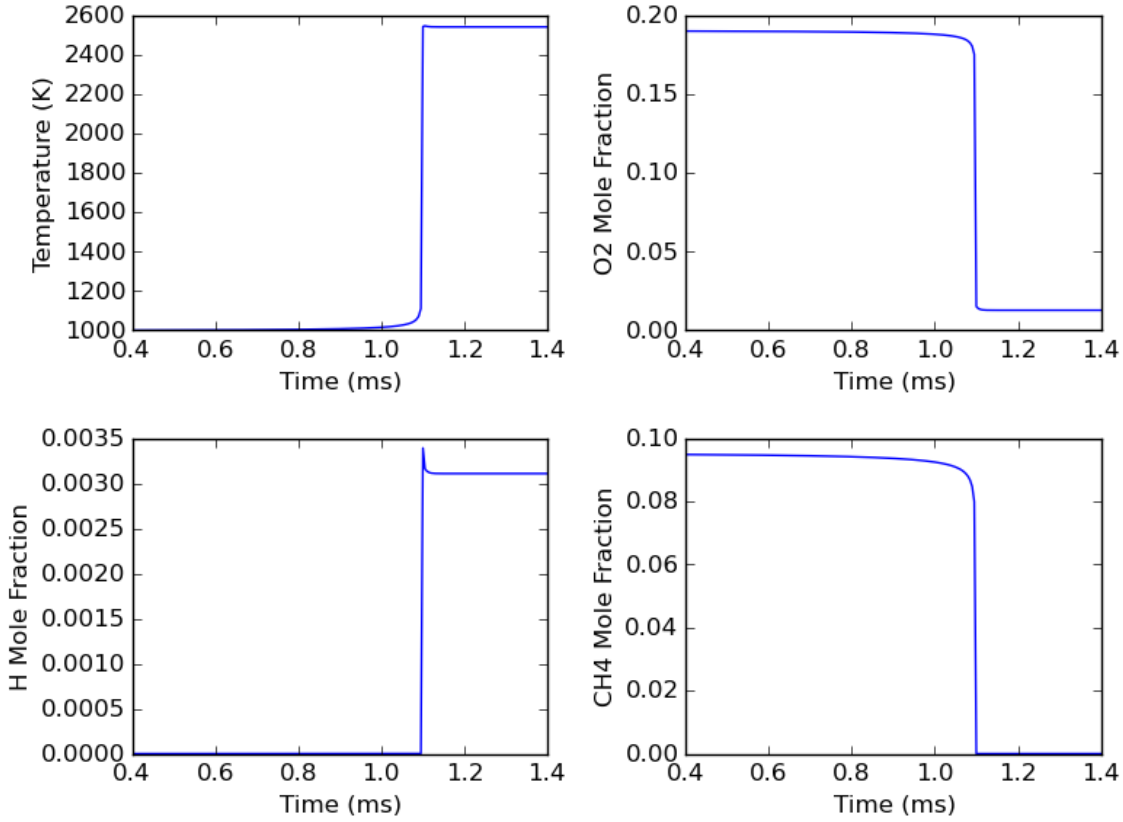

Figure 4.2: Constant pressure autoignition of a methane/air mixture at stoichiometry and 1000K, with the GRIMech3.0 mechanism.

4.2.3 An example of application : mixing two streams (OPTIONAL)

Introduction

Up until now, we have seen how to generate simple closed vessels. However, it is sometimes interesting to mix different streams, or to feed the exhaust of a reservoir with constant parameters to a reacting reactor. In this exercise, we will design a script that simulates the stoichiometric constant volume mixing of a stream of air with a stream of pure gaseous methane, as depicted in fig. 4.3.

Script generation

Go into the folder 'Mixing' and create a file with a '.py' extension, say, 'mixing.py', that you can edit in your favorite text editor :

```
$ touch mixing.py
$ gedit mixing.py &
```

Import cantera at the top of your script, as usual. Then, import the gas for your first stream, from the Cantera file 'air.xml'; and set its state to ambient air composition (although it should be the default setting) :

```
gas_a = ct.Solution('air.xml')
gas_a.TPX = 300.0, ct.one_atm, 'O2:0.21, N2:0.78, AR:0.01'
```

Use the GRIMech3.0 mechanism present in your folder, gri30.cti, to generate a pure stream of gaseous methane (CH4):

```
gas_b = ...
gas_b.TPX = ...
```

Create two reservoirs to hold both the air and the methane, as well as an exhaust reservoir that will receive the mixing reactor's outlet (see fig. 4.3). Note that you will need to initialize the downstream reservoir with the 'gas_b', since the 'gas_a' does not contain any mechanism for methane mixtures :

```
res_a = ct.Reservoir(gas_a)
res_b = ...
downstream = ct.Reservoir(gas_b)
```

Generate a constant volume reactor to receive the two mixing streams, and fill it with a mixture of air. Here again, be careful and use 'gas_b':

```
gas_b.TPX = 300.0,ct.one_atm, 'O2:0.21, N2:0.78, AR:0.01'
mixer = ct.IdealGasReactor(gas_b, energy='on')
```

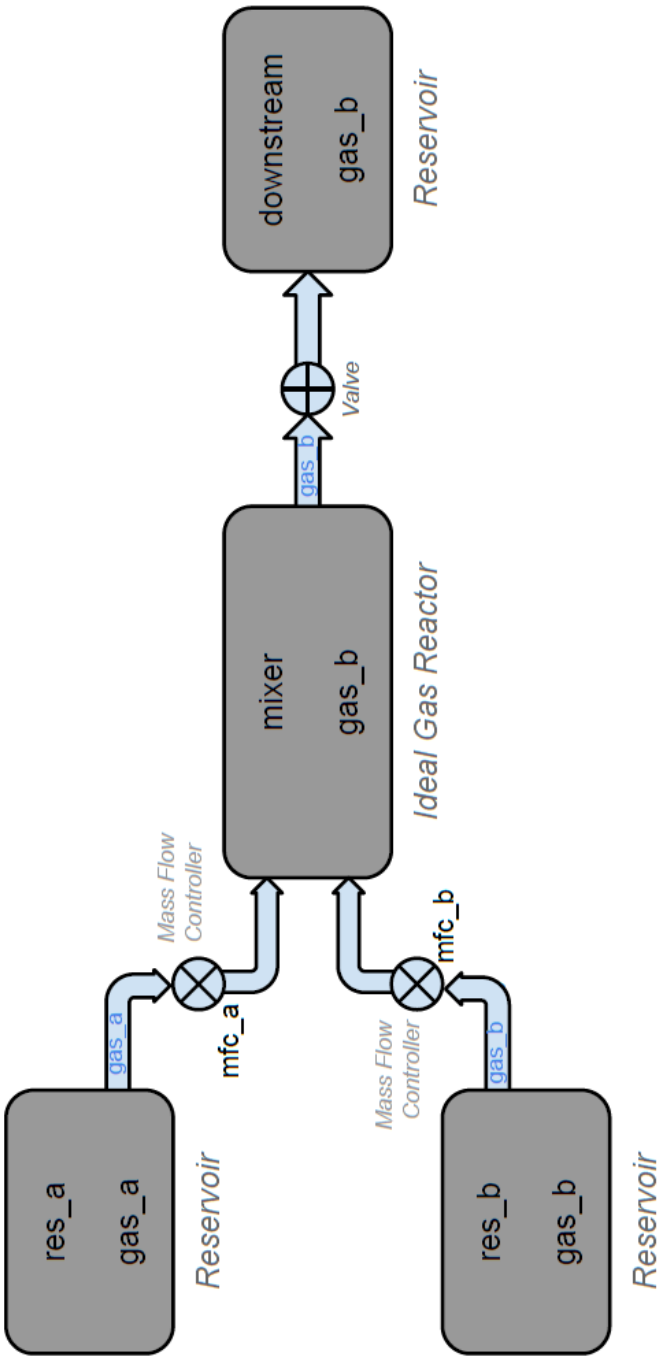

Figure 4.3: Constant volume mixing of two streams of air and methane, with the GRIMech3.0 mechanism.

Next, you need to link all items. To do so, define two flow controllers, from the inlet reservoirs 'res_a' and 'res_b' to the mixing reactor 'mixer', in order to control the mass flow rate into the mixing reactor. This will allow to maintain the inflow mixture at stoichiometry. To do so, note that you will need to extract the mean molecular weight of each stream. Modify your code by adding a line under the 'gas_a' and 'gas_b' state definition :

```
gas_a = ct.Solution('air.xml')
gas_a.TPX = 300.0, ct.one_atm, 'O2:0.21, N2:0.78, AR:0.01'
mw_a = gas_a.mean_molecular_weight/1000 #kg/mol
```

And define your mass flow controllers with them :

```
mfca = ct.MassFlowController(res_a, mixer, mdot=mw_a*2./0.21)
mfcb = ct.MassFlowController(res_b, mixer, mdot=mw_b*1.0)
```

You also need to connect the mixer to the downstream reservoir, here we will use a valve:

```
outlet = ct.Valve(mixer, downstream, K=10.0)
```

This is done in order to maintain the reactor's pressure at the pressure of the downstream reservoir, which is atmospheric pressure (set K to a high enough value).

Next, you can define your equation manager :

```
sim = ct.ReactorNet([mixer])
```

and integrate the equations in time with the 'advance' function. We choose to integrate for a hundred residence times, which should be enough to reach the steady state. It is useful to print results on screen or in a file, so as to have an idea of whether convergence is reached. If it is not the case, just increase the number of residence time integrations:

```
# Since the mixer is a reactor, we need to integrate in time to reach steady
# state. A few residence times should be enough.
print('{0:>14s} {1:>14s} {2:>14s}  {3:>14s}  {4:>14s}'.format(
      't [s]', 'T [K]', 'h [J/kg]', 'P [Pa]', 'X_CH4'))


t = 0.0
for n in range(100):
    tres = mixer.mass/(mfca.mdot(t) + mfcb.mdot(t))
    t += 2*tres
    sim.advance(t)
```

```
    print('{0:14.5g} {1:14.5g} {2:14.5g}  {3:14.5g}  {4:14.5g}'.format(
          t, mixer.T, mixer.thermo.h, mixer.thermo.P, mixer.thermo['CH4'].X[0]))
```

You can add a line to view the state of the gas in the mixer after the simulation :

```
# view the state of the gas in the mixer
print(mixer.thermo.report())
```

Run your script by launching it from your terminal.

```
$ python mixer.py
```

Results

You should get a gas report that looks like this :

```
  gri30:


       temperature              300  K
          pressure           101325  Pa
           density          1.12691  kg/m^3
  mean mol. weight          27.7414  amu

                         1 kg            1 kmol
                    -----------      ------------
          enthalpy    -2.5351e+05       -7.033e+06     J
   internal energy    -3.4342e+05       -9.527e+06     J
           entropy           7222        2.003e+05     J/K
    Gibbs function    -2.4201e+06       -6.714e+07     J
 heat capacity c_p         1070.4         2.97e+04     J/K
 heat capacity c_v         770.71        2.138e+04     J/K

                           X                Y          Chem. Pot. / RT
                    -------------     ------------     ------------
                O2       0.190045         0.219211         -26.3342
               CH4      0.0950226        0.0549513         -54.6765
                N2       0.705882         0.712806         -23.3814
                AR     0.00904977        0.0130318         -23.3151
     [  +49 minor]    6.75663e-31      1.83582e-31
```

Notice that the gas did not ignite, and that the composition is indeed stoichiometric. We could trigger the ignition, by assuming that the reactor is 'already lit'. To do so, you need to initialize the content of the reactor with hot gases. The simplest way to do so is by filling it with stoichiometric mixture of air and methane at equilibrium. This is easily done by replacing the lines :

```
gas_b.TPX = 300.0, ct.one_atm, 'O2:0.21, N2:0.78, AR:0.01'
mixer = ct.IdealGasReactor(gas_b)
```

with :

```
gas_b.TPX = 300.0,ct.one_atm,'O2:1., N2:3.78, CH4:0.5'
gas_b.equilibrate("HP")
mixer = ct.IdealGasReactor(gas_b)
```

Now when you run your script, you should notice that the reactor burns ! The temperature and composition of the reactor should be that of the equilibrium state. *Find the script in Appendix C.3.3.*

```
  gri30:

      temperature         2225.34  K
         pressure          101325  Pa
          density        0.150726  kg/m^3
  mean mol. weight         27.5235  amu

                        1 kg            1 kmol
                     -----------     ------------
          enthalpy    -2.5351e+05      -6.977e+06     J
   internal energy    -9.2575e+05      -2.548e+07     J
           entropy         9836.2       2.707e+05     J/K
    Gibbs function    -2.2142e+07      -6.094e+08     J
 heat capacity c_p         1503.2       4.137e+04     J/K
 heat capacity c_v         1201.1       3.306e+04     J/K

                           X               Y          Chem. Pot. / RT
                     -------------     ------------     ------------
                 H2     0.00379758      0.000278142         -25.3976
                  H    0.000443298       1.6234e-05         -12.5967
                  O    0.000250097      0.000145381         -17.1146
```

O2	0.00512519	0.00595853	-34.4245
OH	0.00309077	0.00190985	-29.9164
H2O	0.182985	0.119771	-42.7174
HO2	5.65629e-07	6.78313e-07	-47.1272
H2O2	4.98718e-08	6.16335e-08	-59.8915
C	3.34832e-10	1.46117e-10	-4.96175
CH	2.26905e-09	1.07329e-09	-13.887
CH2	3.86952e-08	1.97203e-08	-24.8492
CH2(S)	2.81163e-09	1.4329e-09	-24.6677
CH3	5.31788e-07	2.90491e-07	-36.8004
CH4	9.62339e-07	5.60923e-07	-48.2144
CO	0.00944947	0.00961662	-38.7093
CO2	0.0848248	0.135634	-56.0308
HCO	6.55507e-09	6.91108e-09	-49.3739
CH2O	1.59975e-07	1.74522e-07	-54.8062
CH2OH	2.53795e-09	2.86168e-09	-58.996
CH3O	1.3173e-10	1.48533e-10	-57.8179
CH3OH	3.35483e-09	3.90561e-09	-68.9764
C2H	3.3346e-13	3.0325e-13	-30.48
C2H2	7.90405e-11	7.47741e-11	-43.2563
C2H3	8.87672e-13	8.72266e-13	-48.6783
C2H4	2.48123e-11	2.52903e-11	-58.1174
C2H5	4.13776e-13	4.369e-13	-63.5043
C2H6	2.595e-13	2.83506e-13	-74.0743
HCCO	1.2309e-10	1.8349e-10	-51.4245
CH2CO	2.57465e-10	3.93232e-10	-63.6932
HCCOH	5.2631e-12	8.03844e-12	-60.6555
N	1.51208e-08	7.695e-09	-13.759
NH	3.90756e-09	2.13166e-09	-26.0397
NH2	1.0095e-08	5.87671e-09	-36.8968
NH3	2.67378e-08	1.65444e-08	-49.6978
NNH	8.46979e-10	8.93074e-10	-40.2511
NO	0.00142824	0.00155707	-31.3634
NO2	2.66748e-07	4.45869e-07	-48.6093

```
              N2O    1.07917e-07       1.7257e-07        -44.8307
              HNO    2.79413e-08      3.14849e-08        -43.9954
               CN    4.43542e-10      4.19277e-10        -26.5269
              HCN    1.51036e-07      1.48304e-07        -39.0752
             H2CN    1.09595e-12      1.11627e-12        -48.5579
             HCNN    8.92162e-13      1.33005e-12        -41.5252
             HCNO    3.80754e-08      5.95199e-08        -45.5661
             HOCN     3.6565e-09      5.71588e-09        -57.2235
             HNCO    1.75399e-07      2.74186e-07        -59.1872
              NCO    7.27407e-09      1.11045e-08        -46.4238
               N2       0.699624         0.712078        -27.6542
               AR      0.0089787        0.0130318         -26.183
          CH2CHO    1.13568e-14      1.77614e-14        -74.0555
          CH3CHO    3.16313e-13      5.06279e-13        -81.8362
[   +2 minor]    1.45495e-19      2.30142e-19
```

4.2.4 Autoignition timing

Introduction

An interesting feature of 0D simulations is that it allows to compute the temporal evolution of a mixture under specific conditions towards its equilibrium state. That evolution can be viewed as a sort of 'autoignition' of the mixture. To go further, that last exercise will guide you through such a computation of the autoignition timing of a methane/air mixture. Several definitions of the autoignition timing can be found on the literature. The most commonly accepted definition relies on the temperature gradient : the time of the sharpest temperature increase is spotted as being the autoignition point. In order to catch this time with precision, the time step of the simulation should be as small as possible.

A simple script to catch the autoignition timing

Go into the 'Autoignition' subdirectory of the 'Tuto4' directory. You will find a partially filled script 'Autoignition_simple.py' :

```
##############################################################
#
# Autoignition of a methane air mixture at stoichiometry
```

```
#             and atmospheric pressure, for different
#             initial temperature
#
#####################################################################

import cantera as ct
import csv
import numpy as np

#####################################################################

# Mechanism used for the process
gas = ct.Solution('gri30.cti')

# Initial temperature, Pressure and stoichiometry
gas.TPX = 1250, ct.one_atm, 'CH4:0.5, O2:1, N2:3.76'

# Set gas state

# Specify the number of time steps and the time step size
nt = 100000
dt = 1.e-6  # s

# Storage space
mfrac = []
# ...
time = []
temperature = []
HR = []

#####################################################################

# Create the batch reactor
r = ...
```

```
# Now create a reactor network consisting of the single batch reactor
sim = ...

# Run the simulation
# Initial simulation time
current_time = 0.0

# Loop for nt time steps of dt seconds.
for n in range(nt):
    current_time += dt
    sim.advance(current_time)
    time.append(current_time)
    temperature.append(r.T)
    mfrac.append(r.thermo.Y)
    HR.append(- np.dot(gas.net_production_rates, gas.partial_molar_enthalpies))

#####################################################################
# Catch the autoignition timing
#####################################################################

# Get the ignition delay time by the maximum value of the Heat Release rate
Autoignition = time[HR.index(max(HR))]
print('Autoignition time = ' + str(Autoignition))
# Posterity
Autoignition = Autoignition * 1000  # ms
FinalTemp = temperature[nt - 1]

#####################################################################
# Save results
#####################################################################
# write output CSV file for importing into Excel
csv_file = 'Phi-1_P-1_T-1250_UV.csv'
with open(csv_file, 'w') as outfile:
    writer = csv.writer(outfile)
```

```
    writer.writerow(['Auto ignition time [ms]', 'Final Temperature [K]'] + gas.species_names)
    writer.writerow([Autoignition, FinalTemp] + list(mfrac[:]))
print('output written to ' + csv_file)
```

Fill in the blanks by specifying the type of reactor and defining a ReactorNet object, as in the first exercise, section 4.2.1. With this script, we will catch the autoignition timing of a mixture of methane and air, with an initial stoichiometric composition and an initial temperature of 1250 K.

Results

Find the complete script in Appendix C.3.4. You should get a screen printout that looks like this :

```
Autoignition time = (s) 0.0215205
output written to Phi-1_P-1_T-1250_UV.csv
```

It should create a 'Phi-1_P-1_T-1250_UV.csv' file in your directory, the content of which should look something like :

```
Auto ignition time [ms],Final Temperature [K],H2,H,O,O2,...,CH3CHO
21.5205,2836.02735401,0.00136159069637,0.000307040605043,...,1.13610521252e-22
```

You could try to modify this script and catch the autoignition timing for another initial temperature, for example. Be careful as you will certainly have to modify the time step resolution while doing so, under the '#Specify the number of time steps and the time step size' section of the previous script.

Catch the autoignition timing for different initial temperature (OPTIONAL)

Now, it is often of interest to catch the autoignition timing of a mixture for several initial temperature and/or composition in a single script. To do this, you could simply loop upon the initial gaseous composition. In this last exercise, we will modify our previous script to catch the autoignition timing of a mixture of methane and air under atmospheric pressure and stoichiometric condition; but for several different initial temperature. Copy your previous script under a new name, say, 'Autoignition_Tvar.py'. Open this script in your favorite editor. Begin by commenting out the part setting the gas composition :

```
#Initial temperature, Pressure and stoichiometry
#gas.TPX = 1250, one_atm, 'CH4:0.5,O2:1,N2:3.76'
```

And replace it by the specifics of each computation :

```
# Initial temperatures
Temperature_range = list(range(800, 1700, 100))
```

```
# Specify the number of time steps and the time step size
nt = 100000
dt = 1.e-4  # s

# Storing auto ignitions
auto_ignitions = []

for index, Temperature in enumerate(Temperature_range):
    ######################################################################

    # Initial temperature, Pressure and stoichiometry
    gas.TPX = Temperature, ct.one_atm, 'CH4:0.5, O2:1, N2:3.76'

    # Create the batch reactor
    r = ct.IdealGasReactor(gas)
    # Now create a reactor network consisting of the single batch reactor
    sim = ct.ReactorNet([r])

    # Storage space
    mfrac = []
    # ...
    time = []
    temperature = []
    HR = []

    # Run the simulation
    # Initial simulation time
    current_time = 0.0

    # Loop for nt time steps of dt seconds.
    for n in range(nt):
        current_time += dt
        sim.advance(current_time)
```

```
        time.append(current_time)
        temperature.append(r.T)
        mfrac.append(r.thermo.Y)
        HR.append(- np.dot(gas.net_production_rates, gas.partial_molar_enthalpies))

    ################################################################
    # Catch the autoignition timing
    ################################################################

    # Get the ignition delay time by the maximum value of the Heat Release rate
    auto_ignition = time[HR.index(max(HR))]
    print('For T = ' + str(Temperature) + ', Autoignition time = ' + str(auto_ignition) + ' s'
    # Posterity
    FinalTemp = temperature[nt - 1]

    auto_ignitions.append(auto_ignition)

    # ################################################################
    # # Save results
    # ################################################################
    # # write output CSV file for importing into Excel
    # csv_file = 'Phi-1_P-1_T-' + str(Temperature) + '_UV.csv'
    # with open(csv_file, 'w') as outfile:
    #     writer = csv.writer(outfile)
    #     writer.writerow(['Auto ignition time [s]', 'Final Temperature [K]'] + gas.species_na
    #     writer.writerow([auto_ignition, FinalTemp] + list(mfrac[:]))
    # print('output written to ' + csv_file)

T_invert = [1000 / Temperature for Temperature in Temperature_range]
################################################################
# Plot results
################################################################
# create plot
```

```
plt.plot(Temperature_range, auto_ignitions, 'b-o')
plt.xlabel(r'Temperature [K]', fontsize=20)
plt.ylabel("Auto ignition [s]", fontsize=20)
plt.yscale('log')
plt.title(r'Autoignition of $CH_{4}$ + Air mixture at $\Phi$ = 1, and P = 1 bar',
          fontsize=22, horizontalalignment='center')
plt.axis(fontsize=20)
plt.grid()
plt.savefig('Phi-1_P-1_Trange_UV.png', bbox_inches='tight')
plt.show()
```

Notice that the advancement of the reactor is now embedded inside the loop, and is done for each initial condition.

Results

Find the complete script in Appendix C.3.4. You should get a screen printout with the autoignition timing for each initial condition :

```
For T = 800, Autoignition time = 9.999999999990033 s
For T = 900, Autoignition time = 7.055299999996896 s
For T = 1000, Autoignition time = 1.066899999999899 s
For T = 1100, Autoignition time = 0.19789999999999452 s
For T = 1200, Autoignition time = 0.043400000000000216 s
For T = 1300, Autoignition time = 0.011099999999999988 s
For T = 1400, Autoignition time = 0.0032999999999999982 s
For T = 1500, Autoignition time = 0.0011000000000000003 s
For T = 1600, Autoignition time = 0.0004 s
```

The plot it generates is shown in fig. 4.4.

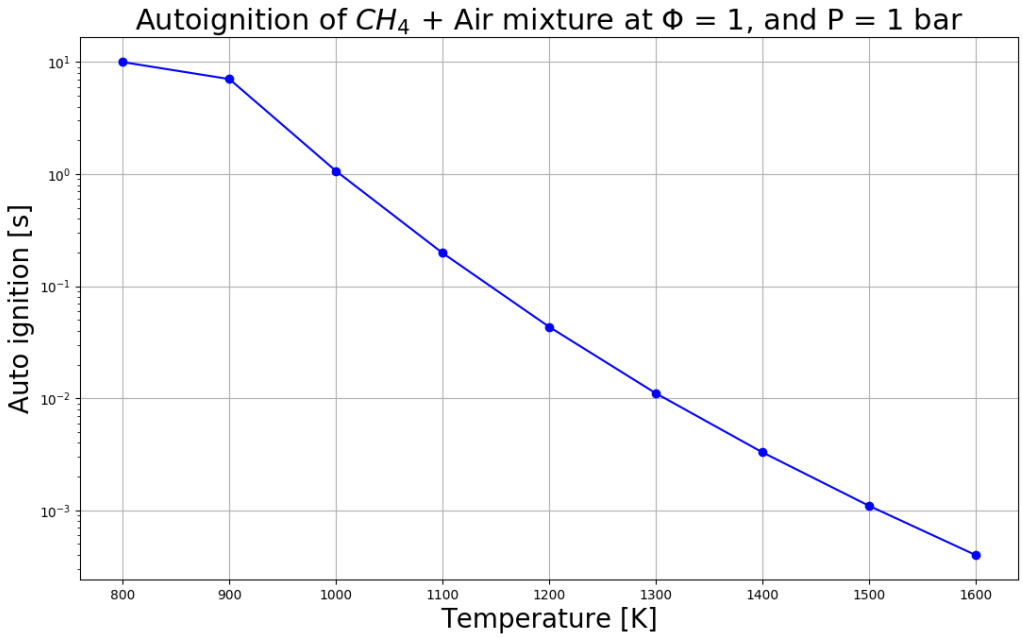

Figure 4.4: Autoignition time of a methane-air mixture at stoichiometry and under atmospheric pressure, for different initial temperatures.

5. *Tutorial 5 : 1D simulations*

5.1 Introduction

The goal of this section is to introduce one-dimensional flame simulations with Cantera. There are five types of such simulations available in Cantera 2.3 : premixed freely propagating flat flames (FreeFlame), burner stabilized flat flames (BurnerFlame), counterflow premixed flames (CounterflowPremixedFlame), counterflow diffusion flames (CounterflowDiffusionFlame) and flames made by a premixed jet impinging on a wall (ImpingingJet). In the following exercise, we will see only FreeFlame, BurnerFlame and CounterflowDiffusionFlame as the 2 remaining configurations are very similar to the CounterflowDiffusionFlame only with different boundary conditions.

An important thing to understand when trying to perform such computations, is that Cantera is not very "robust". If you remember from the lecture, it uses a modified Newton algorithm, the convergence of which is greatly sensitive to the initial conditions and settings of the solver. The type of mechanism used for the chemistry description as well as the transport model chosen can also be the cause of convergence failures... Now, a few solver properties can be tuned to help with the convergence, as we will see in the exercises, but in most cases it is best to start with a "standard" computation (atmospheric conditions, stoichiometry) and to iterate from there: it will be both easier and faster to reach a reliable solution.

Cantera 1D calculations can also be very long. Obviously, whether you choose a detailed or a reduced mechanism for a particular simulation of a mixture will impact your results and performances. Using a detailed mechanism could provide reliable mass fractions and temperature profiles; but the simulation time can take up to several days. Eventually, it all depends on the information you seek, and the accuracy level that is acceptable to you. See fig. 5.1 for an example of laminar flame speed calculation for Kerosene-air mixture with two radically different schemes.

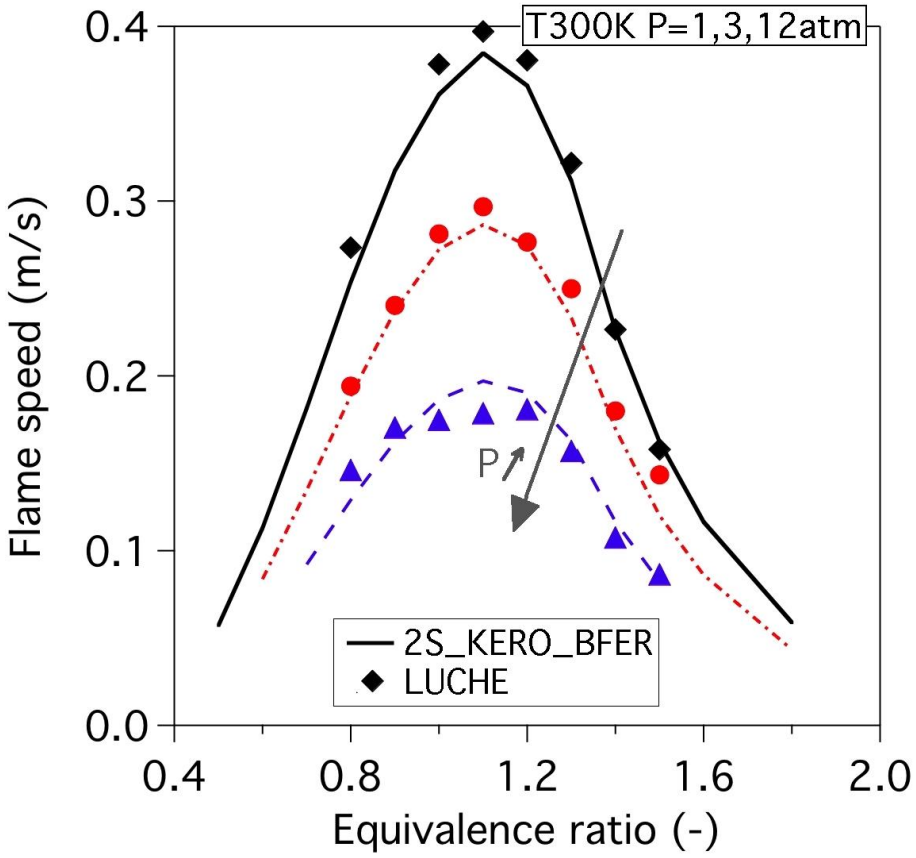

Figure 5.1: Flame speed versus equivalence ratio at 300 K, for three pressures (1, 3 and 12 atm). Comparison between a reduced scheme (from Luche's thesis, 991 reactions and 91 species) and a global scheme (2S_KERO_BFER developped by Benedetta Franzelli at CERFACS, 2 global steps and 6 species)

5.2 General structure of a 1D simulation with Cantera

An interesting feature of all standard 1D flames is that their script can be structured in a similar fashion. With the Python interface, it should be composed of :

- As usual, a package definition section. You will always require cantera and numpy
- The definition of a gaseous object : import a gas and set its state (we have reviewed this step in all previous tutorials)
- A 1D domain, initialy discretized
- And eventually, the flame, with :
  - a gaseous composition (use the gaseous object previously defined)
  - a set of equations to integrate (depending upon the type of flame)
  - inlets/boundaries conditions
- Storage/plot options

The next sections will review the last two steps, and specifics will be provided on each exercise, when relevant.

5.2.1 1D domains and discretization

All 1D simulations require a spatial domain. The simplest way to define such domains is to specify only the width of the domain and Cantera will construct a uniform grid of 9 points:

```
width = 0.02 # m
```

Or with lots of points using numpy :

```
initial_grid = numpy.linspace(0, 0.02, 300)
```

But usually, you will find that the solver prefers to start with a grid that is refined in the boundary regions :

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

It is also best to start with a grid that is not too wide, 2 cm is a good start. Cantera will add points, refining your grid in the region of steepest gradients, according to the level of precision you require (see next section). However, you will need to continue the computation of your flame, starting from a previously stored profile, if you want to extend your domain.

5.2.2 Flame objects

Once your grid is set, the next step is the definition of the flame object. This step depends upon the type of simulation you are running. All flame objects should be initialized by specifying a gaseous composition. With each type of flame comes a set of governing equations, and inlets/boundaries definitions. We will see this in more details in the exercises.

FreeFlame object

For a premixed simulation, this is done through the Cantera "FreeFlame" object :

```
f = FreeFlame(gas, initial_grid)
```

In Cantera, the FreeFlame object is composed of three domains : 'self.inlet', 'self.flame', and 'self.outlet'. Depending on which quantity you are interested in, or want to define, you will need to act on one or the other domain. Some non-physical quantities are also directly applied on the flame. This is pretty self explanatory, as we will see in the exercises.

BurnerFlame object

For a burner simulation, this is done through the Cantera "BurnerFlame" object :

```
f = BurnerFlame(gas, initial_grid)
```

In the same fashion, a burner stabilized flame object is composed of three domains : 'self.burner', 'self.flame', and 'self.outlet'.

CounterflowDiffusionFlame object

For a counterflow flame simulation, this is done through the Cantera "CounterflowDiffusionFlame" object :

```
f = CounterflowDiffusionFlame(gas, initial_grid)
```

In the same fashion, this type of flame object is composed of three domains : 'self.fuel_inlet', 'self.flame', and 'self.oxidizer_inlet'.

5.2.3 Boundary conditions

They are specific to the type of simulation being performed, and will thus be reviewed in the different exercises.

5.2.4 Save and restart your computation : the 'xml' format

You can store a flame solution in different formats (see Appendix B), but you should always call the internal Cantera function 'save' after each evaluations of a flame object f :

```
f.save('yourfile.xml','name_of_solution', 'description_of_solution')
```

This will give you the possibility to restart a flame simulation from a previously stored solution. In order to do so, you will need the '.xml' saving file of a previous simulation, and the 'name_of_solution' of the run you want to restart from. For example, if you saved a solution in a python script with the line:

```
f.save('ch4_adiabatic.xml','energy', 'solution with the energy equation enabled')
```

You can restore it in a new python script by invoking the 'restore' function on a new flame object g :

```
g.restore('ch4_adiabatic.xml', 'energy')
```

Of course, the new flame object must be compatible : you need corresponding species with the same name (the mechanism used can contain more or less species than the mechanism used to compute the stored solution). Be careful, the restore function overwrites the grid you specify. Let's say I create :

```
f = ct.FreeFlame(gas, width=0.02)
```

Save it :

```
f.save('solution.xml')
```

Create a new flame with a different grid:

```
g = ct.FreeFlame(gas, width=0.02)
```

Then I restore the previous solution :

```
g.restore('solution.xml')
```

Printing the grid, here is the output :

```
print(g.grid)
array([0.   , 0.004, 0.006, 0.007, 0.008, 0.01 , 0.012, 0.016, 0.02 ])
```

As you can see, domain length is back to 0.02. There is no function to interpolate restored solution on a new mesh yet.

The 'restore' function is usually called right after the gas is imported and the flame object initialized:

```
g = FreeFlame(gas)
g.restore('ch4_adiabatic.xml', 'energy')
```

Once your flame is restored, you can modify the inlet conditions and flame pressure to compute different conditions; and/or any solver parameters so as to alter the transport model or convergence settings.

NOTE : Save relevant information

It is straightforward, with the new Python interface of Cantera 2.3, to save relevant data, by invoking the special 'write_csv' function:

```
#Write the velocity, temperature, density, and mole fractions to a CSV file
yourflame.write_csv('yourfile.csv', species = 'X', quiet=False)
```

This will write the velocity, temperature, density and mole or mass fractions, respectively, to a CSV file; depending on whether species = 'X' or 'Y' is specified, respectively.

5.3 About transport properties in 1D simulations

Cantera provides a set of transport manager classes that manage the computation of transport properties. Every object representing a phase of matter for which transport properties are needed has a transport manager assigned to it. In Cantera, for gaseous phases, you have the choice between a 'Mixture-averaged' transport model and 'Multicomponent' transport model. Setting a model in a 1D simulation is done with the function 'transport_model' applied on the flame object f :

```
f.transport_model = 'Multi' # or 'Mix'
```

The two transport models are based on the ones described by Kee, Coltrin, and Glarborg in "Theoretical and Practical Aspects of Chemically Reacting Flow Modeling." Basically, in the mixture-averaged formulations, the diffusion velocity of species k is related to the species gradients by a Fickian formula, where only a 'mixture diffusion coefficient' is needed. This is also referred to as the Hirshfelder and Curtis approximation. The multicomponent formulation, on the other hand, takes into consideration the diffusion of species k into each species j.

The multicomponent formulation should always provide you with the most reliable results... but at what cost ? Indeed, using a multicomponent formulation will result in an exponential growth of the required calculation time when your reaction mechanism becomes heavier; whereas this growth is somewhat linear with a mixture-averaged formulation. Furthermore, starting with a multicomponent formulation will almost always end up in a failure, so that if you need to use this formulation you will have to change your transport model in a continuation; or by restoring a solution as we have seen in the previous section. If a continuation is performed where the transport model is changed, it is recommended to leave the other refinement criteria untouched.

5.4 Solver properties

The properties required by the solver to integrate the 1D equations can be summarized as follow:

- the set of equations to solve
- tolerances values for the Newton integration
- maximum number of iterations before a Jacobian is re-computed
- time stepping values for the Newton time-stepping iterations
- grid refinement settings

5.4.1 Set of equations to solve

The first thing to specify is whether you want the energy equation to be solved. If you need to compute a flame with a heavy mechanism, for example, it is often recommended to start by running a first flame without, by setting the flame parameter 'energy_enabled' to 'False':

```
f.energy_enabled = False
```

5.4.2 Newton integration

Tolerances values

Now, the flame will be solved through Newton iterations, the convergence of which is governed by specified relative and absolute error tolerances (rtol and atol, respectively). You will need to specify those tolerances limits. This is done through the 'set_steady_tolerances' and 'set_transient_tolerances' functions, which respectively set the tolerances limits for the Newton and modified time-stepping Newton iterations :

```
#Tolerance properties
tol_ss    = [1.0e-5, 1.0e-9]        # [rtol atol] for steady-state problem
tol_ts    = [1.0e-5, 1.0e-9]        # [rtol atol] for time stepping

f.flame.set_steady_tolerances(default=tol_ss)
f.flame.set_transient_tolerances(default=tol_ts)
```

Note that those functions are applied on the 'flame' itself, as they will be used for computation in the entire flame domain; but could also be specified relative to a chosen subset of species, if needed. Standard values are between 1.0e-05 and 1.0e-13. It is possible to refine their values in a continuation computation.

Jacobian evaluations

Next, specify the maximum number of times the Jacobian can be used before it must be re-evaluated. Recall from the presentation this morning that it might be recomputed before, though. This is done through a specific function 'set_max_jac_age', called on a flame object f:

```
f.set_max_jac_age(50, 50)
```

Standard values are between 30 an 50.

Time-stepping for the modified Newton iterations

Time-stepping for the modified Newton iterations are defined on the flame object f as follows:

```
f.set_time_step(1.0e-5, [2, 5, 10, 20, 80]) #s
```

This means that a first set of 2 timesteps with value 1.0e-5 will be taken before trying a Newton iteration with the solution obtained. If this fails, 5 timesteps with value 1.0e-05 will be taken, and so on. Usually, it is best to start with a time stepping of the order of 1.0e-05, but note that in some cases, with very sensitive chemistries, you might need to go as low as 1.0e-07. The time required to complete your simulation will increase in consequence.

Grid refinement

Finally, you need to specify the grid refinement criteria :

```
f.set_refine_criteria(ratio = 10.0, slope = 1, curve = 1, prune = 0.05)
```

The different fields definitions are :

-ratio : additional points will be added if the ratio of the spacing on either side of a grid point exceeds this value

-slope : maximum difference in value between two adjacent points, scaled by the maximum difference in the profile ($0.0 <$ slope < 1.0). Adds points in regions of high slope.

-curve : maximum difference in slope between two adjacent intervals, scaled by the maximum difference in the profile ($0.0 <$ curve < 1.0). Adds points in regions of high curvature.

-prune : if the slope or curve criteria are satisfied to the level of prune, the grid point is assumed not to be needed and is removed. Set prune significantly smaller than slope and curve. Set to zero to disable pruning the grid.

For a first computation, it is best to put the ratio value to a sufficiently high value; and to let the 'slope' and 'curve' values to 1. Use 'prune' if you see that Cantera tries to refine your grid too much, and that the number of points starts to exceed a reasonable value ($\approx$ 200 points is enough).

5.5 Exercises

5.5.1 A premixed flat flame simulation

Introduction

We will start by computing a simple methane/air flame at stoichiometry, and under atmospheric pressure and temperature, with the GRIMech3.0 mechanism. Note that we will provide the general way of computing such a flame, but that several computations in a row can be performed, loops upon initial conditions can be added and so on ...

Prepare your run

Go into the subfolder '1DpremixedFlame' of the folder 'Tuto5' and open in your favorite text editor the partially filled 'premixed_flame.py'. You will need to specify the mixture composition : up until now, we have seen different ways to define the stoichiometry, you can choose whichever method you prefer.

```
fuel = {'CH4': 1}
oxidizer = {'O2': 1, 'N2': 3.76}


# Set gas state to that of the unburned gas
gas.TP = tin, p
gas.set_equivalence_ratio(phi, fuel, oxidizer)
```

Note the chosen way to specify a grid here, with an array :

```
#Initial grid, chosen to be 0.02cm long :
width = 0.02
```

Next, if we remember the structure of a 1D simulation provided in section 5.2, we will need to define the flame object. Following what was said in section 5.2.2, define a flame object suited to a 1D premixed flame simulation; and the associated inlet conditions, accounting for the fresh gas state.

```
#Create the free laminar premixed flame
f = ...


#set inlet conditions
f.inlet.X = ...
f.inlet.T = ...
```

For a freely propagating 1D flat flame, you only need one boundary condition: you need to specify the fresh gas composition and temperature at the inlet, where the gas is unburned.

Solve your first flame

Once all is properly set, you can launch a first flame calculation. This is done in the script under the section 'First flame'. This section has been filled for you, but you should take the time to read and understand each line. Mostly, the purpose of this section is to properly set the solver properties. You can refer to section 5.4 for information about each function used. Recall that a first flame should be computed without solving the energy equation, as specified through the function 'energy_enabled'.

```
f.energy_enabled = False
```

For the first computation, again depending on the scheme and on your conditions, you can choose to compute the flame on the initial grid or to allow the solver to refine it according to the parameters' configurations previously set with the function 'set_refine_criteria' (see section 5.4.2). This is done through the 'refine_grid' field :

```
refine_grid = False         # True to enable refinement, False to disable


f.solve(loglevel, refine_grid)
```

It is not always necessary to set this to 'False', if your refinement criteria are loose enough.

Launch your script as such :

```
$ python premixed_flame.py
```

And verify that it runs. You should get a screen printout of the sort :

```
...
Take 20 timesteps       0.01498     -1.307
Attempt Newton solution of steady-state problem...    success.

Problem solved on [9] point grid(s).
.....................................................................

grid refinement disabled.
Solution saved to file ch4_adiabatic.xml as solution no_energy.
```

Continue your simulation

Once a first flame is solved, without the energy equation and with very loose grid refinement criteria, you will need to perform a continuation. This is done under the section 'Second flame' and subsequent 'Third flame' and so on of your script. Try first to perform a Second flame by uncommenting the proper section of your script and filling in the blanks :

1. Enable the energy equation
2. Refine the grid refinement criteria : divide all previous values by 2
3. Enable the grid refinement

Then, note that the flame is solved AND saved again, with the special function 'save' on the same '.xml' file than the first flame (see section 5.2.4). *Indeed, each time you call the 'solve' function, your flame is recomputed and all the variables change state. It is possible than convergence may fail and that the solver crashes. Only if you monitor the evolution of your flame in between each computation will you be able to debug/restart your simulation and save precious time !*

Note, next, that the flamespeed will be printed on your screen, through the line :

```
#See the sl to get an idea of whether or not you should continue
print('mixture-averaged flamespeed = 0:7f m/s'.format(f.u[0])) #m/s
```

As a first test for the convergence of your flame, if you do not want to investigate all the variables; you could print the laminar flame on your monitor in between 'solve'. It is accessed through the 'u' function, applied on the flame, which returns the array containing the velocity at each grid point.

At this point, you should launch your script again :

```
$ python premixed_flame.py

...

###############################################################################
Refining grid in flame.
    New points inserted after grid points 16 17 20 21 22 23 24
    to resolve C C2H C2H2 C2H3 C2H4 C2H5 C2H6 C3H7 C3H8 CH CH2 CH2(S) CH2CO
    CH2OH CH3 CH3CHO CH3O HCCO HCCOH HCN HCO N
###############################################################################

.......................................................................
Attempt Newton solution of steady-state problem...    success.

Problem solved on [49] point grid(s).
.......................................................................

no new points needed in flame
Solution saved to file ch4_adiabatic.xml as solution energy.
mixture-averaged flamespeed = 0.389712 m/s
```

Next fill in subsequent sections 'Third flame and so on', by copying information from the section 'Second flame' :

1. Modify the grid refinement criteria
2. Solve the flame
3. Save the flame

Add as many continuation as you see fit, and refine your settings progressively. *Typically, you should find that convergence is reached for values of 'slope' and 'curve' of about 0.05, and 'ratio' of about 2-3.*

```
$ python premixed_flame.py

...

###############################################################################
```

```
Refining grid in flame.
    New points inserted after grid points 55 76 171
    to resolve point 171 point 55 point 76
###############################################################################

.......................................................................
Attempt Newton solution of steady-state problem...    success.

Problem solved on [264] point grid(s).

.......................................................................
no new points needed in flame
Solution saved to file ch4_adiabatic.xml as solution energy continuation.
mixture-averaged flamespeed continuation = 0.379019 m/s
mixture-averaged final T = 2218.049742 K
```

Save your simulation

When you feel confident that your flame has reached convergence, you can save the flame's state in a '.csv' format with the 'write_csv' function. Uncomment the 'Save your results if needed' (last) section of your script :

```
f.write_csv('ch4_adiabatic.csv', quiet=False)
```

This will write the velocity, temperature, density, and specified species profiles (mole 'X' or mass 'Y' fractions) into a 'filename.csv' file (see section 5.2.4)

```
$ python premixed_flame.py

...

###############################################################################
Refining grid in flame.
    New points inserted after grid points 55 76 171
    to resolve point 171 point 55 point 76
###############################################################################
```

```
.....................................................................
Attempt Newton solution of steady-state problem...    success.

Problem solved on [264] point grid(s).

.....................................................................
no new points needed in flame
Solution saved to file ch4_adiabatic.xml as solution energy continuation.
mixture-averaged flamespeed continuation = 0.379019 m/s
mixture-averaged final T = 2218.049742 K
```

This file should look something like :

```
z (m),u (m/s),V (1/s),...,CH2CHO,CH3CHO
0.0,0.374864629014,-1.60997586636e-133,...,2.31960438205e-28,0.0
...
0.02,2.7862260761,0.0,...,5.84177268453e-25,1.37675030861e-26
```

From this file -or directly on the python script- you could generate plots for the temperature, u, rho, mole fractions ...

Change transport model (OPTIONAL)

You could uncomment the lines under the section 'Change transport model' of your script. Notice that in our case, switching to the more accurate multicomponent formulation of the transport properties does not change much the final temperature nor flamespeed ... But the computation time has increased!

```
$ python premixed_flame.py

...

Problem solved on [264] point grid(s).

.....................................................................
no new points needed in flame
Solution saved to file ch4_adiabatic.xml as solution energy_multi.
multicomponent flamespeed = 0.383763 m/s
multicomponent final T = 2218.234672 K
Solution saved to 'ch4_adiabatic.csv'.
```

Results

From the '.csv' file previously generated, you could obtain fig. 5.2. The mixture-averaged laminar flame speed is 0.379019 m/s, and final temperature is 2218.049742 K. *Find a complete script in Appendix C.4.1.*

Restart your computation from an '.xml' file (OPTIONAL)

As an example, take the '.xml' output of your previously computed stoichiometric freely propagating flat flame. Depending on the number of continuation that you performed, your '.xml' file will have more or less solutions to restart from ... Try to generate a new python script; say, 'premixed_flame_continuation.py', that will restart a flame from the last solution in your '.xml' file. Modify the flame settings, so that :

- $\phi = 0.8$
- T inlet = 600 K
- P = 1 atm

You don't need to alter the solver properties.

Results

```
$ python premixed_flame_continuation.py


...
Importing datasets:
axial velocity   radial velocity   temperature   lambda
H2   H   O   O2   OH   H2O   HO2   H2O2
C   CH   CH2   CH2(S)   CH3   CH4   CO   CO2   HCO   CH2O   CH2OH
CH3O   CH3OH   C2H   C2H2   C2H3   C2H4   C2H5   C2H6   HCCO   CH2CO
HCCOH   N   NH   NH2   NH3   NNH   NO   NO2   N2O   HNO   CN   HCN
H2CN   HCNN   HCNO   HOCN   HNCO   NCO   N2   AR   C3H7   C3H8   CH2CHO
CH3CHO
...
Problem solved on [287] point grid(s).


.....................................................................
no new points needed in flame
```

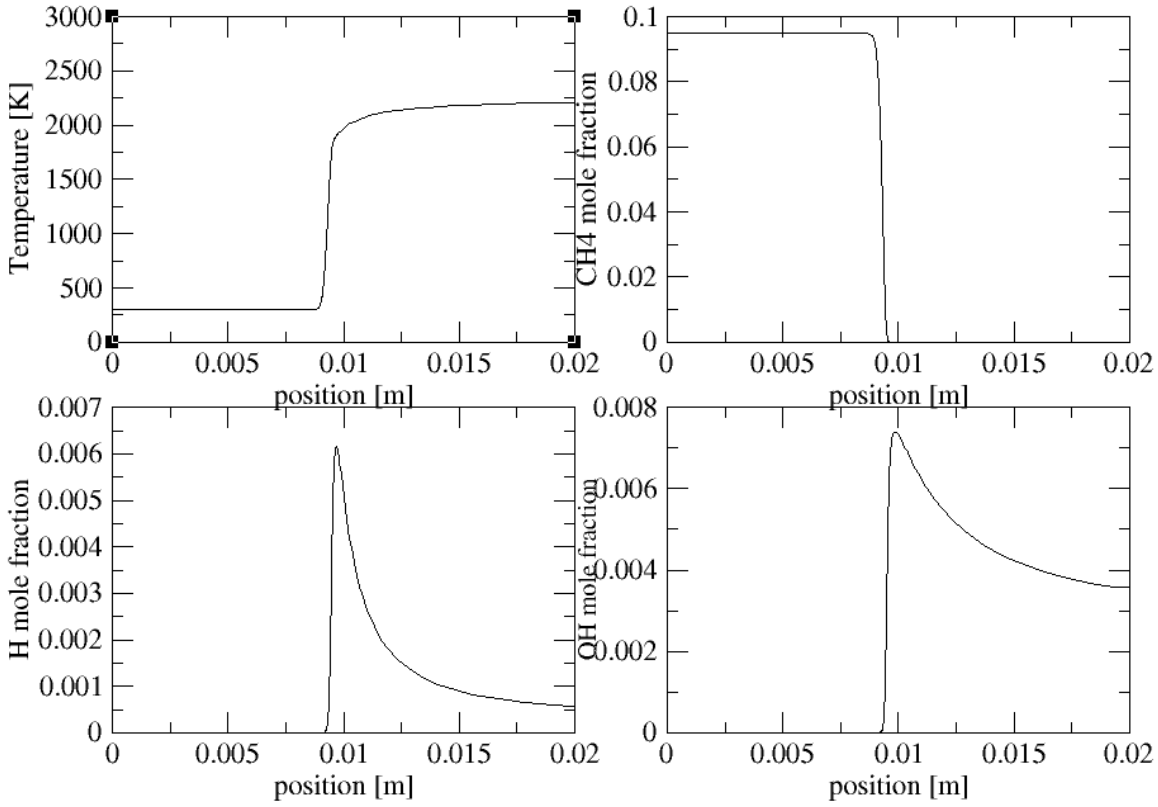

Figure 5.2: Temperature and important species' mole fractions evolution through a free premixed flat flame, computed for a methane-air mixture at stoichimometry and under atmospheric conditions with the GRIMech3.0 mechanism.

```
mixture-averaged flamespeed continuation = 1.062978 m/s
mixture-averaged final T = 2200.509604 K
Solution saved to 'ch4_adiabatic.csv'.
```

The simulation is going faster than if started from scratch. This is really useful when you need to obtain the laminar flame speed for several initial compositions, or under different pressures, for example. To check that the 'restore' function is doing its job, you could try printing the flame's state after a restore, with the 'show_solution()' function applied to the flame object. *Find a complete script un Appendix C.4.2.*

5.5.2 A burner stabilized flat flame simulation

Introduction

As seen in the introduction of this tutorial, another type of premixed flat flame that Cantera can simulated is the burner stabilized flame. The only difference in modelling this sort of premixed flame from the freely propagating one resides in the specification of a mass flow rate as a boundary condition, to "fix" the location of the flame in space. See fig. 5.3.

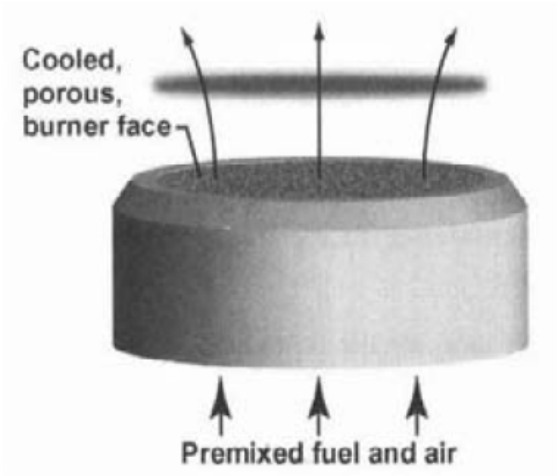

Illustration of a premixed flat-flame burner diffusion flame.

Figure 5.3: From R.J. Kee, M.E. Coltrin and P. Glarborg "Chemically Reacting Flow"

The steps for defining a simulation script are the same as those for a freely propagating flat flame, up to the definition of the flame. From there, the settings will be specific to a burner stabilized flame, but we will notice many similarities.

A simple script

Go into the subfolder '1DburnerFlame' of the 'Tuto5' folder, and open the python script you will find there in your favorite text editor. Remember the steps to define a 1D simulation script with Cantera, section 5.2 :

- a package section
- the gaseous object settings
- a 1D domain
- a flame, with :
  - a gaseous composition

- a set of equations to integrate
- inlet conditions
- specific boundary conditions

Try to understand each line of the script in regards of those aforementioned steps. Notice the additional boundary condition required for a burner stabilized 1D simulation :

```
...
f = ct.BurnerFlame(gas, initial_grid)

f.burner.T = tburner
f.burner.X = reactants

mdot = 0.04
f.burner.mdot = mdot
...
```

Once your flame is properly defined, you need to specify the settings for the solver. This is done in exactly the same fashion as for the freely propagation premixed flat flame.

Next, several flames are computed in a row, with refined settings. Here also, the subsequent solutions are saved in between 'solve' callings; while the final solution is saved in a '.csv' file, to be able to visualize the evolution of the flame properties throughout the domain.

Results

See fig. 5.5 and 5.4. A complete script is provided in Appendix C.4.3.

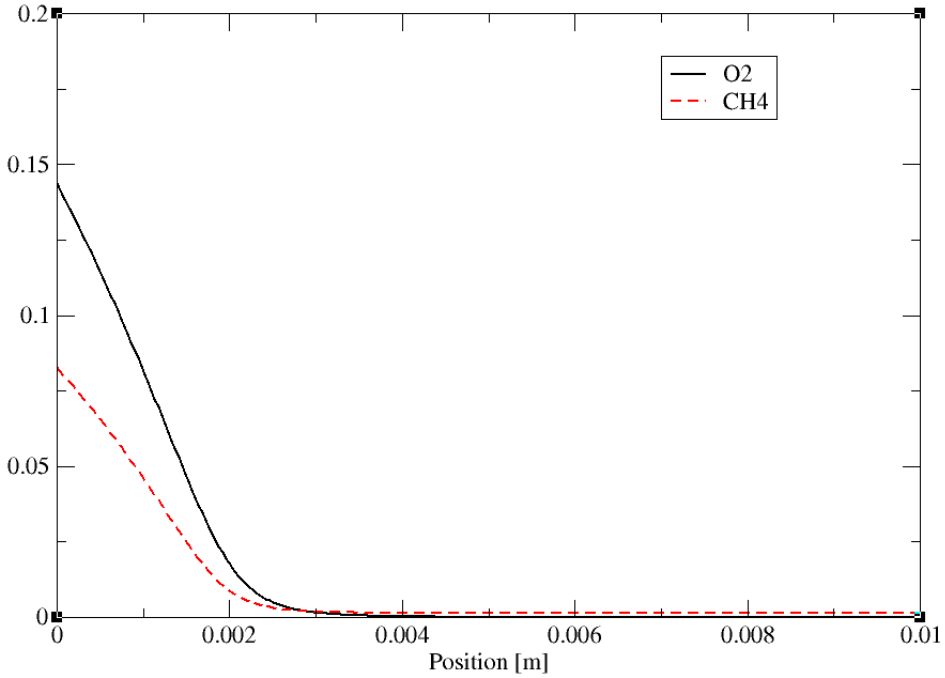

Figure 5.4: Evolution of the main species through a burner stabilized premixed flat flame, computed for a methane-air mixture at $\phi = 1.3$, Tini = 373 K and under atmospheric pressure with the GRIMech3.0 mechanism and the multicomponent diffusion enabled.

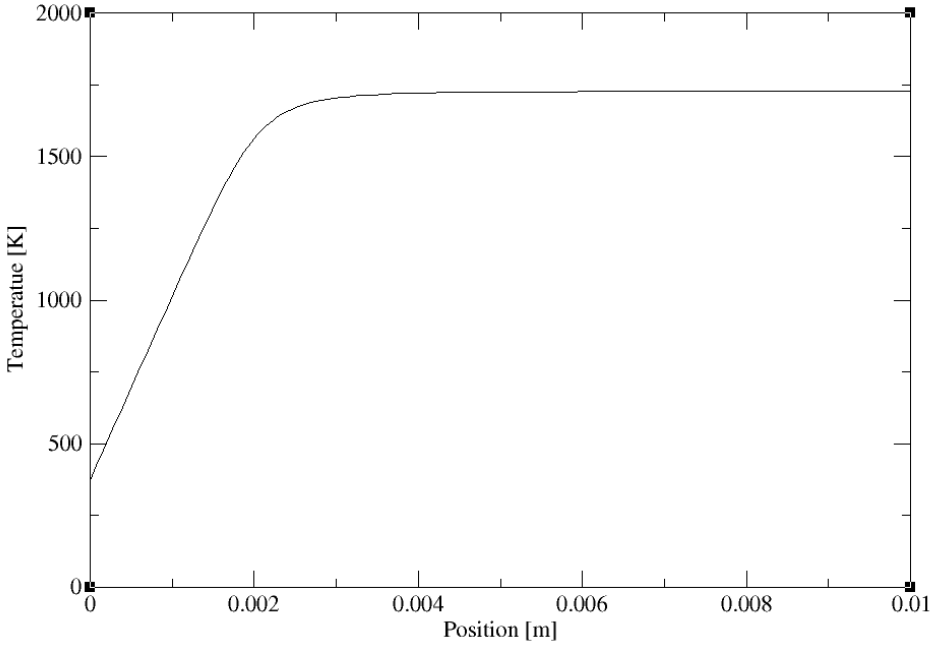

Figure 5.5: Evolution of the temperature through a burner stabilized premixed flat flame, computed for a methane-air mixture at $\phi = 1.3$, Tini = 373 K and under atmospheric pressure with the GRIMech3.0 mechanism and the multicomponent diffusion enabled.

5.5.3 A counterflow diffusion flame simulation

Introduction

This section will be dedicated to the simulation of a counterflow diffusion flame. This is the only configuration for diffusion flames that Cantera version 2.1.1 handles. We will use the GRIMech3.0 mechanism to generate an ethane-air mixture (C_2H_6).

A simple script

Go into the subfolder '1DcounterflowFlame' of the 'Tuto6' folder, and open the python script you will find there in your favorite text editor. Remember the steps to define a 1D simulation script with Cantera, section 5.2, with a few specifics in this case :

- a package section
- the gaseous object settings
- a 1D domain
- the flame components will be :
  - a gaseous composition
  - a set of equations to solve
  - inlets conditions (two inlets)
  - specific boundaries conditions (two inlets)
  - an initial profile

Try to understand each line of the script in regards of those aforementioned steps.

Note that there are now two boundaries : one for the fuel and another for the oxidizer. Try to be consistent with Cantera's logic and always put the fuel inlet on the left and the oxidizer inlet on the right...

Here we start with a domain of about 2 cm, with evenly spaced points. With those settings, the global strain rate will approximatively be equal to 40 s^{-1}.

Note, next, that you are required to specify the fuel for the initial guess: together with the inlet flow rates and composition, it will enable the construction of an initial solution. Be careful that the oxidizer and corresponding stoichiometry must be specified if it is not O_2 ! The initial guess is generated by assuming infinitely fast chemistry.

For the solver, the settings and procedure are the same as for the other type of flames we just reviewed: we start again by disabling the energy equation, then the tolerances for the Newton integrations are specified and a first flame without grid refinement is solved. The simulation is continued by first enabling the energy equation, and then lowering the grid refinements.

Intermediate solutions are systematically stored in a '.xml'. file while the final solution is stored in a '.csv' file for further visualizations.

Results

See fig. 5.6. complete script is provided in Appendix C.4.4.

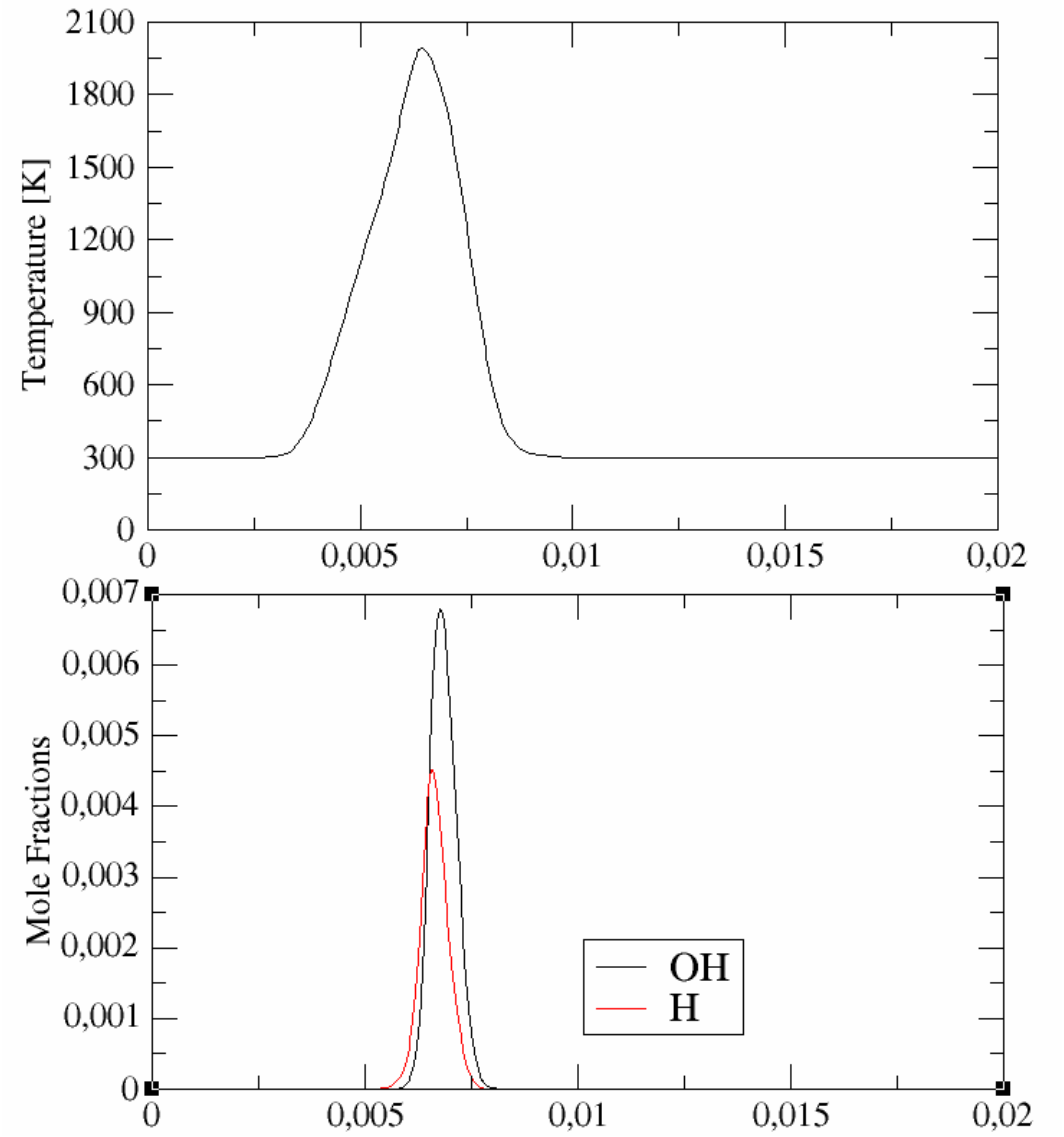

Figure 5.6: Spatial evolution of the temperature and some radicals mole fraction through a diffusion counterflow flame, computed for an ethane-air mixture at T fuel and T air = 300 K, under atmospheric pressure and for $a = 40\ s^{-1}$; with the GRIMech3.0 mechanism.

A. *Appendix: Standard entries and fields found in Cantera's data files*

A.1 Introduction

Remember from Tutorial 2 that a 'data file' is composed of three main parts:

- a phases and interfaces area
- a species area (with, possibily, an element area)
- a reactions area

In each one of those parts, specific information to define phases objects are specified, in the form of **entries** and **directives**, the most common of which we will detail hereafter. Each entry and directive is composed of **fields**, which are assigned values.

A.2 Phases and interfaces

There are several different types of entries, corresponding to different types of phases. Phases are created using one of the following:

1. ideal_gas
2. stoichiometric_solid
3. stoichiometric_liquid
4. metal
5. semiconductor
6. incompressible_solid
7. lattice

8. lattice_solid
9. liquid_vapor
10. redlich_kwong
11. ideal_interface
12. edge

These share many common features, however, which are contained in the phase class of the code.

Fields :

Name

The 'name' field is a string that identifies the phase. It must not contain any whitespace characters or reserved XML characters, and must be unique within the file among all phase definitions of any type. Phases are referenced by name when importing them into an application program, or when defining an interface between phases.

Declaring the elements

The elements that may be present in the phase or interface are declared in the 'elements' field. This must be a string of element symbols separated by spaces and/or commas. Each symbol must either match one listed in the database file elements.xml, or else match the symbol of an element entry defined elsewhere in the data file (See Elements). The elements.xml database contains most elements of the periodic table, with their natural-abundance atomic masses. It also contains a few isotopes (D, Tr), and an element for an electron (E). This pseudo-element can be used to specify the composition of charged species. Note that two-character symbols should have an uppercase first letter, and a lowercase second letter (e.g. Cu, not CU). It should be noted that the order of the element symbols in the string determines the order in which they are stored internally by Cantera. For some calculations, such as multi-phase chemical equilibrium, it is important to synchronize the elements among multiple phases, so that each phase contains the same elements with the same ordering. In such cases, simply use the same string in the 'elements' field for all phases:

```
ideal_gas(name = "gasmix",
          elements = "H C O N Ar",
          ...)
```

Declaring the species

The species that are present are declared in the 'species' field. They are not defined there, only declared. Species definitions may be imported from other files, or species may be defined locally using species entries elsewhere in the file. If a single string of species symbols is given, then it is assumed that these are locally defined. For each one, a corresponding species entry must be present somewhere in the file, either preceding or following the phase entry. Note that the string may extend over multiple lines by delimiting it with triple quotes.

```
species = """ H2  H  O  O2  OH  H2O  HO2  H2O2  C  CH
         CH2  CH2(S)  CH3  CH4  CO  CO2  HCO  CH2O  CH2OH  CH3O
         CH3OH  C2H  C2H2  C2H3  C2H4  C2H5  C2H6  HCCO  CH2CO  HCCOH
         N  NH  NH2  NH3  NNH  NO  NO2  N2O  HNO  CN
         HCN  H2CN  HCNN  HCNO  HOCN  HNCO  NCO  N2  AR  C3H7
         C3H8  CH2CHO  CH3CHO """,


# include all species defined in this file
species = 'all'
```

If the species are imported from another file, instead of being defined locally, then the string should begin with the file name (without extension), followed by a colon:

```
# import selected species from silicon.xml
species = "silicon: SI SI2 SIH SIH2 SIH3 SIH4 SI2H6"
```

In this case, the species definitions will be taken from file silicon.xml, which must exist either in the local directory or somewhere on the Cantera search path. It is also possible to import species from several sources, or mix local definitions with imported ones, by specifying a sequence of strings:

```
species = ["CL2 CL F F2 HF HCL", # defined in this file
           "air: O2 N2 NO", # imported from 'air.xml'
           "ions: CL- F-"] # imported from 'ions.xml'
```

Note that the strings must be separated by commas, and enclosed in square brackets or parentheses.

Declaring the reactions

The reactions amongst the species are declared in the 'reactions' field. Just as with species, reactions may be defined locally in the file, or may be imported from one or more other files. All reactions must only involve species that have been declared for the phase. If all reactions defined locally in a file are to be included in the phase definition, then assign the 'reactions' field the string 'all':

```
# import all reactions defined in this file
reactions = "all"

# import all reactions defined in rxns.xml
reactions = "rxns: all"
```

Unlike species, reactions do not have a name, but do have an optional 'id' field. If the 'id' field is not assigned a value, then when the reaction entry is read it will be assigned a four-digit string encoding the reaction number, beginning with '0001' for the first reaction in the file, and incrementing by one for each new reaction. If only some of the reactions defined in a file are to be included, then a range can be specified using the reaction 'id' fields :

```
# import reactions defined in this file
reactions = 'nox-12 to nox-24'

# import reactions 1-14 in rxns.xml
reactions = "rxns: 0001 to 0014"
```

It is also possible to import reactions from several sources, as for the species :

```
# import reactions from several sources
reactions = ["all",              # all local reactions
             "gas: all",         # all reactions in gas.xml
             "gri30: 0005 to 0008"] # reactions 5 to 8 in gri30.xml
```

Kinetics models for phases

A kinetics model is a set of equations to use to compute reaction rates. In most cases, each type of phase has an associated kinetics model that is used by default, and so the 'kinetics' field does not need to be assigned a value. For example, the ideal_gas entry has an associated kinetics model called GasKinetics that implements mass-action kinetics, computes reverse rates from thermochemistry for reversible reactions, and provides various pressure-independent and pressure-dependent reaction types. Other models could be implemented, and this field would then be used to select the desired model.

Transport models for phases

A transport model is a set of equations used to compute transport properties. For ideal_gas phases for example, two transport models are available; the one desired can be selected by assigning a string to this field.

option	meaning
'no_validation'	Turn off all validation. Use when the definition has been previously validated to speed up importing the definition into an application
'skip_undeclared_elements'	When importing species, skip any containing undeclared elements, rather than flagging them as an error
'skip_undeclared_species'	When importing reactions, skip any containing undeclared species, rather than flagging them as an error
'skip_undeclared_third_bodies'	When importing reactions with third body efficiencies, ignore any efficiencies for undeclared species, rather than flagging them as an error.
'allow_discontinuous_thermo'	Disable the automatic adjustment of NASA polynomials to eliminate discontinuities in enthalpy and entropy at the midpoint temperature.

Table A.1: Summary of the 'options' fields for phases entries

```
ideal_gas(name="gas1",
          ...,
          transport="Multi", # use multicomponent formulation
          ...)

ideal_gas(name="gas2",
          ...,
          transport="Mix", # use mixture-averaged formulation
          ...)
```

Options for phases

The 'options' field is used to indicate how certain conditions should be handled when importing the phase definition. The 'options' field may be assigned a string or a sequence of strings from the table A.2.

Phases for interfaces

The 'phases' field specifies the bulk phases that participate in the heterogeneous reactions. Although in most cases this string will list one or two phases, no limit is placed on the number. This is particularly useful in some electrochemical problems, where reactions take place near the triple-phase bounday where a gas, an electrolyte, and a metal all meet.

Site density for interfaces

The 'site_density' field is the number of adsorption sites per unit area.

Initial state

The phase or interface may be assigned an initial state to which it will be set when the definition is imported into an application and an object created. This is done by assigning field 'initial_state' an embedded entry of type state, which will differ depending on whether a phase or an interface is under consideration. Usually, temperature and pressure will be used for a gas :

```
ideal_gas(name = "gas",
      ...
      initial_state = state(temperature = 300.0,
                        pressure = OneAtm)    )
```

Most of the attributes defined here are immutable, meaning that once the definition has been imported into an application, they cannot be changed by the application. For example, it is not possible to change the elements or the species. The temperature, pressure, and composition, however, are mutable they can be changed. This is why the field defining the state is called the 'initial_state'; the object in the application will be initially set to this state, but it may be changed at any time.

Ideal gas mixtures :

Many combustion simulations make use of reacting ideal gas mixtures. These can be defined using the ideal_gas entry seen above. The Cantera ideal gas model allows any number of species, and any number of reactions among them. An example of an ideal_gas entry is shown below:

```
ideal_gas(name='NOx-submech',
          elements='N O Ar',
          species='gri30: N2 O2 N O NO NO2 N2O AR',
          reactions='gri30:all',
          transport='Mix',
          initial_state=state(temperature=500.0,
                              pressure=(1.0, 'atm'),
                              mole_fractions='N2:0.78, O2:0.21, AR:0.01'),
          options = ('skip_undeclared_elements',
```

```
                    'skip_undeclared_species',
                    'skip_undeclared_third_bodies'))
```

This entry defines an ideal gas mixture that contains 8 species, the definitions of which are imported from file 'gri30.xml'. Transport properties are to be computed using mixture rules, and the state of the gas is to be set initially to 500 K, 1 atm, and an air-like composition. As previously stressed, two transport models can be used, and only one kinetic model is implemented; so that it is not necessary to define it.

A.3 Elements

The element entry defines an element or an isotope of an element. Note that these entries are not often needed, since the the database file 'elements.xml' that comes with Cantera is searched for element definitions when importing phase and interface definitions. An explicit element entry is needed only if an isotope that is not in this file is required:

```
element(symbol='C-13',
        atomic_mass=13.003354826)
element("O-!8", 17.9991603)
```

A.4 Species

Species are a top-level kind of entry of the data file. Their definitions are not embedded in a phase or interface entry.

Fields :

Name

The 'name' field may contain embedded parentheses, + or - signs to indicate the charge, or just about anything else that is printable and not a reserved character in XML. Some example name specifications:

```
name = 'CH4'
name = 'KERO'
name = 'argon_2+'
name = 'C(s)'
```

Elemental composition :

The elemental composition is specified in the atoms entry, as follows:

```
atoms = "C:1 O:2"             # CO2
atoms = "C:1, O:2"            # CO2 with optional comma
```

For surface species, it is possible to omit the 'atoms' field entirely, in which case it is composed of nothing, and represents an empty surface site. This can also be done to represent vacancies in solids. A charged vacancy can be defined to be composed solely of electrons:

```
species(name = 'oxygen-vacancy',
        atoms = 'O:0, E:2',
        ...)
```

Note that an atom number of zero may be given if desired, but is completely equivalent to omitting that element. The number of atoms of an element must be non-negative, except for the special element E that represents an electron.

Thermodynamic properties :

The phase and ideal_interface entries discussed in the last section implement specific models for the thermodynamic properties appropriate for the type of phase or interface they represent. Although each one may use different expressions to compute the properties, they all require thermodynamic property information for the individual species. For the phase types implemented at present, the properties needed are:

1. The molar heat capacity at constant pressure ${c^0}_p(T)$ for a range of temperatures at a reference pressure P_0;
2. The molar enthalpy $h^{(T_0,P_0)}$ at a reference pressure P_0 a reference temperature T_0;
3. The absolute molar entropy $s^{(T_0,P_0)}$ at (T_0, P_0).

Several entries exists to implement different parametrizations (functional form) for the heat capacity. Note that there is no requirement that all species in a phase use the same parametrization; each species can use the one most appropriate to represent how the heat capacity depends upon the temperature. Currently, three entry types are implemented, all of which provide species properties appropriate for models of ideal gas mixtures, ideal solutions, and pure compounds. Non-ideal phase models are not yet implemented, but may be in future releases. When they are, additional entry types may also be added that provide species-specific coefficients required by specific non-ideal equations of state. The only one worth reviewing here is the 'NASA' parametrization, which represents ${c^0}_p(T)$ with a fourth-order polynomial. In Cantera, it is defined by an embedded 'NASA' entry. Very often, two NASA parametrizations are used for two contiguous temperature ranges. This can be specified by assigning the 'thermo' field of the species entry a sequence of two NASA entries:

```
# use one NASA parameterization for T < 1000 K, and another for T > 1000 K.
species(name = "O2",
        atoms = " O:2 ",
        thermo = (
            NASA( [ 200.00, 1000.00], [ 3.782456360E+00, -2.996734160E-03,
                    9.847302010E-06, -9.681295090E-09, 3.243728370E-12,
                    -1.063943560E+03, 3.657675730E+00] ),
            NASA( [ 1000.00, 3500.00], [ 3.282537840E+00, 1.483087540E-03,
                    -7.579666690E-07, 2.094705550E-10, -2.167177940E-14,
                    -1.088457720E+03, 5.453231290E+00] ) ) )
```

In some cases, species properties may only be required at a single temperature or over a narrow temperature range. In such cases, the heat capacity can be approximated as constant, and the 'const_cp' parameterization.

A.5 Reactions

Cantera supports a number of different types of reactions, including several types of homogeneous reactions, surface reactions, and electrochemical reactions. For each, there is a corresponding entry type. The simplest entry type is 'reaction', which can be used for any homogeneous reaction that has a rate expression that obeys the law of mass action, with a rate coefficient that depends only on temperature. All of the entry types that define reactions share some common features.

Fields :

Reaction equation

The reaction equation determines the reactant and product stoichiometry. A relatively simple parsing strategy is currently used, which assumes that all coefficient and species symbols on either side of the equation are delimited by spaces:

```
2 CH2 <=> CH + CH3      # OK
2 CH2<=>CH + CH3        # OK
2CH2 <=> CH + CH3       # error
CH2 + CH2 <=> CH + CH3  # OK
2 CH2 <=> CH+CH3        # error
```

The incorrect versions here would generate undeclared species errors and would halt processing of the data file. In the first case, the error would be that the species $2CH_2$ is undeclared, and in the second case it would be species $CH + CH_3$. Whether the reaction is reversible or not is determined by the form of the equality sign in the reaction equation. If either <=> or= is found, then the reaction is regarded as reversible, and the reverse rate will be computed from detailed balance. If, on the other hand, => is found, the reaction will be treated as irreversible.

Reaction rate

The rate coefficient is specified with an embedded entry corresponding to the rate coefficient type. At present, the only implemented type is the modified Arrhenius function, which is defined with an 'Arrhenius' entry:

```
rate_coeff = Arrhenius(A=1.0e13, n=0, E=(7.3, 'kcal/mol'))
rate_coeff = Arrhenius(1.0e13, 0, (7.3, 'kcal/mol'))
```

As a shorthand, if the 'rate_coeff' field is assigned a sequence of three numbers, these are assumed to be (A,n,E) in the modified Arrhenius function:

```
rate_coeff = [1.0e13, 0, (7.3, 'kcal/mol')] # equivalent to above
```

The units of the pre-exponential factor A can be specified explicitly if desired.

The identifying string An optional identifying string can be entered in the 'id' field, which can then be used in the reactions field of a phase or interface entry, to identify this reaction. If omitted, the reactions are assigned 'id' strings as they are read in, beginning with '0001', '0002', etc. Note that this 'id' string is only used when selectively importing reactions.

The options Certain conditions are normally flagged as errors by Cantera. In some cases, they may not be errors, and the 'options' field can be used to specify how they should be handled.

1. The 'skip' option can be used to temporarily remove this reaction from the phase or interface that imports it, just as if the reaction entry were commented out.

2. Normally, when a reaction is imported into a phase, it is checked to see that it is not a duplicate of another reaction already present in the phase, and an error results if a duplicate is found. But in some cases, it may be appropriate to include duplicate reactions, for example if a reaction can proceed through two distinctly different pathways, each with its own rate expression. If the 'duplicate' option is specified, then the reaction must have a duplicate. Any reaction that specifies that it is a duplicate, but cannot be paired with another reaction generates an error.

3. Cantera normally does not allow negative pre-exponential factors. But if there are duplicate reactions such that their total rate is positive, then negative A parameters are acceptable, as long as the 'negative_A' option is specified.

Efficiencies

In a three-body reaction, different species may be more or less effective in acting as the collision partner. These effects can be accounted for by defining a collision efficiency for each species, through the 'efficiencies' field. Some examples from GRI-Mech 3.0 are shown below:

```
three_body_reaction( "2 O + M <=> O2 + M", [1.20000E+17, -1, 0],
                     " AR:0.83 C2H6:3 CH4:2 CO:1.75 CO2:3.6 H2:2.4 H2O:15.4 ")

three_body_reaction( "O + H + M <=> OH + M", [5.00000E+17, -1, 0],
                     efficiencies = " AR:0.7 C2H6:3 CH4:2 CO:1.5 CO2:2 H2:2 H2O:6 ")

three_body_reaction(
    equation = "H + OH + M <=> H2O + M",
    rate_coeff = [2.20000E+22, -2, 0],
    efficiencies = " AR:0.38 C2H6:3 CH4:2 H2:0.73 H2O:3.65 "
)
```

As always, the field names are optional if the field values are entered in the declaration order.

Falloff parameterization

A falloff reaction is one that has a rate that is dependent upon pressure and temperature. This type of reaction have a 'falloff' field that can be specified through different embedded entries in Cantera. Usually, a 'Troe' entry will be specified :

```
falloff_reaction( "H + CH2 (+ M) <=> CH3 (+ M)",
         kf = [6.00000E+14, 0, 0],
         kf0   = [1.04000E+26, -2.76, 1600],
         falloff = Troe(A = 0.562, T3 = 91, T1 = 5836, T2 = 8552),
         efficiencies = " AR:0.7  C2H6:3  CH4:2  CO:1.5  CO2:2  H2:2  H2O:6 ")
```

Note that if the 'falloff' is omitted in a falloff reaction entry, then a unity falloff function (Lindemann form) is assumed.

B. *Appendix: Standard Python commands for Cantera*

When working with the Python interface, the Python rules of syntax apply. If you are not familiar with this language, do not panic: it will not take you more than a few hours to get acquainted with the few important basic rules. This section is meant to provide you with every Python information you will need to properly use Cantera. The rest is optional, and can help you save time during calculation, or present it in a more elegant way... Nothing you cannot do without !

B.1 Generalities

B.1.1 Indented commands

The first thing you should know is that Python is very fond of intents. When you design a script, every new information needs to be put on a new line, with the same indent than the block of information it belongs to. For example, whenever you start an "if" statement, you will need to indent the subsequent block of information :

```
if expression:
   statement(s)
else:
   statement(s)
```

Notice the ":" following the "if" condition. This is mandatory, and will tell the code that the statements' enunciation will be provided next. Also, note that you do not need and "end" statement: stopping the indentation will inform the code that it is out of the "if" statements.

B.1.2 Comment your results

As in every python script, the # character is used to comment a line. Everything on its right will be ignored by the compiler. Use it to comment properly your scripts or input_files so as to recall what you define:

```
##################################################################
```

```
# Create the gas object
gas = . . .
```

You could also enclose several lines with three " ' ", so that everything in between is commented out. Be careful, though, it does not work for indented commands. Here is an example :

```
#################
#Third flame:
#Uncommented
f.energy_enabled = True
f.set_refine_criteria(ratio=2, slope=0.2, curve=0.2, prune=0.04)

#and solve the problem again
f.solve(loglevel = 1, refine_grid = True)

#save your results
f.save('c2h6_diffusion.xml','energy continuation')
```

VS

```
'''
#################
#Third flame:
#Commented
f.energy_enabled = True
f.set_refine_criteria(ratio=2, slope=0.2, curve=0.2, prune=0.04)

#and solve the problem again
f.solve(loglevel = 1, refine_grid = True)

#save your results
f.save('c2h6_diffusion.xml','energy continuation')
'''
```

B.1.3 Import packages

Another important feature of Python is that depending on what kind of application you will use it for, you will need to use specific functions or functionalities. Those come in the form of "packages" that

needs to be "loaded" whenever needed, usually at the beginning of a script. It is the case when using Cantera through Python : Cantera is just a "package" of functions and objects ! You can import a package in two different ways :

```
from cantera import *
```

or

```
import cantera as ct
```

The second way is preferred when several packages are loaded, some of which might share an object with the same name. In such a case, access the right one with 'ct.' at the beginning :

```
import cantera as ct

gas = ct.Solution('gri30.cti')
```

Now, when working with Cantera, two other packages are usually required. First off, numpy, which is the fundamental package for scientific computing with Python :

```
from numpy import *
```

or

```
import numpy as np
```

and matplotlib, which is the package that will allow you to plot your results :

```
from matplotlib.pylab import *
```

or

```
import matplotlib.pylab as plt
```

The "sys" package and the "csv" package to display and save your results might also be required. Those packages can be loaded with the simple command :

```
import sys
import csv
```

B.1.4 String tricks

A string can be enclosed in double or single quotes, but they must match. To create a string containing single quotes, enclose it in double quotes and vice versa.

```
>>> print 'I love "NY"'
I love "NY"
>>> print "I love 'NY'"
I love 'NY'
```

Another interesting Python feature is the possibility to break a line into several ones :

```
>>> print 'I love "NY"'
I love "NY"
>>> print ('I'
... ' love'
... ' "NY"')
I love "NY"
```

Specifically, with Cantera, try to use the multi-line form whenever possible to make everything easier to understand and modify. Prefer :

```
species = """ CH4   O2   CO2   H2O  CO   H2
              HO2   H    OH    O    CH3  H2O2
              CH2O  HCO  CH3O  N2""",
```

to the equivalent :

```
species = """ CH4  O2  CO2  H2O  CO  H2  HO2  H  OH  O  ...""",
```

B.1.5 Variables

Variables can be used to specify a field that might be re used and/or very long and complicated to write. For example, if multiple phases are to be defined that all need to be at the same initial temperature, define a temperature variable, so that subsequent temperature fields can easily be initialized :

```
T_0 = (300, 'K')
...
...
phaseOne(
    ...
```

```
    temperature = T_0
    )
phaseTwo(
    ...
    temperature = T_0
    )
...
```

B.1.6 Loops

In order to perform several calculations in a row, you will need to loop over an array of relevant datas, and perform a simulation every time. This is done through a "for" or "if" loop, with Python. Let us take an example : say we need to perform several free propagating flame calculations, with the pressure as the varying parameter for example. The script will look something like this :

```
#import
...(import packages)...
#prepare your run
...(set variables)...
nb_P = 25
P_min = 1
P_max = 50
P     = np.zeros(nb_P,'d')
for j in range(0,nb_P):
    P[j]   = P_min + (P_max - P_min)*j/(nb_P - 1)
    gas.TPX = 300, P[j], 'H2:2., O2:1., N2:3.76'

    #Create the free laminar premixed flame
    f = FreeFlame(gas, initial_grid)
    ...(settings and solve)...
```

The rules of indentation discussed previously apply, and don't forget the ":" at the end of the "if" expression. The "if" expression here is given by a

```
range(...)
```

In python, it specifies that the loop will occur 'nb_P' - 0 = 25 - 0 times, with the variable 'j' taking on different values '0', '1', ..., 'nb_P-1' each loop. This 'j' is used inside the loop to seek for the right pressure 'P[j]' value, so that a calculation is performed inside each loop with the right gaseous settings.

Now, the loops could also be on the number of points of your simulation :

```
simulation.n_points
```

Or the number of species, reactions, etc. :

```
gas.species_names   # returns a list of species names
gas.n_species       # number of species
gas.n_reactions     # number of reactions
gas.X               # returns an array of moles fractions
gas.Y               # returns an array of mass fractions
```

But the syntax will remain the same, and will look something like this :

```
for j in range(0,gas.n_species):
    ...(statement(s))...
```

You could also specify that you want to loop through a subset of the number of species in your gas. For example in :

```
for j in range(2,gas.n_species):
    ...(statement(s))...
```

the variable 'j' will take on values from '2' to the number of species - 1.

This way of performing a loop is classic, and will be used throughout the tutorials.

B.2 Storage

B.2.1 Storage variables

In python, whenever you need to store information, you will do so through a variable or a multi-dimensional array. We have seen in the previous paragraph that defining a variable is straightforward; but an array needs to be defined prior to its utilization. Usually, an array of the right length, initialized with '0' will be created, with the help of the numpy package :

```
import numpy as np


...


xeq = np.zeros(gas.n_species)
```

Here, 'xeq' will be an array filled with '0' of length 'gas.n_species'; so, of length the number of species that your gas contains. Remember that indexing the arrays starts at '0', so that in this example, the last array's value will have index 'gas.n_species - 1'. Keeping this in mind, the array's values can be specified or re-specified whenever necessary. For example :

```
xeq[2] = gas.X[2]
```

will store the value of the mole fraction of the *third* species of your mechanism, into the *third* array's value.

Now, mutlidimensional arrays can also be defined. This is done with the simple command :

```
2D_array = np.zeros((gas.n_species,nb_points))
3D_array = np.zeros((gas.n_species,nb_points1,nb_points2))
...
```

This will instantiate multidimensional arrays, of dimensions specified in between the double parenthesis, with zeroes.

B.2.2 Save and display relevant information : loops

Basically, the guidelines are the same than what was discussed in subsection B.1.6.

B.2.3 Save and display relevant information : the %arg operator

Now, when storing information, the %arg operator can prove to be very useful for string formatting. It interprets the argument like a format string to be applied to a following list, and returns the string resulting from this formatting operation. The argument should specify the format : "s" if it is a string; 'X.xf' if it is a double, the 'X' specifying the maximum numbers before the comma and the 'x' specifying the precision after the comma; 'X.xe' if it is a double in the form of an exponential. See the following ways of storing the temperature and species mole fractions of a simulation :

```
# ----------------------------------------------------------------------
#  Print the temperature and mole fractions in
#  a very conservative way
# ----------------------------------------------------------------------
...
gas.equilibrate("HP", solver = 'vcs')


# Balance record
```

```
x       = gas.X
tad     = gas.T


csv_file = open('yourfile.csv','w')
csv_file.write('%s' % ('T (K)'))
for item in gas.species_names:
    csv_file.write(' %s' % (item))
csv_file.write("\n")
csv_file.write('%5.0f' % (tad))
for i in range(gas.n_species):
    csv_file.write(' %10.5e' % (x[i]))
print('Output written to yourfile.csv')
```

This will create a "yourfile.csv" in your working directory, the content of which should be something like :

```
T H OH HO2 H2 H2O H2O2 ...  P2 P2- P2-H
 2348 6.26214e-04 ... 4.14561e-133 5.22651e-133 8.65372e-140
```

So you can see that, even with a very mediocre knowledge of Python, and the attributes' names of the object relevant to your simulations -which can be found through the Cantera's search tool *http://www.cantera.org/docs/sphinx/html/search.html*- you can easily extract the information you seek.

B.2.4 Save relevant information : the 'csv' package

Now, importing the 'csv' package at the beginning of your script could help you and make you save time. See the following way of storing the same information as we did in the previous subsection :

```
# ----------------------------------------------------------------------
#  Print the temperature and mole fractions in
#  a more clever way
# ----------------------------------------------------------------------
import csv
...
gas.equilibrate("HP", solver = 'vcs')


# Balance record
```

```
x       = gas.X
tad     = gas.T


csv_file = 'yourfile.csv'
with open(csv_file, 'w') as outfile:
    writer = csv.writer(outfile)
    writer.writerow(['T (K)'] + gas.species_names)
    writer.writerow([tad] + list(x[i]))
```

See *https://docs.python.org/2/library/csv.html*, for more information on the 'csv' module.

Note that there are thousands of python "tricks" that can make your life easier, but as this is not the goal of this session, we will not expand further; just remember that the internet is full of python addicts that will provide answers to all of your questions.

B.3 Display tricks

With python, it is possible to print information directly onto your terminal, to help you check convergence, monitor a particular variable's evolution ... Here again, the %arg function could come in handy :

```
print "******************************************************* "
print "    Initial state :"
print "******************************************************* "
print "P =  " , "%10.4e  "%  (gas.P)+"    Pa"
print "T =  " , "%10.4e  "%  (gas.T)+"    K"
print "V =  " , "%10.4e  "%  (gas.volume_mass)+"    m3/kg"
print "U =  " , "%10.4e  "%  (gas.int_energy_mass)+"    J/kg"
print "H =  " , "%10.4e  "%  (gas.enthalpy_mass)+"    J/kg"
print "S =  " , "%10.4e  "%  (gas.entropy_mass)+"    J/kg/K"
print ""
print ""
```

Will return something like :

```
*******************************************************
    Initial state :
*******************************************************
```

```
P =   1.0000e+05      Pa
T =   4.0000e+02      K
V =   1.2035e+00      m3/kg
U =   -2.6590e+05      J/kg
H =   -1.4554e+05      J/kg
S =   7.5652e+03      J/kg/K
```

B.4 Plots with Matplotlib

As explained before, you will need to start by loading the matplotlib package at the beginning of your script :

```
from matplotlib.pylab import *
```

Next, all of the plotting functionalities it offers will be available to you. Information can be found online: basically, everything you will want to plot can be done. Here is an example of adiabatic flame temperature versus equivalence ratio for a H_2/Air mixture:

```
...(simulation)...
plot(phi, tad, '-')
title(r'Tad vs. $\Phi$ for $H_{2}/Air$ flames, at P = 1 atm, Tin = 300 K')
xlabel(r'$\Phi$', fontsize=20)
ylabel("Temperature [T]")
show()
```

The 'plot' function in the first line takes two arrays of same size in argument, and displays the evolution of the second one in function of the first one. The '-' argument will generate a dotted line plot. The second line specifies the title of the plot. The third and fourth lines will give labels to the axis. The 'show()' function will display the graph automatically. Sometimes, you might prefer not to be disturbed by plots popping out. In such a case, you could save them instead :

```
#savefig('myplot.png', bbox_inches='tight')
```

Importing the "sys" package at the beginning of your script, along with matplotlib, you could interact with it via arguments passed in your terminal. You could then design a specific plot area at the end of your script, with indentation :

```
if '--plot' in sys.argv:
    ...(plot options)...
```

So that launching the script from your terminal with the '–plot' argument :

```
$ python script_plot.py --plot
```

will result in a plot popping at the end of your simulation, while omitting it will only perform the simulation.

C. *Appendix: Solution to the exercises*

C.1 Tutorial 2

C.1.1 List of h2_sandiego.cti errors

1. bar in line 9 is not in a string format
2. Reaction 21 is not commented out line 237
3. the embedded 'Troe' entry in the 'falloff' field line 223 does not have the right format for its fields. Names of the fields should be specified, as the first one is.

C.1.2 submechanisms.cti

Let us work with a new file, 'submechanisms.cti'. Define :

```
ideal_gas(name = 'hydrogen_submech',
          elements = 'H O',
          species = 'gri30:all',
          reactions = 'gri30:all',
          options = ('skip_undeclared_elements',
                     'skip_undeclared_species',
                     'skip_undeclared_third_bodies'))
```

If we load it into a python environment as we did in the first tutorial, we get a gas mixture containing the 8 species (out of 53 total) that contain only H and O:

```
>>> import cantera as ct
>>> gas = ct.Solution('submechanisms.cti','hydrogen_submech')
>>> gas()


  hydrogen_submech:
```

```
       temperature           0.001  K
          pressure      0.00412448  Pa
           density           0.001  kg/m^3
  mean mol. weight         2.01588  amu

                        1 kg            1 kmol
                     -----------      ------------
         enthalpy     -3.786e+06       -7.632e+06     J
   internal energy     -3.786e+06       -7.632e+06     J
           entropy        6210.88        1.252e+04     J/K
    Gibbs function     -3.786e+06       -7.632e+06     J
 heat capacity c_p        9669.19        1.949e+04     J/K
 heat capacity c_v         5544.7        1.118e+04     J/K

                          X                Y         Chem. Pot. / RT
                    -------------     ------------     ------------
                 H2             1                1          -917934
                  H             0                0
                  O             0                0
                 O2             0                0
                 OH             0                0
                H2O             0                0
                HO2             0                0
               H2O2             0                0
```

Printing the equations returns :

```
>>> for i in range(gas.n_reactions):
...   print gas.reaction_equation(i)
...
2 O + M <=> O2 + M
O + H + M <=> OH + M
O + H2 <=> H + OH
O + HO2 <=> OH + O2
O + H2O2 <=> OH + HO2
```

```
H + O2 + M <=> HO2 + M

H + 2 O2 <=> HO2 + O2

H + O2 + H2O <=> HO2 + H2O

H + O2 <=> O + OH

2 H + M <=> H2 + M

2 H + H2 <=> 2 H2

2 H + H2O <=> H2 + H2O

H + OH + M <=> H2O + M

H + HO2 <=> O + H2O

H + HO2 <=> O2 + H2

H + HO2 <=> 2 OH

H + H2O2 <=> HO2 + H2

H + H2O2 <=> OH + H2O

OH + H2 <=> H + H2O

2 OH (+ M) <=> H2O2 (+ M)

2 OH <=> O + H2O

OH + HO2 <=> O2 + H2O

OH + H2O2 <=> HO2 + H2O

OH + H2O2 <=> HO2 + H2O

2 HO2 <=> O2 + H2O2

2 HO2 <=> O2 + H2O2

OH + HO2 <=> O2 + H2O
```

It is exactly the same to extract a "NOx-less" mechanism for methane/air combustion :

```
ideal_gas(name = 'NOx-less',
          elements = 'gri30:all,
          species = 'gri30:H2  H  O  O2  OH  H2O  HO2  H2O2  C  CH
                   CH2  CH2(S)  CH3  CH4  CO  CO2  HCO  CH2O  CH2OH  CH3O
                   CH3OH  C2H  C2H2  C2H3  C2H4  C2H5  C2H6  HCCO  CH2CO  HCCOH
                   N2  AR  C3H7 C3H8  CH2CHO  CH3CHO',
          reactions = 'gri30:all',
          options = ('skip_undeclared_elements',
                     'skip_undeclared_species',
                     'skip_undeclared_third_bodies'))
```

...

C.2 Tutorial 3

C.2.1 equil_simple.py

```
# homogeneous equilibrium of a gas

import cantera as ct

# create an object representing the gas phase
gas = ct.Solution('gri30.cti')

compo = {}
compo['CH4'] = 0.5
compo['O2'] = 1
compo['N2'] = 3.76

# set initial state
gas.TPX = (300.0, 1.0e05, compo)

gas.equilibrate('TP')

print(gas())
```

C.2.2 AdiabaticFlameT.py

```
"""
Adiabatic flame temperature and equilibrium composition for a fuel/air mixture
as a function of equivalence ratio.
"""

import cantera as ct
import matplotlib.pyplot as plt
import numpy as np
import sys
import csv
```

```
################################################################################
# Edit these parameters to change the initial temperature, the pressure, and
# the phases in the mixture.

# Import the gas :
gas = ct.Solution('mech.cti')

# Choose the equivalence ratio range :
phi_min = 0.3
phi_max = 3
npoints = 100

# Set the gas composition :
T = 300.0
P = 101325.0

# find fuel, nitrogen, and oxygen indices
# to set the composition of the gas
fuel_species = 'C2H4'

################################################################################

# Create some arrays to hold the data with numpy
# 1D arrays : phi, tad
phi = np.zeros(npoints)
tad = np.zeros(npoints)

# 2D arrays :
xeq = np.zeros((gas.n_species, npoints))

# Start the loop on Phi :
for i in range(npoints):
    # Start with setting the composition of the gas
    air_N2_O2_molar_ratio = 3.76
```

```
    phi[i] = phi_min + (phi_max - phi_min) * i / (npoints - 1)

    gas.set_equivalence_ratio(phi[i], {fuel_species: 1}, {'O2': 1, 'N2': air_N2_O2_molar_ratio

    gas.TP = T, P

    # Equilibrate the mixture adiabatically at constant P
    # with the solver vcs
    gas.equilibrate('HP')

    # Save the adiabatic temperature each time
    tad[i] = gas.T
    # you can even print it on screen
    # print("At phi = ","%10.4f"% (phi[i])+ "  Tad = ","%10.4f"% (tad[i]))

    # You could also save the equilibrium mass fractions at each
    # phi, as long as you initialized an array :
    xeq[:, i] = gas.X

# Save your results in a CSV file
csv_file = 'yourfile.csv'
with open(csv_file, 'w') as outfile:
    writer = csv.writer(outfile)
    writer.writerow(['Phi', 'T (K)'] + gas.species_names)
    for i in range(npoints):
        writer.writerow([phi[i], tad[i]] + list(xeq[:, i]))
print("Output written to", "%s" % csv_file)

# Add a plot option
if '--plot' in sys.argv:

    plt.plot(phi, tad, '-')
```

```
    plt.title(r'Tad vs. $\Phi$ for $C_{2}H_{4}/Air$ flames')
    plt.xlabel(r'$\Phi$', fontsize=20)
    plt.ylabel("Adiabatic flame temperature (K)")

    plt.grid()

    # plt.show()
    plt.savefig('plot_tad.png', bbox_inches='tight')
```

C.3 Tutorial 4

C.3.1 reactorUV.py

```
"""
Constant-volume, adiabatic kinetics simulation.
"""

import sys
import numpy as np
import cantera as ct
import matplotlib.pyplot as plt

#############################################################################

# Mechanism used for the process
gas = ct.Solution('gri30.cti')

# Gas state
gas.TPX = 1000.0, ct.one_atm, 'CH4:0.5,O2:1,N2:3.76'

# Create Reactor and fill with gas
r = ct.IdealGasReactor(gas)

# Prepare the simulation with a ReactorNet object
```

```
sim = ct.ReactorNet([r])
time = 4.e-1

# Arrays to hold the datas
times = np.zeros(200)
data = np.zeros((200, 4))

# Advance the simulation in time
# and print the internal evolution of temperature, volume and internal energy
print(('%10s %10s %10s %14s' % ('t [s]', 'T [K]', 'vol [m3]', 'u [J/kg]')))
for n in range(200):
    time += 5.e-3
    sim.advance(time)
    times[n] = time  # time in s
    data[n, 0] = r.T
    data[n, 1:] = r.thermo['O2', 'H', 'CH4'].X
    print(('%10.3e %10.3f %10.3f %14.6e' % (sim.time, r.T,
                                            r.thermo.v, r.thermo.u)))

# Plot the results if matplotlib is installed.
if '--plot' in sys.argv[1:]:

    plt.clf()
    plt.subplot(2, 2, 1)
    plt.plot(times, data[:, 0])
    plt.xlabel('Time (ms)')
    plt.ylabel('Temperature (K)')
    plt.subplot(2, 2, 2)
    plt.plot(times, data[:, 1])
    plt.xlabel('Time (ms)')
    plt.ylabel('O2 Mole Fraction')
    plt.subplot(2, 2, 3)
    plt.plot(times, data[:, 2])
    plt.xlabel('Time (ms)')
```

```
    plt.ylabel('H Mole Fraction')
    plt.subplot(2, 2, 4)
    plt.plot(times, data[:, 3])
    plt.xlabel('Time (ms)')
    plt.ylabel('CH4 Mole Fraction')
    plt.tight_layout()
    plt.show()
else:
    print("To view a plot of these results, run this script with the option --plot")
```

C.3.2 Modified reactorHP.py

```
"""
Constant-pressure, adiabatic kinetics simulation.
"""

import sys
import numpy as np
import cantera as ct
import matplotlib.pyplot as plt

#############################################################################

# Mechanisms used for the process
gri3 = ct.Solution('gri30.xml')
air = ct.Solution('air.xml')

# Gas state
gri3.TPX = 1000.0, ct.one_atm, 'CH4:0.5,O2:1,N2:3.76'

# Create Reactors used for the process and initialize them
r = ct.IdealGasReactor(gri3)
env = ct.Reservoir(air)

# Define a wall between the reactor and the environment, and
```

```
# make it flexible, so that the pressure in the reactor is held
# at the environment pressure.
w = ct.Wall(r, env)
w.expansion_rate_coeff = 1.0e6  # set expansion parameter. dV/dt = KA(P_1 - P_2)
w.area = 1.0

# Prepare the simulation with a ReactorNet object
sim = ct.ReactorNet([r])
time = 4.e-1

# Arrays to hold the datas
times = np.zeros(200)
data = np.zeros((200, 4))

# Advance the simulation in time
print(('%10s %10s %10s %14s' % ('t [s]', 'T [K]', 'P [Pa]', 'h [J/kg]')))
for n in range(200):
    time += 5.e-3
    sim.advance(time)
    times[n] = time  # time in s
    data[n, 0] = r.T
    data[n, 1:] = r.thermo['O2', 'H', 'CH4'].X
    print(('%10.3e %10.3f %10.3f %14.6e' % (sim.time, r.T,
                                            r.thermo.P, r.thermo.h)))

# Plot the results if matplotlib is installed.
if '--plot' in sys.argv[1:]:

    plt.clf()
    plt.subplot(2, 2, 1)
    plt.plot(times, data[:, 0])
    plt.xlabel('Time (ms)')
    plt.ylabel('Temperature (K)')
    plt.subplot(2, 2, 2)
```

```
    plt.plot(times, data[:, 1])
    plt.xlabel('Time (ms)')
    plt.ylabel('O2 Mole Fraction')
    plt.subplot(2, 2, 3)
    plt.plot(times, data[:, 2])
    plt.xlabel('Time (ms)')
    plt.ylabel('H Mole Fraction')
    plt.subplot(2, 2, 4)
    plt.plot(times, data[:, 3])
    plt.xlabel('Time (ms)')
    plt.ylabel('CH4 Mole Fraction')
    plt.tight_layout()
    plt.show()
else:
    print("To view a plot of these results, run this script with the option --plot")
```

C.3.3 mixing.py

```
"""
Mixing two streams.

Since reactors can have multiple inlets and outlets, they can be used to
implement mixers, splitters, etc. In this example, air and methane are mixed
in stoichiometric proportions. Due to the low temperature, no reactions occur.
A reaction may occur if the reactor is set in a hot state prior to mixing.
Note that the air stream and the methane stream use *different* reaction
mechanisms, with different numbers of species and reactions. When gas flows
from one reactor or reservoir to another one with a different reaction
mechanism, species are matched by name. If the upstream reactor contains a
species that is not present in the downstream reaction mechanism, it will be
ignored. In general, reaction mechanisms for downstream reactors should
contain all species that might be present in any upstream reactor.
"""

import cantera as ct
```

```
# Use air for stream a.
gas_a = ct.Solution('air.xml')
gas_a.TPX = 300.0, ct.one_atm, 'O2:0.21, N2:0.78, AR:0.01'
mw_a = gas_a.mean_molecular_weight/1000 #kg/mol


# Use GRI-Mech 3.0 for stream b (methane) and for the mixer. If it is desired
# to have a pure mixer, with no chemistry, use instead a reaction mechanism
# for gas_b that has no reactions.
gas_b = ct.Solution('gri30.xml')
gas_b.TPX = 300.0, ct.one_atm, 'CH4:1'
mw_b = gas_b.mean_molecular_weight/1000 #kg/mol

# Create reservoirs for the two inlet streams and for the outlet stream.  The
# upsteam reservoirs could be replaced by reactors, which might themselves be
# connected to reactors further upstream. The outlet reservoir could be
# replaced with a reactor with no outlet, if it is desired to integrate the
# composition leaving the mixer in time, or by an arbitrary network of
# downstream reactors.
res_a = ct.Reservoir(gas_a)
res_b = ct.Reservoir(gas_b)
downstream = ct.Reservoir(gas_b)

# Create a reactor for the mixer. A reactor is required instead of a
# reservoir, since the state will change with time if the inlet mass flow
# rates change or if there is chemistry occurring.
#gas_b.TPX = 300.0, ct.one_atm, 'O2:0.21, N2:0.78, AR:0.01'
gas_b.TPX = 300.0,ct.one_atm,'O2:1., N2:3.78, CH4:0.5'
gas_b.equilibrate("HP")
mixer = ct.IdealGasReactor(gas_b,energy='on')
print(mixer.thermo.report())

# create two mass flow controllers connecting the upstream reservoirs to the
```

```
# mixer, and set their mass flow rates to values corresponding to
# stoichiometric combustion.
mfca = ct.MassFlowController(res_a, mixer, mdot=mw_a*2./0.21)
mfcb = ct.MassFlowController(res_b, mixer, mdot=mw_b*1.0)

# connect the mixer to the downstream reservoir with a valve.
outlet = ct.Valve(mixer, downstream, K=10)

sim = ct.ReactorNet([mixer])

# Since the mixer is a reactor, we need to integrate in time to reach steady
# state. A few residence times should be enough.
print('{0:>14s} {1:>14s} {2:>14s}  {3:>14s}  {4:>14s}'.format(
      't [s]', 'T [K]', 'h [J/kg]', 'P [Pa]', 'X_CH4'))

t = 0.0
for n in range(100):
    tres = mixer.mass/(mfca.mdot(t) + mfcb.mdot(t))
    t += 2*tres
    sim.advance(t)
    print('{0:14.5g} {1:14.5g} {2:14.5g}  {3:14.5g}  {4:14.5g}'.format(
          t, mixer.T, mixer.thermo.h, mixer.thermo.P, mixer.thermo['CH4'].X[0]))

# view the state of the gas in the mixer
print(mixer.thermo.report())
```

C.3.4 Autoignition scripts

Autoignition_simple.py

```
#######################################################################
#
# Autoignition of a methane air mixture at stoichiometry
#             and atmospheric pressure, for one
#             initial temperature
#
```

```
#####################################################################

import cantera as ct
import csv
import numpy as np

#####################################################################

# Mechanism used for the process
gas = ct.Solution('gri30.cti')

# Initial temperature, Pressure and stoichiometry
gas.TPX = 1250, ct.one_atm, 'CH4:0.5, O2:1, N2:3.76'

# Set gas state

# Specify the number of time steps and the time step size
nt = 100000
dt = 1.e-6  # s

# Storage space
mfrac = []
time = []
temperature = []
HR = []

#####################################################################

# Create the batch reactor
r = ct.IdealGasReactor(gas)
# Now create a reactor network consisting of the single batch reactor
sim = ct.ReactorNet([r])

# Run the simulation
```

```
# Initial simulation time
current_time = 0.0

# Loop for nt time steps of dt seconds.
for n in range(nt):
    current_time += dt
    sim.advance(current_time)
    time.append(current_time)
    temperature.append(r.T)
    mfrac.append(r.thermo.Y)
    HR.append(- np.dot(gas.net_production_rates, gas.partial_molar_enthalpies))

#######################################################################
# Catch the autoignition timing
#######################################################################

# Get the ignition delay time by the maximum value of the Heat Release rate
Autoignition = time[HR.index(max(HR))]
print('Autoignition time = ' + str(Autoignition))
# Posterity
Autoignition = Autoignition * 1000  # ms
FinalTemp = temperature[nt - 1]

#######################################################################
# Save results
#######################################################################
# write output CSV file for importing into Excel
csv_file = 'Phi-1_P-1_T-1250_UV.csv'
with open(csv_file, 'w') as outfile:
    writer = csv.writer(outfile)
    writer.writerow(['Auto ignition time [ms]', 'Final Temperature [K]'] + gas.species_names)
    writer.writerow([Autoignition, FinalTemp] + list(mfrac[:]))
print('output written to ' + csv_file)
```

Autoignition_Tvar.py

```
################################################################
#
# Autoignition of a methane air mixture at stoichiometry
#              and atmospheric pressure, for different
#              initial temperature
#
################################################################

import cantera as ct
import numpy as np
import matplotlib.pyplot as plt

#####################################################################

# Mechanism used for the process
gas = ct.Solution('gri30.cti')

# Initial temperatures
Temperature_range = list(range(800, 1700, 100))

# Specify the number of time steps and the time step size
nt = 100000
dt = 1.e-4  # s

# Storing auto ignitions
auto_ignitions = []

for index, Temperature in enumerate(Temperature_range):
    ###############################################################

    # Initial temperature, Pressure and stoichiometry
    gas.TPX = Temperature, ct.one_atm, 'CH4:0.5, O2:1, N2:3.76'
```

```
    # Create the batch reactor
    r = ct.IdealGasReactor(gas)
    # Now create a reactor network consisting of the single batch reactor
    sim = ct.ReactorNet([r])

    # Storage space
    mfrac = []
    # ...
    time = []
    temperature = []
    HR = []

    # Run the simulation
    # Initial simulation time
    current_time = 0.0

    # Loop for nt time steps of dt seconds.
    for n in range(nt):
        current_time += dt
        sim.advance(current_time)

        time.append(current_time)
        temperature.append(r.T)
        mfrac.append(r.thermo.Y)
        HR.append(- np.dot(gas.net_production_rates, gas.partial_molar_enthalpies))

    ###################################################################
    # Catch the autoignition timing
    ###################################################################

    # Get the ignition delay time by the maximum value of the Heat Release rate
    auto_ignition = time[HR.index(max(HR))]
    print('For T = ' + str(Temperature) + ', Autoignition time = ' + str(auto_ignition) + ' s'
    # Posterity
```

```
    FinalTemp = temperature[nt - 1]

    auto_ignitions.append(auto_ignition)

    # ######################################################################
    # # Save results
    # ######################################################################
    # # write output CSV file for importing into Excel
    # csv_file = 'Phi-1_P-1_T-' + str(Temperature) + '_UV.csv'
    # with open(csv_file, 'w') as outfile:
    #     writer = csv.writer(outfile)
    #     writer.writerow(['Auto ignition time [s]', 'Final Temperature [K]'] + gas.species_na
    #     writer.writerow([auto_ignition, FinalTemp] + list(mfrac[:]))
    # print('output written to ' + csv_file)

T_invert = [1000 / Temperature for Temperature in Temperature_range]
######################################################################
# Plot results
######################################################################
# create plot
plt.plot(Temperature_range, auto_ignitions, 'b-o')
plt.xlabel(r'Temperature [K]', fontsize=20)
plt.ylabel("Auto ignition [s]", fontsize=20)
plt.yscale('log')
plt.title(r'Autoignition of $CH_{4}$ + Air mixture at $\Phi$ = 1, and P = 1 bar',
          fontsize=22, horizontalalignment='center')
plt.axis(fontsize=20)
plt.grid()
plt.savefig('Phi-1_P-1_Trange_UV.png', bbox_inches='tight')
plt.show()
```

C.4 Tutorial 5

C.4.1 premixed_flame.py

```
#################################################################
#
# ADIABATIC_FLAME - A freely-propagating, premixed flat flame
#
#################################################################

# import :
import cantera as ct
import numpy as np

#####################################################################
# Prepare your run
#####################################################################

# Import gas phases with mixture transport model
gas = ct.Solution('gri30.cti')

# Parameter values :
# General
p = 1e5  # pressure
tin = 300.0  # unburned gas temperature
phi = 1

fuel = {'CH4': 1}
oxidizer = {'O2': 1, 'N2': 3.76}

# Set gas state to that of the unburned gas
gas.TP = tin, p
gas.set_equivalence_ratio(phi, fuel, oxidizer)

# Create the free laminar premixed flame
```

```
f = ct.FreeFlame(gas, width=0.02)

# set inlet conditions
f.inlet.X = gas.X
f.inlet.T = gas.T

########################################################################
# Program starts here
########################################################################
# First flame:

# No energy for starters
f.energy_enabled = False

# Tolerance properties
tol_ss = [1.0e-5, 1.0e-9]  # [rtol atol] for steady-state problem
tol_ts = [1.0e-5, 1.0e-9]  # [rtol atol] for time stepping

f.flame.set_steady_tolerances(default=tol_ss)
f.flame.set_transient_tolerances(default=tol_ts)

# Max number of times the Jacobian will be used before it must be re-evaluated
f.set_max_jac_age(50, 50)

# Set time steps whenever Newton convergence fails
f.set_time_step(1.0e-5, [2, 5, 10, 20, 80])  # s

# Refinement criteria
f.set_refine_criteria(ratio=10.0, slope=1, curve=1)

# Calculation
loglevel = 1  # amount of diagnostic output (0
# to 5)
```

```
refine_grid = False  # True to enable refinement, False to
# disable

f.solve(loglevel, refine_grid)

f.save('ch4_adiabatic.xml', 'no energy',
       'solution with no energy')

################
# Second flame:

# Energy equation enabled
f.energy_enabled = True

# Calculation and save of the results
refine_grid = True

# Refinement criteria when energy equation is enabled
f.set_refine_criteria(ratio=5.0, slope=0.5, curve=0.5)

f.solve(loglevel, refine_grid)

f.save('ch4_adiabatic.xml', 'energy',
       'solution with the energy equation enabled')

# See the sl to get an idea of whether or not you should continue
print('mixture-averaged flamespeed = {0:7f} m/s'.format(f.u[0]))  # m/s

################
# Third flame and so on ...:

# Refinement criteria should be changed ...
f.set_refine_criteria(ratio=2.0, slope=0.05, curve=0.05)
```

```
# Calculation and saving of the results should always be done
f.solve(loglevel, refine_grid)

f.save('ch4_adiabatic.xml', 'energy continuation',
       'solution with the energy equation enabled continuation')

points = f.flame.n_points
print('mixture-averaged flamespeed continuation = {0:7f} m/s'.format(f.u[0]))  # m/s
print('mixture-averaged final T = {0:7f} K'.format(f.T[points - 1]))  # K

#################
# Fourth flame and so on ...:

# Switch transport model
f.transport_model = 'Multi'

f.solve(loglevel, refine_grid)

f.save('ch4_adiabatic.xml', 'energy_multi',
       'solution with the multicomponent transport and energy equation enabled')

points = f.flame.n_points
print('multicomponent flamespeed = {0:7f} m/s'.format(f.u[0]))  # m/s
print('multicomponent final T = {0:7f} K'.format(f.T[points - 1]))  # K

#####################################################################
# Save your results if needed
#####################################################################
# Write the velocity, temperature, density, and mole fractions to a CSV file
f.write_csv('ch4_adiabatic.csv', quiet=False)
```

C.4.2 premixed_flame_continuation.py

```
#############################################################
#
```

```
# ADIABATIC_FLAME - A freely-propagating, premixed flat flame
#
#######################################################################

# import :
import cantera as ct

#######################################################################
# Prepare your run
#######################################################################

# Import gas phases with mixture transport model
gas = ct.Solution('gri30.cti')

# Parameter values :
# General
p = 1e5  # pressure
tin = 600.0  # unburned gas temperature
phi = 0.8

fuel = {'CH4': 1}
oxidizer = {'O2': 1, 'N2': 3.76}

# Set gas state to that of the unburned gas
gas.TP = tin, p
gas.set_equivalence_ratio(phi, fuel, oxidizer)

# Create the free laminar premixed flame
f = ct.FreeFlame(gas, width=0.02)

#######################################################################
# Program starts here
#######################################################################
```

```
f.restore('ch4_adiabatic.xml', 'energy continuation')

# Set gas state to that of the unburned gas
gas.TP = tin, p
gas.set_equivalence_ratio(phi, fuel, oxidizer)

# set inlet conditions
f.inlet.X = gas.X
f.inlet.T = gas.T

f.solve()

points = f.flame.n_points
print('mixture-averaged flamespeed continuation = {0:7f} m/s'.format(f.u[0]))  # m/s
print('mixture-averaged final T = {0:7f} K'.format(f.T[points - 1]))  # K

#######################################################################
# Save your results if needed
#######################################################################
# Write the velocity, temperature, density, and mole fractions to a CSV file
f.write_csv('ch4_adiabatic.csv', quiet=False)
```

C.4.3 burner_flame.py

```
"""
A burner-stabilized lean premixed hydrogen-oxygen flame at low pressure.
"""

import cantera as ct

#######################################################################
# Prepare your run
#######################################################################

# Import gas phases with mixture transport model
```

```
gas = ct.Solution('gri30.cti')

# Parameter values :
# General
p = 1e5  # pressure
tin = 373.0  # unburned gas temperature
phi = 1.3

fuel = {'CH4': 1}
oxidizer = {'O2': 1, 'N2': 3.76}

# Set gas state to that of the unburned gas
gas.TP = tin, p
gas.set_equivalence_ratio(phi, fuel, oxidizer)

#################
f = ct.BurnerFlame(gas, width=0.02)

f.burner.T = gas.T
f.burner.X = gas.X

mdot = 0.04
f.burner.mdot = mdot

#################
f.energy_enabled = False

tol_ss = [1.0e-5, 1.0e-9]  # [rtol atol] for steady-state
tol_ts = [1.0e-5, 1.0e-4]  # [rtol atol] for time stepping

f.flame.set_steady_tolerances(default=tol_ss)
f.flame.set_transient_tolerances(default=tol_ts)

f.set_refine_criteria(ratio=10.0, slope=1, curve=1)
```

```
f.set_max_jac_age(50, 50)

f.set_time_step(1.0e-5, [1, 2, 5, 10, 20])

loglevel = 1  # amount of diagnostic output (0 to 5)

refine_grid = True  # True to enable refinement, False to

f.solve(loglevel, refine_grid)

f.save('ch4_burner_flame.xml', 'no_energy',
       'solution with the energy equation disabled')

#################
f.energy_enabled = True

f.set_refine_criteria(ratio=3.0, slope=0.1, curve=0.2)

f.solve(loglevel, refine_grid=True)

import matplotlib.pyplot as plt
plt.plot(f.grid, f.T)
plt.show()

f.save('ch4_burner_flame.xml', 'energy',
       'solution with the energy equation enabled')

#################
f.transport_model = 'Multi'

f.solve(loglevel, refine_grid=True)

f.save('ch4_burner_flame.xml', 'energy_multi',
```

```
       'solution with the energy equation enabled and multicomponent transport')

#################
f.soret_enabled = True

f.solve(loglevel, refine_grid=True)

f.save('ch4_burner_flame.xml', 'energy_soret',
       'solution with the energy equation enabled and multicomponent transport')

#################
f.write_csv('ch4_burner_flame.csv', quiet=False)
```

C.4.4 diffusion_flame.py

```
"""
An opposed-flow ethane/air diffusion flame
"""

import cantera as ct
import numpy as np

#######################################################################
# Create the gas object
gas = ct.Solution('gri30.xml', 'gri30_mix')

# Parameter values :
# General
p = 1e5  # pressure

# Different input parameters
tin_f = 300.0  # fuel inlet temperature
tin_o = 300.0  # oxidizer inlet temperature
mdot_f = 0.24  # fuel inlet kg/m^2/s
mdot_o = 0.72  # oxidizer inlet kg/m^2/s
```

```
comp_f = 'C2H6:1'  # fuel composition
comp_o = 'O2:0.21, N2:0.78, AR:0.01'  # air composition

# Set gas state for the flame domain
# gas.TP = tin_o, p
# Distance between inlets is 2 cm.
# Start with an evenly-spaced 6-point grid.
initial_grid = np.linspace(0, 0.02, 6)

################ generate initial strain info #######
rho_o = p / (8.314 / 0.029 * tin_o)
rho_f = p / (8.314 / 0.030 * tin_f)

vel_o = mdot_o / rho_o
vel_f = mdot_f / rho_f

a = (vel_o + vel_f) / 0.02  # s-1

################ generate initial strain info #######

# Create an object representing the counterflow flame configuration,
# which consists of a fuel inlet on the left, the flow in the middle,
# and the oxidizer inlet on the right.
f = ct.CounterflowDiffusionFlame(gas, initial_grid)

# Set the state of the two inlets
f.fuel_inlet.mdot = mdot_f
f.fuel_inlet.X = comp_f
f.fuel_inlet.T = tin_f

f.oxidizer_inlet.mdot = mdot_o
f.oxidizer_inlet.X = comp_o
f.oxidizer_inlet.T = tin_o
```

```
# construct the initial solution estimate. To do so, it is necessary
# to specify the fuel species. If a fuel mixture is being used,
# specify a representative species here for the purpose of
# constructing an initial guess.
f.set_initial_guess(fuel='C2H6')

###############################################################
# Program starts here
###############################################################
# First flame:
# disable the energy equation
f.energy_enabled = False

# Set error tolerances
tol_ss = [1.0e-5, 1.0e-11]  # [rtol, atol] for steady-state problem
tol_ts = [1.0e-5, 1.0e-11]  # [rtol, atol] for time stepping
f.flame.set_steady_tolerances(default=tol_ss)
f.flame.set_transient_tolerances(default=tol_ts)

# and solve the problem without refining the grid
f.solve(loglevel=1, refine_grid=False)

#################
# Second flame:
# specify grid refinement criteria, turn on the energy equation,
f.energy_enabled = True

f.set_refine_criteria(ratio=3, slope=0.8, curve=0.8)

# and solve the problem again
f.solve(loglevel=1, refine_grid=True)

# save your results
```

```
f.save('c2h6_diffusion.xml', 'energy')

#################
# Third flame:
# specify grid refinement criteria, turn on the energy equation,
f.energy_enabled = True

f.set_refine_criteria(ratio=2, slope=0.2, curve=0.2, prune=0.04)

# and solve the problem again
f.solve(loglevel=1, refine_grid=True)

# save your results
f.save('c2h6_diffusion.xml', 'energy continuation')

#####################################################################
# Save your results if needed
#####################################################################
# write the velocity, temperature, and mole fractions to a CSV file
f.write_csv('c2h6_diffusion.csv', quiet=False)
```