Combustion Toolbox

Release v0.9.6

Alberto Cuadra Lara

Jun 12, 2022

CONTENTS

A MATLAB-GUI based open-source tool for solving gaseous combustion problems.

Robust modular kernel `Robust`, `modular`, and `fast` chemical equilibrium computations.

Interactive App The code is encapsulated in an `user-friendly` GUI with tons of capabilities.

Open source Completely open source, GUI included! Github GPLv3

CHAPTER

ONE

COMBUSTION TOOLBOX CAPABILITIES

Robust chemical equilibrium computations TP HP SP TV EV SV frozen plasma state

Interactive App Over 10K lines of code encapsulated in an `user-friendly` GUI with tons of capabilities. Toolbox Standalone

Shocks and detonations pre- and post-shock states Incident Reflected Oblique Shock polars Regular Reflections

Open source Completely open source, GUI included! Github FileExchange Zenodo GPLv3

Rocket propellant performance Infinite-Chamber-Area Finite-Chamber-Area

Extensive database with NASA's 9-coefficient polynomial fits up to 20000 K

Excellent agreement with `NASA's CEA`, `CANTERA`, `Caltech's SD-Toolbox`, and `TEA`.

Predefined plots

Export results spreadsheet .mat

Compatible windows linux mac

CHAPTER

TWO

START HERE!

Tutorial New to Combustion Toolbox?

Examples See examples of Combustion-Toolbox applications.

Documentation Let's check the documentation of almost (every) functions.

CHAPTER

THREE

GALLERY

We have several examples of what Combustion Toolbox can do. Here we show some results obtained from Combustion Toolbox.

Figure 1: *Performance test, 100 Chapman-Jouguet pre-detonation and post-detonation states for a lean to rich CH4-air mixtures at standard conditions (T1 = 300 K and pressure p1 = 1 bar). The computational time was of 2.86 seconds using a Intel(R) Core(TM) i7-8700 CPU @ 3.20GHz for a set of 24 species considered.*

Figure 2: *Hugoniot curves for different molecular gases at pre-shock temperature T1 = 300 K and pressure p1 = 1 atm [numerical results obtained with Combustion Toolbox (lines) and contrasted with NASA's Chemical Equilibrium with Applications (CEA) code excluding ionization (symbols)].*

Figure 3: *Variation of molar fraction for a CJ detonation for lean to rich CH4-air mixtures at standard conditions (T1 = 300 K and pressure p1 = 1 atm); line: numerical results obtained with Combustion Toolbox; circles: NASA's Chemical Equilibrium with Applications code. The computational time was of 6.68 seconds using a Intel(R) Core(TM) i7-8700 CPU @ 3.20GHz for a set of 26 species considered and a total of 351 case studies.*

Figure 4: *Pressure-deflection shock polar (left) and wave angle-deflection shock polar (right) for an air mixture (78.084% N2, 20.9476% O2, 0.9365% Ar, 0.0319% CO2) at pre-shock temperature T1 = 300 K and pressure p1 = 1 atm, and a range of preshock Mach numbers M1 = [2, 14]; line: considering dissociation, ionization, and recombination in multi-species mixtures; dashed: considering a thermochemically frozen air mixture.*

CHAPTER

FOUR

CONTRIBUTING

Please send feedback or inquiries to acuadra@ing.uc3m.es

Thank you for testing Combustion Toolbox!

CHAPTER

FIVE

ACKNOWLEDGEMENTS

- Combustion Toolbox's color palette is obtained from the following repository: Stephen (2021). ColorBrewer: Attractive and Distinctive Colormaps (https://github.com/DrosteEffect/BrewerMap), GitHub. Retrieved December 3, 2021.
- For validations, Combustion Toolbox uses CPU Info from the following repository: Ben Tordoff (2022). CPU Info (https://github.com/BJTor/CPUInfo/releases/tag/v1.3), GitHub. Retrieved March 22, 2022.

CHAPTER

SIX

DEVELOPERS

- **Alberto Cuadra-Lara** - *Core Developer and App designer*
- **César Huete** - *Developer*
- **Marcos Vera** - *Developer*

Grupo de Mecánica de Fluidos, Universidad Carlos III, Av. Universidad 30, 28911, Leganés, Spain

See also the list of contributors who participated in this project.

CHAPTER

SEVEN

CITING COMBUSTION TOOLBOX

```
@misc{combustiontoolbox,
    author = "Cuadra, A and Huete, C and Vera, M",
    title = "Combustion Toolbox: A MATLAB-GUI based open-source tool for solving␣
↪combustion problems",
    year = 2022,
    note = "Version 0.9.6",
    doi = {https://doi.org/10.5281/zenodo.6635715}
}
```

7.1 Quickstart

Combustion toolbox can be used in two ways:

- Using the MATLAB's desktop environment to obtain all the versatility of the plain code.
- Using the Graphical User Interface (GUI) and forget about code.

1. Download the code from one of the following repositories:

Github

MATLAB FileExchange

<sodipodi:namedview id="namedview14" pagecolor="#ffffff" bordercolor="#666666" borderopacity="1.0" inkscape:pageshadow="2" inkscape:pageopacity="0.0" inkscape:pagecheckerboard="0" showgrid="false" inkscape:zoom="1.2886364" inkscape:cx="-103.59788" inkscape:cy="-17.460317" inkscape:window-width="1920" inkscape:window-height="1009" inkscape:window-x="-8" inkscape:window-y="312" inkscape:window-maximized="1" inkscape:current-layer="Layer_1" fit-margin-top="0" fit-margin-left="0" fit-margin-right="0" fit-margin-bottom="0" />

Zenodo **2.** Open MATLAB with the path set to the downloaded folder

3. To install the GUI execute `Combustion Toolbox.mlappintall` in the installer folder. As simple as that.

7.2 Tutorial

With this short tutorial you will familiarize with the basic functionalities of Combustion Toolbox

7.2.1 Getting Started

Start MATLAB and browse for folder where you have downloaded Combustion Toolbox. To include files in `PATH` run this command in the command window:

```
>> addpath(genpath(pwd))
```

First, using Combustion Toolbox, you have to initialize the tool (load databases, set default variables, ...). To do that at the prompt type:

```
>> self = App
```

If files contained in Combustion Toolbox are correctly declared, you should see something like this:

```
self =

  struct with fields:

           E: [1×1 struct]
           S: [1×1 struct]
           C: [1×1 struct]
        Misc: [1×1 struct]
          PD: [1×1 struct]
          PS: [1×1 struct]
          TN: [1×1 struct]
   DB_master: [1×1 struct]
          DB: [1×1 struct]
```

7.2.2 Setting the state

Indicate temperature [K] and pressure [bar] of the initial mixture

```
>> self = set_prop(self, 'TR', 300, 'pR', 1 * 1.01325);
```

Indicate species and number of moles of each species in the initial mixture

Individual case

For example, for a stochiometric CH4-ideal_air mixture:

```
>> self.PD.S_Fuel     = {'CH4'};
>> self.PD.N_Fuel     = 1;
>> self.PD.S_Oxidizer = {'O2'};
>> self.PD.N_Oxidizer = 2;
>> self.PD.S_Inert    = {'N2'};
>> self.PD.N_Inert    = 7.52;
```

This is the same as specifying a unit value for the equivalence ratio:

```
>> self.PD.S_Fuel     = {'CH4'};
>> self.PD.S_Oxidizer = {'O2'};
>> self.PD.S_Inert    = {'N2'};
>> self = set_prop(self, 'phi', 1);
>> self.PD.proportion_inerts_O2 = 79/21;
```

The last two lines of code establish the equivalence relation and the proportion of the inert species over the oxidazer, respectively. The number of moles is calculated considering that the number of moles of fuel is one.

Several cases

Combustion Toolbox also allows the computation of a range of values of different properties. For example, in case we want to compute a range of values of the equivalence ratio, e.g., phi = 0.5:0.01:5, do this:

```
>> self.PD.S_Fuel     = {'CH4'};
>> self.PD.S_Oxidizer = {'O2'};
>> self.PD.S_Inert    = {'N2'};
>> self = set_prop(self, 'phi', 0.5:0.01:5);
>> self.PD.proportion_inerts_O2 = 79/21;
```

7.2.3 Chemical Equilibrium

Depending on the problem you want to solve, you may need to configure additional inputs. For example, to compute the equilibrium composition at a defined temperature and pressure (TP) we have to set these values as

```
>> self = set_prop(self, 'pP', self.PD.pR.value, 'TP', 3000);
```

and to solve the aforementioned problem, run

```
>> self = SolveProblem(self, 'TP');
```

The results are contained in `self.PS`. By default, this routine print the results through the command window (default: self.Misc.FLAG_RESULTS=true) which gives for the stoichiometric case (phi=1):

```
COMPUTING Nº MOLES FROM EQUIVALENCE RATIO (PHI).
************************************************************
------------------------------------------------------------
Problem type: TP  | phi = 1.0000
------------------------------------------------------------
                  |    REACTANTS    |      PRODUCTS
T [K]             |       300.0000  |      3000.0000
p [bar]           |         1.0132  |         1.0132
r [kg/m3]         |         1.1225  |         0.1029
h [kJ/kg]         |      -254.5296  |      2574.2795
e [kJ/kg]         |      -344.7953  |      1589.5140
g [kJ/kg]         |     -2428.4002  |    -30246.9221
s [kJ/(kg-K)]     |         7.2462  |        10.9404
W [g/mol]         |        27.6333  |        25.3293
(dlV/dlp)T [-]    |                 |        -1.0285
(dlV/dlT)p [-]    |                 |         1.5830
```

(continues on next page)

(continued from previous page)

```
cp [kJ/(kg-K)] |         1.0786  |         5.5609
gamma [-]      |         1.3869  |         1.1357
sound vel [m/s]|       353.8198  |      1057.5349
-------------------------------------------------------------
REACTANTS               Xi [-]
N2                   7.1493e-01
O2                   1.9005e-01
CH4                  9.5023e-02
MINORS[+21]          0.0000e+00

TOTAL                1.0000e+00
-------------------------------------------------------------
PRODUCTS                Xi [-]
N2                   6.4771e-01
H2O                  1.1162e-01
CO                   5.8485e-02
OH                   3.5936e-02
H2                   3.0822e-02
CO2                  2.8615e-02
H                    2.7583e-02
O2                   2.5914e-02
O                    1.8096e-02
NO                   1.5206e-02
N                    1.1123e-05
NH                   9.5805e-07
HCO                  1.2464e-07
NH2                  1.1860e-07
NH3                  3.0265e-08
HCN                  4.4103e-09
CN                   4.0110e-10
CH                   5.7109e-13
CH3                  1.5595e-13
CH4                  1.1358e-14
MINORS[+4]           3.5228e-16

TOTAL                1.0000e+00
-----------------------------------------------------------------------------
*****************************************************************************
```

7.2.4 Postprocessed results (predefined plots for several cases)

There are some predefined charts based on the selected problem, in case you have calculated multiple cases. Just calling the routine

```
>> postResults(self);
```

will reproduce **Figure 1** which represents the variation of the molar fraction with the equivalence ratio for lean to rich CH4-ideal_air mixtures at 3000 [K] and 1.01325 [bar].

Figure 1: *Example TP: variation of molar fraction for lean to rich CH4-ideal_air mixtures at 3000 [K] and 1.01325 [bar], a set of 26 species considered and a total of 451 case studies. The computational time was of 3.04 seconds using a Intel(R) Core(TM) i7-8700 CPU @ 3.20GHz.*

7.2.5 Congratulations!

Congratulations you have finished the Combustion Toolbox Matlab tutorial! You should now be ready to begin using Combustion Toolbox on your own.

7.3 Examples

Combustion Toolbox can be used through:

- the user-friendly Graphic User Interface (GUI),
- the desktop environment.

7.3.1 Examples GUI

Here we summarize many of the computations that can be performed using the Combustion Toolbox GUI.

7.3.2 Examples desktop environment

Here we have a collection of examples included in Combustion Toolbox.

Example_TP.m

```
% -------------------------------------------------------------------------
% EXAMPLE: TP
%
% Compute equilibrium composition at defined temperature (e.g., 3000 K) and
% pressure (e.g., 1.01325 bar) for lean to rich CH4-air mixtures at
% standard conditions, a set of 26 species considered and a set of
% equivalence ratios phi contained in (0.5, 5) [-]
%
% Soot formation == {'CO2','CO','H2O','H2','O2','N2','He','Ar','Cbgrb',...
%                    'C2','C2H4','CH','CH','CH3','CH4','CN','H',...
%                    'HCN','HCO','N','NH','NH2','NH3','NO','O','OH'}
%
% See wiki or ListSpecies() for more predefined sets of species
%
% @author: Alberto Cuadra Lara
%          PhD Candidate - Group Fluid Mechanics
%          Universidad Carlos III de Madrid
%
% Last update Oct 22 2021
% -------------------------------------------------------------------------

%% INITIALIZE
self = App('Soot formation');
%% INITIAL CONDITIONS
self = set_prop(self, 'TR', 300, 'pR', 1 * 1.01325, 'phi', 0.5:0.01:5);
self.PD.S_Fuel     = {'CH4'};
self.PD.S_Oxidizer = {'O2'};
```

(continues on next page)

(continued from previous page)

```
self.PD.S_Inert    = {'N2', 'Ar', 'CO2'};
self.PD.proportion_inerts_O2 = [78.084, 0.9365, 0.0319] ./ 20.9476;
%% ADDITIONAL INPUTS (DEPENDS OF THE PROBLEM SELECTED)
self = set_prop(self, 'pP', self.PD.pR.value, 'TP', 3000);
%% SOLVE PROBLEM
self = SolveProblem(self, 'TP');
%% DISPLAY RESULTS (PLOTS)
postResults(self);
```

Example_TV.m

```
% -------------------------------------------------------------------------
% EXAMPLE: TV
%
% Compute equilibrium composition at defined temperature (e.g., 3000 K) and
% constant volume for lean to rich CH4-air mixtures at standard conditions,
% a set of 26 species considered and a set of equivalence ratios (phi)
% contained in (0.5, 5) [-]
%
% Soot formation == {'CO2','CO','H2O','H2','O2','N2','He','Ar','Cbgrb',...
%                    'C2','C2H4','CH','CH','CH3','CH4','CN','H',...
%                    'HCN','HCO','N','NH','NH2','NH3','NO','O','OH'}
%
% See wiki or ListSpecies() for more predefined sets of species
%
% @author: Alberto Cuadra Lara
%          PhD Candidate - Group Fluid Mechanics
%          Universidad Carlos III de Madrid
%
% Last update Oct 22 2021
% -------------------------------------------------------------------------

%% INITIALIZE
self = App('Soot formation');
%% INITIAL CONDITIONS
self = set_prop(self, 'TR', 300, 'pR', 1 * 1.01325, 'phi', 0.5:0.01:5);
self.PD.S_Fuel     = {'CH4'};
self.PD.S_Oxidizer = {'O2'};
self.PD.S_Inert    = {'N2', 'Ar', 'CO2'};
self.PD.proportion_inerts_O2 = [78.084, 0.9365, 0.0319] ./ 20.9476;
%% ADDITIONAL INPUTS (DEPENDS OF THE PROBLEM SELECTED)
self = set_prop(self, 'TP', 3000);
%% SOLVE PROBLEM
self = SolveProblem(self, 'TV');
%% DISPLAY RESULTS (PLOTS)
postResults(self);
```

Example_HP.m

```
% ------------------------------------------------------------------------
% EXAMPLE: HP
%
% Compute adiabatic temperature and equilibrium composition at constant
% pressure (e.g., 1.01325 bar) for lean to rich CH4-air mixtures at
% standard conditions, a set of 26 species considered and a set of
% equivalence ratios phi contained in (0.5, 5) [-]
%
% Soot formation == {'CO2','CO','H2O','H2','O2','N2','He','Ar','Cbgrb',...
%                    'C2','C2H4','CH','CH','CH3','CH4','CN','H',...
%                    'HCN','HCO','N','NH','NH2','NH3','NO','O','OH'}
%
% See wiki or ListSpecies() for more predefined sets of species
%
% @author: Alberto Cuadra Lara
%          PhD Candidate - Group Fluid Mechanics
%          Universidad Carlos III de Madrid
%
% Last update Oct 22 2021
% ------------------------------------------------------------------------

%% INITIALIZE
self = App('Soot formation');
%% INITIAL CONDITIONS
self = set_prop(self, 'TR', 300, 'pR', 1 * 1.01325, 'phi', 0.5:0.01:5);
self.PD.S_Fuel     = {'CH4'};
self.PD.S_Oxidizer = {'O2'};
self.PD.S_Inert    = {'N2', 'Ar', 'CO2'};
self.PD.proportion_inerts_O2 = [78.084, 0.9365, 0.0319] ./ 20.9476;
%% ADDITIONAL INPUTS (DEPENDS OF THE PROBLEM SELECTED)
self = set_prop(self, 'pP', self.PD.pR.value);
%% SOLVE PROBLEM
self = SolveProblem(self, 'HP');
%% DISPLAY RESULTS (PLOTS)
postResults(self);
```

Example_HP_MIXTEMP.m

```
% ------------------------------------------------------------------------
% EXAMPLE: HP MIXTEMP
%
% Compute adiabatic temperature and equilibrium composition at constant
% pressure (e.g., 1.01325 bar) for lean to rich CH4-air mixtures at
% standard conditions except for the air which is at 380 K. Also, a set
% of 26 species considered and a set of equivalence ratios phi contained
% in (0.5, 5) [-]
%
% Soot formation == {'CO2','CO','H2O','H2','O2','N2','He','Ar','Cbgrb',...
%                    'C2','C2H4','CH','CH','CH3','CH4','CN','H',...
```

(continues on next page)

(continued from previous page)

```
%                   'HCN','HCO','N','NH','NH2','NH3','NO','O','OH'}
%
% See wiki or ListSpecies() for more predefined sets of species
%
% @author: Alberto Cuadra Lara
%          PhD Candidate - Group Fluid Mechanics
%          Universidad Carlos III de Madrid
%
% Last update Feb 19 2022
% -------------------------------------------------------------------------

%% INITIALIZE
self = App('Soot formation');
%% INITIAL CONDITIONS
self = set_prop(self, 'pR', 1 * 1.01325, 'phi', 0.5:0.01:5);
self.PD.S_Fuel     = {'CH4'};
self.PD.S_Oxidizer = {'O2'};
self.PD.S_Inert    = {'N2', 'Ar', 'CO2'};
self.PD.proportion_inerts_O2 = [78.084, 0.9365, 0.0319] ./ 20.9476;

self.PD.T_Fuel     = 300;
self.PD.T_Oxidizer = 380;
self.PD.T_Inert    = [380, 380, 380];
%% ADDITIONAL INPUTS (DEPENDS OF THE PROBLEM SELECTED)
self = set_prop(self, 'pP', self.PD.pR.value);
%% SOLVE PROBLEM
self = SolveProblem(self, 'HP');
%% DISPLAY RESULTS (PLOTS)
postResults(self);
```

Example_HP_PROPELLANTS.m

```
% -------------------------------------------------------------------------
% EXAMPLE: HP PROPELLANTS
%
% Compute adiabatic temperature and equilibrium composition at constant
% pressure (e.g., 1.01325 bar) for lean to rich LH2-LOX mixtures at
% standard conditions, a set of 24 species considered and a set of
% equivalence ratios phi contained in (0.5, 5) [-]
%
% HYDROGEN_L == {'H','H2O','OH','H2','O','O3','O2','HO2','H2O2',...
%                'H2bLb','O2bLb'}
%
% See wiki or ListSpecies() for more predefined sets of species
%
% @author: Alberto Cuadra Lara
%          PhD Candidate - Group Fluid Mechanics
%          Universidad Carlos III de Madrid
%
% Last update Feb 19 2022
```

(continues on next page)

(continued from previous page)

```
% ------------------------------------------------------------------------

%% INITIALIZE
self = App('HYDROGEN_L');
%% INITIAL CONDITIONS
self = set_prop(self, 'TR', 300, 'pR', 1 * 1.01325, 'phi', 0.2:0.05:5);
self.PD.S_Fuel     = {'H2bLb'};
self.PD.S_Oxidizer = {'O2bLb'};
%% ADDITIONAL INPUTS (DEPENDS OF THE PROBLEM SELECTED)
self = set_prop(self, 'pP', self.PD.pR.value);
%% SOLVE PROBLEM
self = SolveProblem(self, 'HP');
%% DISPLAY RESULTS (PLOTS)
postResults(self);
```

Example_EV.m

```
% ------------------------------------------------------------------------
% EXAMPLE: EV
%
% Compute equilibrium composition at adiabatic temperature and constant
% volume for lean to rich CH4-air mixtures at standard conditions, a set
% of 26 species considered and a set of equivalence ratios (phi) contained
% in (0.5, 5) [-]
%
% Soot formation == {'CO2','CO','H2O','H2','O2','N2','He','Ar','Cbgrb',...
%                    'C2','C2H4','CH','CH','CH3','CH4','CN','H',...
%                    'HCN','HCO','N','NH','NH2','NH3','NO','O','OH'}
%
% See wiki or ListSpecies() for more predefined sets of species
%
% @author: Alberto Cuadra Lara
%          PhD Candidate - Group Fluid Mechanics
%          Universidad Carlos III de Madrid
%
% Last update Oct 22 2021
% ------------------------------------------------------------------------

%% INITIALIZE
self = App('Soot formation');
%% INITIAL CONDITIONS
self = set_prop(self, 'TR', 300, 'pR', 1 * 1.01325, 'phi', 0.5:0.01:5);
self.PD.S_Fuel     = {'CH4'};
self.PD.S_Oxidizer = {'O2'};
self.PD.S_Inert    = {'N2', 'Ar', 'CO2'};
self.PD.proportion_inerts_O2 = [78.084, 0.9365, 0.0319] ./ 20.9476;
%% ADDITIONAL INPUTS (DEPENDS OF THE PROBLEM SELECTED)
% No additional data required. The internal energy and volume are constants.
%% SOLVE PROBLEM
self = SolveProblem(self, 'EV');
```

(continues on next page)

(continued from previous page)

```
%% DISPLAY RESULTS (PLOTS)
postResults(self);
```

Example_SP.m

```
% -------------------------------------------------------------------------
% EXAMPLE: SP
% Compute Isentropic compression/expansion and equilibrium composition at
% a defined set of pressure (1.01325, 1013.25 bar) for a rich CH4-air mixture
% at standard conditions, a set of 26 species considered, and a equivalence
% ratio phi 1.5 [-]
%
% Soot formation == {'CO2','CO','H2O','H2','O2','N2','He','Ar','Cbgrb',...
%                    'C2','C2H4','CH','CH','CH3','CH4','CN','H',...
%                    'HCN','HCO','N','NH','NH2','NH3','NO','O','OH'}
%
% See wiki or ListSpecies() for more predefined sets of species
%
% @author: Alberto Cuadra Lara
%          PhD Candidate - Group Fluid Mechanics
%          Universidad Carlos III de Madrid
%
% Last update Feb 18 2022
% -------------------------------------------------------------------------

%% INITIALIZE
self = App('Soot formation');
%% INITIAL CONDITIONS
self = set_prop(self, 'TR', 300, 'pR', 1 * 1.01325, 'phi', 1.5);
self.PD.S_Fuel     = {'CH4'};
self.PD.S_Oxidizer = {'O2'};
self.PD.S_Inert    = {'N2', 'Ar', 'CO2'};
self.PD.proportion_inerts_O2 = [78.084, 0.9365, 0.0319] ./ 20.9476;
%% ADDITIONAL INPUTS (DEPENDS OF THE PROBLEM SELECTED)
self = set_prop(self, 'pP', 1.01325 * logspace(0, 3, 200));
%% SOLVE PROBLEM
self = SolveProblem(self, 'SP');
%% DISPLAY RESULTS (PLOTS)
postResults(self);
```

Example_SV.m

```
% -------------------------------------------------------------------------
% EXAMPLE: SV
% Compute Isentropic compression/expansion and equilibrium composition at
% a defined set of volume ratios (0.5, 2) for a lean CH4-air mixture at
% 700 K and 10 bar, a set of 26 species considered, and a equivalence
% ratio phi 0.5 [-]
%
```

(continues on next page)

(continued from previous page)

```
% Soot formation == {'CO2','CO','H2O','H2','O2','N2','He','Ar','Cbgrb',...
%                     'C2','C2H4','CH','CH','CH3','CH4','CN','H',...
%                     'HCN','HCO','N','NH','NH2','NH3','NO','O','OH'}
%
% See wiki or ListSpecies() for more predefined sets of species
%
% @author: Alberto Cuadra Lara
%          PhD Candidate - Group Fluid Mechanics
%          Universidad Carlos III de Madrid
%
% Last update Feb 19 2022
% -------------------------------------------------------------------------

%% INITIALIZE
self = App('Soot formation');
%% INITIAL CONDITIONS
self = set_prop(self, 'TR', 700, 'pR', 10, 'phi', 0.5);
self.PD.S_Fuel     = {'CH4'};
self.PD.S_Oxidizer = {'O2'};
self.PD.S_Inert    = {'N2', 'Ar', 'CO2'};
self.PD.proportion_inerts_O2 = [78.084, 0.9365, 0.0319] ./ 20.9476;
%% ADDITIONAL INPUTS (DEPENDS OF THE PROBLEM SELECTED)
self = set_prop(self, 'vP_vR', 0.5:0.01:2);
%% SOLVE PROBLEM
self = SolveProblem(self, 'SV');
%% DISPLAY RESULTS (PLOTS)
postResults(self);
```

Example_SV_FROZEN.m

```
% -------------------------------------------------------------------------
% EXAMPLE: SV FROZEN
% Compute Isentropic compression/expansion and equilibrium composition at
% a defined set of volume ratios (0.5, 2) for a lean CH4-air mixture at
% 700 K and 10 bar, frozen chemistry, and a equivalence ratio phi 0.5 [-]
%
% LS == {'CH4', 'O2', 'N2', 'Ar', 'CO2'}
%
% See wiki or ListSpecies() for more predefined sets of species
%
% @author: Alberto Cuadra Lara
%          PhD Candidate - Group Fluid Mechanics
%          Universidad Carlos III de Madrid
%
% Last update Feb 19 2022
% -------------------------------------------------------------------------

%% INITIALIZE
self = App({'CH4', 'O2', 'N2', 'Ar', 'CO2'});
%% INITIAL CONDITIONS
```

(continues on next page)

(continued from previous page)

```
self = set_prop(self, 'TR', 700, 'pR', 10, 'phi', 0.5);
self.PD.S_Fuel     = {'CH4'};
self.PD.S_Oxidizer = {'O2'};
self.PD.S_Inert    = {'N2', 'Ar', 'CO2'};
self.PD.proportion_inerts_O2 = [78.084, 0.9365, 0.0319] ./ 20.9476;
%% ADDITIONAL INPUTS (DEPENDS OF THE PROBLEM SELECTED)
self = set_prop(self, 'vP_vR', 0.5:0.01:2);
%% SOLVE PROBLEM
self = SolveProblem(self, 'SV');
%% DISPLAY RESULTS (PLOTS)
postResults(self);
```

Example_SHOCK_I.m

```
% -------------------------------------------------------------------------
% EXAMPLE: SHOCK_I
%
% Compute pre-shock and post-shock state for a planar incident shock wave
% at standard conditions, a set of 20 species considered and a set of
% initial shock front velocities (u1) contained in (360, 20000) [m/s]
%
% Air == {'O2','N2','O','O3','N','NO','NO2','NO3','N2O','N2O3','N2O4',...
%         'N3','C','CO','CO2','Ar','H2O','H2','H','He'}
%
% See wiki or ListSpecies() for more predefined sets of species
%
% @author: Alberto Cuadra Lara
%          PhD Candidate - Group Fluid Mechanics
%          Universidad Carlos III de Madrid
%
% Last update March 17 2022
% -------------------------------------------------------------------------

%% INITIALIZE
self = App('Air');
%% INITIAL CONDITIONS
self = set_prop(self, 'TR', 300, 'pR', 1 * 1.01325);
self.PD.S_Oxidizer = {'O2'};
self.PD.S_Inert    = {'N2', 'Ar', 'CO2'};
self.PD.proportion_inerts_O2 = [78.084, 0.9365, 0.0319] ./ 20.9476;
%% ADDITIONAL INPUTS (DEPENDS OF THE PROBLEM SELECTED)
u1 = logspace(2, 5, 500); u1 = u1(u1<20000); u1 = u1(u1>=360);
self = set_prop(self, 'u1', u1);
%% SOLVE PROBLEM
self = SolveProblem(self, 'SHOCK_I');
%% DISPLAY RESULTS (PLOTS)
postResults(self);
```

Example_SHOCK_I_IONIZATION.m

```
% -------------------------------------------------------------------------
% EXAMPLE: SHOCK_I_IONIZATION
%
% Compute pre-shock and post-shock state for a planar incident shock wave
% at standard conditions, a set of 39 species considered and a set of
% initial shock front velocities (u1) contained in (360, 13000) [m/s]
%
% Air_ions == {'O2','N2','O','O3','N','NO','NO2','NO3','N2O','N2O3',...
%              'N2O4','N3','eminus','Nminus','Nplus','NOplus','NO2minus',...
%              'NO3minus','N2plus','N2minus','N2Oplus','Oplus','Ominus',...
%              'O2plus', 'O2minus','CO2','CO','COplus','C','Cplus',...
%              'Cminus','CN','CNplus','CNminus','CNN','NCO','NCN','Ar',...
%              'Arplus'}
%
% See wiki or ListSpecies() for more predefined sets of species
%
% @author: Alberto Cuadra Lara
%          PhD Candidate - Group Fluid Mechanics
%          Universidad Carlos III de Madrid
%
% Last update March 17 2022
% -------------------------------------------------------------------------

%% INITIALIZE
self = App('Air_ions');
%% INITIAL CONDITIONS
self = set_prop(self, 'TR', 300, 'pR', 1 * 1.01325);
self.PD.S_Oxidizer = {'O2'};
self.PD.S_Inert    = {'N2', 'Ar', 'CO2'};
self.PD.proportion_inerts_O2 = [78.084, 0.9365, 0.0319] ./ 20.9476;
%% ADDITIONAL INPUTS (DEPENDS OF THE PROBLEM SELECTED)
u1 = logspace(2, 5, 500); u1 = u1(u1<13000); u1 = u1(u1>=360);
self = set_prop(self, 'u1', u1);
%% SOLVE PROBLEM
self = SolveProblem(self, 'SHOCK_I');
%% DISPLAY RESULTS (PLOTS)
postResults(self);
```

Example_SHOCK_R.m

```
% -------------------------------------------------------------------------
% EXAMPLE: SHOCK_R
%
% Compute pre-shock and post-shock state for a planar reflected shock wave
% at standard conditions, a set of 20 species considered and a set of
% initial shock front velocities (u1) contained in (360, 9000) [m/s]
%
% Air == {'O2','N2','O','O3','N','NO','NO2','NO3','N2O','N2O3','N2O4',...
%         'N3','C','CO','CO2','Ar','H2O','H2','H','He'}
```

(continues on next page)

(continued from previous page)

```
%
% See wiki or ListSpecies() for more predefined sets of species
%
% @author: Alberto Cuadra Lara
%          PhD Candidate - Group Fluid Mechanics
%          Universidad Carlos III de Madrid
%
% Last update March 17 2022
% -------------------------------------------------------------------------

%% INITIALIZE
self = App('Air');
%% INITIAL CONDITIONS
self = set_prop(self, 'TR', 300, 'pR', 1 * 1.01325);
self.PD.S_Oxidizer = {'O2'};
self.PD.S_Inert    = {'N2', 'Ar', 'CO2'};
self.PD.proportion_inerts_O2 = [78.084, 0.9365, 0.0319] ./ 20.9476;
%% ADDITIONAL INPUTS (DEPENDS OF THE PROBLEM SELECTED)
u1 = logspace(2, 5, 500); u1 = u1(u1<9000); u1 = u1(u1>=360);
self = set_prop(self, 'u1', u1);
%% SOLVE PROBLEM
self = SolveProblem(self, 'SHOCK_R');
%% DISPLAY RESULTS (PLOTS)
postResults(self);
```

Example_SHOCK_OBLIQUE_BETA.m

```
% -------------------------------------------------------------------------
% EXAMPLE: SHOCK_OBLIQUE_BETA
%
% Compute pre-shock and post-shock state for a oblique incident shock wave
% at standard conditions, a set of 20 species considered, a initial
% shock front velocities u1 = a1 * 10 [m/s], and a set of wave angles
% beta = [20:5:85] [deg]
%
% Air_ions == {'O2','N2','O','O3','N','NO','NO2','NO3','N2O','N2O3',...
%              'N2O4','N3','eminus','Nminus','Nplus','NOplus','NO2minus',...
%              'NO3minus','N2plus','N2minus','N2Oplus','Oplus','Ominus',...
%              'O2plus', 'O2minus','CO2','CO','COplus','C','Cplus',...
%              'Cminus','CN','CNplus','CNminus','CNN','NCO','NCN','Ar',...
%              'Arplus'}
%
% See wiki or ListSpecies() for more predefined sets of species
%
% @author: Alberto Cuadra Lara
%          PhD Candidate - Group Fluid Mechanics
%          Universidad Carlos III de Madrid
%
% Last update March 24 2022
% -------------------------------------------------------------------------
```

(continues on next page)

(continued from previous page)

```

%% INITIALIZE
self = App('Air_ions');
% self = App({'O2', 'N2', 'Ar', 'CO2'}); % Frozen
% self = App({'O2'}); % Frozen
%% INITIAL CONDITIONS
self = set_prop(self, 'TR', 300, 'pR', 1 * 1.01325);
self.PD.S_Oxidizer = {'O2'};
self.PD.S_Inert    = {'N2', 'Ar', 'CO2'};
self.PD.proportion_inerts_O2 = [78.084, 0.9365, 0.0319] ./ 20.9476;
%% ADDITIONAL INPUTS (DEPENDS OF THE PROBLEM SELECTED)
overdriven = 10;
self = set_prop(self, 'u1', 3.472107491008314e+02 * overdriven, 'beta', 20:5:85);
%% SOLVE PROBLEM
self = SolveProblem(self, 'SHOCK_OBLIQUE');
%% DISPLAY RESULTS (PLOTS)
postResults(self);
```

Example_SHOCK_OBLIQUE_THETA.m

```
% -------------------------------------------------------------------------
% EXAMPLE: SHOCK_OBLIQUE_THETA
%
% Compute pre-shock and post-shock state for a oblique incident shock wave
% at standard conditions, a set of 20 species considered, a initial
% shock front velocities u1 = a1 * 10 [m/s], and a set of deflection angle
% theta = [5:5:40] [deg]
%
% Air_ions == {'O2','N2','O','O3','N','NO','NO2','NO3','N2O','N2O3',...
%              'N2O4','N3','eminus','Nminus','Nplus','NOplus','NO2minus',...
%              'NO3minus','N2plus','N2minus','N2Oplus','Oplus','Ominus',...
%              'O2plus', 'O2minus,'CO2','CO','COplus','C','Cplus',...
%              'Cminus','CN','CNplus','CNminus','CNN','NCO','NCN','Ar',...
%              'Arplus'}
%
% See wiki or ListSpecies() for more predefined sets of species
%
% @author: Alberto Cuadra Lara
%          PhD Candidate - Group Fluid Mechanics
%          Universidad Carlos III de Madrid
%
% Last update March 24 2022
% -------------------------------------------------------------------------

%% INITIALIZE
self = App('Air_ions');
% self = App({'O2', 'N2', 'Ar', 'CO2'}); % Frozen
% self = App({'O2'}); % Frozen
%% INITIAL CONDITIONS
self = set_prop(self, 'TR', 300, 'pR', 1 * 1.01325);
```

(continues on next page)

(continued from previous page)

```
self.PD.S_Oxidizer = {'O2'};
self.PD.S_Inert    = {'N2', 'Ar', 'CO2'};
self.PD.proportion_inerts_O2 = [78.084, 0.9365, 0.0319] ./ 20.9476;
%% ADDITIONAL INPUTS (DEPENDS OF THE PROBLEM SELECTED)
overdriven = 10;
self = set_prop(self, 'u1', 3.472107491008314e+02 * overdriven, 'theta', 5:5:40);
%% SOLVE PROBLEM
self = SolveProblem(self, 'SHOCK_OBLIQUE');
%% DISPLAY RESULTS (PLOTS)
postResults(self);
```

Example_SHOCK_OBLIQUE_R.m

```
% -------------------------------------------------------------------------
% EXAMPLE: SHOCK_OBLIQUE_R
%
% Compute pre-shock and post-shock state (incident and reflected) for a
% oblique incident shock wave at standard conditions, a set of 20 species
% considered, a initial shock front velocities u1 = a1 * 10 [m/s], and a
% deflection angle theta = 20 [deg]
%
% Air_ions == {'O2','N2','O','O3','N','NO','NO2','NO3','N2O','N2O3',...
%              'N2O4','N3','eminus','Nminus','Nplus','NOplus','NO2minus',...
%              'NO3minus','N2plus','N2minus','N2Oplus','Oplus','Ominus',...
%              'O2plus', 'O2minus','CO2','CO','COplus','C','Cplus',...
%              'Cminus','CN','CNplus','CNminus','CNN','NCO','NCN','Ar',...
%              'Arplus'}
%
% See wiki or ListSpecies() for more predefined sets of species
%
% @author: Alberto Cuadra Lara
%          PhD Candidate - Group Fluid Mechanics
%          Universidad Carlos III de Madrid
%
% Last update March 24 2022
% -------------------------------------------------------------------------

%% INITIALIZE
self = App('Air_ions');
% self = App({'O2', 'N2', 'Ar', 'CO2'}); % Frozen
% self = App({'O2'}); % Frozen
%% INITIAL CONDITIONS
self = set_prop(self, 'TR', 300, 'pR', 1 * 1.01325);
self.PD.S_Oxidizer = {'O2'};
self.PD.S_Inert    = {'N2', 'Ar', 'CO2'};
self.PD.proportion_inerts_O2 = [78.084, 0.9365, 0.0319] ./ 20.9476;
%% ADDITIONAL INPUTS (DEPENDS OF THE PROBLEM SELECTED)
overdriven = 10;
self = set_prop(self, 'u1', 3.472107491008314e+02 * overdriven, 'theta', 20);
%% SOLVE PROBLEM
```

(continues on next page)

(continued from previous page)

```
self = SolveProblem(self, 'SHOCK_OBLIQUE_R');
%% DISPLAY RESULTS (PLOTS)
postResults(self);
```

Example_SHOCK_POLAR.m

```
% -------------------------------------------------------------------------
% EXAMPLE: SHOCK_POLAR
%
% Compute shock polar plots at standard conditions, a set of 39 species
% considered, and a set of initial shock front velocities u1/a1 = [2:2:14]
%
% Air_ions == {'O2','N2','O','O3','N','NO','NO2','NO3','N2O','N2O3',...
%              'N2O4','N3','eminus','Nminus','Nplus','NOplus','NO2minus',...
%              'NO3minus','N2plus','N2minus','N2Oplus','Oplus','Ominus',...
%              'O2plus', 'O2minus','CO2','CO','COplus','C','Cplus',...
%              'Cminus','CN','CNplus','CNminus','CNN','NCO','NCN','Ar',...
%              'Arplus'}
%
% See wiki or ListSpecies() for more predefined sets of species
%
% @author: Alberto Cuadra Lara
%          PhD Candidate - Group Fluid Mechanics
%          Universidad Carlos III de Madrid
%
% Last update March 21 2021
% -------------------------------------------------------------------------

%% INITIALIZE
% self = App('Air_ions');
self = App({'O2', 'N2', 'Ar', 'CO2'}); % Frozen
% self = App({'O2', 'N2'}); % Frozen
% self = App({'O2'}); % Frozen
%% INITIAL CONDITIONS
self = set_prop(self, 'TR', 300, 'pR', 1 * 1.01325);
self.PD.S_Oxidizer = {'O2'};
% self.PD.S_Inert    = {'N2'};
% self.PD.proportion_inerts_O2 = 79/21;
self.PD.S_Inert    = {'N2', 'Ar', 'CO2'};
self.PD.proportion_inerts_O2 = [78.084, 0.9365, 0.0319] ./ 20.9476;
%% ADDITIONAL INPUTS (DEPENDS OF THE PROBLEM SELECTED)
% range1 = logspace(0, 1, 300); range1 = range1(range1 < 5);
% overdriven = [range1, linspace(5, 14, 30)]; overdriven = overdriven(overdriven > 1);
overdriven = 2:2:14;
% overdriven = [2, 3, 5, 10];
% overdriven = 2.85;
self = set_prop(self, 'u1', 3.472107491008314e+02 * overdriven);
%% SOLVE PROBLEM
self = SolveProblem(self, 'SHOCK_POLAR');
%% DISPLAY RESULTS (PLOTS)
postResults(self);
```

Example_SHOCK_POLAR_R.m

```
% ------------------------------------------------------------------------
% EXAMPLE: SHOCK_POLAR_REFLECTED
%
% Compute shock polar plots at standard conditions, a set of 39 species
% considered, and a set of initial shock front velocities u1/a1 = [2:2:14]
%
% Air_ions == {'O2','N2','O','O3','N','NO','NO2','NO3','N2O','N2O3',...
%              'N2O4','N3','eminus','Nminus','Nplus','NOplus','NO2minus',...
%              'NO3minus','N2plus','N2minus','N2Oplus','Oplus','Ominus',...
%              'O2plus', 'O2minus,'CO2','CO','COplus','C','Cplus',...
%              'Cminus','CN','CNplus','CNminus','CNN','NCO','NCN','Ar',...
%              'Arplus'}
%
% See wiki or ListSpecies() for more predefined sets of species
%
% @author: Alberto Cuadra Lara
%          PhD Candidate - Group Fluid Mechanics
%          Universidad Carlos III de Madrid
%
% Last update March 21 2021
% ------------------------------------------------------------------------

%% INITIALIZE
self = App('Air_ions');
% self = App({'O2', 'N2', 'Ar', 'CO2'}); % Frozen
% self = App({'O2'}); % Frozen
%% INITIAL CONDITIONS
self = set_prop(self, 'TR', 300, 'pR', 1 * 1.01325);
self.PD.S_Oxidizer = {'O2'};
self.PD.S_Inert    = {'N2', 'Ar', 'CO2'};
self.PD.proportion_inerts_O2 = [78.084, 0.9365, 0.0319] ./ 20.9476;
%% ADDITIONAL INPUTS (DEPENDS OF THE PROBLEM SELECTED)
% range1 = logspace(0, 1, 300); range1 = range1(range1 < 5);
% overdriven = [range1, linspace(5, 14, 30)]; overdriven = overdriven(overdriven > 1);
% overdriven = 2:2:14;
% overdriven = [2, 3, 5, 14];
overdriven = 6.5;
self = set_prop(self, 'u1', 3.472107491008314e+02 * overdriven, 'theta', [5:5:31.77]);
%% SOLVE PROBLEM
self = SolveProblem(self, 'SHOCK_POLAR_R');
%% DISPLAY RESULTS (PLOTS)
postResults(self);
```

Example_DET.m

```
% -------------------------------------------------------------------------
% EXAMPLE: DET
%
% Compute pre-shock and post-shock state for a planar detonation
% considering Chapman-Jouguet (CJ) theory for lean to rich CH4-air mixtures
% at standard conditions, a set of 26 species considered and a set of
% equivalence ratios (phi) contained in (0.5, 5) [-]
%
%
% Soot formation == {'CO2','CO','H2O','H2','O2','N2','He','Ar','Cbgrb',...
%                    'C2','C2H4','CH','CH','CH3','CH4','CN','H',...
%                    'HCN','HCO','N','NH','NH2','NH3','NO','O','OH'}
%
% See wiki or ListSpecies() for more predefined sets of species
%
% @author: Alberto Cuadra Lara
%          PhD Candidate - Group Fluid Mechanics
%          Universidad Carlos III de Madrid
%
% Last update Oct 22 2021
% -------------------------------------------------------------------------

%% INITIALIZE
self = App('Soot Formation');
%% INITIAL CONDITIONS
self = set_prop(self, 'TR', 300, 'pR', 1 * 1.01325, 'phi', 0.5:0.01:4);
self.PD.S_Fuel     = {'CH4'};
self.PD.S_Oxidizer = {'O2'};
self.PD.S_Inert    = {'N2', 'Ar', 'CO2'};
self.PD.proportion_inerts_O2 = [78.084, 0.9365, 0.0319] ./ 20.9476;
%% ADDITIONAL INPUTS (DEPENDS OF THE PROBLEM SELECTED)
% No additional data required. The initial velocity is unique for CJ
% condition
%% SOLVE PROBLEM
self = SolveProblem(self, 'DET');
%% DISPLAY RESULTS (PLOTS)
postResults(self);
```

Example_DET_R.m

```
% -------------------------------------------------------------------------
% EXAMPLE: DET REFLECTED
%
% Compute pre-shock and post-shock state for a reflected planar detonation
% considering Chapman-Jouguet (CJ) theory for lean to rich CH4-air mixtures
% at standard conditions, a set of 26 species considered and a set of
% equivalence ratios (phi) contained in (0.5, 5) [-]
%
%
```

(continues on next page)

(continued from previous page)

```
% Soot formation == {'CO2','CO','H2O','H2','O2','N2','He','Ar','Cbgrb',...
%                    'C2','C2H4','CH','CH','CH3','CH4','CN','H',...
%                    'HCN','HCO','N','NH','NH2','NH3','NO','O','OH'}
%
% See wiki or ListSpecies() for more predefined sets of species
%
% @author: Alberto Cuadra Lara
%          PhD Candidate - Group Fluid Mechanics
%          Universidad Carlos III de Madrid
%
% Last update April 04 2022
% -------------------------------------------------------------------------

%% INITIALIZE
self = App('Soot Formation');
%% INITIAL CONDITIONS
self = set_prop(self, 'TR', 300, 'pR', 1 * 1.01325, 'phi', 0.5:0.01:5);
self.PD.S_Fuel     = {'CH4'};
self.PD.S_Oxidizer = {'O2'};
self.PD.S_Inert    = {'N2', 'Ar', 'CO2'};
self.PD.proportion_inerts_O2 = [78.084, 0.9365, 0.0319] ./ 20.9476;
%% ADDITIONAL INPUTS (DEPENDS OF THE PROBLEM SELECTED)
% No additional data required. The initial velocity is unique for CJ
% condition
%% SOLVE PROBLEM
self = SolveProblem(self, 'DET_R');
%% DISPLAY RESULTS (PLOTS)
postResults(self);
```

Example_DET_OVERDRIVEN.m

```
% -------------------------------------------------------------------------
% EXAMPLE: DET_OVERDRIVEN
%
% Compute pre-shock and post-shock state for a planar overdriven detonation
% considering Chapman-Jouguet (CJ) theory for a stoichiometric CH4-air
% mixture at standard conditions, a set of 26 species considered and a set
% of overdrives contained in (1,10) [-].
%
% Soot formation == {'CO2','CO','H2O','H2','O2','N2','He','Ar','Cbgrb',...
%                    'C2','C2H4','CH','CH','CH3','CH4','CN','H',...
%                    'HCN','HCO','N','NH','NH2','NH3','NO','O','OH'}
%
% See wiki or ListSpecies() for more predefined sets of species
%
% @author: Alberto Cuadra Lara
%          PhD Candidate - Group Fluid Mechanics
%          Universidad Carlos III de Madrid
%
% Last update March 17 2022
```

(continues on next page)

(continued from previous page)

```
% -------------------------------------------------------------------------

%% INITIALIZE
self = App('Soot Formation');
%% INITIAL CONDITIONS
self = set_prop(self, 'TR', 300, 'pR', 1 * 1.01325, 'phi', 1);
self.PD.S_Fuel     = {'CH4'};
self.PD.S_Oxidizer = {'O2'};
self.PD.S_Inert    = {'N2', 'Ar', 'CO2'};
self.PD.proportion_inerts_O2 = [78.084, 0.9365, 0.0319] ./ 20.9476;
%% ADDITIONAL INPUTS (DEPENDS OF THE PROBLEM SELECTED)
overdriven = 1:0.1:10;
self = set_prop(self, 'overdriven', overdriven);
% condition
%% SOLVE PROBLEM
self = SolveProblem(self, 'DET_OVERDRIVEN');
%% DISPLAY RESULTS (PLOTS)
postResults(self);
```

Example_DET_OVERDRIVEN_R.m

```
% -------------------------------------------------------------------------
% EXAMPLE: DET OVERDRIVEN REFLECTED
%
% Compute pre-shock and post-shock state for a reflected planar overdriven
% detonation considering Chapman-Jouguet (CJ) theory for a stoichiometric
% CH4-air mixture at standard conditions, a set of 26 species considered
% and a set of overdrives contained in (1,10) [-].
%
% Soot formation == {'CO2','CO','H2O','H2','O2','N2','He','Ar','Cbgrb',...
%                    'C2','C2H4','CH','CH','CH3','CH4','CN','H',...
%                    'HCN','HCO','N','NH','NH2','NH3','NO','O','OH'}
%
% See wiki or ListSpecies() for more predefined sets of species
%
% @author: Alberto Cuadra Lara
%          PhD Candidate - Group Fluid Mechanics
%          Universidad Carlos III de Madrid
%
% Last update March 24 2022
% -------------------------------------------------------------------------

%% INITIALIZE
self = App('Soot Formation');
%% INITIAL CONDITIONS
self = set_prop(self, 'TR', 300, 'pR', 1 * 1.01325, 'phi', 1);
self.PD.S_Fuel     = {'CH4'};
self.PD.S_Oxidizer = {'O2'};
self.PD.S_Inert    = {'N2', 'Ar', 'CO2'};
self.PD.proportion_inerts_O2 = [78.084, 0.9365, 0.0319] ./ 20.9476;
```

(continues on next page)

(continued from previous page)

```
%% ADDITIONAL INPUTS (DEPENDS OF THE PROBLEM SELECTED)
overdriven = 1:0.1:10;
self = set_prop(self, 'overdriven', overdriven);
%% SOLVE PROBLEM
self = SolveProblem(self, 'DET_OVERDRIVEN_R');
%% DISPLAY RESULTS (PLOTS)
postResults(self);
```

Example_DET_OBLIQUE_BETA.m

```
% -------------------------------------------------------------------------
% EXAMPLE: DET_OBLIQUE_BETA
%
% Compute pre-shock and post-shock state for a oblique overdriven detonation
% considering Chapman-Jouguet (CJ) theory for a stoichiometric CH4-air
% mixture at standard conditions, a set of 26 species considered, an
% overdrive of 4 and a set of wave angles [15:5:80] [deg].
%
% Soot formation == {'CO2','CO','H2O','H2','O2','N2','He','Ar','Cbgrb',...
%                    'C2','C2H4','CH','CH','CH3','CH4','CN','H',...
%                    'HCN','HCO','N','NH','NH2','NH3','NO','O','OH'}
%
% See wiki or ListSpecies() for more predefined sets of species
%
% @author: Alberto Cuadra Lara
%          PhD Candidate - Group Fluid Mechanics
%          Universidad Carlos III de Madrid
%
% Last update March 25 2022
% -------------------------------------------------------------------------

%% INITIALIZE
self = App('Soot Formation');
%% INITIAL CONDITIONS
self = set_prop(self, 'TR', 300, 'pR', 1 * 1.01325, 'phi', 1);
self.PD.S_Fuel     = {'CH4'};
self.PD.S_Oxidizer = {'O2'};
self.PD.S_Inert    = {'N2', 'Ar', 'CO2'};
self.PD.proportion_inerts_O2 = [78.084, 0.9365, 0.0319] ./ 20.9476;
%% ADDITIONAL INPUTS (DEPENDS OF THE PROBLEM SELECTED)
overdriven = 4;
self = set_prop(self, 'overdriven', overdriven, 'beta', 15:5:80);
%% SOLVE PROBLEM
self = SolveProblem(self, 'DET_OBLIQUE');
%% DISPLAY RESULTS (PLOTS)
postResults(self);
```

Example_DET_OBLIQUE_THETA.m

```
% -------------------------------------------------------------------------
% EXAMPLE: DET_OBLIQUE_THETA
%
% Compute pre-shock and post-shock state for a oblique overdriven detonation
% considering Chapman-Jouguet (CJ) theory for a stoichiometric CH4-air
% mixture at standard conditions, a set of 24 species considered, an
% overdrive of 4 and a set of deflection angles [10:5:40] [deg].
%
% Soot formation == {'CO2', 'CO', 'H2O', 'H2', 'O2', 'N2', 'He', 'Ar',...
%                    'HCN','H','OH','O','CN','NH3','CH4','C2H4','CH3',...
%                    'NO','HCO','NH2','NH','N','CH','Cbgrb'}
%
% See wiki or ListSpecies() for more predefined sets of species
%
% @author: Alberto Cuadra Lara
%          PhD Candidate - Group Fluid Mechanics
%          Universidad Carlos III de Madrid
%
% Last update March 25 2022
% -------------------------------------------------------------------------

%% INITIALIZE
self = App('Soot Formation');
%% INITIAL CONDITIONS
self = set_prop(self, 'TR', 300, 'pR', 1 * 1.01325, 'phi', 1);
self.PD.S_Fuel     = {'CH4'};
self.PD.S_Oxidizer = {'O2'};
self.PD.S_Inert    = {'N2', 'Ar', 'CO2'};
self.PD.proportion_inerts_O2 = [78.084, 0.9365, 0.0319] ./ 20.9476;
%% ADDITIONAL INPUTS (DEPENDS OF THE PROBLEM SELECTED)
overdriven = 4;
self = set_prop(self, 'overdriven', overdriven, 'theta', 15);
%% SOLVE PROBLEM
self = SolveProblem(self, 'DET_OBLIQUE');
%% DISPLAY RESULTS (PLOTS)
postResults(self);
```

Example_ROCKET.m

```
% -------------------------------------------------------------------------
% EXAMPLE: ROCKET Propellants
%
% Compute adiabatic temperature and equilibrium composition at constant
% pressure (e.g., 1.01325 bar) for lean to rich LH2-LOX mixtures at
% standard conditions, a set of 24 species considered and a set of
% equivalence ratios phi contained in (2, 8) [-]
%
% HYDROGEN_L == {'H','H2O','OH','H2','O','O3','O2','HO2','H2O2',...
%                'H2bLb','O2bLb'}
```

(continues on next page)

(continued from previous page)

```
%
% See wiki or ListSpecies() for more predefined sets of species
%
% @author: Alberto Cuadra Lara
%          PhD Candidate - Group Fluid Mechanics
%          Universidad Carlos III de Madrid
%
% Last update March 24 2022
% -------------------------------------------------------------------------

%% INITIALIZE
self = App('HYDROGEN_L');
%% INITIAL CONDITIONS
self = set_prop(self, 'TR', 90, 'pR', 1 * 1.01325, 'phi', 2:0.05:8);
self.PD.S_Fuel     = {'H2bLb'};
self.PD.S_Oxidizer = {'O2bLb'};
%% SOLVE PROBLEM
self = SolveProblem(self, 'ROCKET');
%% DISPLAY RESULTS (PLOTS)
postResults(self);
```

7.4 Validations

A set of the results obtained using Combustion Toolbox, CEA-NASA, CANTERA & SD-Toolbox1, and TEA.

1 The Shock & Detonation Toolbox uses the Cantera software package as kernel for the thermochemical calculations.

For the sake of clarity, we only show a reduced set of species in the validation of the mole fractions. To run all the validations contrasted with CEA at once, at the prompt type:

```
>> run_validations_CEA
```

7.4.1 Validation TP 1

- Contrasted with: NASA's Chemical Equilibrium with Applications software
- Problem type: Equilibrium composition at defined T and p
- Temperature [K] = 2500
- Pressure [bar] = 1
- Initial mixture [moles]:
- List of species considered = `ListSpecies('Soot formation Extended')`
- URL Folder Results CEA: ./Validations/CEA/Data/TP

To repeat the results, run:

```
>> run_validation_TP_CEA_1.m
```

7.4.2 Validation TP 2

- Contrasted with: NASA's Chemical Equilibrium with Applications software
- Problem type: Equilibrium composition at defined T and p
- Temperature [K] = 2500
- Pressure [bar] = 1
- Initial mixture [moles]:
- List of species considered = `ListSpecies('Soot formation Extended')`
- URL Folder Results CEA: ./Validations/CEA/Data/TP

To repeat the results, run:

```
>> run_validation_TP_CEA_2.m
```

7.4.3 Validation TP 3

- Contrasted with: NASA's Chemical Equilibrium with Applications software
- Problem type: Equilibrium composition at defined T and p
- Temperature [K] = 2500
- Pressure [bar] = 1
- Initial mixture [moles]:
- List of species considered = `ListSpecies('Soot formation Extended')`
- URL Folder Results CEA: ./Validations/CEA/Data/TP

To repeat the results, run:

```
>> run_validation_TP_CEA_3.m
```

7.4.4 Validation TP 4

- Contrasted with: NASA's Chemical Equilibrium with Applications software
- Problem type: Equilibrium composition at defined T and p
- Temperature [K] = 2500
- Pressure [bar] = 1
- Initial mixture [moles]:
- List of species considered = `ListSpecies('Soot formation Extended')`
- URL Folder Results CEA: ./Validations/CEA/Data/TP

To repeat the results, run:

```
>> run_validation_TP_CEA_4.m
```

7.4.5 Validation HP 1

- Contrasted with: NASA's Chemical Equilibrium with Applications software
- Problem type: Adiabatic T and composition at constant p
- Temperature [K] = 300
- Pressure [bar] = 1
- Initial mixture [moles]:
- List of species considered = `ListSpecies('Soot formation Extended')`
- URL Folder Results CEA: ./Validations/CEA/Data/HP

To repeat the results, run:

```
>> run_validation_HP_CEA_1.m
```

7.4.6 Validation HP 2

- Contrasted with: NASA's Chemical Equilibrium with Applications software
- Problem type: Adiabatic T and composition at constant p
- Temperature [K] = 300
- Pressure [bar] = 1
- Initial mixture [moles]:
- List of species considered = `ListSpecies('Soot formation Extended')`
- URL Folder Results CEA: ./Validations/CEA/Data/HP

To repeat the results, run:

```
>> run_validation_HP_CEA_2.m
```

7.4.7 Validation HP 3

- Contrasted with: NASA's Chemical Equilibrium with Applications software
- Problem type: Adiabatic T and composition at constant p
- Temperature [K] = 300
- Pressure [bar] = 1
- Initial mixture [moles]:
- List of species considered = `ListSpecies('Soot formation Extended')`
- URL Folder Results CEA: ./Validations/CEA/Data/HP

To repeat the results, run:

```
>> run_validation_HP_CEA_3.m
```

7.4.8 Validation HP 4

- Contrasted with: NASA's Chemical Equilibrium with Applications software
- Problem type: Adiabatic T and composition at constant p
- Temperature [K] = 300
- Pressure [bar] = 1
- Initial mixture [moles]:
- List of species considered = `ListSpecies('Soot formation Extended')`
- URL Folder Results CEA: ./Validations/CEA/Data/HP

To repeat the results, run:

```
>> run_validation_HP_CEA_4.m
```

7.4.9 Validation DET 1

- Contrasted with: NASA's Chemical Equilibrium with Applications software
- Problem type: Chapman-Jouget Detonation
- Temperature [K] = 300
- Pressure [bar] = 1
- Initial mixture [moles]:
- List of species considered = `ListSpecies('Soot formation Extended')`
- URL Folder Results CEA: ./Validations/CEA/Data/HP

To repeat the results, run:

```
>> run_validation_DET_CEA_1.m
```

7.4.10 Validation DET 2

- Contrasted with: NASA's Chemical Equilibrium with Applications software
- Problem type: Chapman-Jouget Detonation
- Temperature [K] = 300
- Pressure [bar] = 1
- Initial mixture [moles]:
- List of species considered = `ListSpecies('Soot formation Extended')`
- URL Folder Results CEA: ./Validations/CEA/Data/DET

To repeat the results, run:

```
>> run_validation_DET_CEA_2.m
```

7.4.11 Validation DET 3

- Contrasted with: NASA's Chemical Equilibrium with Applications software
- Problem type: Chapman-Jouget Detonation
- Temperature [K] = 300
- Pressure [bar] = 1
- Initial mixture [moles]:
- List of species considered = `ListSpecies('Soot formation Extended')`
- URL Folder Results CEA: ./Validations/CEA/Data/DET

To repeat the results, run:

```
>> run_validation_DET_CEA_3.m
```

7.4.12 Validation DET 4

- Contrasted with: NASA's Chemical Equilibrium with Applications software
- Problem type: Chapman-Jouget Detonation
- Temperature [K] = 300
- Pressure [bar] = 1
- Initial mixture [moles]:
- List of species considered = `ListSpecies('Soot formation Extended')`
- URL Folder Results CEA: ./Validations/CEA/Data/DET

To repeat the results, run:

```
>> run_validation_DET_CEA_4.m
```

7.4.13 Validation SHOCK IONIZATION 1

- Contrasted with: NASA's Chemical Equilibrium with Applications software
- Problem type: Shock incident
- Temperature [K] = 300
- Pressure [bar] = 1
- Initial mixture [moles]:
- List of species considered = ListSpecies('Air_ions')
- URL Folder Results CEA: ./Validations/CEA/Data/Shocks

To repeat the results, run:

```
>> run_validation_SHOCK_IONIZATION_CEA_1.m
```

7.4.14 Validation SHOCK REFLECTED IONIZATION 1

- Contrasted with: NASA's Chemical Equilibrium with Applications software
- Problem type: Shock reflected
- Temperature [K] = 300
- Pressure [bar] = 1
- Initial mixture [moles]:
- List of species considered = `ListSpecies('Air_ions')`
- URL Folder Results CEA: ./Validations/CEA/Data/Shocks

To repeat the results, run:

```
>> run_validation_SHOCK_R_IONIZATION_CEA_1.m
```

7.4.15 Validation SHOCK POLAR 1

- Contrasted with: Caltech's SD Toolbox and CANTERA
- Problem type: Shock Polar
- Temperature [K] = 300
- Pressure [bar] = 1.01325
- Initial mixture [moles]:
- List of species considered = Frozen
- URL Folder Results SDToolbox: ./Validations/SDToolbox/Data

To repeat the results, run:

```
>> run_validation_SHOCK_POLAR_SDToolbox_1.m
```

7.4.16 Validation ROCKET 1

- Contrasted with: NASA's Chemical Equilibrium with Applications software
- Problem type: ROCKET
- Description: Equilibrium composition at exit of the rocket nozzle
- Temperature Fuel [K] = 298.15
- Temperature Oxid [K] = 90.17
- Chamber pressure [bar] = 22
- Finite-Area-Chamber model (FAC)
- Area ratio $A_{\text{chamber}}/A_{\text{throat}}$ = 2
- Area ratio $A_{\text{exit}}/A_{\text{throat}}$ = [1:2.6]
- Initial mixture [moles]:
- List of species considered = HC/O2/N2 PROPELLANTS

- URL Folder Results CEA: ./Validations/CEA/ROCKET

To repeat the results, run:

```
>> run_validation_ROCKET_CEA_1.m
>> run_validation_ROCKET_CEA_16.m
```

7.4.17 Validation TEA 1

- Contrasted with: Thermochemical Equilibrium Abundances of chemical species software
- Problem type: Equilibrium composition at defined T and p
- Temperature [K] = linspace(500, 5000)
- Pressure [bar] = logspace(-5, 2)
- Initial mixture [moles]:
  - H = 1.0000000000e+00
  - He = 8.5113803820e-02
  - C = 2.6915348039e-04
  - N = 6.7608297539e-05
  - O = 4.8977881937e-04
- List of species considered = {'C', 'CH4', 'CO2', 'CO', 'H2', 'H', 'H2O', 'He', 'N2', 'N', 'NH3', 'O'}
- URL Results TEA: https://github.com/dzesmin/TEA/blob/master/doc/examples/quick_example/results/quick_example.tea

To repeat the results, run:

```
>> run_validation_TP_TEA_1.m
```

7.4.18 Validation TEA 2

- Contrasted with: Thermochemical Equilibrium Abundances of chemical species software
- Problem type: Equilibrium composition at defined T and p
- Description: Thermochemical equilibrium vertical distribution of WASP-43b with a metallicity $\zeta = 1$ assuming a T-P profile
- Temperature [K] = [958.36, 1811.89]
- Pressure [bar] = [2.3988e-06, 31.6230]
- Initial mixture: Computed from solar abundances assuming a metallicity zeta = 1
- List of species considered = {'C2H2_acetylene', 'C2H4', 'C', 'CH4', 'CO2', 'CO', 'H2', 'H2O', 'H2S', 'H', 'HCN', 'He', 'SH', 'N2', 'N', 'NH3', 'O', 'S'}
- URL Results TEA: https://github.com/dzesmin/RRC-BlecicEtal-2015a-ApJS-TEA/tree/master/Fig6/WASP43b-solar

To repeat the results, run:

```
>> run_validation_TP_TEA_2.m
```

7.4.19 Validation TEA 3

- Contrasted with: Thermochemical Equilibrium Abundances of chemical species software
- Problem type: Equilibrium composition at defined T and p
- Description: Thermochemical equilibrium vertical distribution of WASP-43b with a metallicity $\zeta = 10$ assuming a T-P profile
- Temperature [K] = [958.36, 1811.89]
- Pressure [bar] = [2.3988e-06, 31.6230]
- Initial mixture: Computed from solar abundances assuming a metallicity zeta = 10
- List of species considered = {'C2H2_acetylene', 'C2H4', 'C', 'CH4', 'CO2', 'CO', 'H2', 'H2O', 'H2S', 'H', 'HCN', 'He', 'SH', 'N2', 'N', 'NH3', 'O', 'S'}
- URL Results TEA: https://github.com/dzesmin/RRC-BlecicEtal-2015a-ApJS-TEA/tree/master/Fig6/WASP43b-10xsolar

To repeat the results, run:

```
>> run_validation_TP_TEA_3.m
```

7.4.20 Validation TEA 4

- Contrasted with: Thermochemical Equilibrium Abundances of chemical species software
- Problem type: Equilibrium composition at defined T and p
- Description: Thermochemical equilibrium vertical distribution of WASP-43b with a metallicity $\zeta = 50$ assuming a T-P profile
- Temperature [K] = [958.36, 1811.89]
- Pressure [bar] = [2.3988e-06, 31.6230]
- Initial mixture: Computed from solar abundances assuming a metallicity zeta = 50
- List of species considered = {'C2H2_acetylene', 'C2H4', 'C', 'CH4', 'CO2', 'CO', 'H2', 'H2O', 'H2S', 'H', 'HCN', 'He', 'SH', 'N2', 'N', 'NH3', 'O', 'S'}
- URL Results TEA: https://github.com/dzesmin/RRC-BlecicEtal-2015a-ApJS-TEA/tree/master/Fig6/WASP43b-50xsolar

To repeat the results, run:

```
>> run_validation_TP_TEA_4.m
```

7.5 Publications

7.5.1 Citing Combustion Toolbox

If you use Combustion Toolbox in a publication, please cite it using the following reference:

- *Cuadra, A., Huete, C., & Vera, M. (2022). Combustion Toolbox: A MATLAB-GUI based open-source tool for solving combustion problems. (v0.9.6). Zenodo. https://doi.org/10.5281/zenodo.6635715.*

It can be handy the BibTeX format:

```
@misc{combustiontoolbox,
    author = "Cuadra, A and Huete, C and Vera, M",
    title = "Combustion Toolbox: A MATLAB-GUI based open-source tool for solving␣
↪combustion problems",
    year = 2022,
    note = "Version 0.9.6",
    doi = {https://doi.org/10.5281/zenodo.6635715}
}
```

7.5.2 Contributions

Here we have some contributions, work in progress articles, and publications in which Combustion Toolbox has been used.

Journal articles

- Cuadra, A, Huete, C., & Vera, M. (2022). Development of a wider-scope thermochemical code **(work in progress)**.
- Juan Sánchez-Monreal, Alberto Cuadra-Lara, César Huete, Marcos Vera (2022). SimEx: A tool for rapid evaluation of the effects of explosions **(work in progress)**.
- Huete, C., Cuadra, A., Vera, M., & Urzay, J. (2021). Thermochemical effects on hypersonic shock waves interacting with weak turbulence. Physics of Fluids 33, 086111 (2021) **(featured article)**. https://doi.org/10.1063/5.0059948
- Cuadra, A., Huete, C., & Vera, M. (2020). Effect of equivalence ratio fluctuations on planar detonation discontinuities. Journal of Fluid Mechanics, 903, A30. https://doi.org/10.1017/jfm.2020.651

Conference contributions

- Cuadra, A., Huete, C., Vera, M., & Urzay, J. (2021). Theory of turbulence augmentation across hypersonic shock waves. In 74th Annual Meeting of the Division of Fluid Dynamics (APS DFD), Phoenix, US.
- Cuadra, A., Huete, C., & Vera, M. (2021). Effect of fuel mass fraction heterogeneity on the detonation propagation speed. In 25th International Congress of Theoretical and Applied Mechanics (ICTAM), Milano, Italy.
- Cuadra, A., & Vera, M. (2019). Development and validation of a new MATLAB®/GUI based thermochemical code. In 11th International Mediterranean Combustion Symposium (MSC), Tenerife, Spain.
- Cuadra, A., & Vera, M. (2019). Development of a GUI-based thermochemical code with teaching and research applications. In 1st Colloquium of the Spanish Theoretical and Applied Mechanics Society (STAMS), Madrid, Spain.

Seminars & Workshops

- Cuadra, A., Huete, C. & Vera, M. (2021). Development of an open-source thermochemical code: Fundamentals and application to shock turbulence interaction problems in the hypersonic regime. Seminar presented as part of the PhD Programme in Mechatronics Engineering, Málaga, Spain

PhD, MSc & BSc thesis

- Cuadra, A (2022). Development of a wide-spectrum thermochemical code with application to the analysis of combustion problems and high energy materials. Universidad Carlos III de Madrid, Spain (PhD thesis - work in progress). Advisors: Marcos Vera & César Huete.
- Aguilar, C (2022). CT-ROCKET: A MATLAB-GUI based thermochemical code to estimate rocket propellant performance. Universidad Carlos III de Madrid, Spain (BSc thesis - work in progress). Advisors: Alberto Cuadra.
- Cuadra, A (2019). Development of a GUI-based thermochemical code with teaching and research applications. Universidad Carlos III de Madrid, Spain (MSc thesis). Advisors: Marcos Vera.

7.6 Functions

Here we can find the documentation for the routines implemented in Combustion Toolbox. We have different modules:

- settings: routines to generate a self struct variable with all the data necessary (mixtures, conditions, databases),
- databases: routines to generate the databases using the NASA's 9 coefficient polynomials fits,
- solver: routines to solve different type of problems implemented,
- gui: routines to generate the app, all the necessary functions to be compatible with the plain code and extend its functionality.

Caution: The documentation is under development

7.6.1 Database

Combustion Toolbox generates its own databases taking into account the NASA-9 polynomials fitting to evaluate the dimensionless thermodynamic functions of the species for the specific heat, enthalpy, and entropy as function of temperature, namely

$$\begin{aligned} c_p^\circ/R &= a_1T^{-2} + a_2T^{-1} + a_3 + a_4T + a_5T^2 + a_6T^3 + a_7T^4, && (7.1)\\ h^\circ/RT &= -a_1T^{-2} + a_2T^{-1}\ln T + a_3 + a_4T/2 + a_5T^2/3 + a_6T^3/4 && (7.2)\\ &\quad +a_7T^4/5 + a_8/T, && (7.3)\\ s^\circ/R &= -a_1T^{-2}/2 - a_2T^{-1} + a_3\ln T + a_4T + a_5T^2/2 + a_6T^3/3 && (7.4)\\ &\quad +a_7T^4/4 + a_9, && (7.5) \end{aligned}$$

where a_i from $i = 1, \ldots, 7$ are the temperature coefficients and $i = 8, 9$ are the integration constants, respectively. Depending of the species the polynomials fit up to 20000 K [1]. These values are available in the source code and can be also obtained from NASA's thermo build website.

To compute the dimensionless Gibbs energy, $g_i^\circ(T)/RT$, from NASA's polynomials we use the next expression

$$g_i^\circ/RT = h^\circ/RT - s^\circ/R, \tag{7.6}$$

or equivalently

$$g_i^\circ/RT = -a_1T^{-2}/2 + a_2T^{-1}(1+\ln T) + a_3(1-\ln T) - a_4T/2 - a_5T^2/6 - a_6T^3/12 \quad (7.7)$$
$$-a_7T^4/20 + a_8/T - a_9. \quad (7.8)$$

This data is collected and formatted into a more accessible structure. We can distinguish from:

- `DB_master`: structured database from NASA's thermo file.
- `DB`: structured database with piecewise cubic Hermite interpolating polynomials and linear extrapolation for faster data access.

The use of piecewise cubic Hermite interpolating polynomials increments the performance of Combustion Toolbox in approximate 200% as shown in **Figure 1** obtaining the same results as seen in **Figure 2**. To evaluate the thermodynamic functions, e.g., the Gibbs energy [kJ/mol] function, or the thermal enthalpy [kJ/mol] of CO_2 at $T = 2000$ K is as simple as using these callbacks

```
>> g0_CO2  = species_g0('CO2', 2000, DB)
>> DhT_CO2 = species_DhT('CO2', 2000, DB)
```

Figure 1: *Performance test, execution times for over 10^5 calculations of the specific heat at constant pressure, enthalpy, Gibbs energy, and entropy, denoted as c_p, h_0, g_0, and s_0, respectively, using the NASA's 9 coefficient polynomials (dark blue) and the piecewise cubic Hermite interpolating polynomials (teal). The test has been carried out with an Intel(R) Core(TM) i7-8700 CPU @ 3.20GHz. Note: lower is better.*

Figure 2: *Comparison of entropy [kJ/(mol-K)] as a function of temperature [K] obtained using the piecewise cubic Hermite interpolating polynomials (lines) and using the NASA's 9 coefficient polynomials (symbols) for a set of species.*

Another important parameter comes from the conservation of mass, which is the stoichiometric matrix A_0, by generalizing this constraint condition we have

$$\sum_{j=1}^{\mathrm{NS}} a_{ij} n_j - b_i^\circ = 0, \quad (7.9)$$

or in matricial form

$$\underbrace{\begin{pmatrix} a_{11} & a_{21} & \cdots & a_{\mathrm{NS}1} \\ a_{12} & a_{22} & \cdots & a_{\mathrm{NS}2} \\ \vdots & \vdots & & \vdots \\ a_{1\mathrm{NE}} & a_{2\mathrm{NE}} & \cdots & a_{\mathrm{NSNE}} \end{pmatrix}}_{\mathbf{A^T}} \underbrace{\begin{pmatrix} n_1 \\ n_2 \\ \vdots \\ n_{\mathrm{NS}} \end{pmatrix}}_{\mathbf{N}} - \underbrace{\begin{pmatrix} b_1 \\ b_2 \\ \vdots \\ b_{\mathrm{NE}} \end{pmatrix}}_{\mathbf{b^\circ}} = 0, \quad (7.10)$$

where a_{ij} are the stoichiometric coefficients of the species, which represent the number of atoms of element i per mole of species j. The number of moles and the total number of atoms of the j species and i element reads n_j and b_i, respectively. This is computed during the initialization of the variable `self`. Using one of the predefined list of species, e.g., `soot formation`, this can be initializate as

```
self = App('soot formation')
```

A simple test can be performed by considering the global exothermic reaction of hydrogen bromide with n_j moles

$$\mathrm{H_2} + \mathrm{Br_2} \rightleftharpoons 2\mathrm{HBr} \quad (7.11)$$

which have only three species involve. The system obtained is

$$\begin{pmatrix} 0 & 2 & 1 \\ 2 & 0 & 1 \end{pmatrix} \begin{pmatrix} n_{H_2} \\ n_{Br_2} \\ n_{HBr} \end{pmatrix} - \begin{pmatrix} b_{Br}^\circ \\ b_H^\circ \end{pmatrix} = 0. \quad (7.12)$$

A quick check using Combustion Toolbox:

```
>> self = App({'H2', 'Br2', 'HBr'});
>> print_stoichiometric_matrix(self, 'transpose')
```

```
Transpose stoichiometric matrix:

         H2    Br2    HBr
         __    ___    ___

   BR    0      2      1
   H     2      0      1
```

Routines

FullName2name(*species*)

Get full name of the given species

Parameters **species** (str) – Chemical species

Returns *name (str)* – Full name of the given species

check_DB(*self*, *DB_master*, *DB*, *varargin*)

Include not defined species in database from master database

Parameters

- **self** (struct) – struct with elements data
- **DB_master** (struct) – Database with the thermodynamic data of the chemical species
- **DB** (struct) – Database with custom thermodynamic polynomials functions generated from NASAs 9 polynomials fits

Optional Args: LS_check (cell)

Returns

Tuple containing –

- DB (struct): Database with custom thermodynamic polynomials functions generated from NASAs 9 polynomials fits
- E (struct): Elements data
- S (struct): Slements data
- C (struct): Constant data

compute_change_moles_gas_reaction(*element_matrix*, *swtCondensed*)

In order to compute the internal energy of formation from the enthalpy of formation of a given species, we must determine the change in moles of gases during the formation reaction of a mole of that species starting from the elements in their reference state.

Notes: The only elements that are stable as diatomic gases are elements 1 (H), 8 (N), 9 (O), 10 (F), and 18 (Cl). The remaining elements that are stable as (monoatomic) gases are the noble gases He (3), Ne (11), Ar (19), Kr (37), Xe (55), and Rn (87), which do not form any compound.

Parameters

- **element_matrix** (float) – Element matrix of the species
- **swtCondensed** (float) – 0 or 1 indicating gas or condensed species

Returns *Delta_n (float)* – Change in moles of gases during the formation reaction of a mole of that species starting from the elements in their reference state

compute_interval_NASA(*species*, *T*, *DB*, *tRange*, *ctTInt*)
Compute interval NASA polynomials

Parameters

- **species** (str) – Chemical species
- **T** (float) – Temperature [K]
- **DB** (struct) – Database with custom thermodynamic polynomials functions generated from NASAs 9 polynomials fits
- **tRange** (cell) – Ranges of temperatures [K]
- **ctTInt** (float) – Number of intervals of temperatures

Returns *tInterval (float)* – Index of the interval of temperatures

detect_location_of_phase_specifier(*species*)
Detect the location of the opening pharenthesis of the phase identifier (if any)

Parameters **species** (str) – Chemical species

Returns *n_open_parenthesis (float)* – Index of the location of the open parenthesis

generate_DB(*DB_master*)
Generate Database (DB) with thermochemical interpolation curves for the species contained in DB_master

Parameters **DB_master** (struct) – Database with the thermodynamic data of the chemical species

Returns *DB (struct)* – Database with custom thermodynamic polynomials functions generated from NASAs 9 polynomials fits

generate_DB_Theo()
Generate database for theoretical computation of the jump conditions of a diatomic species only considering dissociation.

Returns *DB_Theo (struct)* – Database with quantum data of several diatomic species

generate_DB_master(*varargin*)
Generate Mater Database (DB_master) with the thermodynamic data of the chemical species

Optional args:

- reducedDB (flag): Flag indicating reduced database
- thermoFile (file): File with NASA's thermodynamic database

Returns *DB_master (struct)* – Database with the thermodynamic data of the chemical species

generate_DB_master_reduced(*DB_master*)
Generate Reduced Mater Database (DB_master_reduced) with the thermodynamic data of the chemical species

Parameters **DB_master** (struct) – Database with the thermodynamic data of the chemical species

Returns *DB_master_reduced (struct)* – Reduced database with the thermodynamic data of the chemical species

get_g0(*self*, *species*, *T*, *DB*)
Compute Compute Gibbs energy [kJ/mol] of the species at the given temperature [K] using NASA's 9 polynomials

Parameters

- **self** (str) – Data of the mixture, conditions, and databases
- **species** (str) – Chemical species
- **temperature** (float) – Range of temperatures to evaluate [K]
- **DB** (struct) – Database with custom thermodynamic polynomials functions generated from NASAs 9 polynomials fits

Returns *g0 (float)* – Gibbs energy [kJ/mol]

get_interval(*species*, *T*, *DB*)
Get interval of the NASA's polynomials from the Database (DB) for the given species and temperature [K].

Parameters

- **species** (str) – Chemical species
- **T** (float) – Temperature [K]
- **DB** (struct) – Database with custom thermodynamic polynomials functions generated from NASAs 9 polynomials fits

Returns *tInterval (float)* – Index of the interval of temperatures

get_reference_elements_with_T_intervals()
Get list with reference form of elements and its temperature intervals

Returns *list (cell)* – List with reference form of elements and its temperature intervals

get_speciesProperties(*DB*, *species*, *T*, *MassOrMolar*, *echo*)
Calculates the thermodynamic properties of any species included in the NASA database

Parameters

- **DB** (struct) – Database with custom thermodynamic polynomials functions generated from NASAs 9 polynomials fits
- **species** (str) – Chemical species
- **T** (float) – Temperature [K]
- **MassOrMolar** (str) – Label indicating mass [kg] or molar [mol] units
- **echo** (float) – 0 or 1 indicating species not found

Returns

Tuple containing

- txFormula (str): Chemical formula
- mm (float): Molar weight [g/mol]
- cP0 (float): Specific heat at constant pressure [J/(mol-k)]
- cV0 (float): Specific heat at constant volume [J/(mol-k)]
- hf0 (float): Enthalpy of formation [J/mol]
- h0 (float): Enthalpy [J/mol]
- ef0 (float): Internal energy of formation [J/mol]

- e0 (float): Enthalpy [J/mol]
- s0 (float): Entropy [J/(mol-k)]
- Dg0 (float): Gibbs energy [J/mol]

isRefElm(*reference_elements*, *species*, *T*)
Check if the given species is a reference element

Parameters

- **reference_elements** (cell) – List of reference elements with temperature intervals [K]
- **species** (str) – Chemical species
- **T** (float) – Temperature

Returns *name (str)* – Full name of the given species

name_with_parenthesis(*species*)
Update the name of the given string with parenthesis. The character b if comes in pair represents parenthesis in the NASA's database

Parameters **species** (str) – Chemical species in NASA's Database format

Returns *species_with (str)* – Chemical species with parenthesis

set_element_matrix(*txFormula*, *elements*)
Compute element matrix of the given species

Parameters **txFormula** (str) – Chemical formula

Returns *element_matrix(float)* – Element matrix

Example

For CO2

element_matrix = [7, 9; 1, 2]

That is, the species contains 1 atom of element 7 (C) and 2 atoms of element 9 (O)

---

set_g0(*LS*, *T*, *DB*)
Function that computes the vector of gibbs free energy for the given set of species [J/mol]

Parameters

- **LS** (cell) – List of species
- **T** (float) – Temperature [K]
- **DB** (struct) – Database with custom thermodynamic polynomials functions generated from NASAs 9 polynomials fits

Returns *g0 (float)* – Gibbs energy [J/mol]

set_h0(*LS*, *T*, *DB*)
Function that computes the vector of enthalpies for the given set of species [J/mol]

Parameters

- **LS** (cell) – List of species
- **T** (float) – Temperature [K]

- **DB** (`struct`) – Database with custom thermodynamic polynomials functions generated from NASAs 9 polynomials fits

Returns *h0 (float)* – Gibbs energy [J/mol]

`set_reference_form_of_elements`
Get list with reference form of elements

Returns *list (cell)* – List with reference form of elements

`set_reference_form_of_elements_with_T_intervals`
Get list with reference form of elements and its temperature intervals

Returns *list (cell)* – List with reference form of elements and its temperature intervals

`species_DeT`(*species*, *T*, *DB*)
Compute thermal internal energy [kJ/mol] of the species at the given temperature [K] using piecewise cubic Hermite interpolating polynomials and linear extrapolation

Parameters

- **species** (`str`) – Chemical species
- **T** (`float`) – Temperature [K]
- **DB** (`struct`) – Database with custom thermodynamic polynomials functions generated from NASAs 9 polynomials fits

Returns *DeT (float)* – Thermal internal energy [kJ/mol]

`species_DeT_NASA`(*species*, *temperature*, *DB*)
Compute thermal internal energy [kJ/mol] of the species at the given temperature [K] using NASA's 9 polynomials

Parameters

- **species** (`str`) – Chemical species
- **temperature** (`float`) – Range of temperatures to evaluate [K]
- **DB** (`struct`) – Database with custom thermodynamic polynomials functions generated from NASAs 9 polynomials fits

Returns *DeT (float)* – Thermal internal energy [kJ/mol]

`species_DhT`(*species*, *T*, *DB*)
Compute thermal enthalpy [kJ/mol] of the species at the given temperature [K] using piecewise cubic Hermite interpolating polynomials and linear extrapolation

Parameters

- **species** (`str`) – Chemical species
- **T** (`float`) – Temperature [K]
- **DB** (`struct`) – Database with custom thermodynamic polynomials functions generated from NASAs 9 polynomials fits

Returns *DhT (float)* – Thermal enthalpy [kJ/mol]

`species_DhT_NASA`(*species*, *temperature*, *DB*)
Compute thermal enthalpy [kJ/mol] of the species at the given temperature [K] using NASA's 9 polynomials

Parameters

- **species** (`str`) – Chemical species

- **temperature** (float) – Range of temperatures to evaluate [K]
- **DB** (struct) – Database with custom thermodynamic polynomials functions generated from NASAs 9 polynomials fits

Returns *DhT (float)* – Thermal enthalpy [kJ/mol]

species_cP(*species*, *T*, *DB*)

Compute specific heat at constant pressure [J/(mol-K)] of the species at the given temperature [K] using piecewise cubic Hermite interpolating polynomials and linear extrapolation

Parameters

- **species** (str) – Chemical species
- **T** (float) – Temperature [K]
- **DB** (struct) – Database with custom thermodynamic polynomials functions generated from NASAs 9 polynomials fits

Returns *cP (float)* – Specific heat at constant pressure [J/(mol-K)]

species_cP_NASA(*species*, *temperature*, *DB*)

Compute specific heats at constant pressure and at constant volume [J/(mol-K)] of the species at the given temperature [K] using NASA's 9 polynomials

Parameters

- **species** (str) – Chemical species
- **temperature** (float) – Range of temperatures to evaluate [K]
- **DB** (struct) – Database with custom thermodynamic polynomials functions generated from NASAs 9 polynomials fits

Returns

Tuple containing

- cP (float): Specific heat at constant pressure [J/(mol-K)]
- cV (float): Specific heat at constant volume [J/(mol-K)]

species_cV(*species*, *T*, *DB*)

Compute specific heat at constant volume [J/(mol-K)] of the species at the given temperature [K] using piecewise cubic Hermite interpolating polynomials and linear extrapolation

Parameters

- **species** (str) – Chemical species
- **T** (float) – Temperature [K]
- **DB** (struct) – Database with custom thermodynamic polynomials functions generated from NASAs 9 polynomials fits

Returns *cV (float)* – Specific heat at constant volume [J/(mol-K)]

species_cV_NASA(*species*, *temperature*, *DB*)

Compute specific heat at constant volume [J/(mol-K)] of the species at the given temperature [K] using NASA's 9 polynomials

Parameters

- **species** (str) – Chemical species
- **temperature** (float) – Range of temperatures to evaluate [K]

- **DB** (`struct`) – Database with custom thermodynamic polynomials functions generated from NASAs 9 polynomials fits

Returns *cV (float)* – Specific heat at constant volume [J/(mol-K)]

species_e0_NASA(*species*, *temperature*, *DB*)

Compute internal energy and the thermal internal energy [kJ/mol] of the species at the given temperature [K] using NASA's 9 polynomials

Parameters

- **species** (`str`) – Chemical species
- **temperature** (`float`) – Range of temperatures to evaluate [K]
- **DB** (`struct`) – Database with custom thermodynamic polynomials functions generated from NASAs 9 polynomials fits

Returns

Tuple containing

- e0 (float): Internal energy [kJ/mol]
- DeT (float): Thermal internal energy [kJ/mol]

species_g0(*species*, *T*, *DB*)

Compute Gibbs energy [kJ/mol] of the species at the given temperature [K] using piecewise cubic Hermite interpolating polynomials and linear extrapolation

Parameters

- **species** (`str`) – Chemical species
- **T** (`float`) – Temperature [K]
- **DB** (`struct`) – Database with custom thermodynamic polynomials functions generated from NASAs 9 polynomials fits

Returns *g0 (float)* – Gibbs energy [kJ/mol]

species_g0_NASA(*species*, *temperature*, *DB*)

Compute Compute Gibbs energy [kJ/mol] of the species at the given temperature [K] using NASA's 9 polynomials

Parameters

- **species** (`str`) – Chemical species
- **temperature** (`float`) – Range of temperatures to evaluate [K]
- **DB** (`struct`) – Database with custom thermodynamic polynomials functions generated from NASAs 9 polynomials fits

Returns *g0 (float)* – Gibbs energy [kJ/mol]

species_h0(*species*, *T*, *DB*)

Compute enthalpy [kJ/mol] of the species at the given temperature [K] using piecewise cubic Hermite interpolating polynomials and linear extrapolation

Parameters

- **species** (`str`) – Chemical species
- **T** (`float`) – Temperature [K]

- **DB** (struct) – Database with custom thermodynamic polynomials functions generated from NASAs 9 polynomials fits

Returns *h0 (float)* – enthalpy [kJ/mol]

species_h0_NASA(*species*, *temperature*, *DB*)

Compute enthalpy and thermal enthalpy [kJ/mol] of the species at the given temperature [K] using NASA's 9 polynomials

Parameters

- **species** (str) – Chemical species
- **temperature** (float) – Range of temperatures to evaluate [K]
- **DB** (struct) – Database with custom thermodynamic polynomials functions generated from NASAs 9 polynomials fits

Returns

Tuple containing

- h0 (float): Enthalpy [kJ/mol]
- DhT (float): Thermal enthalpy [kJ/mol]

species_s0(*species*, *T*, *DB*)

Compute entropy [kJ/(mol-K)] of the species at the given temperature [K] using piecewise cubic Hermite interpolating polynomials and linear extrapolation

Parameters

- **species** (str) – Chemical species
- **T** (float) – Temperature [K]
- **DB** (struct) – Database with custom thermodynamic polynomials functions generated from NASAs 9 polynomials fits

Returns *s0 (float)* – Entropy [kJ/(mol-K)]

species_s0_NASA(*species*, *temperature*, *DB*)

Compute entropy [kJ/(mol-K)] of the species at the given temperature [K] using NASA's 9 polynomials

Parameters

- **species** (str) – Chemical species
- **temperature** (float) – Range of temperatures to evaluate [K]
- **DB** (struct) – Database with custom thermodynamic polynomials functions generated from NASAs 9 polynomials fits

Returns *s0 (float)* – Entropy [kJ/(mol-K)]

species_thermo_NASA(*species*, *temperature*, *DB*)

Compute thermodynamic function using NASA's 9 polynomials

Parameters

- **species** (str) – Chemical species
- **temperature** (float) – Range of temperatures to evaluate [K]
- **DB** (struct) – Database with custom thermodynamic polynomials functions generated from NASAs 9 polynomials fits

Returns

Tuple containing

- cP (float): Specific heat at constant pressure [J/(mol-K)]
- cV (float): Specific heat at constant volume [J/(mol-K)]
- h0 (float): Enthalpy [kJ/mol]
- DhT (float): Thermal enthalpy [kJ/mol]
- e0 (float): Internal energy [kJ/mol]
- DeT (float): Thermal internal energy [kJ/mol]
- s0 (float): Entropy [J/(mol-K)]
- g0 (float): Gibbs energy [kJ/mol]

`unpack_NASA_coefficients`(*species*, *DB*)

Unpack NASA's polynomials coefficients from database

Parameters

- **species** (`str`) – Chemical species
- **DB** (`struct`) – Database with custom thermodynamic polynomials functions generated from NASAs 9 polynomials fits

Returns

Tuple containing

- a (cell): Temperature coefficients
- b (cell): Integration constants
- tRange (cell): Ranges of temperatures [K]
- tExponents (cell): Exponent coefficients
- ctTInt (float): Number of intervals of temperatures
- txFormula (str): Chemical formula
- swtCondensed (float): 0 or 1 indicating gas or condensed phase, respectively

---

1. McBride, Bonnie J. NASA Glenn coefficients for calculating thermodynamic properties of individual species. National Aeronautics and Space Administration, John H. Glenn Research Center at Lewis Field, 2002.

7.6.2 Settings

Routines to initialize the main module, print results, plots, and external functions. We can distinguish four folders:

- self (main struct variable),
- general functions,
- display functions,
- external functions.

self

Routines to initialize the main module. All the data is save in a self variable (struct).

App

Routines to initialize the main module.

App(*varargin*)
Generate self variable with all the data required to initialize the computations

Parameters **empty** (none) – Generate default self variable assuming as products LS = Soot formation

Optional Args:

- LS (cell): List of species
- obj (class): Class combustion_toolbox_app (GUI)
- type (str): If value is fast initialize from the given Databases
- DB_master (struct): Master database
- DB (struct) : Database with custom thermodynamic polynomials functions generated from NASAs 9 polynomials fits

Returns *self (struct)* – Data of the mixture (initialization - empty), conditions, and databases

ContainedElements(*self*)
Obtain containted elements from the given set of species (reactants and products)

Parameters **self** (struct) – Data of the mixture, conditions, and databases

Returns *self (struct)* – Data of the mixture, conditions, and databases

Initialize(*self*)

This routine have three tasks:

- Check that all species are contained in the Database
- Establish cataloged list of species according to the state of the phase (gaseous or condensed). It also obtains the indices of cryogenic liquid species, e.g., liquified gases
- Compute Stoichiometric Matrix

Parameters **self** (struct) – Data of the mixture, conditions, and databases

Returns *self (struct)* – Data of the mixture, conditions, and databases

set_DB(*self*, *FLAG_REDUCED_DB*, *FLAG_FAST*)
Generate Database with custom polynomials from DB_master

Parameters

- **self** (struct) – Data of the mixture, conditions, and databases
- **FLAG_REDUCED_DB** (bool) – Flag compute from reduced database
- **FLAG_FAST** (bool) – Flag load databases

Returns *self (struct)* – Data of the mixture, conditions, and databases

Constants

Routines to initialize the Constants submodule in the self variable (struct).

`Constants()`
Initialize struct with constants data

Returns *self (struct)* – struct with constants data

Elements

Routines to initialize the Elements submodule in the self variable (struct).

`Elements()`
Initialize struct with elements data

Returns *self (struct)* – struct with elements data

`set_elements()`
Set cell with elements name

Returns

Tuple containing –

- elements (cell): Elements
- NE (struct): Number of elements

Miscellaneous

Routines to initialize the Miscellaneous submodule in the self variable (struct).

`Miscellaneous()`
Initialize struct with miscellaneous data

Returns *self (struct)* – struct with miscellaneous data

ProblemDescription

Routines to initialize the ProblemDescription submodule in the self variable (struct).

`ProblemDescription()`
Initialize struct with problem description data

Returns *self (struct)* – struct with problem description data

ProblemSolution

Routines to initialize the ProblemSolution submodule in the self variable (struct).

`ProblemSolution()`
Initialize struct with problem solution data

Returns *self (struct)* – struct with problem solution data

Species

Routines to initialize the Species submodule in the self variable (struct).

`ListSpecies(varargin)`
Set list of species in the mixture (reactants and products)

Predefined list of species:

- SOOT FORMATION (default)
- COMPLETE
- HC/O2/N2 EXTENDED
- SOOT FORMATION EXTENDED
- NASA ALL
- AIR, DISSOCIATED AIR
- AIR IONS, AIR_IONS
- IDEAL_AIR, AIR_IDEAL
- HYDROGEN
- HYDROGEN_L, HYDROGEN (L)
- HC/O2/N2 PROPELLANTS

Parameters `empty` (`none`) – return default list of species (soot formation)

Optional Args:

- self (struct): Data of the mixture, conditions, and databases
- LS (cell): Name list species / list of species
- EquivalenceRatio (float): Equivalence ratio
- EquivalenceRatio_soot (float): Equivalence ratio in which theoretically appears soot

Returns *self (struct)* – Data of the mixture, conditions, and databases

`Species()`
Initialize struct with problem solution data

Returns *self (struct)* – struct with problem solution data

`get_index_ions`(*species*)
Get index of ions for the given list of species

Parameters `species` (`str`) – List of species

Returns *index (float)* – Index of ions

TunningProperties

Routines to initialize the TunningProperties submodule in the self variable (struct).

`TunningProperties`()
Initialize struct with tunning properties attributes

`FLAG_FAST`
Flag indicating use guess composition of the previous computation (default: false)

Type bool

`tolN`
Tolerance of the Gibbs minimization method (default: 1e-15)

Type float

`tolE`
Tolerance of the mass balance (default: 1e-06)

Type float

`tol_pi_e`
Tolerance of the dimensionless Lagrangian multiplier - ions (default: 1e-04)

Type float

`tol0`
Tolerance of the root finding algorithm (default: 1e-03)

Type float

`root_method`
Method for root finding (default: newton)

Type function

`itMax`
Max number of iterations - root finding method - HP, EV, SP, SV (default: 30)

Type float

`root_T0_l`
First guess T [K] left branch - root finding method (default: 300)

Type float

`root_T0_r`
First guess T [K] right branch - root finding method (default: 1500)

Type float

`root_T0`
Guess T[K] if it's of previous range - root finding method (default: 2000)

Type float

`tol_shocks`
Tolerance of shocks routines (default: 5e-05)

Type float

`it_shocks`
Max number of iterations - shocks and detonations (default: 50)

Type float

`volume_ratio`
Initial guess volume ratio shocks (default: 5)

Type float

`tol_oblique`
Tolerance oblique shocks (default: 1e-03)

Type float

`it_oblique`
Max number of iterations - oblique shocks (default: 20)

Type float

`N_points_polar`
Number of points to compute shock polar (default: 100)

Type float

`tol_rocket`
Tolerance rocket performance (default: 1e-04)

Type float

`it_rocket`
Max number of iterations - rocket performance (default: 10)

Type float

Returns *self (struct)* – struct with tunning properties data

7.6.3 Solvers

Routines to solve different type of problems. We can distinguish four modules:

Functions

A collection of functions necessary to obtain different data in the solver module. Here we can find:

- general functions,
- root finding algorithms,
- thermodynamics properties.

General functions

A collection of general functions necessary to obtain different data in the solver module.

Routines

CalculatePhic(*Fuel*, *Ninerts*, *phi*, *TP*, *pP*, *DB*)

Compute equivalence ratio in which soot appears considering complete combustion

Parameters

- **Fuel** (struct) – Struct mix with all the properties of the Fuel mixture
- **Ninerts** (float) – Number of moles of the inerts species
- **phi** (float) – Equivalence ratio [-]
- **TP** (float) – Temperature [K]
- **pP** (float) – Pressure [bar]
- **DB** (struct) – Database

Returns *phi_c (float)* – Equivalence ratio in which soot appears [-]

ComputeProperties(*self*, *SpeciesMatrix*, *p*, *T*)

Compute properties from the given SpeciesMatrix at pressure p [bar] and temperature T [K]

Parameters

- **self** (struct) – Data of the mixture, conditions, and databases
- **SpeciesMatrix** (float) – Matrix with the properties values of the mixture
- **p** (float) – Pressure [bar]
- **T** (float) – Temperature [K]

Returns *mix (struct)* – Properties of the mixture

Compute_YFuel(*mix*, *mix_Fuel*)

Compute fuel mass fraction [-]

Parameters

- **mix** (struct) – Properties of the mixture (fuel + oxidizer + inerts)
- **mix_Fuel** (struct) – Properties of the mixture (fuel)

Returns *Yi_Fuel (float)* – Mass fractions of the fuel mixture

Compute_density(*mix*)

Get density of the set of mixtures

Parameters **mix** (struct) – Properties of the mixture/s

Returns *rho (float)* – Vector with the densities of all the mixtures

Compute_phi_c(*Fuel*)

Compute guess of equivalence ratio in which soot appears considering complete combustion

Parameters **Fuel** (struct) – Struct mix with all the properties of the Fuel mixture

Returns *phi_c (float)* – Equivalence ratio in which soot appears [-]

SetSpecies(*self*, *species*, *moles*, *T*)
Fill the properties matrix with the data of the mixture

Parameters

- **self** (struct) – Data of the mixture, conditions, and databases
- **species** (cell) – Species contained in the system
- **moles** (float) – Moles of the species in the mixture [mol]
- **T** (float) – Temperature [K]

Returns *M (float)* – properties matrix

Stoich_Matrix(*self*)
Initialize the stoichiometric matrix and properties matrix

Parameters **self** (struct) – Data of the mixture, conditions, and databases

Returns *self (struct)* – Data of the mixture, conditions, and databases

cell2vector(*value*, *varargin*)
Convert values of a individual cell into a vector. If the value correspond with a struct it can return as a vector the values of a given fieldname.

Parameters **value** (cell or struct) – Data of the mixture, conditions, and databases

Optional Args: field (str): Fieldname of the given value (struct)

Returns *vector (any)* – Vector with the values of the individual cell/fieldname (struct)

compute_first_derivative(*x*, *y*)
Compute first central derivate using a non-uniform grid

Parameters

- **x** (float) – Grid values
- **y** (float) – Values for the corresponding grid

Returns *dxdy (float)* – Value of the first derivate for the given grid and its corresponding values

compute_temperature_mixture(*self*, *species*, *moles*, *temperatures*)
Compute equilibrium temperature [K] of a gaseous mixture compound of n species with species at different temperatures.

Parameters

- **self** (struct) – Data of the mixture, conditions, and databases
- **species** (str) – Cell array with the species of the mixture
- **moles** – Moles of the species in the mixture [mol]
- **temperatures** – Vector or cell array with the temperatures of each species

Returns *T (float)* – Temperature of the mixture at equilibrium

find_ind(*LS*, *species*)
Find the index of the species based on the given list (LS)

Parameters

- **LS** (cell) – List of species
- **species** (cell) – Species to find index values

Returns *index (float)* – List with the index of the species based on the given list (LS)

`get_order`(*value*)
Get order of magnitude of a number in base 10

Parameters **value** (`float`) – number in base 10

Returns *order (float)* – order of magnitude of a number in base 10

`get_partial_derivative`(*self*, *mix*)
Get value of the partial derivative for the set problem type [kJ/K] (HP, EV) or [kJ/K^2] (SP, SV)

Parameters

- **self** (`struct`) – Data of the mixture, conditions, and databases
- **mix** (`struct`) – Properties of the mixture

Returns *value (float)* – Value of the partial derivative for the set problem type [kJ/K] (HP, EV) or [kJ/K^2] (SP, SV)

`get_transformation`(*self*, *field*)
Get the corresponding value of the field in Problem Description (PD)

Parameters

- **self** (`struct`) – Data of the mixture, conditions, and databases
- **field** (`str`) – Fieldname in Problem Description (PD)

Returns *value (float)* – Value/s assigned to the field

`list_phase_species`(*self*, *LS*)
Establish cataloged list of species according to the state of the phase (gaseous or condensed). It also obtains the indices of cryogenic liquid species, i.e., liquified gases.

Parameters

- **self** (`struct`) – Data of the mixture, conditions, and databases
- **LS** (`cell`) – List of species

Returns *self (struct)* – Data of the mixture, conditions, and databases

`set_prop`(*self*, *varargin*)
Assign property values to the respective variables

Parameters **self** (`struct`) – Data of the mixture, conditions, and databases

Optional Args:

- field (str): Fieldname in Problem Description (PD)
- value (float): Value/s to assing in the field in Problem Description (PD)

Returns *self (struct)* – Data of the mixture, conditions, and databases

`set_transformation`(*self*, *field*, *value*)
Set the corresponding value of the field in Problem Description (PD)

Parameters

- **self** (`struct`) – Data of the mixture, conditions, and databases
- **field** (`str`) – Fieldname in Problem Description (PD)
- **value** (`float`) – Value/s to assign to the field

Returns *self (struct)* – Data of the mixture, conditions, and databases

soundspeed_eq(*self*, *mix*, *phi*, *P0*, *T0*)
Compute speed of sound at equilibrium

Parameters

- **self** (struct) – Data of the mixture, conditions, and databases
- **mix** (struct) – Struct mix with all the properties of the mixture
- **phi** (float) – Equivalence ratio [-]
- **P0** (float) – Pressure [bar]
- **T0** (float) – Temperature [K]

Returns *sound (float)* – sound speed [m/s]

vector2cell(*value*)
Create cell array from vector

Parameters **value** (any) – Vector with data of any type

Returns *c (cell)* – Cell with the values of the vector

Root finding algorithms

Roots algorithm used to obtain the temperature at equilibrium for a given thermochemical transformation. The methods implemented are:

- Newton-Raphson method.
- Steffensen-Aitken method.

Routines

print_error_root(*it*, *itMax*, *T*, *STOP*)
Print error of the method if the number of iterations is greater than maximum iterations allowed

Parameters

- **it** (float) – Number of iterations executed in the method
- **itMax** (float) – Maximum nNumber of iterations allowed in the method
- **T** (float) – Temperature [K]
- **STOP** (float) – Relative error [-]

newton(*self*, *mix1*, *pP*, *field*, *x0*)
Find the temperature [K] (root) for the set chemical transformation at equilibrium using the Newton-Raphson method

Parameters

- **self** (struct) – Data of the mixture, conditions, and databases
- **mix1** (struct) – Properties of the initial mixture
- **pP** (float) – Pressure [bar]
- **field** (str) – Fieldname in Problem Description (PD)

- **x0** (float) – Guess temperature [K]

Returns

Tuple containing

- x (float): Temperature at equilibrium [K]
- STOP (float): Relative error [-]

get_gpoint(*self*, *mix1*, *pP*, *field*, *x0*)

Get fixed point of a function based on the chemical transformation

Parameters

- **self** (struct) – Data of the mixture, conditions, and databases
- **mix1** (struct) – Properties of the initial mixture
- **pP** (float) – Pressure [bar]
- **field** (str) – Fieldname in Problem Description (PD)
- **x0** (float) – Guess temperature [K]

Returns

Tuple containing

- gpoint (float): Fixed point of the function [kJ] (HP, EV) or [kJ/K] (SP, SV)
- gpoint_relative (float): Fixed relative point of the function [kJ] (HP, EV) or [kJ/K] (SP, SV)

get_point(*x_vector*, *g_vector*)

Get point of the fixed point function

Parameters

- **x_vector** (float) – Guess temperature [K]
- **g_vector** (struct) – Fixed points of the function [kJ] (HP, EV) or [kJ/K] (SP, SV)

Returns *point (float)* – Point of the function [K]

get_point_aitken(*x0*, *g_vector*)

Get fixed point of a function based on the chemical transformation using the Aitken acceleration method

Parameters

- **x0** (float) – Guess temperature [K]
- **g_vector** (struct) – Fixed points of the function [kJ] (HP, EV) or [kJ/K] (SP, SV)

Returns *point (float)* – Point of the function [K]

steff(*self*, *mix1*, *pP*, *field*, *x0*)

Find the temperature [K] (root) for the set chemical transformation at equilibrium using the Steffenson-Aitken method

Parameters

- **self** (struct) – Data of the mixture, conditions, and databases
- **mix1** (struct) – Properties of the initial mixture
- **pP** (float) – Pressure [bar]
- **field** (str) – Fieldname in Problem Description (PD)
- **x0** (float) – Guess temperature [K]

Returns

Tuple containing

- x (float): Temperature at equilibrium [K]
- STOP (float): Relative error [-]

`steff_guess`(*self*, *mix1*, *pP*, *field*)
Find a estimate of the temperature for the set chemical equilibrium transformation

Parameters

- **self** (struct) – Data of the mixture, conditions, and databases
- **mix1** (struct) – Properties of the initial mixture
- **pP** (float) – Pressure [bar]
- **field** (str) – Fieldname in Problem Description (PD)

Returns *x0 (float)* – Guess temperature [K]

Thermodynamic properties

Functions to obtain thermodynamic properties from a given mixture.

Routines

`adiabaticIndex`(*mix*)
Get the adiabatic index [-] of the mixture from the ratio of the specific heat capacities

Parameters **mix** (struct) – Properties of the mixture

Returns *value (float)* – Adiabatic index of the mixture [-]

`adiabaticIndex_sound`(*mix*)
Get the adiabatic index [-] of the mixture from definition of sound velocity

Parameters **mix** (struct) – Properties of the mixture

Returns *value (float)* – Adiabatic index [-] of the mixture

`cp_mass`(*mix*)
Get the mass-basis specific heat at constant pressure [kJ/kg-K] of the mixture

Parameters **mix** (struct) – Properties of the mixture

Returns *value (float)* – Mass-basis specific heat at constant pressure [kJ/kg-K] of the mixture

`cp_mole`(*mix*)
Get the mole-basis specific heat at constant pressure [kJ/mol-K] of the mixture

Parameters **mix** (struct) – Properties of the mixture

Returns *value (float)* – Mole-basis specific heat at constant pressure [kJ/mol-K] of the mixture

`cv_mass`(*mix*)
Get the mass-basis specific heat at constant volume [kJ/kg-K] of the mixture

Parameters **mix** (struct) – Properties of the mixture

Returns *value (float)* – Mass-basis specific heat at constant volume [kJ/kg-K] of the mixture

`cv_mole`(*mix*)
Get the mole-basis specific heat at constant volume [kJ/mol-K] of the mixture

Parameters **mix** (`struct`) – Properties of the mixture

Returns *value (float)* – Mole-basis specific heat at constant volume [kJ/mol-K] of the mixture

`density`(*mix*)
Get the density [kg/m3] of the mixture

Parameters **mix** (`struct`) – Properties of the mixture

Returns *value (float)* – density [kg/m3] of the mixture

`enthalpy_mass`(*mix*)
Get the mass specific enthalpy [kJ/kg] of the mixture

Parameters **mix** (`struct`) – Properties of the mixture

Returns *value (float)* – Mass-basis specific enthalpy [kJ/kg] of the mixture

`enthalpy_mole`(*mix*)
Get the mole specific enthalpy [kJ/mol] of the mixture

Parameters **mix** (`struct`) – Properties of the mixture

Returns *value (float)* – Mole-basis specific enthalpy [kJ/mol] of the mixture

`entropy_mass`(*mix*)
Get the mass specific entropy [kJ/kg-K] of the mixture

Parameters **mix** (`struct`) – Properties of the mixture

Returns *value (float)* – Mass-basis specific entropy [kJ/kg-K] of the mixture

`entropy_mole`(*mix*)
Get the mole specific entropy [kJ/mol-K] of the mixture

Parameters **mix** (`struct`) – Properties of the mixture

Returns *value (float)* – Mole-basis specific entropy [kJ/mol-K] of the mixture

`equivalenceRatio`(*mix*)
Get the equivalence ratio of the initial mixture [-]

Parameters **mix** (`struct`) – Properties of the mixture

Returns *value (float)* – Equivalence ratio of the initial mixture [-]

`gibbs_mass`(*mix*)
Get the mass specific gibbs free energy [kJ/kg] of the mixture

Parameters **mix** (`struct`) – Properties of the mixture

Returns *value (float)* – Mass-basis specific gibbs free energy [kJ/kg] of the mixture

`gibbs_mole`(*mix*)
Get the mole specific gibbs free energy [kJ/mol] of the mixture

Parameters **mix** (`struct`) – Properties of the mixture

Returns *value (float)* – Mole-basis specific gibbs free energy [kJ/mol] of the mixture

`intEnergy_mass`(*mix*)
Get the mass specific internal energy [kJ/kg] of the mixture

Parameters **mix** (struct) – Properties of the mixture

Returns *value (float)* – Mass-basis specific internal energy [kJ/kg] of the mixture

intEnergy_mole(*mix*)
Get the mole specific internal energy [kJ/mol] of the mixture

Parameters **mix** (struct) – Properties of the mixture

Returns *value (float)* – Mole-basis specific internal energy [kJ/mol] of the mixture

massFractions(*mix*)
Get the mass fractions of all the species in the mixture [-]

Parameters **mix** (struct) – Properties of the mixture

Returns *value (float)* – Mass fractions of all the species in the mixture [-]

meanMolecularWeight(*mix*)
Get the mean molecular weight [g/mol] of the mixture

Parameters **mix** (struct) – Properties of the mixture

Returns *value (float)* – Mean molecular weight [g/mol] of the mixture

moleFractions(*mix*)
Get the mole fractions of all the species in the mixture [-]

Parameters **mix** (struct) – Properties of the mixture

Returns *value (float)* – Mole fractions of all the species in the mixture [-]

moles(*mix*)
Get the moles [mol] of all the species in the mixture

Parameters **mix** (struct) – Properties of the mixture

Returns *value (float)* – Moles [mol] of all the species in the mixture

pressure(*mix*)
Get the pressure [bar] in the mixture

Parameters **mix** (struct) – Properties of the mixture

Returns *value (float)* – Pressure [bar] in the mixture

soundspeed(*mix*)
Get the speed of sound [m/s] in the mixture

Parameters **mix** (struct) – Properties of the mixture

Returns *value (float)* – Speed of sound [m/s] in the mixture

temperature(*mix*)
Get the temperature [K] in the mixture

Parameters **mix** (struct) – Properties of the mixture

Returns *value (float)* – Temperature [K] in the mixture

velocity_relative(*mix*)
Get the velocity of the gases relative to the shock front [m/s] in the mixture

Parameters **mix** (struct) – Properties of the mixture

Returns *value (float)* – Velocity of the gases relative to the shock front [m/s] in the mixture

Chemical equilibrium

In this section, you will find the documentation of the kernel of the code, which is used to obtain the chemical equilibrium composition for a desired thermochemical transformation, e.g., constant enthalpy and pressure. It also includes routines to compute chemical equilibrium assuming a complete combustion and the calculation of the thermodynamic derivates. The code stems from the minimization of the free energy of the system by using Lagrange multipliers combined with a Newton-Raphson method, upon condition that initial gas properties are defined by two functions of states, e.g., temperature and pressure.

Note: The kernel of the code is based on Gordon, S., & McBride, B. J. (1994). NASA reference publication, 1311.

Thermodynamic derivatives

All thermodynamic first derivatives can be obtained with any set of three independent first derivatives [1]. Combustion Toolbox computes all thermodynamic first derivatives from $(\partial \ln V/\partial \ln T)_p$, $(\partial \ln V/\partial \ln p)_T$, and $(\partial h/\partial T)_p = c_p$. Considering the ideal equation of state

$$pV = nRT \tag{7.13}$$

and applying logarithms to both sides

$$\ln V = n + \ln R + \ln T - \ln p \tag{7.14}$$

is readily seen that

$$\left(\frac{\partial \ln V}{\partial \ln T}\right)_p = 1 + \left(\frac{\partial \ln n}{\partial \ln T}\right)_p, \tag{7.15}$$

$$\left(\frac{\partial \ln V}{\partial \ln p}\right)_T = -1 + \left(\frac{\partial \ln n}{\partial \ln p}\right)_T. \tag{7.16}$$

To compute c_p we have to distinguish between the frozen contribution and the reaction contribution

$$c_p = c_{p,f} + c_{p,r} \tag{7.17}$$

given by the following relations

$$c_{p,f} = \sum_{j=1}^{\mathrm{NS}} n_j c_{p,f}^{\circ}, \tag{7.18}$$

$$c_{p,r} = \frac{1}{T}\left[\sum_{j=1}^{\mathrm{NS}} [1 + \delta_j(n_j - 1)] h_j^{\circ} \left(\frac{\partial \eta_j}{\partial \ln T}\right)\right], \tag{7.19}$$

with $\eta_j = \ln n_j$ and $\delta_j = 1$ for $j = 1, \ldots, NG$ (non-condensed species), and $\eta_j = n_j$ and $\delta_j = 0$ for $j = NG + 1, \ldots, NS$ (condensed species).

Derivatives with respect to temperature

$$\delta_j \left(\frac{\partial \eta_j}{\partial \ln T}\right)_p - \sum_{i=1}^{\mathrm{NE}} a_{ij} \left(\frac{\partial \pi_i}{\partial \ln T}\right)_p - \delta_j \left(\frac{\partial \ln n}{\partial \ln T}\right)_p = \frac{h_j^{\circ}}{RT}, \qquad \text{for } j = 1, \ldots, \mathrm{NS}$$

$$\sum_{j=1}^{\mathrm{NS}} a_{ij}[1 + \delta_j(n_j - 1)] \left(\frac{\partial \eta_j}{\partial \ln T}\right)_p = 0, \quad \text{for } i = 1, \ldots, \mathrm{NE}$$

$$\sum_{j=1}^{\mathrm{NG}} n_j \left(\frac{\partial \eta_j}{\partial \ln T}\right)_p - n \left(\frac{\partial \ln n}{\partial \ln T}\right)_p = 0,$$

Derivatives with respect to pressure

$$\delta_j \left(\frac{\partial \eta_j}{\partial \ln p}\right)_T - \sum_{i=1}^{\text{NE}} a_{ij} \left(\frac{\partial \pi_i}{\partial \ln p}\right)_T - \delta_j \left(\frac{\partial \ln n}{\partial \ln p}\right)_T = -\delta, \qquad \text{for } j = 1, \ldots, \text{NS}$$

$$\sum_{j=1}^{\text{NS}} a_{ij}[1 + \delta_j(n_j - 1)] \left(\frac{\partial \eta_j}{\partial \ln p}\right)_T = 0, \quad \text{for } i = 1, \ldots, \text{NE}$$

$$\sum_{j=1}^{\text{NG}} n_j \left(\frac{\partial \eta_j}{\partial \ln p}\right)_T - n \left(\frac{\partial \ln n}{\partial \ln p}\right)_T = 0.$$

Routines

complete_combustion(*self*, *mix*, *phi*)
Solve chemical equilibrium for CHNO mixtures assuming a complete combustion

Parameters

- **self** (struct) – Data of the mixture, conditions, and databases
- **mix** (struct) – Properties of the initial mixture
- **phi** (float) – Equivalence ratio [-]

Returns

Tuple containing

- moles (float): Equilibrium composition [moles] at defined temperature
- species (str): Species considered in the complemte combustion model

equilibrate(*self*, *mix1*, *pP*, *varargin*)
Obtain properties at equilibrium for the set thermochemical transformation

Parameters

- **self** (struct) – Data of the mixture, conditions, and databases
- **mix1** (struct) – Properties of the initial mixture
- **pP** (float) – Pressure [bar]

Optional Args: mix2 (struct): Properties of the final mixture (previous calculation)

Returns *mix2 (struct)* – Properties of the final mixture

equilibrate_T(*self*, *mix1*, *pP*, *TP*, *varargin*)
Obtain equilibrium properties and composition for the given temperature [K] and pressure [bar]

Parameters

- **self** (struct) – Data of the mixture, conditions, and databases
- **mix1** (struct) – Properties of the initial mixture
- **pP** (float) – Pressure [bar]

- **TP** (float) – Temperature [K]

Optional Args: guess_moles (float): mixture composition [mol] of a previous computation

Returns *mix2 (struct)* – Properties of the final mixture

equilibrium(*self*, *pP*, *TP*, *mix1*, *guess_moles*)

Obtain equilibrium composition [moles] for the given temperature [K] and pressure [bar]. The code stems from the minimization of the free energy of the system by using Lagrange multipliers combined with a Newton-Raphson method, upon condition that initial gas properties are defined by two functions of states. e.g., temperature and pressure.

This method is based on Gordon, S., & McBride, B. J. (1994). NASA reference publication, 1311.

Parameters

- **self** (struct) – Data of the mixture, conditions, and databases
- **pP** (float) – Pressure [bar]
- **TP** (float) – Temperature [K]
- **mix1** (struct) – Properties of the initial mixture
- **guess_moles** (float) – mixture composition [mol] of a previous computation

Returns

Tuple containing

- N0 (float): Equilibrium composition [moles] for the given temperature [K] and pressure [bar]
- STOP (float): Relative error [-]

equilibrium_dT(*self*, *moles*, *T*, *mix1*)

Obtain thermodynamic derivative of the moles of the species and of the moles of the mixture respect to temperature from a given composition [moles] at equilibrium

Parameters

- **self** (struct) – Data of the mixture, conditions, and databases
- **moles** (float) – Equilibrium composition [moles]
- **T** (float) – Temperature [K]
- **mix1** (struct) – Properties of the initial mixture

Returns

Tuple containing

- dNi_T (float): Thermodynamic derivative of the moles of the species respect to temperature
- dN_T (float): Thermodynamic derivative of the moles of the mixture respect to temperature

equilibrium_dp(*self*, *moles*, *mix1*)

Obtain thermodynamic derivative of the moles of the species and of the moles of the mixture respect to pressure from a given composition [moles] at equilibrium

Parameters

- **self** (struct) – Data of the mixture, conditions, and databases
- **moles** (float) – Equilibrium composition [moles]
- **mix1** (struct) – Properties of the initial mixture

Returns

Tuple containing

- dNi_p (float): Thermodynamic derivative of the moles of the species respect to pressure
- dN_p (float): Thermodynamic derivative of the moles of the mixture respect to pressure

equilibrium_ions(*self*, *pP*, *TP*, *mix1*, *guess_moles*)

Obtain equilibrium composition [moles] for the given temperature [K] and pressure [bar] considering ionization of the species. The code stems from the minimization of the free energy of the system by using Lagrange multipliers combined with a Newton-Raphson method, upon condition that initial gas properties are defined by two functions of states. e.g., temperature and pressure.

This method is based on Gordon, S., & McBride, B. J. (1994). NASA reference publication, 1311.

Parameters

- **self** (struct) – Data of the mixture, conditions, and databases
- **pP** (float) – Pressure [bar]
- **TP** (float) – Temperature [K]
- **mix1** (struct) – Properties of the initial mixture
- **guess_moles** (float) – mixture composition [mol] of a previous computation

Returns

Tuple containing

- N0 (float): Equilibrium composition [moles] for the given temperature [K] and pressure [bar]
- STOP (float): Relative error [-]

equilibrium_reduced(*self*, *pP*, *TP*, *mix1*, *guess_moles*)

Obtain equilibrium composition [moles] for the given temperature [K] and pressure [bar]. The code stems from the minimization of the free energy of the system by using Lagrange multipliers combined with a Newton-Raphson method, upon condition that initial gas properties are defined by two functions of states. e.g., temperature and pressure. The algorithm implemented take advantage of the sparseness of the upper left submatrix obtaining a matrix A of size NE + NS-NG + 1.

This method is based on Gordon, S., & McBride, B. J. (1994). NASA reference publication, 1311.

Parameters

- **self** (struct) – Data of the mixture, conditions, and databases
- **pP** (float) – Pressure [bar]
- **TP** (float) – Temperature [K]
- **mix1** (struct) – Properties of the initial mixture
- **guess_moles** (float) – mixture composition [mol] of a previous computation

Returns

Tuple containing

- N0 (float): Equilibrium composition [moles] for the given temperature [K] and pressure [bar]
- STOP (float): Relative error [-]

1. McBride, Bonnie J. Computer program for calculation of complex chemical equilibrium compositions and applications. Vol. 2. NASA Lewis Research Center, 1996.

Shocks and detonations waves

In this section, you will find the documentation of the routines implemented to obtain the pre-shock and post-shock states of a set of fluid dynamic problems related with the shock waves and detonations waves.

Note: The kernel of the incident, reflected, and Chapman-Jouguet detonations are based on Gordon, S., & McBride, B. J. (1994). NASA reference publication, 1311.

Routines

`cj_detonation`(*self*, *mix1*, *varargin*)

Compute pre-shock and post-shock states of a Chapman-Jouguet detonation

This method is based on Gordon, S., & McBride, B. J. (1994). NASA reference publication, 1311.

Parameters

- **self** (`struct`) – Data of the mixture, conditions, and databases
- **mix1** (`struct`) – Properties of the mixture in the pre-shock state

Optional Args: mix2 (struct): Properties of the mixture in the post-shock state (previous calculation)

Returns

Tuple containing

- mix1 (struct): Properties of the mixture in the pre-shock state
- mix2 (struct): Properties of the mixture in the post-shock state

`compute_guess_det`(*self*, *mix1*, *phi*, *overdriven*)

Obtain guess of the jump conditions for a Chapman-Jouguet detonation. Only valid if the mixture have CHON. It computes the guess assuming first a complete combustion, next it recomputes assuming an incomplete combustion from the composition obtained in the previous step.

Parameters

- **self** (`struct`) – Data of the mixture, conditions, and databases
- **mix1** (`struct`) – Properties of the mixture in the pre-shock state
- **phi** (`float`) – Equivalence ratio [-]
- **overdriven** (`float`) – Overdriven ratio [-] respect to the sound velocity of the mixture

Returns

Tuple containing

- P (float): Pressure ratio [-]
- T (float): Temperature ratio [-]
- M1 (float): Pre-shock Mach number [-]
- R (float): Density ratio [-]
- Q (float): Dimensionless Heat release []

- STOP (float): Relative error [-]

det_oblique_beta(*self*, *mix1*, *overdriven*, *beta*, *varargin*)

Compute pre-shock and post-shock states of an oblique detonation wave given the wave angle (one solution)

Parameters

- **self** (struct) – Data of the mixture, conditions, and databases
- **mix1** (struct) – Properties of the mixture in the pre-shock state
- **overdriven** (float) – Overdriven factor [-]
- **beta** (float) – Wave angle [deg] of the incident oblique detonation

Optional Args: mix2 (struct): Properties of the mixture in the post-shock state (previous calculation)

Returns

Tuple containing

- mix1 (struct): Properties of the mixture in the pre-shock state
- mix2 (struct): Properties of the mixture at the post-shock state

det_oblique_theta(*self*, *mix1*, *overdriven*, *theta*, *varargin*)

Compute pre-shock and post-shock states of an oblique detonation wave given the deflection angle.

Two solutions:

- Weak detonation
- Strong detonation

Parameters

- **self** (struct) – Data of the mixture, conditions, and databases
- **mix1** (struct) – Properties of the mixture in the pre-shock state
- **overdriven** (float) – Overdriven factor [-]
- **theta** (float) – Deflection angle [deg]

Optional Args:

- mix2_1 (struct): Properties of the mixture in the post-shock state - weak detonation (previous calculation)
- mix2_2 (struct): Properties of the mixture in the post-shock state - strong detonation (previous calculation)

Returns

Tuple containing

- mix1 (struct): Properties of the mixture in the pre-shock state
- mix2_1 (struct): Properties of the mixture in the post-shock state - weak detonation
- mix2_2 (struct): Properties of the mixture in the post-shock state - strong detonation

overdriven_detonation(*self*, *mix1*, *overdriven*, *varargin*)

Compute pre-shock and post-shock states of an overdriven planar detonation

Parameters

- **self** (struct) – Data of the mixture, conditions, and databases
- **mix1** (struct) – Properties of the mixture in the pre-shock state
- **overdriven** (float) – Overdriven factor [-]

Optional Args: mix2 (struct): Properties of the mixture in the post-shock state (previous calculation)

Returns

Tuple containing

- mix1 (struct): Properties of the mixture in the pre-shock state
- mix2 (struct): Properties of the mixture in the post-shock state

shock_ideal_gas(*gamma*, *M1*)

Compute jump conditions assuming a thermochemically frozen gas

Parameters

- **gamma** (float) – Adiabatic index [-]
- **M1** (float) – Pre-shock Mach number [-]

Returns

Tuple containing

- R (float): Density ratio [-]
- P (float): Pressure ratio [-]
- T (float): Temperature ratio [-]

shock_incident(*self*, *mix1*, *u1*, *varargin*)

Compute pre-shock and post-shock states of a planar incident shock wave

This method is based on Gordon, S., & McBride, B. J. (1994). NASA reference publication, 1311.

Parameters

- **self** (struct) – Data of the mixture, conditions, and databases
- **mix1** (struct) – Properties of the mixture in the pre-shock state
- **u1** (float) – Pre-shock velocity [m/s]

Optional Args: mix2 (struct): Properties of the mixture in the post-shock state (previous calculation)

Returns

Tuple containing

- mix1 (struct): Properties of the mixture in the pre-shock state
- mix2 (struct): Properties of the mixture in the post-shock state

shock_oblique_beta(*self*, *mix1*, *u1*, *beta*, *varargin*)

Compute pre-shock and post-shock states of an oblique shock wave given the wave angle (one solution)

Parameters

- **self** (struct) – Data of the mixture, conditions, and databases

- **mix1** (struct) – Properties of the mixture in the pre-shock state
- **u1** (float) – Pre-shock velocity [m/s]
- **beta** (float) – Wave angle [deg] of the incident oblique shock

Optional Args: mix2 (struct): Properties of the mixture in the post-shock state (previous calculation)

Returns

Tuple containing

- mix1 (struct): Properties of the mixture in the pre-shock state
- mix2 (struct): Properties of the mixture at the post-shock state

shock_oblique_reflected_theta(*self*, *mix1*, *u2*, *theta*, *mix2*, *varargin*)

Compute pre-shock and post-shock states of an oblique reflected shock wave given the deflection angle.

Two solutions:

- Weak shock
- Strong shock

Parameters

- **self** (struct) – Data of the mixture, conditions, and databases
- **mix1** (struct) – Properties of the mixture in the pre-shock state of the incident shock
- **u2** (float) – Post-shock velocity [m/s] of the incident shock
- **theta** (float) – Deflection angle [deg]
- **mix2** (struct) – Properties of the mixture in the post-shock state of the incident shock

Optional Args: mix5_1 (struct): Properties of the mixture in the post-shock state of the reflected shock - weak shock (previous calculation) mix5_2 (struct): Properties of the mixture in the post-shock state of the reflected shock - strong shock (previous calculation)

Returns

Tuple containing

- mix1 (struct): Properties of the mixture in the pre-shock state of the incident shock
- mix2 (struct): Properties of the mixture in the post-shock state of the incident shock
- mix5_1 (struct): Properties of the mixture in the post-shock state of the reflected shock - weak shock
- mix5_2 (struct): Properties of the mixture in the post-shock state of the reflected shock - strong shock

shock_oblique_theta(*self*, *mix1*, *u1*, *theta*, *varargin*)

Compute pre-shock and post-shock states of an oblique shock wave given the deflection angle.

Two solutions:

- Weak shock
- Strong shock

Parameters

- **self** (struct) – Data of the mixture, conditions, and databases
- **mix1** (struct) – Properties of the mixture in the pre-shock state
- **u1** (float) – Pre-shock velocity [m/s]
- **theta** (float) – Deflection angle [deg]

Optional Args:

- mix2_1 (struct): Properties of the mixture in the post-shock state - weak shock (previous calculation)
- mix2_2 (struct): Properties of the mixture in the post-shock state - strong shock (previous calculation)

Returns

Tuple containing

- mix1 (struct): Properties of the mixture in the pre-shock state
- mix2_1 (struct): Properties of the mixture in the post-shock state - weak shock
- mix2_2 (struct): Properties of the mixture in the post-shock state - strong shock

shock_polar(*self*, *mix1*, *u1*, *varargin*)

Compute shock polars of an oblique shock wave

Parameters

- **self** (struct) – Data of the mixture, conditions, and databases
- **mix1** (struct) – Properties of the mixture in the pre-shock state
- **u1** (float) – Pre-shock velocity [m/s]

Optional Args: mix2 (struct): Properties of the mixture in the post-shock state (previous calculation)

Returns

Tuple containing

- mix1 (struct): Properties of the mixture in the pre-shock state
- mix2 (struct): Properties of the mixture at the post-shock state with the shock polar results

shock_reflected(*self*, *mix1*, *u1*, *mix2*, *varargin*)

Compute pre-shock and post-shock states of a planar reflected shock wave

This method is based on Gordon, S., & McBride, B. J. (1994). NASA reference publication, 1311.

Parameters

- **self** (struct) – Data of the mixture, conditions, and databases
- **mix1** (struct) – Properties of the mixture in the pre-shock state of the incident shock
- **u1** (float) – Pre-shock velocity [m/s]
- **mix2** (struct) – Properties of the mixture at the post-shock state of the incident shock

Optional Args: mix5 (struct): Properties of the mixture in the post-shock state of the reflected shock (previous calculation)

Returns

Tuple containing

- mix1 (struct): Properties of the mixture in the pre-shock state of the incident shock
- mix2 (struct): Properties of the mixture at the post-shock state of the incident shock
- mix5 (struct): Properties of the mixture in the post-shock state of the reflected shock

Rocket propellant performance

In this section, you will find the documentation of the routines implemented to obtain the rocket propellant performance. There are two models:

- IAC: Infinite-Area-Chamber,
- FAC: Finite-Area-Chamber (under development).

Note: This module is based on Gordon, S., & McBride, B. J. (1994). NASA reference publication, 1311.

Warning: This module is under development.

Routines

compute_IAC_model(*self*, *mix2*, *mix3*)

Compute thermochemical composition for the Infinite-Area-Chamber (IAC) model

This method is based on Gordon, S., & McBride, B. J. (1994). NASA reference publication, 1311.

Parameters

- **self** (struct) – Data of the mixture, conditions, and databases
- **mix2** (struct) – Properties of the mixture at the outlet of the chamber
- **mix3** (struct) – Properties of the mixture at the throat (previous calculation)

Returns *mix3 (struct)* – Properties of the mixture at the throat

compute_chemical_equilibria(*self*, *mix1*, *pP*, *mix2*)

Compute chemical equilibria for the 2 given thermodynamic states, e.g., enthalpy-pressure (HP)

Parameters

- **self** (struct) – Data of the mixture, conditions, and databases
- **mix1** (struct) – Properties of the initial mixture
- **pP** (float) – Pressure [bar]
- **mix2** (struct) – Properties of the final mixture (previous calculation)

Returns *mix2 (struct)* – Properties of the final mixture

compute_rocket_parameters(*mix2*, *mix3*)
Compute Rocket performance parameters at the throat

This method is based on Gordon, S., & McBride, B. J. (1994). NASA reference publication, 1311.

Parameters

- **mix2** (struct) – Properties of the mixture at the outlet of the chamber
- **mix3** (struct) – Properties of the mixture at the throat

Returns *mix3 (struct)* – Properties of the mixture at the throat

guess_pressure_IAC_model(*mix*)
Compute pressure guess [bar] at the throat considering an Infinite-Area-Chamber (IAC)

This method is based on Gordon, S., & McBride, B. J. (1994). NASA reference publication, 1311.

Parameters **mix** (struct) – Properties of the mixture

Returns *pressure (float)* – Pressure [bar]

rocket_performance(*self*, *mix1*, *varargin*)
Routine that computes the propellant rocket performance

Methods implemented:

- Infinite-Area-Chamber (IAC)
- Finite-Area-Chamber (FAC) - NOT YET

This method is based on Gordon, S., & McBride, B. J. (1994). NASA reference publication, 1311.

Parameters

- **self** (struct) – Data of the mixture, conditions, and databases
- **mix1** (struct) – Properties of the initial mixture

Optional Args:

- mix2 (struct): Properties of the mixture at the outlet of the chamber (previous calculation)
- mix3 (struct): Properties of the mixture at the throat (previous calculation)

Returns

Tuple containing

- mix1 (struct): Properties of the initial mixture
- mix2 (struct): Properties of the mixture at the outlet of the chamber
- mix3 (struct): Properties of the mixture at the throat

Routines

SolveProblem(*self*, *ProblemType*)
Solve the given ProblemType with the conditions and mixture specified in self

Parameters

- **self** (struct) – Data of the mixture, conditions, and databases
- **ProblemType** (str) – Tag of the problem to solve

Returns *self (struct)* – Data of the mixtures (initial and final), conditions, databases

MATLAB MODULE INDEX

S

INDEX

G

I

L

M

N

O

P

R

S

T

U

V