Combustion Toolbox

Release v1.0.5

Alberto Cuadra Lara

Apr 09, 2024

CONTENTS

CHAPTER

ONE

COPYRIGHT, TRADEMARKS, AND ATTRIBUTIONS

Note: This document is in a state of continuous development and you are examining a snapshot in time.

1.1 Attributions

- Alberto Cuadra Lara - Lead developer
- César Huete - Advisor
- Marcos Vera - Advisor

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

Other contributors: Samuel Delbarre and Carlos Aguilar Borasteros.

1.2 HTML Version and Source Code

An HTML version of this manual is freely available via https://combustion-toolbox-website.readthedocs.io/. The source code for the examples used in this manual are available within Combustion Toolbox distribution, which is available via https://github.com/AlbertoCuadra/combustion_toolbox. The online documentation is hosted on Read the Docs and is regularly updated from its GitHub repository. All function headers are written following Google's Python-style docstrings. New routines are automatically included in the online documentation using GitHub Actions.

CHAPTER

TWO

PUBLICATIONS

2.1 Citing Combustion Toolbox

Note: The scientific paper on the Combustion Toolbox is currently under review. In the meantime, you can cite Cuadra [2023] (see Chapter 2 and Appendices A-B).

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

- *Cuadra, A., Huete, C., & Vera, M. (2024). Combustion Toolbox: A MATLAB-GUI based open-source tool for solving gaseous combustion problems. Version 1.0.5. Zenodo. doi:10.5281/zenodo.5554911.*
- *Cuadra, A. (2023). Development of a wide-spectrum thermochemical code with application to planar reacting and non-reacting shocks. PhD thesis, Universidad Carlos III de Madrid. Available at http://hdl.handle.net/10016/38179.*

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 gaseous combustion problems}",
    year    = 2024,
    note    = "Version 1.0.5",
    doi     = {https://doi.org/10.5281/zenodo.5554911}
}

@phdthesis{cuadra2023_thesis,
    title   = {Development of a wide-spectrum thermochemical code with␣
↪application to planar reacting and non-reacting shocks},
```

(continues on next page)

(continued from previous page)

```
    author  = {Cuadra, A.},
    year    = 2023,
    month   = {May},
    address = {Madrid, Spain},
    note    = {Available at \url{http://hdl.handle.net/10016/38179}},
    school  = {Universidad Carlos III de Madrid},
    type    = {PhD thesis}
}
```

2.2 Contributions

Here are some contributions, work-in-progress articles, and publications where the Combustion Toolbox has been used, as well as other codes that utilize this toolbox.

2.2.1 Journal articles

- Chang, X., Hongting, M., & Xiaohui, Y. (2024). Comprehensive performance evaluation method of working fluids for high temperature heat pump based on multi-objective optimization. Applied Thermal Engineering, 123102. doi:10.1016/j.applthermaleng.2024.123102.
- Chen, J., Sun, M., Li, P., An, B., Jiaoru, W., & Li, M. (2024). Effects of excess oxidizer coefficient on RBCC engine performance in ejector mode: A theoretical investigation. Energy, 289, 130070. doi:10.1016/j.energy.2023.130070.
- Cuadra, A., Huete, C., & Vera, M. (2023). Combustion Toolbox: An open-source thermochemical code for gas- and condensed-phase problems involving chemical equilibrium **(under review)**.
- Dahake, A., Singh, R. K., & Singh, A. V. (2023). Dual behavior of hydrogen peroxide in gaseous detonations. Shock Waves, 33(5), 401-414. doi:10.1007/s00193-023-01142-5.
- Sánchez, J., Cuadra, A., Huete, C., Vera, M. (2022). SimEx: A tool for rapid evaluation of the effects of explosions. Applied Sciences, 12(18), 9191. doi:10.3390/app12189101.
- 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)**. doi: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. doi:10.1017/jfm.2020.651.

2.2.2 Other codes that use the Combustion Toolbox

- Sánchez, J., Cuadra, A., Huete, C., Vera, M. (2022). SimEx: A tool for rapid evaluation of the effects of explosions. Applied Sciences, 12(18), 9191. doi:10.3390/app12189101.

2.2.3 Conference contributions

- Cuadra, A., Vera, M., Di Renzo, M. & Huete, C. (2023). Linear Theory of Hypersonic Shocks Interacting with Turbulence in Air. In 2023 AIAA SciTech Forum, National Harbor, USA. doi:10.2514/6.2023-0075.
- Cuadra, A., Huete, C., & Vera, M. (2022). Desarrollo de un código termoquímico para la evaluación de las propiedades teóricas de explosivos (CT-EXPLO) y la estimación del rendimiento de motores cohete (CT-ROCKET). In IX Congreso Nacional de I+D en Defensa y Seguridad, Pontevedra, Spain.
- Cuadra, A., Huete, C., & Vera, M. (2022). Amplificación de la turbulencia a través de una onda de choque en régimen hipersónico. In IX Congreso Nacional de I+D en Defensa y Seguridad, Pontevedra, Spain.
- Cuadra, A., Huete, C., & Vera, M. (2022). Combustion Toolbox: a MATLAB-GUI based open-source tool for solving combustion problems. In 12th National and 3rd International Conference on Engineering Thermodynamics (CNIT), Madrid, Spain.
- Cuadra, A., Huete, C., & Vera, M. (2022). Theory of turbulence augmentation across hypersonic shock waves in air. In 1st Spanish Fluid Mechanics Conference (SFMC), Cádiz, Spain.
- 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, USA.
- 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.

2.2.4 Seminars & Workshops

- Cuadra, A., C. T. Williams, Di Renzo, M., Vera, M., & Huete, C. (2023). Direct numerical simulations and linear analysis for hypersonic shock-turbulence interaction in air. In 4th Spanish HPC Combustion Workshop, Barcelona, Spain.
- Cuadra, A., Huete, C. & Vera, M. (2023). Linear analysis on shock-turbulence interaction implemented with the Combustion Toolbox. Seminar presented during the research-stay with Prof. M. Di Renzo, Lecce, Italy
- 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

2.2.5 PhD, MSc & BSc thesis

- Cuadra, A. (2023). Development of a wide-spectrum thermochemical code with application to planar reacting and non-reacting shocks. Universidad Carlos III de Madrid, Spain (PhD thesis). 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). Advisor: 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). Advisor: Marcos Vera.

CHAPTER

THREE

INSTALLATION

3.1 Downloading the Code

You can download the Combustion Toolbox from several sources:

GitHub MATLAB FileExchange Zenodo

3.2 Installing the Code

To install the Combustion Toolbox, you can use the provided `INSTALL.m` file. Here's how to install the toolbox:

1. Navigate to the directory where you downloaded the code.
2. Run the `INSTALL.m` file using the following command in the MATLAB Command Window:

```
INSTALL()
```

3. This will add the necessary folders to the MATLAB path and also install the Combustion Toolbox GUI. You can now use the Combustion Toolbox in your MATLAB code.

If you wish to install the GUI only, you can run the following command:

```
INSTALL('install', 'gui')
```

Alternatively, you can execute the `combustion_toolbox_app.mlappinstall` file in the MATLAB Command Window to install the GUI, which will be available through the MATLAB Apps Toolbar.

To install the standalone version, simply execute `combustion_toolbox_standalone_windows.exe` for Windows or `combustion_toolbox_standalone_macos.app` for macOS, both found in the installer folder. This royalty-free version requires MATLAB Runtime framework (automatically installed during installation, **requires an internet connection**).

3.3 Using the Combustion Toolbox

The 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.

To use the Combustion Toolbox in the MATLAB desktop environment, you can call the functions from your MATLAB code directly, e.g., `App()`.

To use the GUI, simply type `combustion_toolbox` or `combustion_toolbox_app` in the MATLAB Command Window, or click on the app icon in the MATLAB apps toolbar. This will open the Combustion Toolbox GUI, which provides a user-friendly interface for accessing and using the package.

Note: If you encounter any issues during the installation process or while using the Combustion Toolbox, please refer to the "Issues" section of the GitHub repository or contact us directly at acuadra@ing.uc3m.es for assistance.

CHAPTER

FOUR

TUTORIALS

Combustion Toolbox can be used in two different ways: as a GUI-based application or as plain code. The GUI-based application is the easiest way to use the code, but it is limited to the capabilities of the GUI. The plain code mode is more flexible and allows the user to access all the capabilities of the code. In this section, we will cover a collection of tutorials that will help you to get started with the Combustion Toolbox.

4.1 Basics

In this section we will cover the basic functionalities of the Combustion Toolbox (CT).

Briefly, the main steps required to solve a problem with CT are as follows:

1. Initialize CT (load databases, set default variables, etc.).
2. Define the initial state (temperature, pressure, composition).
3. Set chemical system (species that can appear in the chemical transformations).
4. Select the problem type (e.g., an adibatic isobaric combustion).
5. Define additional parameters (depends on the problem selected).
6. Solve the problem.
7. Plot the results.

4.1.1 Framework initialization

To begin, start MATLAB and navigate to the folder where you have downloaded the Combustion Toolbox. To include files in PATH, run this command in the command window:

```
INSTALL()
```

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

```
self = App()
```

If files contained in CT are correctly defined, 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]
```

This `self` variable encapsulates essential shared data necessary for calculations and typically serves as the first argument in most CT routines. Thus, the data passed between the functions has been organized in a hierarchical tree structure (except for the GUI, which is based on OOP) as shown in Fig.1, namely:

- `self (App)`: parent node; contains all the data of the code, e.g., databases, input values, and results.
- `Constants (C)`: contains constant values.
- `Elements (E)`: contains data of the chemical elements in the problem (names and indices for fast data access).
- `Species (S)`: contains data of the chemical species in the problem (names and indices for fast data access), as well as lists (cells) with the species for complete combustion.
- `Problem Description (PD)`: contains data of the problem to solve, e.g., initial mixture (composition, temperature, pressure), problem type, and its configuration.
- `Problem Solution (PS)`: contains results (mixtures).

- `Tuning Properties (TN)`: contains parameters that control the numerical error of the algorithms implemented in the different modules.
- `Miscellaneous (Misc)`: contains values that configure the auto-generated plots and export setup, as well as flags.
- `Database master (DB_master)`: a structured thermochemical database including data from McBride [2002], Burcat and Ruscic [2005].
- `Database (DB)`: a structured thermochemical database with *griddedInterpolant* objects (see MATLAB built-in function griddedInterpolant.m) that contain piecewise cubic Hermite interpolating polynomials (PCHIP) [Fritsch and Carlson, 1980] for faster data access.

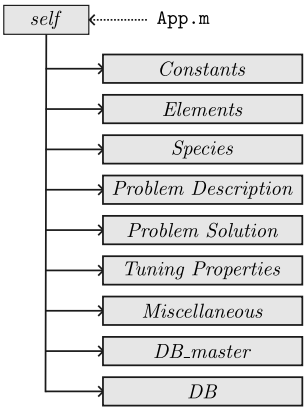


Figure 1: *Combustion Toolbox hierarchical data tree structure, where* `App.m` *is the initialization function.*

4.1.2 Accessing the databases

In the preceding section, we stated that to initialize the Combustion Toolbox (load databases, default parameters, etc.), we have to run the following command:

```
self = App();
```

Now, let's delve into how we can access the databases incorporated within the Combustion Toolbox (CT). There are two main types of databases: `DB_master` and `DB`. The former comprises all data from from McBride [2002], Burcat and Ruscic [2005], while the latter contains the same data in a more optimized format, ensuring faster data access.

Tip: Distinguishing between `DB_master` and `DB` is crucial because the latter, due to the utilization of *griddedInterpolant* objects, has a significantly larger size (~36 MB). A streamlined version of `DB` can be generated to conserve memory and reduce loading time (~1 second for 3603 species) when initializing CT. To achieve this, execute:

```
self.DB = generate_DB(self.DB_master, {'O2', 'N2'});
```

Subsequently, save the self.DB variable in the `databases` folder as DB.mat to be loaded in the remaining sessions.

Finding specific chemical species

The thermodynamic data of the species, e.g., Cgr, can be accessed as follows:

```
self.DB.Cbgrb;
```

This command will yield information similar to the following:

```
      FullName: 'C(gr)'
          name: 'Cbgrb'
      comments: 'Graphite. Ref-Elm. TRC(4/83) vc,uc,tc1000-1002.
↪'
     txFormula: 'C   1.00    0.00    0.00    0.00    0.00'
            mm: 12.0107
            hf: 0
            ef: 0
         phase: 1
             T: [200 229.1457 258.2915 287.4372 316.5829 345.7286 ... ] (1×200␣
↪double)
       cPcurve: [1×1 griddedInterpolant]
       h0curve: [1×1 griddedInterpolant]
       s0curve: [1×1 griddedInterpolant]
       g0curve: [1×1 griddedInterpolant]
        ctTInt: 3
        tRange: {[200 600]  [600 2000]  [2000 6000]}
    tExponents: {[-2 -1 0 1 2 3 4 0]  [-2 -1 0 1 2 3 4 0]  [-2 -1 0 1 2 3 4 0]}
             a: {[1×8 double]  [1×8 double]  [1×8 double]}
             b: {[8.9439e+03 -72.9582]  [1.3984e+04 -44.7718]  [5.8481e+03 -23.
↪5093]}
```

These details offer comprehensive insights into the thermodynamic properties of the specified species in the Combustion Toolbox database. We have the following fields:

- `FullName`: full name of the species.
- `name`: short name of the species as defined in the database.
- `comments`: comments about the species.

- `txFormula`: chemical formula of the species.
- `mm`: molar mass [g/mol].
- `hf`: standard enthalpy of formation [J/mol].
- `ef`: standard internal energy of formation [J/mol].
- `phase`: phase of the species (0: gas, 1: liquid or solid).
- `T`: temperature vector [K].
- `cPcurve`: *griddedInterpolant* object containing the standard specific heat capacity at constant pressure [J/mol-K].
- `h0curve`: *griddedInterpolant* object containing the standard enthalpy [J/mol].
- `s0curve`: *griddedInterpolant* object containing the standard entropy [J/mol-K].
- `g0curve`: *griddedInterpolant* object containing the standard Gibbs free energy [J/mol].
- `ctTInt`: number of temperature intervals.
- `tRange`: temperature intervals.
- `tExponents`: temperature exponents for the standard specific heat capacity at constant pressure, standard enthalpy, and standard entropy, respectively.
- `a`: coefficients of the polynomial.
- `b`: coefficients of the polynomial.

Finding list of chemical species with given chemical elements

To find the list of species that contain only some chemical elements, e.g., O and H, we can use the following command:

NASA database

```
list_species = find_products(self, {'O', 'H'})
```

which will yield the following output:

```
list_species =

  1×15 cell array

  Columns 1 through 7
```

(continues on next page)

(continued from previous page)

```
    {'HO2'}    {'H2O'}    {'H2O2'}    {'OH'}    {'H2Obcrb'}    {'H2ObLb'}    {
↪'H2O2bLb'}

  Columns 8 through 15

    {'O'}    {'O2'}    {'O3'}    {'O2bLb'}    {'O3bLb'}    {'H'}    {'H2'}    {
↪'H2bLb'}
```

Note: By default, the find_products.m function looks for species in the NASA database, includes condensed species, and excludes ionized species. To search for species in Burcat's database and include ionized species, set `flag_burcat` and `flag_ion` options to `true`. Alternatively, we can modify the default value in the Species.m file. Chemical species from the Third Millennium database (Burcat) are indicated with the subscript `_M`.

Third Millennium database (Burcat)

```
list_species = find_products(self, {'O', 'H'}, 'flag_burcat', true)
```

which will yield the following output:

```
list_species =

  1×25 cell array

  Columns 1 through 7

    {'HO2'}    {'H2O'}    {'H2O2'}    {'OH'}    {'H2Obcrb'}    {'H2ObLb'}    {
↪'H2O2bLb'}

  Columns 8 through 14

    {'OH_M'}    {'HO2_M'}    {'HO3_M'}    {'H2O2_M'}    {'H2O3_M'}    {'HOOOH_M
↪'}    {'O'}

  Columns 15 through 21

    {'O2'}    {'O3'}    {'O2bLb'}    {'O3bLb'}    {'O_M'}    {'O2_M'}    {'O3_M
↪'}
```

(continues on next page)

(continued from previous page)

```
  Columns 22 through 25

    {'O4_M'}    {'H'}    {'H2'}    {'H2bLb'}
```

Note: Chemical species from the Third Millennium database (Burcat) are indicated with the subscript `_M`. By default, the find_products.m function looks for species in the NASA database, includes condensed species, and excludes ionized species. To search for species in Burcat's database, we have enabled the `flag_burcat` option, setting it to `true`. Alternatively, we can modify the default value in the Species.m file.

Both and ionized species

```
list_species = find_products(self, {'O', 'H'}, 'flag_burcat', true, , 'flag_ion
↪', true)
```

which will yield the following output:

```
list_species =

  1×52 cell array

  Columns 1 through 6

    {'HO2minus'}    {'H2Oplus'}    {'H3Oplus'}    {'OHplus'}    {'OHminus'}
↪{'HO2plus_M'}

  Columns 7 through 11

    {'HO2minus_M'}    {'HO3plus_M'}    {'HO3minus_M'}    {'H2O2plus_M'}    {
↪'H2O3plus_M'}

  Columns 12 through 17

    {'H3O2plus_M'}    {'Oplus'}    {'Ominus'}    {'O2plus'}    {'O2minus'}    {
↪'O3plus_M'}

  Columns 18 through 23
```

(continues on next page)

(continued from previous page)

```
    {'O3minus_M'}    {'O4plus_M'}    {'O4minus_M'}    {'Hplus'}    {'Hminus'} ␣
↪   {'H2plus'}

  Columns 24 through 29

    {'H2minus'}    {'H2minus_M'}    {'H3plus_M'}    {'eminus'}    {'HO2'}    {
↪'H2O'}

  Columns 30 through 36

    {'H2O2'}    {'OH'}    {'H2Obcrb'}    {'H2ObLb'}    {'H2O2bLb'}    {'OH_M'}␣
↪    {'HO2_M'}

  Columns 37 through 43

    {'HO3_M'}    {'H2O2_M'}    {'H2O3_M'}    {'HOOOH_M'}    {'O'}    {'O2'}
↪{'O3'}

  Columns 44 through 51

    {'O2bLb'}    {'O3bLb'}    {'O_M'}    {'O2_M'}    {'O3_M'}    {'O4_M'}    {
↪'H'}    {'H2'}

  Column 52

    {'H2bLb'}
```

Note: Chemical species from the Third Millennium database (Burcat) are indicated with the subscript `_M`. By default, the find_products.m function looks for species in the NASA database, includes condensed species, and excludes ionized species. To search for species in Burcat's database and include ionized species, we have enabled the `flag_burcat` and `flag_ion` options, setting both to `true`. Alternatively, we can modify the default flag values in the Species.m file.

Tip: The same procedure can be used to identify all possible products after a chemical transformation given a set of chemical species (reactants), as described in *Defining chemical system*.

4.1.3 Defining initial state of a mixture

The initial state of a mixture (`reactants`) is defined by its chemical composition, temperature, and pressure. For this example, let's assume that we have a stoichiometric mixture of methane (CH_4) and ideal-air (79% N_2 and 21% O_2 in volume) at 300 K and 1 bar.

Tip: Remember, that initializing the Combustion Toolbox (CT) is the first step to start working with the code. To do that, type the following at the prompt:

```
self = App();
```

Defining initial chemical species

In the Combustion Toolbox (CT), the chemical species of the initial mixture are categorized as follows:

- `Fuel`: chemical species considered as fuel.
- `Oxidizer`: chemical species that react with the fuel.
- `Inert`: chemical species that remain inert and do not react with the fuel or oxidizer, maintaining a frozen composition.

Considering that all the species react, we can write at the prompt:

```
self.PD.S_Fuel     = {'CH4'};
self.PD.S_Oxidizer = {'N2', 'O2'};
```

Defining initial composition

The above classification enables us to specify the chemical composition of a premixed combustion system using the equivalence ratio, defined as

$$\phi = \frac{\text{fuel-oxidizer-ratio}}{(\text{fuel-oxidizer-ratio})_{\text{st}}} = \frac{n_{\text{fuel}}/n_{\text{oxidizer}}}{(n_{\text{fuel}}/n_{\text{oxidizer}})_{\text{st}}}. \tag{4.1}$$

Here, n represents the number of moles, and the subscript `st` denotes the stoichiometric value. When $\phi = 1$, the mixture is said to be stoichiometric, i.e., the fuel and oxidizer are present in the exact proportions required for complete combustion. If $\phi < 1$ the mixture is said to be fuel-lean, and if $\phi > 1$ the mixture is said to be fuel-rich.

Note: The number of moles have to be specified in the same order as the corresponding set of species.

The initial composition can be defined with the number of moles n_j of each chemical species in the mixture, or with the equivalence ratio ϕ.

Number of moles

For a stoichiometric mixture of methane and ideal-air, we have $n_{\mathrm{CH_4}} = 1$, $n_{\mathrm{O_2}} = 2$, and $n_{\mathrm{N_2}} = 2 \cdot 79/21$. This can be defined as follows:

```
self.PD.N_Fuel = 1;
self.PD.N_Oxidizer = 2 * [1, 79/21];
```

where the factor 2 in front of the oxidizer is due to the stoichiometric coefficients of the reaction.

Equivalence ratio

For a stoichiometric mixture of methane and ideal-air, we have $\phi = 1$. However, we have to specify the number of moles of oxidizers relative to O_2 to define the composition properly, thus

```
self = set_prop(self, 'phi', 1);
self.PD.ratio_oxidizers_O2 = [79, 21] / 21;
```

Tip: There are chemical calculations in which is not necessary to differentiate between fuel, oxidizer, and inert species. In those cases, we can just consider all the species as fuel and provide their corresponding number of moles.

Defining initial temperature and pressure

To define the initial temperature and pressure of the mixture, we can use the set_prop.m function as follows:

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

Warning: The `default units` for temperature and pressure are `K` and `bar`, respectively.

Summary

Number of moles

```
% Initialize CT
self = App();
% Define initial chemical species
self.PD.S_Fuel     = {'CH4'};
self.PD.S_Oxidizer = {'N2', 'O2'};
% Define initial composition (in moles)
self.PD.N_Fuel = 1;
self.PD.N_Oxidizer = 2 * [1, 79/21];
% Define initial temperature and pressure
self = set_prop(self, 'TR', 300, 'pR', 1);
```

Equivalence ratio

```
% Initialize CT
self = App();
% Define initial chemical species
self.PD.S_Fuel     = {'CH4'};
self.PD.S_Oxidizer = {'N2', 'O2'};
% Define initial composition (with equivalence ratio)
self = set_prop(self, 'phi', 1);
self.PD.ratio_oxidizers_O2 = [79, 21] / 21;
% Define initial temperature and pressure
self = set_prop(self, 'TR', 300, 'pR', 1);
```

4.1.4 Defining chemical system

The chemical system refer to the chemical species that may appear as products. Expanding upon the earlier example, we will now establish the chemical system for the complete combustion of a stoichiometric CH_4-ideal air mixture, represented by the global reaction

$$\underbrace{CH_4 + 2(O_2 + 3.76N_2)}_{\text{reactants (initial state)}} \rightleftharpoons \underbrace{CO_2 + 2H_2O + 7.52N_2}_{\text{products (final state)}}. \tag{4.2}$$

Given the above global reaction, the products include CO_2, H_2O, and N_2, which we can define as follows:

```
self.S.LS = {'N2', 'CO2', 'H2O'};
```

and that is all we need to define a chemical system in CT.

Tip: When we have prior knowledge of the chemical species involved in the problem, we can streamline the initialization of CT by directly specifying those species. Consequently, we only need to include these specific chemical species in our problem, writing at the prompt:

```
self = App({'N2', 'CO2', 'H2O'});
```

Identifying all possible products

Combustion problems typically entail numerous chemical species, and occasionally we lack prior knowledge of all relevant species in the system. In such cases, we can recall in find_products.m routine introduced in *Accessing the databases*. This routine allows us to identify all potential chemical species resulting from a chemical transformation (products), given a set of species (reactants). For this example, we can write at the prompt:

```
self.S.LS = find_products(self, {'CH4', 'O2', 'N2'})
```

which will yield a list of +200 chemical species. By default, the find_products.m function scans for species in the NASA database, includes condensed species, and excludes ionized species. To search for species in Burcat's database and include ionized species, we have to enable the `flag_burcat` and `flag_ion` options, setting both to `true`, as follows:

```
self.S.LS = find_products(self, {'CH4', 'O2', 'N2'}, 'flag_burcat', true,
↪'flag_ion', true);
```

This modification generates a list of +1000 chemical species. Alternatively, we can modify the default flag values in the Species.m file.

Tip: In cases where no chemical system is defined, CT automatically identifies all potential products given a set of reactants, i.e., it uses the find_products.m routine to construct the chemical system, taking into account the default flag values `flag_burcat`, `flag_ion`, `flag_condensed` defined in Species.m.

Using predefined chemical systems

The Combustion Toolbox incorporates a range of predefined chemical systems, outlined in the list_species.m routine. For example, some of the predefined chemical systems are defined below:

Air

- Calculations for air.
- Neglects ionization of chemical species.

```
self.S.LS = list_species('air');
```

or equivalently

```
self = list_species(self, 'air');
```

Chemical species included:

```
self.S.LS = {'CO2', 'CO', 'O2', 'N2', 'Ar', 'O', 'O3', ...
             'N', 'NO', 'NO2', 'NO3', 'N2O', 'N2O3', ...
             'N2O4', 'N3', 'C'};
```

Air ions

- Calculations for air.
- Considers ionization of chemical species.

```
self.S.LS = list_species('air ions');
```

or equivalently

```
self = list_species(self, 'air ions');
```

Chemical species included:

```
self.S.LS = {'eminus', 'Ar', 'Arplus', 'C', 'Cplus', 'Cminus', ...
             'CN', 'CNplus', 'CNminus', 'CNN', 'CO', 'COplus', ...
             'CO2', 'CO2plus', 'C2', 'C2plus', 'C2minus', 'CCN', ...
             'CNC', 'OCCN', 'C2N2', 'C2O', 'C3', 'C3O2', 'N', ...
             'Nplus', 'Nminus', 'NCO', 'NO', 'NOplus', 'NO2', ...
             'NO2minus', 'NO3', 'NO3minus', 'N2', 'N2plus', ...
```

(continues on next page)

(continued from previous page)

```
              'N2minus', 'NCN', 'N2O', 'N2Oplus', 'N2O3', 'N2O4', ...
              'N2O5', 'N3', 'O', 'Oplus', 'Ominus', 'O2', 'O2plus', ...
              'O2minus', 'O3'};
```

HC/O2/N2

- Calculations for hydrocarbon (HC) combustion in air.
- Neglects most dissociation chemical species and excludes ions.

```
self.S.LS = list_species('HC/O2/N2');
```

or equivalently

```
self = list_species(self, 'HC/O2/N2');
```

Chemical species included:

```
self.S.LS = {'CO2', 'CO', 'H2O', 'H2', 'O2', 'N2', 'Ar'}
```

Soot formation

- Calculations for hydrocarbon (HC) combustion in air.
- Considers soot formation and a small set of minor species.
- Excludes ionized chemical species.

```
self.S.LS = list_species('soot formation');
```

or equivalently

```
self = list_species(self, 'soot formation');
```

Chemical species included:

```
self.S.LS = {'CO2', 'CO', 'H2O', 'H2', 'O2', 'N2', 'Ar', 'Cbgrb', ...
             'C2', 'C2H4', 'CH', 'CH', 'CH3', 'CH4', 'CN', 'H', ...
             'HCN', 'HCO', 'N', 'NH', 'NH2', 'NH3', 'NO', 'O', 'OH'};
```

Soot formation extended

- Calculations for hydrocarbon (HC) combustion in air.
- Considers soot formation and a large set of minor species.
- Excludes ionized chemical species.

```
self.S.LS = list_species('soot formation extended');
```

or equivalently

```
self = list_species(self, 'soot formation extended');
```

Chemical species included:

```
self.S.LS = {'CO2', 'CO', 'H2O', 'H2', 'O2', 'N2', 'Ar', 'Cbgrb', ...
             'C2', 'C2H', 'C2H2_acetylene', 'C2H2_vinylidene', ...
             'C2H3_vinyl', 'C2H4', 'C2H5', 'C2H5OH', 'C2H6', ...
             'C2N2', 'C2O', 'C3', 'C3H3_1_propynl', ...
             'C3H3_2_propynl', 'C3H4_allene', 'C3H4_propyne', ...
             'C3H5_allyl', 'C3H6O_acetone', 'C3H6_propylene', ...
             'C3H8', 'C4', 'C4H2_butadiyne', 'C5', 'C6H2', 'C6H6', ...
             'C8H18_isooctane', 'CH', 'CH2', 'CH2CO_ketene', ...
             'CH2OH', 'CH3', 'CH3CHO_ethanal', 'CH3CN', ...
             'CH3COOH', 'CH3O', 'CH3OH', 'CH4', 'CN', 'COOH', 'H', ...
             'H2O2', 'HCCO', 'HCHO_formaldehy', 'HCN', 'HCO', ...
             'HCOOH', 'HNC', 'HNCO', 'HNO', 'HO2', 'N', 'N2O', ...
             'NCO', 'NH', 'NH2', 'NH2OH', 'NH3', 'NO', 'NO2', ...
             'O', 'OCCN', 'OH', 'C3O2', 'C4N2', 'CH3CO_acetyl', ...
             'C4H6_butadiene', 'C4H6_1butyne', 'C4H6_2butyne', ...
             'C2H4O_ethylen_o', 'CH3OCH3', 'C4H8_1_butene', ...
             'C4H8_cis2_buten', 'C4H8_isobutene', ...
             'C4H8_tr2_butene', 'C4H9_i_butyl', 'C4H9_n_butyl', ...
             'C4H9_s_butyl', 'C4H9_t_butyl', 'C6H5OH_phenol', ...
             'C6H5O_phenoxy', 'C6H5_phenyl', 'C7H7_benzyl', ...
             'C7H8', 'C8H8_styrene', 'C10H8_naphthale'};
```

HC/O2/N2 propellants

- Calculations for hydrocarbon (HC) combustion (propellants) in air.
- Considers soot formation and a large set of minor species.
- Excludes ionized chemical species.

```
self.S.LS = list_species('HC/O2/N2 propellants');
```

or equivalently

```
self = list_species(self, 'HC/O2/N2 propellants');
```

Chemical species included:

```
self.S.LS = {'CO2', 'CO', 'H2O', 'H2', 'O2', 'N2', 'Ar', 'Cbgrb', ...
             'C2', 'C2H', 'C2H2_acetylene', 'C2H2_vinylidene', ...
             'C2H3_vinyl', 'C2H4', 'C2H5', 'C2H5OH', 'C2H6', ...
             'C2N2', 'C2O', 'C3', 'C3H3_1_propynl', ...
             'C3H3_2_propynl', 'C3H4_allene', 'C3H4_propyne', ...
             'C3H5_allyl', 'C3H6O_acetone', 'C3H6_propylene', ...
             'C3H8', 'C4', 'C4H2_butadiyne', 'C5', 'C6H2', 'C6H6', ...
             'C8H18_isooctane', 'CH', 'CH2', 'CH2CO_ketene', ...
             'CH2OH', 'CH3', 'CH3CHO_ethanal', 'CH3CN', ...
             'CH3COOH', 'CH3O', 'CH3OH', 'CH4', 'CN', 'COOH', 'H', ...
             'H2O2', 'HCCO', 'HCHO_formaldehy', 'HCN', 'HCO', ...
             'HCOOH', 'HNC', 'HNCO', 'HNO', 'HO2', 'N', 'N2O', ...
             'NCO', 'NH', 'NH2', 'NH2OH', 'NH3', 'NO', 'NO2', ...
             'O', 'OCCN', 'OH', 'C3O2', 'C4N2', 'RP_1', 'H2bLb', ...
             'O2bLb'};
```

Si/HC/O2/N2 propellants

- Calculations for Silicon (Si) combustion (propellants) in air.
- Considers soot formation and a minor set of minor species.
- Excludes ionized chemical species.

```
self.S.LS = list_species('Si/HC/O2/N2 propellants');
```

or equivalently

```
self = list_species(self, 'Si/HC/O2/N2 propellants');
```

Chemical species included:

```
self.S.LS = {'CO2', 'CO', 'H2O', 'H2', 'O2', 'N2', 'Ar', 'Cbgrb', ...
             'C2', 'C2H4', 'CH', 'CH', 'CH3', 'CH4', 'CN', 'H', ...
             'H2O2', 'HCN', 'HCO', 'N', 'NH', 'NH2', 'NH3', 'NO', 'O', 'OH', ..
↪.
             'O2bLb', 'Si', 'SiH', 'SiH2', 'SiH3', 'SiH4', 'SiO2', 'SiO', ...
             'SibLb', 'SiO2bLb', 'Si2'};
```

Summary

During initialization

```
% Initialize CT and define chemical system
self = App({'CO2', 'CO', 'H2O', 'H2', 'O2', 'N2', 'Ar'});
```

After initialization

```
% Initialize CT
self = App();
% Define chemical system
self = list_species(self, {'CO2', 'CO', 'H2O', 'H2', 'O2', 'N2', 'Ar'});
```

Using predefined chemical systems

```
% Initialize CT and define chemical system
self = App('HC/O2/N2');
```

4.2 Chemical equilibrium

In this section, we will cover how to perform chemical equilibrium calculations with the Combustion Toolbox. Chemical equilibrium is solved via minimization of the Gibbs/Helmholtz free energy combining the method of Lagrange multipliers with a multidimensional Newton-Raphson (NR) method, based on the mathematical formulation set forth by NASA in its CEA code [Gordon and McBride, 1994]. Our dedicated CT-EQUIL module has been developed to precisely determine the equilibrium composition of multi-component gas mixtures undergoing essential thermochemical transformations. These transformations occur from an initial state (reactants), defined by its composition, temperature, and pressure, to a final state (products), characterized by a set of chemical species (in gaseous— including ions—or pure condensed phase) along with two thermodynamic state functions, such as enthalpy and pressure, e.g., for isobaric combustion processes.

The CT-EQUIL module facilitates the computation of chemical equilibrium composition and thermodynamic properties for a range of specified state function pairs:

- `TP` (temperature and pressure),
- `HP` (enthalpy and pressure),
- `SP` (entropy and pressure),
- `TV` (temperature and volume),
- `EV` (internal energy and volume),
- `SV` (entropy and volume).

Additionally, Combustion Toolbox enables the computation of chemical equilibrium under various assumptions regarding the final gas mixture, including calorically perfect gas, calorically imperfect gas with frozen chemistry, or calorically imperfect gas with equilibrium chemistry, including dissociation and ionization.

In many practical applications, the equilibrium temperature of a system is not initially determined, thereby necessitating the provision of supplementary information to close the problem. This additional information may be obtained from an enthalpy, internal energy, or entropy conservation equation, subject to the requirement that the corresponding state function f remains unchanged, namely

$$\Delta f\left(T\right) \equiv f_{\mathrm{F}}\left(T\right) - f_{\mathrm{I}}\left(T_{\mathrm{I}}\right) = 0, \tag{4.3}$$

where the subscripts F and I refer here to the final and initial states of the mixture, respectively. Unlike in NASA's CEA code, we have increased the flexibility of the CT-EQUIL module by decoupling this additional equation and retrieved the new condition by using a second-order NR method

$$T_{k+1} = T_k - \frac{f\left(T_k\right)}{f'\left(T_k\right)}. \tag{4.4}$$

4.2.1 Chemical equilibrium at constant temperature and pressure

As a first example, let's compute the equilibrium composition of a stoichiometric CH_4-ideal air mixture at 3000 K and 1.01325 bar (1 atm). To obtain the equilibrium composition, we have to define the chemical species that can appear in the final state. The combustion of typical hydrocarbon fuels, such as methane, is a complex process that involves hundreds of chemical species. However, the number of species that have a significant impact on the properties of the final mixture is much smaller. Combustion Toolbox provides a list of predefined species that can be used to compute the equilibrium composition of a mixture. The list of predefined species can be found in the routine list_species.m. Alternatively, refer to the Using predefined chemical systems tutorial section.

For a preliminary analysis, we will consider a set of 23 species:

- CO_2, CO, H_2O, H_2, O_2, N_2, Cgr,
- C_2, C_2H_4, CH, CH_3, CH_4, CN, H,
- HCN, HCO, N, NH, NH_2, NH_3, NO, O, OH,

which typically appear in the combustion of hydrocarbon fuels with air. This set of species is defined in the routine list_species.m as `Soot formation`, which also includes Ar.

Defining chemical system

Plain code

To initialize the Combustion Toolbox (CT) with this set of species, type the following at the MATLAB prompt:

Predefined list of species

```
self = App('Soot formation');
```

Specifiying a custom list of species

```
self = App({'CO2', 'CO', 'H2O', 'H2', 'O2', 'N2', 'Cbgrb', ...
            'C2', 'C2H4', 'CH', 'CH3', 'CH4', 'CN', 'H', ...
            'HCN', 'HCO', 'N', 'NH', 'NH2', 'NH3', 'NO', 'O', 'OH'});
```

GUI

Note: This section is under development. Stay tuned!

Setting the initial thermodynamic state

Plain code

To indicate the temperature [K] and pressure [bar] of the initial mixture, type:

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

GUI

Note: This section is under development. Stay tuned!

Setting the initial chemical composition

Plain code

To indicate the species and chemical composition in the initial mixture:

Individual study

For a stochiometric CH_4-ideal air mixture:

```
self.PD.S_Fuel     = {'CH4'};
self.PD.N_Fuel     = 1;
self.PD.S_Oxidizer = {'N2', 'O2'};
self.PD.N_Oxidizer = [7.52, 2];
```

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

```
self.PD.S_Fuel     = {'CH4'};
self.PD.S_Oxidizer = {'N2', 'O2'};
self.PD.ratio_oxidizers_O2 = [79, 21]/21;
self = set_prop(self, 'phi', 1);
```

The last two lines of code establish the proportion of the oxidizers species over O_2 and the equivalence ratio, respectively. The number of moles are calculated considering that the number of moles of fuel is one by default.

Parametric study

For a lean-to-rich CH_4-ideal air mixtures:

```
self.PD.S_Fuel     = {'CH4'};
self.PD.S_Oxidizer = {'N2', 'O2'};
self.PD.ratio_oxidizers_O2 = [79, 21]/21;
self = set_prop(self, 'phi', 0.5:0.01:4);
```

The last two lines of code establish the proportion of the oxidizers species over O_2 and the equivalence ratio, respectively. The number of moles are calculated considering that the number of moles of fuel is one by default.

GUI

Note: This section is under development. Stay tuned!

Set chemical transformation

Plain code

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

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

Note: Here, the pressure of the final state has been defined using the defined value of the initial state, `self.PD.pR.value`.

GUI

Note: This section is under development. Stay tuned!

Perform chemical calculations

Plain code

To solve the aforementioned problem, run

```
self = solve_problem(self, 'TP');
```

GUI

Note: This section is under development. Stay tuned!

Results

Plain code

The results are contained in `self.PS`. By default, `solve_problem.m` prints the results through the command window, 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
```

(continues on next page)

(continued from previous page)

```
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
cp [kJ/(kg-K)]  |          1.0786  |          5.5609
gamma [-]       |          1.3869  |          1.1680
gamma_s [-]     |          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[+22]          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
```

(continues on next page)

(continued from previous page)

```
CH4                     1.1358e-14
MINORS[+5]              0.0000e+00

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

Note: The `solve_problem.m` function prints the results in the command window by default. To avoid this, write before:

```
self.Misc.FLAG_RESULTS = false
```

or modify its default value directly in `miscellaneous.m`.

GUI

Note: This section is under development. Stay tuned!

Postprocessed results (predefined plots for several cases)

Plain code

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

```
post_results(self);
```

will reproduce **Figure 1**, which represents the variation of the molar fraction with the equivalence ratio for lean to rich CH_4-ideal air mixtures at 3000 [K] and 1.01325 [bar].

Figure 1: *Example TP: variation of molar fraction for lean to rich CH$_4$-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 2.39 seconds using a Intel(R) Core(TM) i7-11800H CPU @ 2.30GHz.*

GUI

Note: This section is under development. Stay tuned!

Summary

Individual study

```
% Initialize CT and define chemical system
self = App('Soot formation');

% Define initial chemical composition
self.PD.S_Fuel     = {'CH4'};
self.PD.S_Oxidizer = {'N2', 'O2'};
self.PD.ratio_oxidizers_O2 = [79, 21]/21;
self = set_prop(self, 'phi', 1);

% Define initial thermochemical state
self = set_prop(self, 'TR', 300, 'pR', 1 * 1.01325);

% Set chemical transformation
self = set_prop(self, 'pP', self.PD.pR.value, 'TP', 3000);

% Perform chemical calculations
self = solve_problem(self, 'TP');
```

Parametric study

```
% Initialize CT and define chemical system
self = App('Soot formation');

% Define initial chemical composition
self.PD.S_Fuel     = {'CH4'};
self.PD.S_Oxidizer = {'N2', 'O2'};
self.PD.ratio_oxidizers_O2 = [79, 21]/21;
self = set_prop(self, 'phi', 0.5:0.01:4);

% Define initial thermochemical state
```

(continues on next page)

(continued from previous page)

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

% Set chemical transformation
self = set_prop(self, 'pP', self.PD.pR.value, 'TP', 3000);

% Perform chemical calculations
self = solve_problem(self, 'TP');

% Postprocess results
post_results(self);
```

4.2.2 Chemical equilibrium at constant enthalpy and pressure

Let's consider the adiabatic combustion of acetylene (C_2H_2) with ideal air (79% N_2, 21% O_2 in volume) at constant pressure with $p = 1.01325$ bar (1 atm) and an equivalence ratio $\phi \in (0.5, 5)$. In this scenario, the equilibrium temperature of the system, denoted as T, is not initially determined, and we need to provide supplementary information to close the problem. For an adiabatic isobaric process, the enthalpy of the system remains constant, thus

$$\Delta f\left(T\right) \equiv h_{\mathrm{F}}\left(T\right) - h_{\mathrm{I}}\left(T_{\mathrm{I}}\right) = 0, \tag{4.5}$$

where the subscripts F and I refer here to the final (products) and initial (reactants) states of the mixture, respectively.

Similar to the preceding case, we will consider a set of 23 species involved in the combustion of hydrocarbon fuels with air, namely:

- CO_2, CO, H_2O, H_2, O_2, N_2, Cgr,
- C_2, C_2H_4, CH, CH_3, CH_4, CN, H,
- HCN, HCO, N, NH, NH_2, NH_3, NO, O, OH.

This set, referred to as 'Soot formation,' is defined in the routine list_species.m and includes Ar.

Tip: The Combustion Toolbox offers various predefined list of species. To explore the complete list, type help list_species at the MATLAB prompt or refer to the Using predefined chemical systems tutorial section.

Let's start by initializing the Combustion Toolbox and defining the chemical system and the initial state of the mixture (chemical composition, temperature, and pressure):

```
% Initialize CT and define chemical system
self = App('Soot formation');

% Define initial chemical composition
self.PD.S_Fuel     = {'C2H2_acetylene'};
self.PD.S_Oxidizer = {'N2', 'O2'};
self.PD.ratio_oxidizers_O2 = [79, 21]/21;
self = set_prop(self, 'phi', 0.5:0.01:5);

% Define initial thermochemical state
self = set_prop(self, 'TR', 300, 'pR', 1 * 1.01325);
```

To close the problem, we maintain constant enthalpy and pressure. This problem type is denoted as HP in the Combustion Toolbox. By default, the code ensures that the enthalpy of the final state equals that of the initial state; however, we must specify the pressure of the final state using the pP property:

```
% Set chemical transformation
self = set_prop(self, 'pP', self.PD.pR.value);
```

Note: All parameters defining the problem are stored within the `PD` property of the `self` structure. While we consistently refer to self for ease of understanding, you may rename it to any other name as needed.

Finally, we can solve the problem and visualize the results:

```
% Perform chemical calculations
self = solve_problem(self, 'HP');

% Postprocess results
post_results(self);
```

If everything goes well, the last promt should display the following message:

```
******************************************************
------------------------------------------------------
Problem type: HP  | phi = 0.5000
------------------------------------------------------
                  |    REACTANTS    |      PRODUCTS
T [K]             |       300.0000  |      1776.8223
p [bar]           |         1.0132  |         1.0132
r [kg/m3]         |         1.1674  |         0.2011
h [kJ/kg]         |       321.9941  |       321.9941
```

(continues on next page)

(continued from previous page)

```
e [kJ/kg]       |        235.1950  |       -181.7992
g [kJ/kg]       |      -1768.9710  |     -15487.8789
s [kJ/(kg-K)]   |          6.9699  |          8.8978
W [g/mol]       |         28.7369  |         29.3242
(dlV/dlp)T [-]  |                  |         -1.0000
(dlV/dlT)p [-]  |                  |          1.0017
cp [kJ/(kg-K)]  |          1.0364  |          1.3539
gamma [-]       |          1.3873  |          1.2660
gamma_s [-]     |          1.3873  |          1.2660
sound vel [m/s] |        347.0091  |        798.6200
-------------------------------------------------------
REACTANTS                Xi [-]
N2                    7.5816e-01
O2                    2.0154e-01
C2H2_acetylene        4.0307e-02
MINORS[+22]           0.0000e+00

TOTAL                 1.0000e+00
-------------------------------------------------------
PRODUCTS                 Xi [-]
N2                    7.7233e-01
O2                    1.0142e-01
CO2                   8.2220e-02
H2O                   4.0942e-02
NO                    2.6404e-03
OH                    3.6588e-04
CO                    4.1312e-05
O                     3.0818e-05
H2                    5.5206e-06
H                     6.8730e-07
N                     2.1015e-11
NH                    7.3010e-13
NH3                   3.5644e-13
NH2                   1.4186e-13
HCO                   1.7907e-14
MINORS[+10]           0.0000e+00

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

Two plots should also appear, illustrating the equilibrium composition of the system as a function of the

equivalence ratio ϕ, and various properties (temperature, pressure, density, enthalpy, internal energy, Gibbs free energy, entropy, and adiabatic index) of the products as a function of ϕ.

Combustion Toolbox performs the calculations from the last case to the first one, and the results are stored in the `PS` property of the `self` structure.

Tip: The convergence tolerances for all the algorithms implemented in the Combustion Toolbox are stored in the TN (tuning) property within the self structure. Default values are specified in TuningProperties.m. For instance, the default tolerance for molar composition is 10^{-14}, for the multi-dimensional Newton-Raphson solver (where temperature is known) it is 10^{-5}, and for the Newton-Raphson solver (when temperature is not known) used to satisfy conservation equation for enthalpy, internal energy, or entropy, the tolerance is set to 10^{-4}.

These parameters ca be easily adjusted by setting the corresponding property in the `TN` structure. For example, to decrease the tolerance (increase precision) for molar composition to 10^{-32}, type

```
self.TN.tolN = 1e-32;
```

prior to solving the problem. To visualize these changes (command window and default plots), execute:

```
self.C.mintol_display = 1e-32;
```

Summary

```
% Initialize CT and define chemical system
self = App('Soot formation');

% Define initial chemical composition
self.PD.S_Fuel     = {'C2H2_acetylene'};
self.PD.S_Oxidizer = {'N2', 'O2'};
self.PD.ratio_oxidizers_O2 = [79, 21]/21;
self = set_prop(self, 'phi', 0.5:0.01:5);

% Define initial thermochemical state
self = set_prop(self, 'TR', 300, 'pR', 1 * 1.01325);

% Set chemical transformation
self = set_prop(self, 'pP', self.PD.pR.value);

% Perform chemical calculations
self = solve_problem(self, 'HP');
```

(continues on next page)

(continued from previous page)

```
% Postprocess results
post_results(self);
```

Congratulations!

Congratulations you have finished the Combustion Toolbox MATLAB tutorial! You should now be ready to begin using the Combustion Toolbox on your own (see the `examples` folder).

CHAPTER

FIVE

EXAMPLES

Note: Please note that this website is currently under development. In the meantime, we have provided a collection of examples already included in the Combustion Toolbox repository for your convenience.

Combustion Toolbox can be used through:

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

5.1 Examples GUI

In this section, we will highlight various computations that can be easily performed using the Combustion Toolbox GUI.

Warning: This section is currently under development. We apologize for any inconvenience this may cause.

5.2 Examples desktop environment

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

5.2.1 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 24 species considered and a set of
% equivalence ratios phi contained in (0.5, 5) [-]
%
% Soot formation == {'CO2','CO','H2O','H2','O2','N2','Ar','Cbgrb',...
%                    'C2','C2H4','CH','CH3','CH4','CN','H',...
%                    'HCN','HCO','N','NH','NH2','NH3','NO','O','OH'}
%
% See wiki or list_species() for more predefined sets of species
%
% @author: Alberto Cuadra Lara
%          PhD Candidate - Group Fluid Mechanics
%          Universidad Carlos III de Madrid
%
% Last update July 22 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 = {'N2', 'O2', 'Ar', 'CO2'};
self.PD.ratio_oxidizers_O2 = [78.084, 20.9476, 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 = solve_problem(self, 'TP');
%% DISPLAY RESULTS (PLOTS)
post_results(self);
```

5.2.2 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 24 species considered and a set of equivalence ratios (phi)
% contained in (0.5, 5) [-]
%
% Soot formation == {'CO2','CO','H2O','H2','O2','N2','Ar','Cbgrb',...
%                    'C2','C2H4','CH','CH3','CH4','CN','H',...
%                    'HCN','HCO','N','NH','NH2','NH3','NO','O','OH'}
%
% See wiki or list_species() for more predefined sets of species
%
% @author: Alberto Cuadra Lara
%          PhD Candidate - Group Fluid Mechanics
%          Universidad Carlos III de Madrid
%
% Last update July 22 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 = {'N2', 'O2', 'Ar', 'CO2'};
self.PD.ratio_oxidizers_O2 = [78.084, 20.9476, 0.9365, 0.0319] ./ 20.9476;
%% ADDITIONAL INPUTS (DEPENDS OF THE PROBLEM SELECTED)
self = set_prop(self, 'TP', 3000);
%% SOLVE PROBLEM
self = solve_problem(self, 'TV');
%% DISPLAY RESULTS (PLOTS)
post_results(self);
```

5.2.3 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 24 species considered and a set of
% equivalence ratios phi contained in (0.5, 5) [-]
%
% Soot formation == {'CO2','CO','H2O','H2','O2','N2','Ar','Cbgrb',...
%                    'C2','C2H4','CH','CH3','CH4','CN','H',...
%                    'HCN','HCO','N','NH','NH2','NH3','NO','O','OH'}
%
% See wiki or list_species() for more predefined sets of species
%
% @author: Alberto Cuadra Lara
%          PhD Candidate - Group Fluid Mechanics
%          Universidad Carlos III de Madrid
%
% Last update July 22 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 = {'N2', 'O2', 'Ar', 'CO2'};
self.PD.ratio_oxidizers_O2 = [78.084, 20.9476, 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 = solve_problem(self, 'HP');
%% DISPLAY RESULTS (PLOTS)
post_results(self);
```

5.2.4 Example_HP_COMPLETE_INCOMPLETE.m

```
% -------------------------------------------------------------------------
% EXAMPLE: HP COMPLETE VS INCOMPLETE
%
% Compute adiabatic temperature and equilibrium composition at constant
% pressure (e.g., 1.01325 bar) for lean to rich natural gas-air mixtures at
% standard conditions, a set of equivalence ratios phi contained in
% (0.5, 3) [-], and two different sets of species "Complete" or "Incomplete"
%
% * Complete:
%       - lean = {'CO2', 'H2O', 'N2', 'Ar', 'O2'};                (equivalence ratio
↪< 1)
%       - rich = {'CO2', 'H2O', 'N2', 'Ar', 'CO', 'H2'};          (equivalence ratio >␣
↪1)
%       - soot = {'N2', 'Ar', 'CO', 'H2', 'Cbgrb', 'CO2', 'H2O'}; (equivalence ratio >␣
↪equivalence ratio soot)
%
% * Incomplete:
%       - Soot formation == {'CO2','CO','H2O','H2','O2','N2','Ar','Cbgrb',...
%                           'C2','C2H4','CH','CH3','CH4','CN','H',...
%                           'HCN','HCO','N','NH','NH2','NH3','NO','O','OH'}
%
% See wiki or list_species() for more predefined sets of species
%
% @author: Alberto Cuadra Lara
%          PhD Candidate - Group Fluid Mechanics
%          Universidad Carlos III de Madrid
%
% Last update Oct 13 2022
% -------------------------------------------------------------------------

%% COMPLETE COMBUSTION
% Initialization
self = App('Complete');
% Set fuel composition
self.PD.S_Fuel = {'CH4', 'C2H6', 'C3H8', 'C4H10_isobutane', 'H2'};
self.PD.N_Fuel = [0.8, 0.05, 0.05, 0.05, 0.05];
% Set oxidizer composition
self = set_air(self, false);
% Set temperature, pressure and equivalence ratio
self = set_prop(self, 'TR', 300, 'pR', 1.01325, 'phi', 0.5:0.02:3);
```

(continues on next page)

(continued from previous page)

```
% Set constant pressure for products
self = set_prop(self, 'pP', self.PD.pR.value);
% Solve Problem
self = solve_problem(self, 'HP');
% Save results
results_complete = self;
%% INCOMPLETE COMBUSTION
% Set product species at equilibrium
self = App('copy', self, 'Soot formation');
% Solve Problem
self = solve_problem(self, 'HP');
% Save results
results_incomplete = self;
%% COMPARE RESULTS
mix1 = results_complete.PS.strR; % Is the same as the incomplete case (same␣
↪initial mixture)
mix2_complete = results_complete.PS.strP;
mix2_incomplete = results_incomplete.PS.strP;

phi = cell2vector(mix1, 'phi');

self.Misc.config.title = '\rm{Complete\ vs\ Incomplete}';
self.Misc.config.labelx = 'Equivalence ratio $\phi$';
self.Misc.config.labely = 'Temperature $T$ [K]';
legend_name = {'Complete', 'Incomplete'};

ax = plot_figure('phi', phi, 'T', mix2_complete, 'config', self.Misc.config);
ax = plot_figure('phi', phi, 'T', mix2_incomplete, 'config', self.Misc.config,
↪'ax', ax, 'legend', legend_name, 'color', 'auto');
ax.Legend.Location = 'northeast';
```

5.2.5 Example_HP_HUMID_AIR.m

```
% -------------------------------------------------------------------------
% EXAMPLE: HP - Humid air
%
% Compute adiabatic temperature and equilibrium composition at constant
% pressure (e.g., 1.01325 bar) for lean to rich CH4-ideal_air mixtures at
% standard conditions, a set of 10 species considered, and excess of air
% of 15%, and a set of relative humidity (0, 20, 40, 60, 80, 100) [%]
```

(continues on next page)

(continued from previous page)

```
%
% LS == {'CO2', 'CO', 'H2O', 'H2', 'O2', 'N2', 'NO', 'OH', 'O', 'H'}
%
% See wiki or list_species() for more predefined sets of species
%
% Results compared with [1]
%
% [1] Sánchez, Y. A. C., Piedrahita, C. A. R., & Serrano, E. G. F. (2014).
%     Influencia del aire húmedo en la combustión del metano. Scientia et
%     technica, 19(4), 364-370.
%
% @author: Alberto Cuadra Lara
%          Postdoctoral researcher - Group Fluid Mechanics
%          Universidad Carlos III de Madrid
%
% Last update Feb 21 2024
% ------------------------------------------------------------------------

% User inputs
air_excess = 15;              % [%]
humidity_relative = 0:20:100; % [%]
T = [10:2:60] + 273.15;       % [K]
p = 1 * 1.01325;              % [bar]

% Initialization
self = App();
DB_master = self.DB_master;
DB = self.DB;
LS = {'CO2', 'CO', 'H2O', 'H2', 'O2', 'N2', 'NO', 'OH', 'O', 'H'};

% Definitions
phi_t = 2; % stoichiometric number of moles of air per mole of fuel
phi = 1 / (1 + air_excess / 100); % equivalence ratio [-]
mm_air = self.DB.Air.mm; % [g/mol]
mm_H2O = self.DB.H2O.mm; % [g/mol]
n_air = 4.76; % total number of moles of stoichiometric ideal dry air (79% N2␣
↪and 21% O2 in volume)

% Miscellaneous
config = self.Misc.config;
config.innerposition = [0.2 0.2 0.35 0.5];  % Set figure inner position␣
```

(continues on next page)

(continued from previous page)

```
↪[normalized]
config.outerposition = [0.2 0.2 0.35 0.5];  % Set figure outer position␣
↪[normalized]
colors = brewermap(length(humidity_relative), 'Greys');
FLAG_FIRST = true;

for j = length(humidity_relative):-1:1
    % Get specific humidity
    W = humidity_specific(T, p, humidity_relative(j));
    % Get moles H2O
    n_H2O = W * n_air * (mm_air / mm_H2O);
    % Solve problem
    for i = length(T):-1:1
        % Initialization
        self = App('fast', DB_master, DB, LS);
        % Initial conditions
        self = set_prop(self, 'TR', T(i), 'pR', p, 'phi', phi);
        self.PD.S_Fuel     = {'CH4'};
        self.PD.S_Oxidizer = {'N2', 'O2', 'H2O'};
        self.PD.N_Oxidizer = phi_t ./ phi .* [79/21, 1, n_H2O(i)];
        % self.PD.ratio_oxidizers_O2 = [79, 21, nH2O(i) * 21] / 21;
        % Set additional inputs (depends of the problem selected)
        self = set_prop(self, 'pP', self.PD.pR.value);
        % Solve problem
        self = solve_problem(self, 'HP');
        % Get data
        mix1{i} = self.PS.strR{1};
        mix2{i} = self.PS.strP{1};
    end

    if FLAG_FIRST
        index_CO2 = find_ind(self.S.LS, 'CO2');
        index_H2O = find_ind(self.S.LS, 'H2O');
        index_CH4 = find_ind(self.S.LS, 'CH4');
        FLAG_FIRST = false;
    end

    % Extract data
    mR = cell2vector(mix1, 'mi');
    TR = cell2vector(mix1, 'T');
    Yi_R = cell2vector(mix1, 'Yi');
```

(continues on next page)

(continued from previous page)

```
    Yi_R_CH4 = Yi_R(index_CH4, :);

    mP = cell2vector(mix2, 'mi');
    TP = cell2vector(mix2, 'T');
    Yi_P = cell2vector(mix2, 'Yi');
    Xi_P = cell2vector(mix2, 'Xi');
    Yi_P_H2O = Yi_P(index_H2O, :);
    Xi_P_CO2 = Xi_P(index_CO2, :);

    m_CH4 = mR .* Yi_R_CH4;
    m_H2O = mP .* Yi_P_H2O;

    % Plot results
    if ~exist('ax1', 'var')
        ax1 = set_figure(config);
        ax2 = set_figure(config);
        ax3 = set_figure(config);
    end

    % Adiabatic flame temperature vs. reactant's temperature
    ax1 = plot_figure('T_R\ [\rm K]', TR, 'T_P\ [\rm K]', TP, 'ax', ax1, 'color
↪', colors(j, :));
    % Molar fraction of CO2 vs. reactant's temperature
    ax2 = plot_figure('T_R\ [\rm K]', TR, 'X_{\rm CO_2}', Xi_P_CO2, 'ax', ax2,
↪'color', colors(j, :));
    % Water content after the combustion process (kg H2O / kg fuel) vs.␣
↪reactant's temperature
    ax3 = plot_figure('T_R\ [\rm K]', TR, '\rm Water\ content\ [kg_{H_2O}/kg_
↪{CH_4}]', m_H2O./m_CH4, 'ax', ax3, 'color', colors(j, :));
end
```

5.2.6 Example_HP_PRESSURE.m

```
% -------------------------------------------------------------------------
% EXAMPLE: HP - EFFECT OF PRESSURE
%
% Compute adiabatic temperature and equilibrium composition at constant
% pressure (e.g., 1 bar) for lean to rich natural gas-air mixtures at
% standard conditions, a set of equivalence ratios phi contained in
% (0.5, 3.5) [-], and two different sets of species "Complete" or "Incomplete"
```

(continues on next page)

(continued from previous page)

```
% The incomplete combustion case is evaluated at different pressures to see
% Le Chatelier's principle, i.e., an increase of pressure shifts
% equilibrium to the side of the reaction with fewer number of moles of
% gas (higher pressures implie less dissociation). A comparison of the
% adiabatic flame temperature, adiabatic index, and Gibbs free energy is
% included.
%
% * Complete:
%       - lean = {'CO2', 'H2O', 'N2', 'Ar', 'O2'};                (equivalence ratio < 1)
%       - rich = {'CO2', 'H2O', 'N2', 'Ar', 'CO', 'H2'};          (equivalence ratio > 1)
%       - soot = {'N2', 'Ar', 'CO', 'H2', 'Cbgrb', 'CO2', 'H2O'}; (equivalence ratio > equivalence ratio soot)
%
% * Incomplete:
%       - Soot formation == {'CO2','CO','H2O','H2','O2','N2','Ar','Cbgrb',...
%                            'C2','C2H4','CH','CH3','CH4','CN','H',...
%                            'HCN','HCO','N','NH','NH2','NH3','NO','O','OH'}
%
% See wiki or list_species() for more predefined sets of species
%
% @author: Alberto Cuadra Lara
%          PhD Candidate - Group Fluid Mechanics
%          Universidad Carlos III de Madrid
%
% Last update July 29 2022
% -------------------------------------------------------------------------

%% COMPLETE COMBUSTION
% Initialization
self = App('Complete');
% Set fuel composition
self.PD.S_Fuel = {'CH4', 'C2H6', 'C3H8', 'C4H10_isobutane', 'H2'};
self.PD.N_Fuel = [0.8, 0.05, 0.05, 0.05, 0.05];
% Set oxidizer composition
self = set_air(self, false);
% Set temperature, pressure and equivalence ratio
self = set_prop(self, 'TR', 300, 'pR', 1, 'phi', 0.5:0.005:3.5);
% Set constant pressure for products
self = set_prop(self, 'pP', self.PD.pR.value);
```

(continues on next page)

(continued from previous page)

```
% Solve Problem
self = solve_problem(self, 'HP');
% Save results
results_complete = self;
%% INCOMPLETE COMBUSTION
pressure_vector = self.PD.pR.value * [1, 10, 100];
for i = length(pressure_vector):-1:1
    % Set product species at equilibrium
    self = App('copy', self, 'Soot formation');
    % Set pressure
    self = set_prop(self, 'pP', pressure_vector(i), 'pP', pressure_vector(i));
    % Solve Problem
    self = solve_problem(self, 'HP');
    % Save results
    results_incomplete{i} = self;
end
%% Comparison of adiabatic flame temperature
mix1 = results_complete.PS.strR; % Is the same as the incomplete case (same␣
↪initial mixture)
mix2_complete = results_complete.PS.strP;

phi = cell2vector(mix1, 'phi');

self.Misc.config.title = '\rm{Complete\ vs\ Incomplete\ at\ different\␣
↪pressures}';
self.Misc.config.labelx = 'Equivalence ratio $\phi$';
self.Misc.config.labely = 'Temperature $T$ [K]';
legend_name = {'Complete'};

ax = plot_figure('phi', phi, 'T', mix2_complete, 'config', self.Misc.config);

for i = 1:length(pressure_vector)
    mix2_incomplete = results_incomplete{i}.PS.strP;
    ax = plot_figure('phi', phi, 'T', mix2_incomplete, 'config', self.Misc.
↪config, 'ax', ax, 'color', 'auto');
    legend_name(i+1) = {sprintf('$p_{%d} = %.4g$ bar', i, pressure(mix2_
↪incomplete{1}))};
end

set_legends(ax, legend_name, 'FLAG_SPECIES', false)
ax.Legend.Location = 'northeast';
```

(continues on next page)

(continued from previous page)

```
%% Comparison of the adiabatic index
self.Misc.config.labely = 'Adibatic index $\gamma_s$';

ax = plot_figure('phi', phi, 'gamma_s', mix2_complete, 'config', self.Misc.
↪config);

for i = 1:length(pressure_vector)
    mix2_incomplete = results_incomplete{i}.PS.strP;
    ax = plot_figure('phi', phi, 'gamma_s', mix2_incomplete, 'config', self.
↪Misc.config, 'ax', ax, 'color', 'auto');
end

set_legends(ax, legend_name, 'FLAG_SPECIES', false)
ax.Legend.Location = 'best';
%% Comparison of the Gibbs energy
self.Misc.config.labely = 'Gibbs free energy $g$ [kJ/kg]';

ax = plot_figure('phi', phi, 'g', mix2_complete, 'config', self.Misc.config);

for i = 1:length(pressure_vector)
    mix2_incomplete = results_incomplete{i}.PS.strP;
    ax = plot_figure('phi', phi, 'g', mix2_incomplete, 'config', self.Misc.
↪config, 'ax', ax, 'color', 'auto');
end

set_legends(ax, legend_name, 'FLAG_SPECIES', false)
ax.Legend.Location = 'best';
```

5.2.7 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 24 species considered and a set of equivalence ratios phi contained
% in (0.5, 5) [-]
%
% Soot formation == {'CO2','CO','H2O','H2','O2','N2','Ar','Cbgrb',...
```

(continues on next page)

(continued from previous page)

```
%                   'C2','C2H4','CH','CH3','CH4','CN','H',...
%                   'HCN','HCO','N','NH','NH2','NH3','NO','O','OH'}
%
% See wiki or list_species() for more predefined sets of species
%
% @author: Alberto Cuadra Lara
%          PhD Candidate - Group Fluid Mechanics
%          Universidad Carlos III de Madrid
%
% Last update July 22 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 = {'N2', 'O2', 'Ar', 'CO2'};
self.PD.ratio_oxidizers_O2 = [78.084, 20.9476, 0.9365, 0.0319] ./ 20.9476;

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

5.2.8 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',...
```

(continues on next page)

(continued from previous page)

```
%                'H2bLb','O2bLb'}
%
% See wiki or list_species() 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('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 = solve_problem(self, 'HP');
%% DISPLAY RESULTS (PLOTS)
post_results(self);
```

5.2.9 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 24 species considered and a set of equivalence ratios (phi) contained
% in (0.5, 5) [-]
%
% Soot formation == {'CO2','CO','H2O','H2','O2','N2','Ar','Cbgrb',...
%                    'C2','C2H4','CH','CH3','CH4','CN','H',...
%                    'HCN','HCO','N','NH','NH2','NH3','NO','O','OH'}
%
% See wiki or list_species() for more predefined sets of species
%
```

(continues on next page)

(continued from previous page)

```
% @author: Alberto Cuadra Lara
%          PhD Candidate - Group Fluid Mechanics
%          Universidad Carlos III de Madrid
%
% Last update July 22 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 = {'N2', 'O2', 'Ar', 'CO2'};
self.PD.ratio_oxidizers_O2 = [78.084, 20.9476, 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 = solve_problem(self, 'EV');
%% DISPLAY RESULTS (PLOTS)
post_results(self);
```

5.2.10 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 24 species considered, and a equivalence
% ratio phi 1.5 [-]
%
% Soot formation == {'CO2','CO','H2O','H2','O2','N2','Ar','Cbgrb',...
%                    'C2','C2H4','CH','CH3','CH4','CN','H',...
%                    'HCN','HCO','N','NH','NH2','NH3','NO','O','OH'}
%
% See wiki or list_species() for more predefined sets of species
%
% @author: Alberto Cuadra Lara
%          PhD Candidate - Group Fluid Mechanics
%          Universidad Carlos III de Madrid
%
```

(continues on next page)

(continued from previous page)

```
% Last update July 22 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 = {'N2', 'O2', 'Ar', 'CO2'};
self.PD.ratio_oxidizers_O2 = [78.084, 20.9476, 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 = solve_problem(self, 'SP');
%% DISPLAY RESULTS (PLOTS)
post_results(self);
```

5.2.11 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 24 species considered, and a equivalence
% ratio phi 0.5 [-]
%
% Soot formation == {'CO2','CO','H2O','H2','O2','N2','Ar','Cbgrb',...
%                    'C2','C2H4','CH','CH3','CH4','CN','H',...
%                    'HCN','HCO','N','NH','NH2','NH3','NO','O','OH'}
%
% See wiki or list_species() for more predefined sets of species
%
% @author: Alberto Cuadra Lara
%          PhD Candidate - Group Fluid Mechanics
%          Universidad Carlos III de Madrid
%
% Last update July 22 2022
% -------------------------------------------------------------------------

%% INITIALIZE
```

(continues on next page)

(continued from previous page)

```
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 = {'N2', 'O2', 'Ar', 'CO2'};
self.PD.ratio_oxidizers_O2 = [78.084, 20.9476, 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 = solve_problem(self, 'SV');
%% DISPLAY RESULTS (PLOTS)
post_results(self);
```

5.2.12 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 list_species() for more predefined sets of species
%
% @author: Alberto Cuadra Lara
%          PhD Candidate - Group Fluid Mechanics
%          Universidad Carlos III de Madrid
%
% Last update July 22 2022
% -------------------------------------------------------------------------

%% INITIALIZE
self = App({'CH4', 'O2', 'N2', 'Ar', 'CO2'});
%% INITIAL CONDITIONS
self = set_prop(self, 'TR', 700, 'pR', 10, 'phi', 0.5);
self.PD.S_Fuel     = {'CH4'};
self.PD.S_Oxidizer = {'N2', 'O2', 'Ar', 'CO2'};
self.PD.ratio_oxidizers_O2 = [78.084, 20.9476, 0.9365, 0.0319] ./ 20.9476;
%% ADDITIONAL INPUTS (DEPENDS OF THE PROBLEM SELECTED)
```

(continues on next page)

(continued from previous page)

```
self = set_prop(self, 'vP_vR', 0.5:0.01:2);
%% SOLVE PROBLEM
self = solve_problem(self, 'SV');
%% DISPLAY RESULTS (PLOTS)
post_results(self);
```

5.2.13 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 16 species considered and a set of
% initial shock front velocities (u1) contained in (sound velocity, 20000) [m/s]
%
% Air == {'O2','N2','O','O3','N','NO','NO2','NO3','N2O','N2O3','N2O4',...
%         'N3','C','CO','CO2','Ar'}
%
% See wiki or list_species() for more predefined sets of species
%
% @author: Alberto Cuadra Lara
%          PhD Candidate - Group Fluid Mechanics
%          Universidad Carlos III de Madrid
%
% Last update July 22 2022
% -------------------------------------------------------------------------

%% INITIALIZE
self = App('Air');
%% INITIAL CONDITIONS
self = set_prop(self, 'TR', 300, 'pR', 1 * 1.01325);
self.PD.S_Oxidizer = {'N2', 'O2', 'Ar', 'CO2'};
self.PD.N_Oxidizer = [78.084, 20.9476, 0.9365, 0.0319] ./ 20.9476;
sound_velocity = compute_sound(self.PD.TR.value, self.PD.pR.value, self.PD.S_Oxidizer, self.PD.N_Oxidizer, 'self', self);
%% ADDITIONAL INPUTS (DEPENDS OF THE PROBLEM SELECTED)
u1 = logspace(2, 5, 500); u1 = u1(u1 < 20000); u1 = u1(u1 >= 1 * sound_velocity);
self = set_prop(self, 'u1', u1);
```

(continues on next page)

(continued from previous page)

```
%% SOLVE PROBLEM
self = solve_problem(self, 'SHOCK_I');
%% DISPLAY RESULTS (PLOTS)
post_results(self);
```

5.2.14 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 51 species considered and a set of
% initial shock front velocities (u1) contained in (360, 13000) [m/s]
%
% Air_ions == {'eminus', 'Ar', 'Arplus', 'C', 'Cplus', 'Cminus', ...
%              'CN', 'CNplus', 'CNminus', 'CNN', 'CO', 'COplus', ...
%              'CO2', 'CO2plus', 'C2', 'C2plus', 'C2minus', 'CCN', ...
%              'CNC', 'OCCN', 'C2N2', 'C2O', 'C3', 'C3O2', 'N', ...
%              'Nplus', 'Nminus', 'NCO', 'NO', 'NOplus', 'NO2', ...
%              'NO2minus', 'NO3', 'NO3minus', 'N2', 'N2plus', ...
%              'N2minus', 'NCN', 'N2O', 'N2Oplus', 'N2O3', 'N2O4', ...
%              'N2O5', 'N3', 'O', 'Oplus', 'Ominus', 'O2', 'O2plus', ...
%              'O2minus', 'O3'}
%
% See wiki or list_species() for more predefined sets of species
%
% @author: Alberto Cuadra Lara
%          PhD Candidate - Group Fluid Mechanics
%          Universidad Carlos III de Madrid
%
% Last update July 22 2022
% -------------------------------------------------------------------------

%% INITIALIZE
self = App('Air_ions');
%% INITIAL CONDITIONS
self = set_prop(self, 'TR', 300, 'pR', 1 * 1.01325);
self.PD.S_Oxidizer = {'N2', 'O2', 'Ar', 'CO2'};
self.PD.N_Oxidizer = [78.084, 20.9476, 0.9365, 0.0319] ./ 20.9476;
%% ADDITIONAL INPUTS (DEPENDS OF THE PROBLEM SELECTED)
```

(continues on next page)

(continued from previous page)

```
u1 = logspace(2, 5, 500); u1 = u1(u1<20000); u1 = u1(u1>=347.25);
self = set_prop(self, 'u1', u1);
%% SOLVE PROBLEM
self = solve_problem(self, 'SHOCK_I');
%% DISPLAY RESULTS (PLOTS)
post_results(self);
```

5.2.15 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 16 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'}
%
% See wiki or list_species() for more predefined sets of species
%
% @author: Alberto Cuadra Lara
%          PhD Candidate - Group Fluid Mechanics
%          Universidad Carlos III de Madrid
%
% Last update July 22 2022
% -------------------------------------------------------------------------

%% INITIALIZE
self = App('Air');
%% INITIAL CONDITIONS
self = set_prop(self, 'TR', 300, 'pR', 1 * 1.01325);
self.PD.S_Oxidizer = {'N2', 'O2', 'Ar', 'CO2'};
self.PD.N_Oxidizer = [78.084, 20.9476, 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 = solve_problem(self, 'SHOCK_R');
%% DISPLAY RESULTS (PLOTS)
```

(continues on next page)

(continued from previous page)

```
post_results(self);
```

5.2.16 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 51 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 == {'eminus', 'Ar', 'Arplus', 'C', 'Cplus', 'Cminus', ...
%              'CN', 'CNplus', 'CNminus', 'CNN', 'CO', 'COplus', ...
%              'CO2', 'CO2plus', 'C2', 'C2plus', 'C2minus', 'CCN', ...
%              'CNC', 'OCCN', 'C2N2', 'C2O', 'C3', 'C3O2', 'N', ...
%              'Nplus', 'Nminus', 'NCO', 'NO', 'NOplus', 'NO2', ...
%              'NO2minus', 'NO3', 'NO3minus', 'N2', 'N2plus', ...
%              'N2minus', 'NCN', 'N2O', 'N2Oplus', 'N2O3', 'N2O4', ...
%              'N2O5', 'N3', 'O', 'Oplus', 'Ominus', 'O2', 'O2plus', ...
%              'O2minus', 'O3'}
%
% See wiki or list_species() for more predefined sets of species
%
% @author: Alberto Cuadra Lara
%          PhD Candidate - Group Fluid Mechanics
%          Universidad Carlos III de Madrid
%
% Last update July 22 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 = {'N2', 'O2', 'Ar', 'CO2'};
self.PD.N_Oxidizer = [78.084, 20.9476, 0.9365, 0.0319] ./ 20.9476;
%% ADDITIONAL INPUTS (DEPENDS OF THE PROBLEM SELECTED)
```

(continues on next page)

(continued from previous page)

```
overdriven = 10;
self = set_prop(self, 'u1', 3.472107491008314e+02 * overdriven, 'beta', 20:5:
↪85);
%% SOLVE PROBLEM
self = solve_problem(self, 'SHOCK_OBLIQUE');
%% DISPLAY RESULTS (PLOTS)
post_results(self);
```

5.2.17 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 51 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 == {'eminus', 'Ar', 'Arplus', 'C', 'Cplus', 'Cminus', ...
%              'CN', 'CNplus', 'CNminus', 'CNN', 'CO', 'COplus', ...
%              'CO2', 'CO2plus', 'C2', 'C2plus', 'C2minus', 'CCN', ...
%              'CNC', 'OCCN', 'C2N2', 'C2O', 'C3', 'C3O2', 'N', ...
%              'Nplus', 'Nminus', 'NCO', 'NO', 'NOplus', 'NO2', ...
%              'NO2minus', 'NO3', 'NO3minus', 'N2', 'N2plus', ...
%              'N2minus', 'NCN', 'N2O', 'N2Oplus', 'N2O3', 'N2O4', ...
%              'N2O5', 'N3', 'O', 'Oplus', 'Ominus', 'O2', 'O2plus', ...
%              'O2minus', 'O3'}
%
% See wiki or list_species() for more predefined sets of species
%
% @author: Alberto Cuadra Lara
%          PhD Candidate - Group Fluid Mechanics
%          Universidad Carlos III de Madrid
%
% Last update July 22 2022
% -------------------------------------------------------------------------

%% INITIALIZE
self = App('Air_ions');
%% INITIAL CONDITIONS
```

(continues on next page)

(continued from previous page)

```
self = set_prop(self, 'TR', 300, 'pR', 1 * 1.01325);
self.PD.S_Oxidizer = {'N2', 'O2', 'Ar', 'CO2'};
self.PD.N_Oxidizer = [78.084, 20.9476, 0.9365, 0.0319] ./ 20.9476;
%% ADDITIONAL INPUTS (DEPENDS OF THE PROBLEM SELECTED)
Mach_number = 5;
self = set_prop(self, 'u1', 3.472107491008314e+02 * Mach_number, 'theta', 5:5:
↪40);
%% SOLVE PROBLEM
self = solve_problem(self, 'SHOCK_OBLIQUE');
%% DISPLAY RESULTS (PLOTS)
post_results(self);
```

5.2.18 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 51 species
% considered, a initial shock front velocities u1 = a1 * 10 [m/s], and a
% deflection angle theta = 20 [deg]
%
% Air_ions == {'eminus', 'Ar', 'Arplus', 'C', 'Cplus', 'Cminus', ...
%              'CN', 'CNplus', 'CNminus', 'CNN', 'CO', 'COplus', ...
%              'CO2', 'CO2plus', 'C2', 'C2plus', 'C2minus', 'CCN', ...
%              'CNC', 'OCCN', 'C2N2', 'C2O', 'C3', 'C3O2', 'N', ...
%              'Nplus', 'Nminus', 'NCO', 'NO', 'NOplus', 'NO2', ...
%              'NO2minus', 'NO3', 'NO3minus', 'N2', 'N2plus', ...
%              'N2minus', 'NCN', 'N2O', 'N2Oplus', 'N2O3', 'N2O4', ...
%              'N2O5', 'N3', 'O', 'Oplus', 'Ominus', 'O2', 'O2plus', ...
%              'O2minus', 'O3'}
%
% See wiki or list_species() for more predefined sets of species
%
% @author: Alberto Cuadra Lara
%          PhD Candidate - Group Fluid Mechanics
%          Universidad Carlos III de Madrid
%
% Last update July 22 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 = {'N2', 'O2', 'Ar', 'CO2'};
self.PD.N_Oxidizer = [78.084, 20.9476, 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
self = solve_problem(self, 'SHOCK_OBLIQUE_R');
%% DISPLAY RESULTS (PLOTS)
post_results(self);
```

5.2.19 Example_SHOCK_POLAR.m

```
% -------------------------------------------------------------------------
% EXAMPLE: SHOCK_POLAR
%
% Compute shock polar plots at standard conditions, a set of 51 species
% considered, and a set of initial shock front Mach numbers = [2, 3, 5, 14]
%
% Air_ions == {'eminus', 'Ar', 'Arplus', 'C', 'Cplus', 'Cminus', ...
%              'CN', 'CNplus', 'CNminus', 'CNN', 'CO', 'COplus', ...
%              'CO2', 'CO2plus', 'C2', 'C2plus', 'C2minus', 'CCN', ...
%              'CNC', 'OCCN', 'C2N2', 'C2O', 'C3', 'C3O2', 'N', ...
%              'Nplus', 'Nminus', 'NCO', 'NO', 'NOplus', 'NO2', ...
%              'NO2minus', 'NO3', 'NO3minus', 'N2', 'N2plus', ...
%              'N2minus', 'NCN', 'N2O', 'N2Oplus', 'N2O3', 'N2O4', ...
%              'N2O5', 'N3', 'O', 'Oplus', 'Ominus', 'O2', 'O2plus', ...
%              'O2minus', 'O3'}
%
% See wiki or list_species() for more predefined sets of species
%
% @author: Alberto Cuadra Lara
%          PhD Candidate - Group Fluid Mechanics
%          Universidad Carlos III de Madrid
```

(continues on next page)

(continued from previous page)

```
%
% Last update Jan 10 2023
% -------------------------------------------------------------------------

%% INITIALIZE
self = App('Air_ions');
%% INITIAL CONDITIONS
self = set_prop(self, 'TR', 300, 'pR', 1 * 1.01325);
self.PD.S_Oxidizer = {'N2', 'O2', 'Ar', 'CO2'};
self.PD.N_Oxidizer = [78.084, 20.9476, 0.9365, 0.0319] ./ 20.9476;
%% ADDITIONAL INPUTS (DEPENDS OF THE PROBLEM SELECTED)
self = set_prop(self, 'M1', [2, 3, 5, 14]);
%% SOLVE PROBLEM
self = solve_problem(self, 'SHOCK_POLAR');
%% DISPLAY RESULTS (PLOTS)
post_results(self);
```

5.2.20 Example_SHOCK_POLAR_R.m

```
% -------------------------------------------------------------------------
% EXAMPLE: SHOCK_POLAR_REFLECTED_THERMO
%
% Compute shock polar plots at T1 = 226.65 K and p1 = 0.0117 bar
% (altitude 30 km), a set of 28 species considered, an initial
% shock front Mach number = 20, deflection angle theta = 35, and different
% thermochemical models (chemical equilibrium and frozen chemistry).
%
% LS== {'eminus', 'Ar', 'Arplus', 'N', ...
%       'Nplus', 'Nminus', 'NO', 'NOplus', 'NO2', ...
%       'NO2minus', 'NO3', 'NO3minus', 'N2', 'N2plus', ...
%       'N2minus', 'N2O', 'N2Oplus', 'N2O3', 'N2O4', ...
%       'N2O5', 'N3', 'O', 'Oplus', 'Ominus', 'O2', 'O2plus', ...
%       'O2minus', 'O3'}
%
% See wiki or list_species() for more predefined sets of species
%
% @author: Alberto Cuadra Lara
%          PhD Candidate - Group Fluid Mechanics
%          Universidad Carlos III de Madrid
%
```

(continues on next page)

(continued from previous page)

```
% Last update Apr 13 2023
% -------------------------------------------------------------------------

LS = {'eminus', 'Ar', 'Arplus', 'N', ...
      'Nplus', 'Nminus', 'NO', 'NOplus', 'NO2', ...
      'NO2minus', 'NO3', 'NO3minus', 'N2', 'N2plus', ...
      'N2minus', 'N2O', 'N2Oplus', 'N2O3', 'N2O4', ...
      'N2O5', 'N3', 'O', 'Oplus', 'Ominus', 'O2', 'O2plus', ...
      'O2minus', 'O3'};

% Initialize
self = App(LS);
% Miscellaneous
self.TN.FLAG_FAST = true;
self.Misc.FLAG_RESULTS = false;
if FLAG_FROZEN
    self.Misc.config.linestyle = '--';
end
% Thermochemical model
self.PD.FLAG_FROZEN = false;
% Initial conditions
self = set_prop(self, 'TR', 226.65, 'pR', 0.0117);
self.PD.S_Oxidizer = {'N2', 'O2', 'Ar'};
self.PD.N_Oxidizer = [78, 21, 1] ./ 21;
% Additional inputs (depends of the problem selected)
self = set_prop(self, 'M1', 20, 'theta', 35);
% Solve problem
self = solve_problem(self, 'SHOCK_POLAR_R');
% Display results (plots)
post_results(self);
```

5.2.21 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 24 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','Ar','Cbgrb',...
%                    'C2','C2H4','CH','CH3','CH4','CN','H',...
%                    'HCN','HCO','N','NH','NH2','NH3','NO','O','OH'}
%
% See wiki or list_species() for more predefined sets of species
%
% @author: Alberto Cuadra Lara
%          PhD Candidate - Group Fluid Mechanics
%          Universidad Carlos III de Madrid
%
% Last update July 22 2022
% -------------------------------------------------------------------------

%% 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 = {'N2', 'O2', 'Ar', 'CO2'};
self.PD.ratio_oxidizers_O2 = [78.084, 20.9476, 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 = solve_problem(self, 'DET');
%% DISPLAY RESULTS (PLOTS)
post_results(self);
```

5.2.22 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 24 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','Ar','Cbgrb',...
%                    'C2','C2H4','CH','CH3','CH4','CN','H',...
%                    'HCN','HCO','N','NH','NH2','NH3','NO','O','OH'}
%
% See wiki or list_species() for more predefined sets of species
%
% @author: Alberto Cuadra Lara
%          PhD Candidate - Group Fluid Mechanics
%          Universidad Carlos III de Madrid
%
% Last update July 22 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 = {'N2', 'O2', 'Ar', 'CO2'};
self.PD.ratio_oxidizers_O2 = [78.084, 20.9476, 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 = solve_problem(self, 'DET_R');
%% DISPLAY RESULTS (PLOTS)
post_results(self);
```

5.2.23 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','Ar','Cbgrb',...
```

(continues on next page)

(continued from previous page)

```
%                   'C2','C2H4','CH','CH3','CH4','CN','H',...
%                   'HCN','HCO','N','NH','NH2','NH3','NO','O','OH'}
%
% See wiki or list_species() for more predefined sets of species
%
% @author: Alberto Cuadra Lara
%          PhD Candidate - Group Fluid Mechanics
%          Universidad Carlos III de Madrid
%
% Last update July 22 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 = {'N2', 'O2', 'Ar', 'CO2'};
self.PD.ratio_oxidizers_O2 = [78.084, 20.9476, 0.9365, 0.0319] ./ 20.9476;
%% ADDITIONAL INPUTS (DEPENDS OF THE PROBLEM SELECTED)
drive_factor = 1:0.1:10;
self = set_prop(self, 'drive_factor', drive_factor);
%% SOLVE PROBLEM
self = solve_problem(self, 'DET_OVERDRIVEN');
%% DISPLAY RESULTS (PLOTS)
post_results(self);
```

5.2.24 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 24 species considered
% and a set of overdrives contained in (1,10) [-].
%
% Soot formation == {'CO2','CO','H2O','H2','O2','N2','Ar','Cbgrb',...
%                   'C2','C2H4','CH','CH3','CH4','CN','H',...
%                   'HCN','HCO','N','NH','NH2','NH3','NO','O','OH'}
```

(continues on next page)

(continued from previous page)

```
%
% See wiki or list_species() for more predefined sets of species
%
% @author: Alberto Cuadra Lara
%          PhD Candidate - Group Fluid Mechanics
%          Universidad Carlos III de Madrid
%
% Last update July 22 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 = {'N2', 'O2', 'Ar', 'CO2'};
self.PD.ratio_oxidizers_O2 = [78.084, 20.9476, 0.9365, 0.0319] ./ 20.9476;
%% ADDITIONAL INPUTS (DEPENDS OF THE PROBLEM SELECTED)
drive_factor = 1:0.1:5;
self = set_prop(self, 'drive_factor', drive_factor);
%% SOLVE PROBLEM
self = solve_problem(self, 'DET_OVERDRIVEN_R');
%% DISPLAY RESULTS (PLOTS)
post_results(self);
```

5.2.25 Example_DET_UNDERDRIVEN.m

```
% -------------------------------------------------------------------------
% EXAMPLE: DET_UNDERDRIVEN
%
% Compute pre-shock and post-shock state for a planar under-driven detonation
% considering Chapman-Jouguet (CJ) theory for a stoichiometric CH4-air
% mixture at standard conditions, a set of 24 species considered and a set
% of underdrives contained in (1,10) [-].
%
% Soot formation == {'CO2','CO','H2O','H2','O2','N2','Ar','Cbgrb',...
%                    'C2','C2H4','CH','CH3','CH4','CN','H',...
%                    'HCN','HCO','N','NH','NH2','NH3','NO','O','OH'}
%
% See wiki or list_species() for more predefined sets of species
```

(continues on next page)

(continued from previous page)

```
%
% @author: Alberto Cuadra Lara
%          PhD Candidate - Group Fluid Mechanics
%          Universidad Carlos III de Madrid
%
% Last update July 22 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 = {'N2', 'O2', 'Ar', 'CO2'};
self.PD.ratio_oxidizers_O2 = [78.084, 20.9476, 0.9365, 0.0319] ./ 20.9476;
%% ADDITIONAL INPUTS (DEPENDS OF THE PROBLEM SELECTED)
drive_factor = 1:0.1:10;
self = set_prop(self, 'drive_factor', drive_factor);
%% SOLVE PROBLEM
self = solve_problem(self, 'DET_UNDERDRIVEN');
%% DISPLAY RESULTS (PLOTS)
post_results(self);
```

5.2.26 Example_DET_UNDERDRIVEN_R.m

```
% -------------------------------------------------------------------------
% EXAMPLE: DET UNDERDRIVEN REFLECTED
%
% Compute pre-shock and post-shock state for a reflected planar underdriven
% detonation considering Chapman-Jouguet (CJ) theory for a stoichiometric
% CH4-air mixture at standard conditions, a set of 24 species considered
% and a set of overdrives contained in (1,10) [-].
%
% Soot formation == {'CO2','CO','H2O','H2','O2','N2','Ar','Cbgrb',...
%                    'C2','C2H4','CH','CH3','CH4','CN','H',...
%                    'HCN','HCO','N','NH','NH2','NH3','NO','O','OH'}
%
% See wiki or list_species() for more predefined sets of species
%
% @author: Alberto Cuadra Lara
```

(continues on next page)

(continued from previous page)

```
%          PhD Candidate - Group Fluid Mechanics
%          Universidad Carlos III de Madrid
%
% Last update Oct 07 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 = {'N2', 'O2', 'Ar', 'CO2'};
self.PD.ratio_oxidizers_O2 = [78.084, 20.9476, 0.9365, 0.0319] ./ 20.9476;
%% ADDITIONAL INPUTS (DEPENDS OF THE PROBLEM SELECTED)
drive_factor = 1:0.1:5;
self = set_prop(self, 'drive_factor', drive_factor);
%% SOLVE PROBLEM
self = solve_problem(self, 'DET_UNDERDRIVEN_R');
%% DISPLAY RESULTS (PLOTS)
post_results(self);
```

5.2.27 Example_DET_OVERDRIVEN_AND_UNDERDRIVEN.m

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

(continues on next page)

(continued from previous page)

```
%
% Last update Oct 12 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 = {'N2', 'O2', 'Ar', 'CO2'};
self.PD.ratio_oxidizers_O2 = [78.084, 20.9476, 0.9365, 0.0319] ./ 20.9476;
%% ADDITIONAL INPUTS (DEPENDS OF THE PROBLEM SELECTED)
drive_factor = 1:0.1:10;
self = set_prop(self, 'drive_factor', drive_factor);
%% SOLVE PROBLEMS
self = solve_problem(self, 'DET_OVERDRIVEN');
self_under = solve_problem(self, 'DET_UNDERDRIVEN');
%% APPEND RESULTS
self.PS.strR = append_cells(flip(self_under.PS.strR), self.PS.strR);
self.PS.strP = append_cells(flip(self_under.PS.strP), self.PS.strP);
%% DISPLAY RESULTS (PLOTS)
post_results(self);
```

5.2.28 Example_DET_OBLIQUE_BETA.m

```
% -------------------------------------------------------------------------
% EXAMPLE: DET_OBLIQUE_BETA
%
% Compute pre-shock and post-shock state for a oblique 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 wave angles [15:5:80] [deg].
%
% Soot formation == {'CO2','CO','H2O','H2','O2','N2','Ar','Cbgrb',...
%                    'C2','C2H4','CH','CH3','CH4','CN','H',...
%                    'HCN','HCO','N','NH','NH2','NH3','NO','O','OH'}
%
% See wiki or list_species() for more predefined sets of species
%
% @author: Alberto Cuadra Lara
```

(continues on next page)

(continued from previous page)

```
%          PhD Candidate - Group Fluid Mechanics
%          Universidad Carlos III de Madrid
%
% Last update Oct 07 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 = {'N2', 'O2', 'Ar', 'CO2'};
self.PD.ratio_oxidizers_O2 = [78.084, 20.9476, 0.9365, 0.0319] ./ 20.9476;
%% ADDITIONAL INPUTS (DEPENDS OF THE PROBLEM SELECTED)
drive_factor = 4;
self = set_prop(self, 'drive_factor', drive_factor, 'beta', 15:5:80);
%% SOLVE PROBLEM
self = solve_problem(self, 'DET_OBLIQUE');
%% DISPLAY RESULTS (PLOTS)
post_results(self);
```

5.2.29 Example_DET_OBLIQUE_THETA.m

```
% -------------------------------------------------------------------------
% EXAMPLE: DET_OBLIQUE_THETA
%
% Compute pre-shock and post-shock state for a oblique 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 [15:5:50] [deg].
%
% Soot formation == {'CO2','CO','H2O','H2','O2','N2','Ar','Cbgrb',...
%                    'C2','C2H4','CH','CH3','CH4','CN','H',...
%                    'HCN','HCO','N','NH','NH2','NH3','NO','O','OH'}
%
% See wiki or list_species() for more predefined sets of species
%
% @author: Alberto Cuadra Lara
%          PhD Candidate - Group Fluid Mechanics
%          Universidad Carlos III de Madrid
```

(continues on next page)

(continued from previous page)

```
%
% Last update Nov 12 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 = {'N2', 'O2', 'Ar', 'CO2'};
self.PD.ratio_oxidizers_O2 = [78.084, 20.9476, 0.9365, 0.0319] ./ 20.9476;
%% ADDITIONAL INPUTS (DEPENDS OF THE PROBLEM SELECTED)
drive_factor = 4;
self = set_prop(self, 'drive_factor', drive_factor, 'theta', 15:5:50);
%% SOLVE PROBLEM
self = solve_problem(self, 'DET_OBLIQUE');
%% DISPLAY RESULTS (PLOTS)
post_results(self);
```

5.2.30 Example_DET_POLAR.m

```
% -------------------------------------------------------------------------
% EXAMPLE: DET_POLAR
%
% Compute detonation polar curves at standard conditions, a set of 30 species
% considered, and a set of pre-shock Mach numbers M1 = [5, 6, 7, 8, 10], or
% what is the same, drive factors M1/M1_cj = [1.0382, 1.2458, 1.4534,...
% 1.6611, 2.0763]
%
% Hydrogen == {'H2O','H2','O2','N2','He','Ar','H','HNO',...
%              'HNO3','NH','NH2OH','NO3','N2H2','N2O3','N3','OH',...
%              'HNO2','N','NH3','NO2','N2O','N2H4','N2O5','O','O3',...
%              'HO2','NH2','H2O2','N3H','NH2NO2'}
%
% See wiki or list_species() for more predefined sets of species
%
% @author: Alberto Cuadra Lara
%          PhD Candidate - Group Fluid Mechanics
%          Universidad Carlos III de Madrid
%
```

(continues on next page)

(continued from previous page)

```
% Last update Oct 07 2022
% -------------------------------------------------------------------------

%% INITIALIZE
self = App('Hydrogen');
%% INITIAL CONDITIONS
self = set_prop(self, 'TR', 300, 'pR', 1 * 1.01325, 'phi', 1);
self.PD.S_Fuel = {'H2'};
self.PD.S_Oxidizer = {'N2', 'O2'};
self.PD.ratio_oxidizers_O2 = [79, 21]/21;
%% ADDITIONAL INPUTS (DEPENDS OF THE PROBLEM SELECTED)
self = set_prop(self, 'drive_factor', [1.0382, 1.2458, 1.4534, 1.6611, 2.
↪0763]);
%% TUNING PROPERTIES
self.TN.N_points_polar = 300; % Number of points to compute polar curves
%% SOLVE PROBLEM
self = solve_problem(self, 'DET_POLAR');
%% DISPLAY RESULTS (PLOTS)
post_results(self);
```

5.2.31 Example_DET_POLAR_SONIC_AND_MAX.m

```
% -------------------------------------------------------------------------
% EXAMPLE: DET_POLAR_SONIC_AND_MAX
%
% Compute detonation polar curves at standard conditions, a set of 30 species
% considered, and a set of pre-shock Mach numbers M1 = [1.01*M1cj:15], or
% what is the same, drive factors M1/M1_cj = [1.01:2.9069]
%
% Hydrogen == {'H2O','H2','O2','N2','He','Ar','H','HNO',...
%              'HNO3','NH','NH2OH','NO3','N2H2','N2O3','N3','OH',...
%              'HNO2','N','NH3','NO2','N2O','N2H4','N2O5','O','O3',...
%              'HO2','NH2','H2O2','N3H','NH2NO2'}
%
% See wiki or list_species() for more predefined sets of species
%
% @author: Alberto Cuadra Lara
%          PhD Candidate - Group Fluid Mechanics
%          Universidad Carlos III de Madrid
%
```

(continues on next page)

(continued from previous page)

```
% Last update Oct 07 2022
% -------------------------------------------------------------------------

%% INITIALIZE
self = App('Hydrogen');
%% INITIAL CONDITIONS
self = set_prop(self, 'TR', 300, 'pR', 1 * 1.01325, 'phi', 1);
self.PD.S_Fuel = {'H2'};
self.PD.S_Oxidizer = {'N2', 'O2'};
self.PD.ratio_oxidizers_O2 = [79, 21]/21;
%% ADDITIONAL INPUTS (DEPENDS OF THE PROBLEM SELECTED)
self = set_prop(self, 'drive_factor', 1.01:0.01:2.9069);
%% TUNING PROPERTIES
self.TN.N_points_polar = 300;  % Number of points to compute shock polar
%% SOLVE PROBLEM
self = solve_problem(self, 'DET_POLAR');
%% GET POINTS
theta_sonic = cell2vector(self.PS.strP, 'theta_sonic');
beta_sonic = cell2vector(self.PS.strP, 'beta_sonic');
theta_max = cell2vector(self.PS.strP, 'theta_max');
beta_max = cell2vector(self.PS.strP, 'beta_max');

for i = length(self.PS.strP):-1:1
    postshock_u2n(:, i) = self.PS.strP{i}.polar.un;
    postshock_u2(:, i) = self.PS.strP{i}.polar.u;
    postshock_a(:, i) = self.PS.strP{i}.polar.sound;
    beta(:, i) = self.PS.strP{i}.polar.beta;
    theta(:, i) = self.PS.strP{i}.polar.theta;
end

M2 = postshock_u2 ./ postshock_a;
M2n = postshock_u2n ./ postshock_a;

for i = length(self.PS.strP):-1:1
    ind_row_sonic(i) =  find(M2(:, i) >= 1, 1);
    ind_row_sonic_n(i) =  find(M2n(:, i) >= 1, 1);
    beta_sonic_2(i) = beta(ind_row_sonic(i), i);
    theta_sonic_2(i) = theta(ind_row_sonic(i), i);
end
%% DISPLAY RESULTS (PLOTS)
post_results(self);
```

(continues on next page)

(continued from previous page)

```
%% SMOOTH RESULTS - FOURIER
start_point_sonic = [0 0 0 0 0 0.0508682282375078];
start_point_sonic_2 = [0 0 0 0 0 0.0661385531723257];
start_point_max = [0 0 0 0 0 0.0656724073958934];
[beta_sonic_smooth, theta_sonic_smooth] = smooth_data([90, beta_sonic], [0, theta_sonic], start_point_sonic);
[theta_max_smooth, beta_max_smooth] = smooth_data([0, theta_max], [90, beta_max], start_point_max);
[theta_sonic_2_smooth, beta_sonic_2_smooth] = smooth_data([0, theta_sonic_2], [90, beta_sonic_2], start_point_sonic_2);
%% PLOT
f = figure(2); ax = gca;
plot_figure('$\theta_{\rm M2n sonic}$', theta_sonic_smooth, '$\beta_{\rm M2n sonic}$', beta_sonic_smooth, 'linestyle', 'k:', 'color', 'auto', 'ax', ax)
plot_figure('$\theta_{\rm max}$', theta_max_smooth, '$\beta_{\rm max}$', beta_max_smooth, 'linestyle', 'k:', 'color', 'auto', 'ax', ax)
plot_figure('$\theta_{\rm M2 sonic}$', theta_sonic_2_smooth, '$\beta_{\rm M2 sonic}$', beta_sonic_2_smooth, 'linestyle', 'k:', 'color', 'auto', 'ax', ax)
```

5.2.32 Example_ROCKET_IAC.m

```
% -------------------------------------------------------------------------
% EXAMPLE: ROCKET Propellants considering an Infinite-Area-Chamber (IAC)
%
% Compute rocket propellant performance considering an Infinite-Area-Chamber
% for lean to rich LH2-LOX mixtures at 101.325 bar, a set of 11 species
% considered, a set of equivalence ratios phi contained in (2, 5) [-], and
% an exit area ratio A_exit/A_throat = 3
%
% HYDROGEN_L == {'H','H2O','OH','H2','O','O3','O2','HO2','H2O2',...
%                'H2bLb','O2bLb'}
%
% See wiki or list_species() for more predefined sets of species
%
% @author: Alberto Cuadra Lara
%          PhD Candidate - Group Fluid Mechanics
%          Universidad Carlos III de Madrid
%
% Last update Jul 30 2022
% -------------------------------------------------------------------------
```

(continues on next page)

(continued from previous page)

```

%% INITIALIZE
self = App('HYDROGEN_L');
%% INITIAL CONDITIONS
self = set_prop(self, 'TR', 90, 'pR', 100 * 1.01325, 'phi', 1:0.05:5);
self.PD.S_Fuel     = {'H2bLb'};
self.PD.S_Oxidizer = {'O2bLb'};
self.PD.FLAG_IAC = true;
%% ADDITIONAL INPUTS (DEPENDS OF THE PROBLEM SELECTED)
self = set_prop(self, 'Aratio', 3);
%% SOLVE PROBLEM
self = solve_problem(self, 'ROCKET');
%% DISPLAY RESULTS (PLOTS)
post_results(self);
```

5.2.33 Example_ROCKET_FAC.m

```
% -------------------------------------------------------------------------
% EXAMPLE: ROCKET Propellants considering an Finite-Area-Chamber (FAC)
%
% Compute rocket propellant performance considering an Finite-Area-Chamber
% with an area ratio of the combustion chamber A_chamber/A_throat = 2
% for lean to rich LH2-LOX mixtures at 101.325 bar, a set of 24 species
% considered, a set of equivalence ratios phi contained in (2, 5) [-], and
% an exit area ratio A_exit/A_throat = 3
%
% HYDROGEN_L == {'H','H2O','OH','H2','O','O3','O2','HO2','H2O2',...
%                'H2bLb','O2bLb'}
%
% See wiki or list_species() for more predefined sets of species
%
% @author: Alberto Cuadra Lara
%          PhD Candidate - Group Fluid Mechanics
%          Universidad Carlos III de Madrid
%
% Last update Jul 30 2022
% -------------------------------------------------------------------------

%% INITIALIZE
self = App('HYDROGEN_L');
```

(continues on next page)

(continued from previous page)

```
%% INITIAL CONDITIONS
self = set_prop(self, 'TR', 90, 'pR', 100 * 1.01325, 'phi', 1:0.05:5);
self.PD.S_Fuel     = {'H2bLb'};
self.PD.S_Oxidizer = {'O2bLb'};
self.PD.FLAG_IAC = false;
%% ADDITIONAL INPUTS (DEPENDS OF THE PROBLEM SELECTED)
self = set_prop(self, 'Aratio_c', 2, 'Aratio', 3);
%% SOLVE PROBLEM
self = solve_problem(self, 'ROCKET');
%% DISPLAY RESULTS (PLOTS)
post_results(self);
```

5.2.34 Example_EXOPLANET_WASP43b_1.m

```
% -------------------------------------------------------------------------
% EXAMPLE: EXOPLANET WASP-43b - METALLICITY 1
%
% Compute equilibrium vertical composition with a metallicity 1 of WASP-43b
%
% URL RESULTS TEA:
% https://github.com/dzesmin/RRC-BlecicEtal-2015a-ApJS-TEA/tree/master/Fig6/
↪WASP43b-solar
%
% @author: Alberto Cuadra Lara
%          PhD Candidate - Group Fluid Mechanics
%          Universidad Carlos III de Madrid
%
% Last update Oct 12 2022
% -------------------------------------------------------------------------

LS = {'C2H2_acetylene', 'C2H4', 'C', 'CH4', 'CO2', 'CO', 'H2', 'H2O', 'H2S', 'H
↪', 'HCN', 'He', 'HS_M', 'N2', 'N', 'NH3', 'O', 'S'};
%% INITIALIZE
self = App(LS);
%% INITIAL CONDITIONS
metallicity = 1;
Fuel = {'H', 'He', 'C', 'N', 'O', 'S'};
Ni_abundances = abundances2moles(Fuel, 'abundances.txt', metallicity)';
T = linspace(100, 4000, 300);
p = logspace(-5, 2, 300);
```

(continues on next page)

(continued from previous page)

```

self.PD.S_Fuel = Fuel;
self.PD.N_Fuel = Ni_abundances;
self = set_prop(self, 'TR', 300, 'pR', 1);
self = set_prop(self, 'TP', T, 'pP', p);
%% SOLVE PROBLEM
self = solve_problem(self, 'TP');
%% POSTPROCESSING CONFIGURATION
self.Misc.config.label_type = 'long';
%% DISPLAY RESULTS (PLOTS)
plot_molar_fractions(self, self.PS.strP, 'Xi', 'p', 'ydir', 'reverse', 'xscale
↪', 'log');
```

CHAPTER

SIX

VALIDATIONS

A set of the results obtained using Combustion Toolbox, NASA's CEA [Gordon and McBride, 1994], Cantera [Goodwin *et al.*, 2021], Caltech's SD-Toolbox [Browne *et al.*, 2008, Browne *et al.*, 2008], and TEA [Blecic *et al.*, 2016].

Note: Caltech's SD-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:

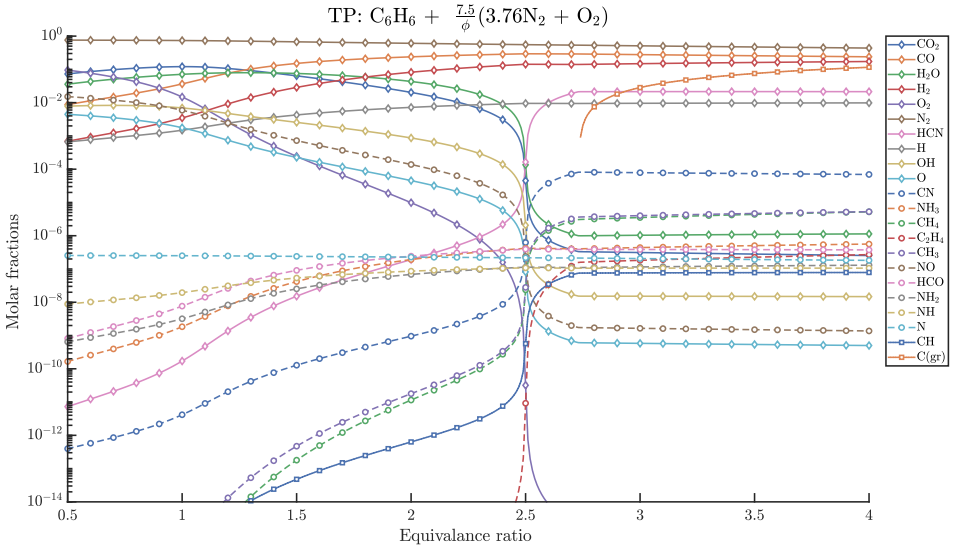
```
run_validations_CEA
```

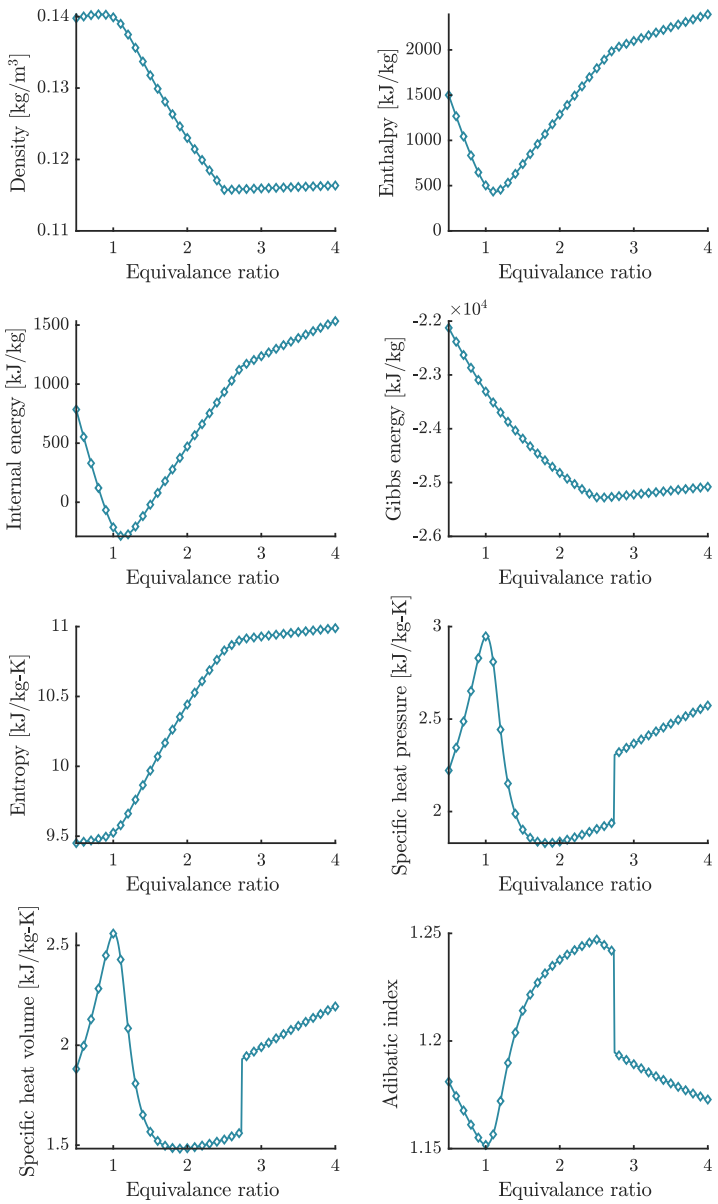
6.1 Validation TP 1

- Contrasted with: NASA's Chemical Equilibrium with Applications software
- Problem type: Equilibrium composition at defined temperature and pressure
- Temperature [K] = 2500
- Pressure [bar] = 1
- Initial mixture [moles]: $C_6H_6 + \frac{7.5}{\phi}\,(3.76N_2 + O_2)$, with equivalence ratio $\phi \in [0.5, 4]$
- List of species considered = `list_species('Soot formation Extended')`
- URL Folder Results CEA: ./validations/cea/data/tp

To repeat the results, run:

```
run_validation_TP_CEA_1.m
```



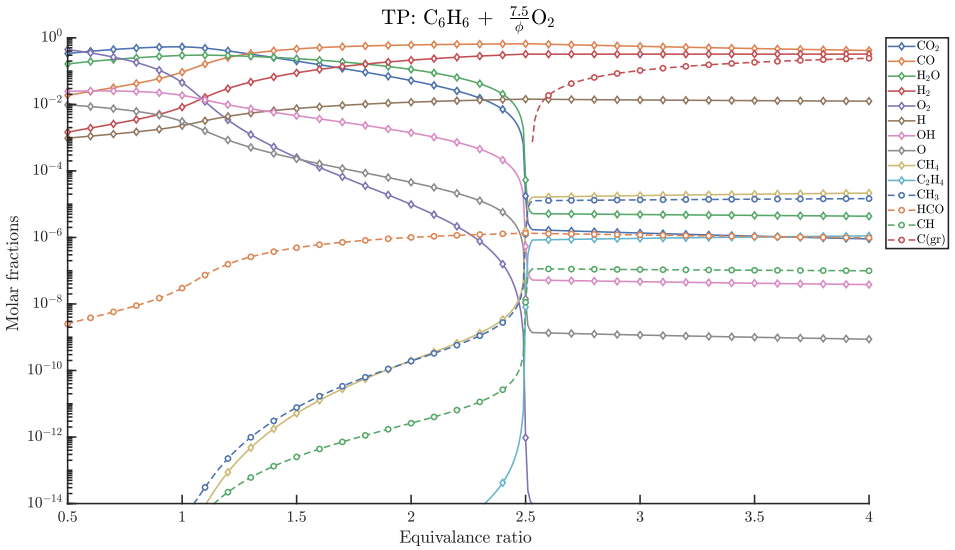

Density [kg/m^3]
0.14
0.13
0.12
0.11
1
2
3
4
Equivalance ratio
Enthalpy [kJ/kg]
2000
1500
1000
500
0
Internal energy [kJ/kg]
Gibbs energy [kJ/kg]
$\times 10^4$
-2.2
-2.3
-2.4
-2.5
-2.6
Entropy [kJ/kg-K]
11
10.5
10
9.5
Specific heat pressure [kJ/kg-K]
3
2.5
2
Specific heat volume [kJ/kg-K]
Adibatic index
1.25
1.2
1.15

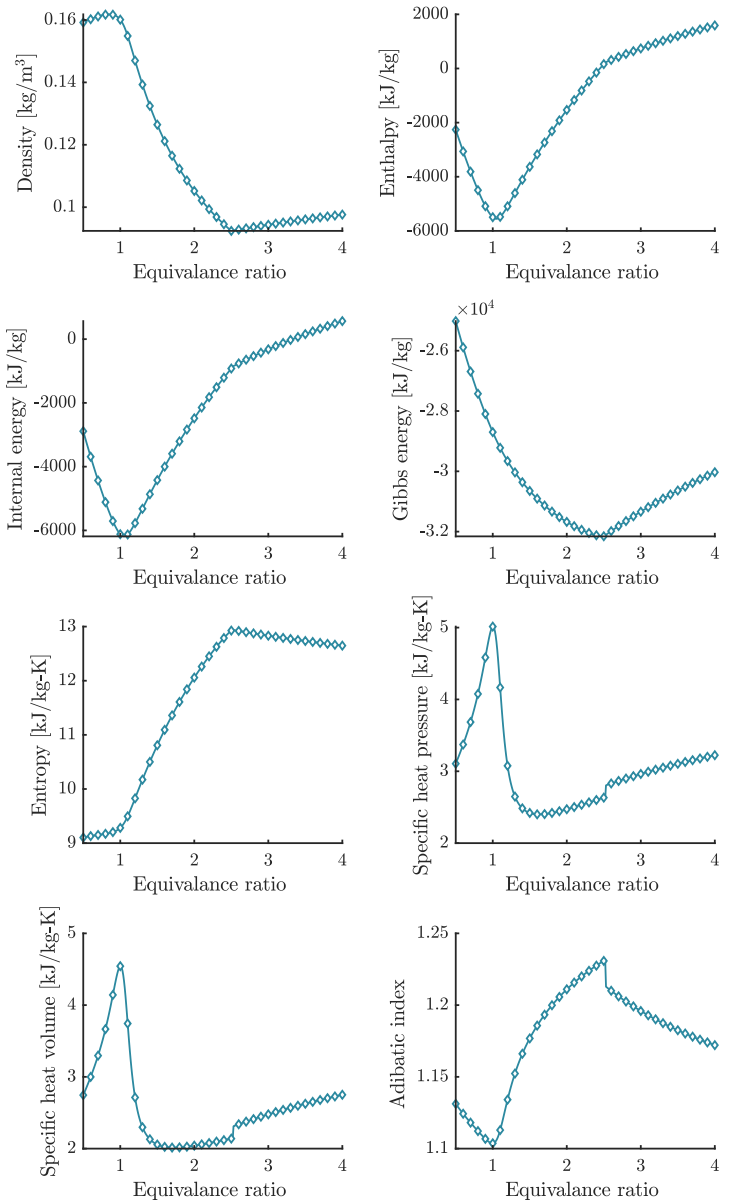
6.2 Validation TP 2

- Contrasted with: NASA's Chemical Equilibrium with Applications software
- Problem type: Equilibrium composition at defined temperature and pressure
- Temperature [K] = 2500
- Pressure [bar] = 1
- Initial mixture [moles]: $C_6H_6 + \frac{7.5}{\phi}O_2$, with equivalence ratio $\phi \in [0.5, 4]$
- List of species considered = `list_species('Soot formation Extended')`
- URL Folder Results CEA: ./validations/cea/data/tp

To repeat the results, run:

```
run_validation_TP_CEA_2.m
```



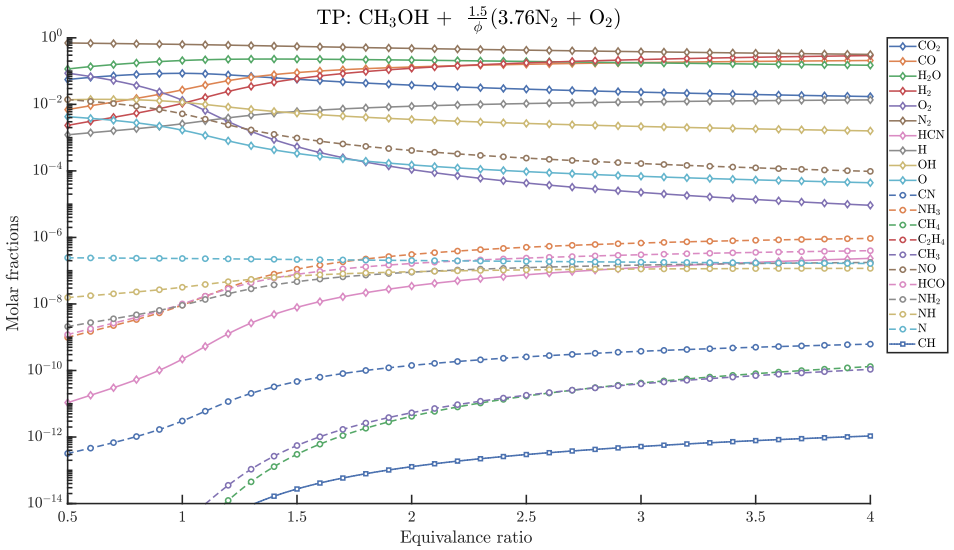

Density [kg/m^3]
0.16
0.14
0.12
0.1
1
2
3
4
Equivalance ratio
Enthalpy [kJ/kg]
2000
0
-2000
-4000
-6000
Equivalance ratio
Internal energy [kJ/kg]
0
-2000
-4000
-6000
Equivalance ratio
Gibbs energy [kJ/kg]
×10^4
-2.6
-2.8
-3
-3.2
Equivalance ratio
Entropy [kJ/kg-K]
13
12
11
10
9
Equivalance ratio
Specific heat pressure [kJ/kg-K]
5
4
3
2
Equivalance ratio
Specific heat volume [kJ/kg-K]
5
4
3
2
Equivalance ratio
Adibatic index
1.25
1.2
1.15
1.1
Equivalance ratio

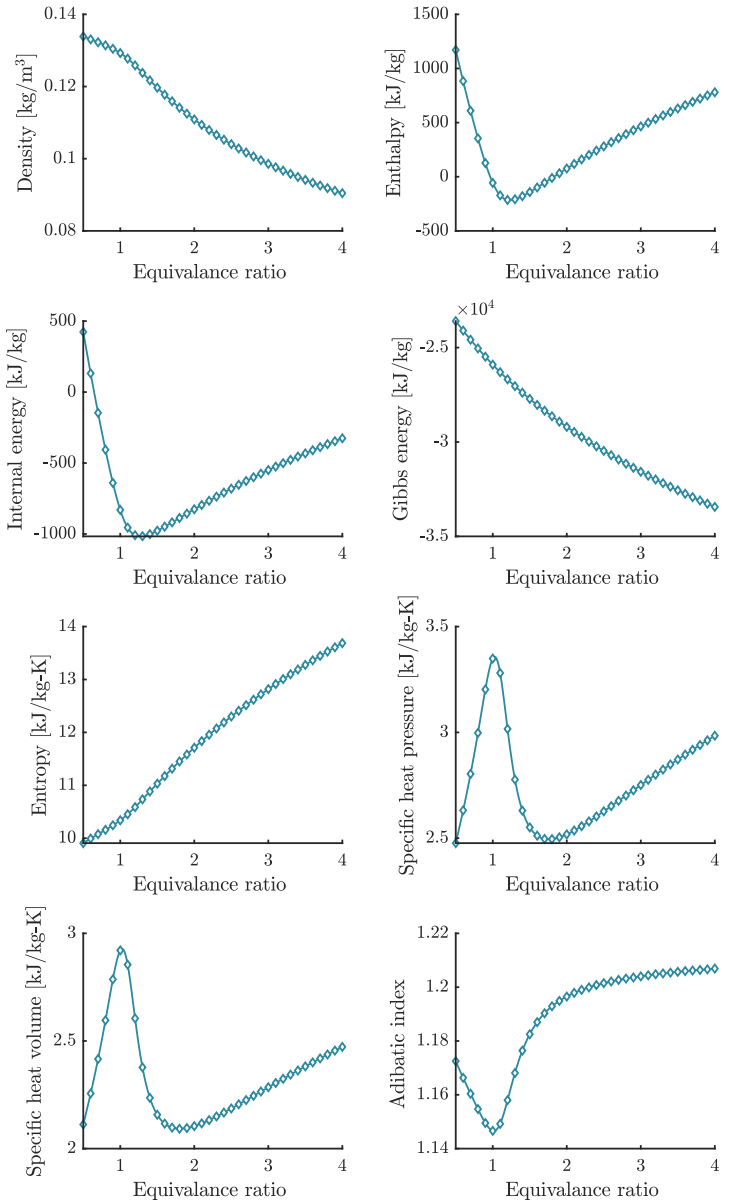
6.3 Validation TP 3

- Contrasted with: NASA's Chemical Equilibrium with Applications software
- Problem type: Equilibrium composition at defined temperature and pressure
- Temperature [K] = 2500
- Pressure [bar] = 1
- Initial mixture [moles]: $CH_3OH + \frac{1.5}{\phi}\,(3.76N_2 + O_2)$, with equivalence ratio $\phi \in [0.5, 4]$
- List of species considered = `list_species('Soot formation Extended')`
- URL Folder Results CEA: ./validations/cea/data/tp

To repeat the results, run:

```
run_validation_TP_CEA_3.m
```



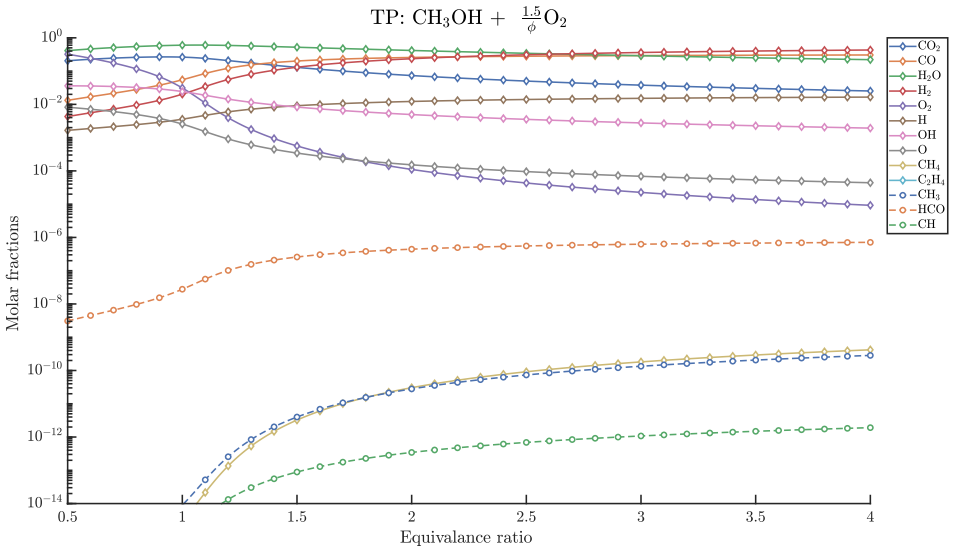

Density [kg/m^3]
0.14
0.12
0.1
0.08
1
2
3
4
Equivalance ratio
Enthalpy [kJ/kg]
1500
1000
500
0
-500
Internal energy [kJ/kg]
500
-1000
Gibbs energy [kJ/kg]
$\times 10^4$
-2.5
-3
-3.5
Entropy [kJ/kg-K]
14
13
12
11
10
Specific heat pressure [kJ/kg-K]
3.5
2.5
Specific heat volume [kJ/kg-K]
Adibatic index
1.22
1.2
1.18
1.16
1.14

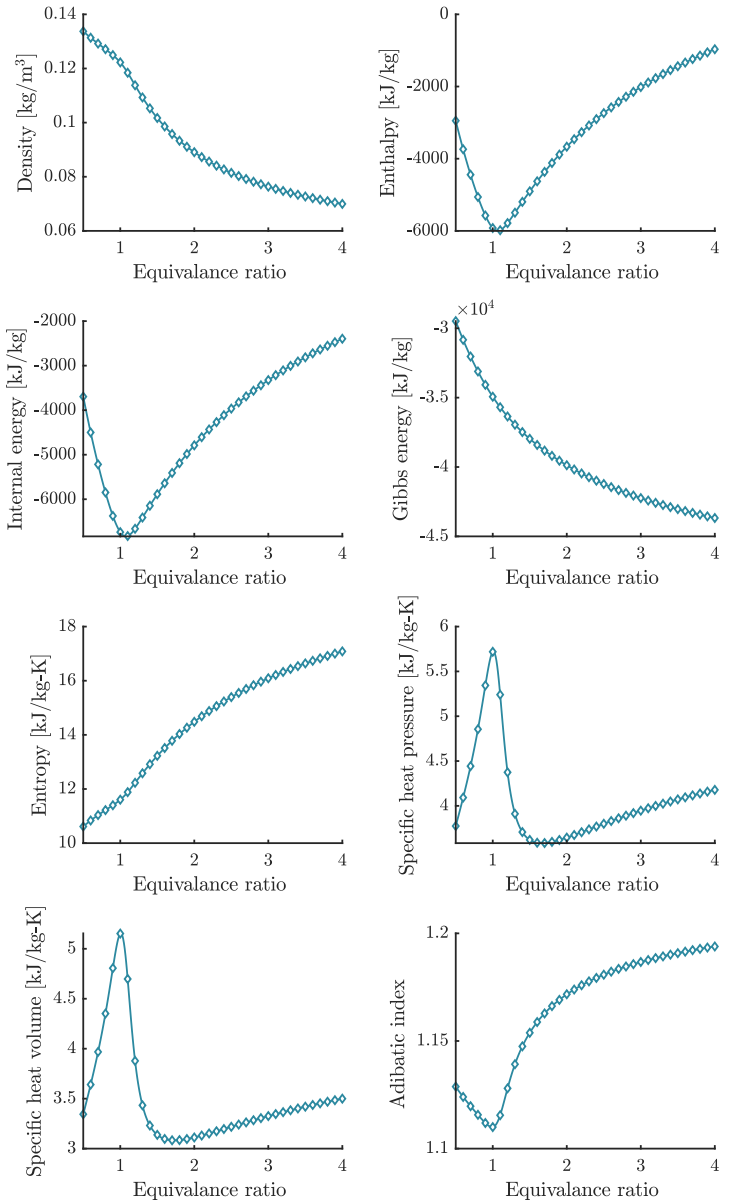
6.4 Validation TP 4

- Contrasted with: NASA's Chemical Equilibrium with Applications software
- Problem type: Equilibrium composition at defined temperature and pressure
- Temperature [K] = 2500
- Pressure [bar] = 1
- Initial mixture [moles]: $CH_3OH + \frac{1.5}{\phi} O_2$, with equivalence ratio $\phi \in [0.5, 4]$
- List of species considered = `list_species('Soot formation Extended')`
- URL Folder Results CEA: ./validations/cea/data/tp

To repeat the results, run:

```
run_validation_TP_CEA_4.m
```



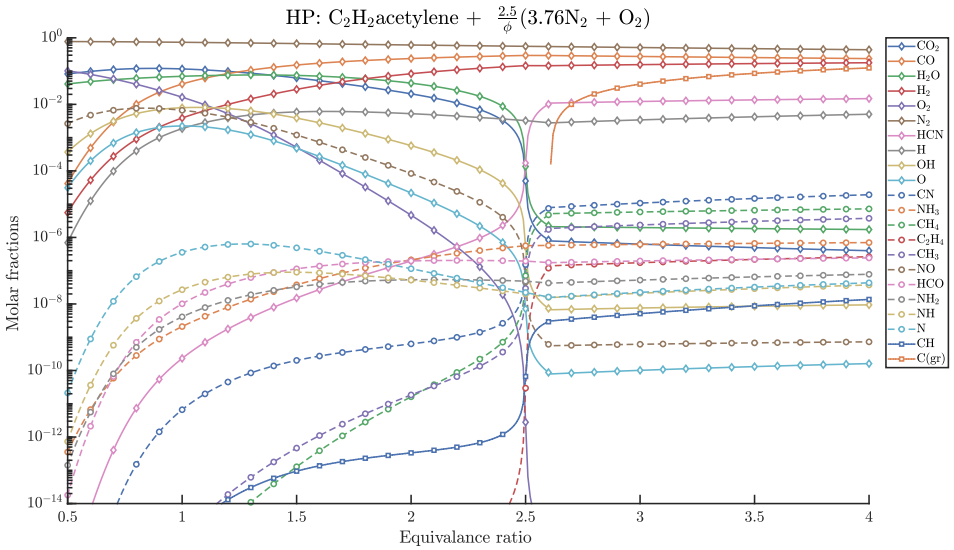

Density [kg/m^3]
Equivalance ratio
Enthalpy [kJ/kg]
Equivalance ratio
Internal energy [kJ/kg]
Equivalance ratio
Gibbs energy [kJ/kg]
×10^4
Equivalance ratio
Entropy [kJ/kg-K]
Equivalance ratio
Specific heat pressure [kJ/kg-K]
Equivalance ratio
Specific heat volume [kJ/kg-K]
Equivalance ratio
Adibatic index
Equivalance ratio

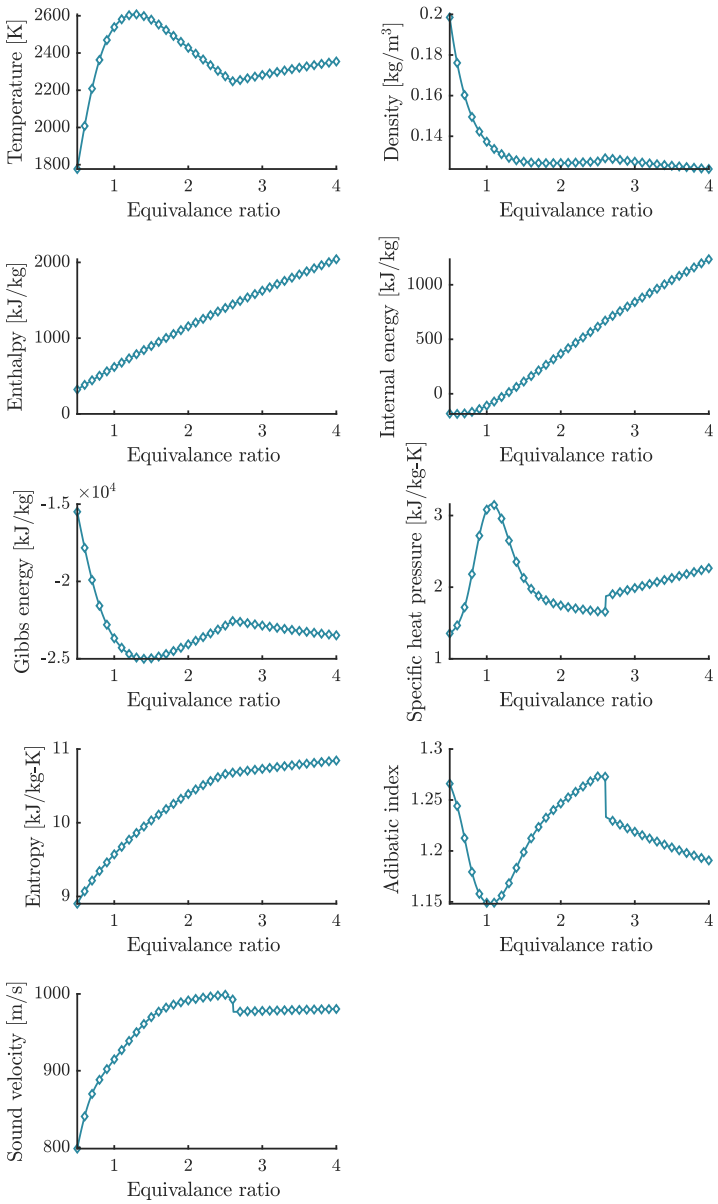
6.5 Validation HP 1

- Contrasted with: NASA's Chemical Equilibrium with Applications software
- Problem type: Adiabatic temperature and composition at constant pressure
- Temperature [K] = 300
- Pressure [bar] = 1
- Initial mixture [moles]: C_2H_2acetylene $+ \frac{2.5}{\phi}(3.76N_2 + O_2)$, with equivalence ratio $\phi \in [0.5, 4]$
- List of species considered = `list_species('Soot formation Extended')`
- URL Folder Results CEA: ./validations/cea/data/hp

To repeat the results, run:

```
run_validation_HP_CEA_1.m
```



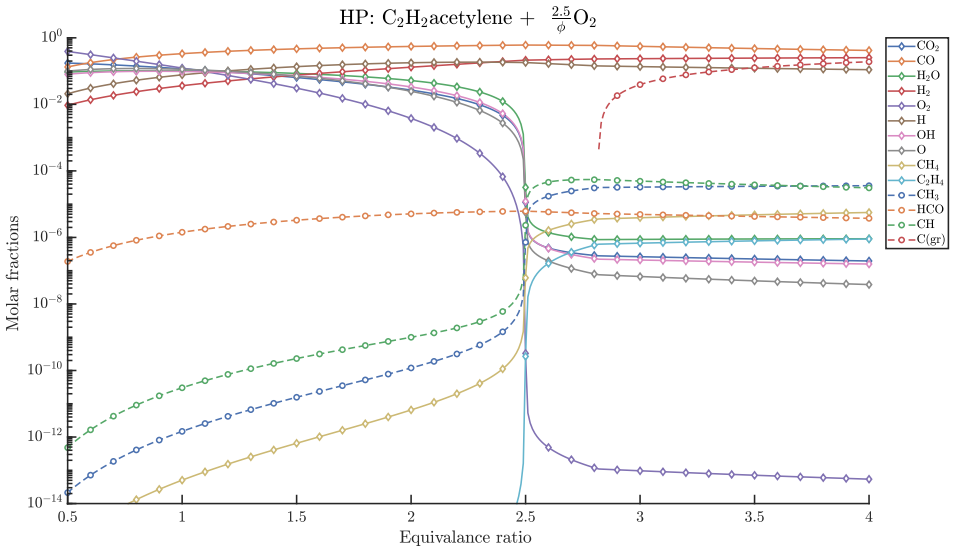

Temperature [K]
2600
2400
2200
2000
1800
Density [kg/m^3]
0.2
0.18
0.16
0.14
Enthalpy [kJ/kg]
2000
1000
0
Internal energy [kJ/kg]
1000
500
0
Gibbs energy [kJ/kg]
$\times 10^4$
-1.5
-2
-2.5
Specific heat pressure [kJ/kg-K]
3
2
1
Entropy [kJ/kg-K]
11
10
9
Adibatic index
1.3
1.25
1.2
1.15
Sound velocity [m/s]
1000
900
800
1
2
3
4
Equivalance ratio

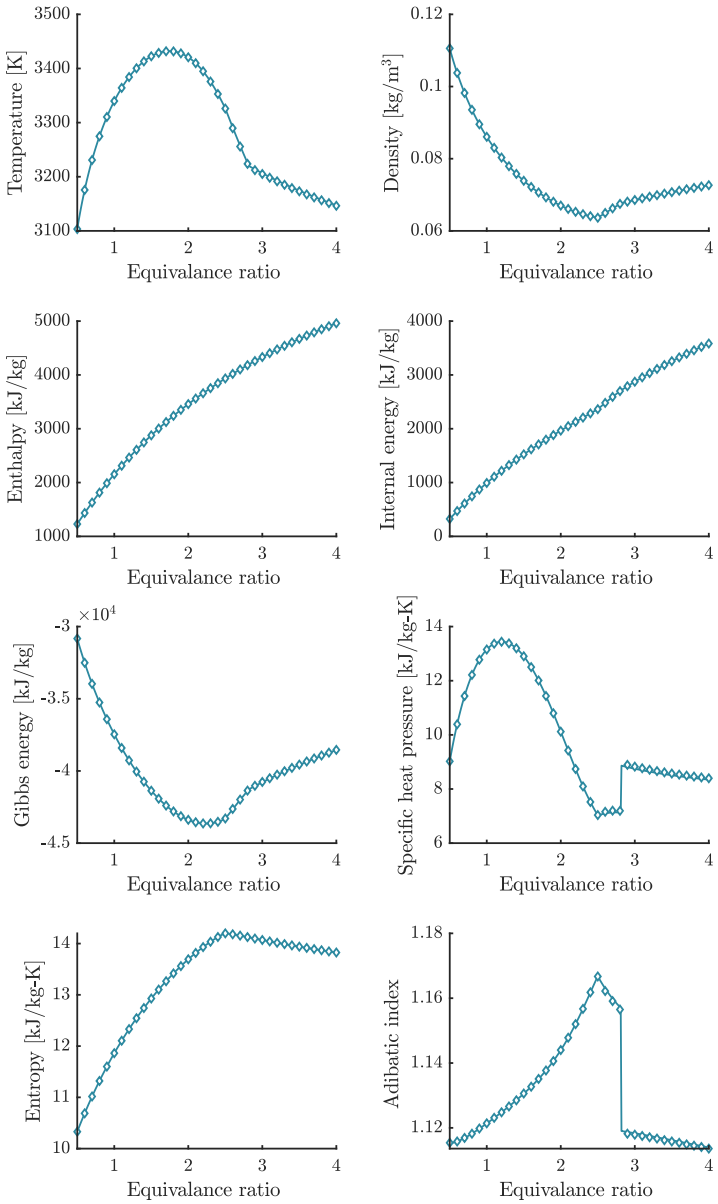
6.6 Validation HP 2

- Contrasted with: NASA's Chemical Equilibrium with Applications software
- Problem type: Adiabatic temperature and composition at constant pressure
- Temperature [K] = 300
- Pressure [bar] = 1
- Initial mixture [moles]: C_2H_2acetylene + $\frac{2.5}{\phi}O_2$, with equivalence ratio $\phi \in [0.5, 4]$
- List of species considered = `list_species('Soot formation Extended')`
- URL Folder Results CEA: ./validations/cea/data/hp

To repeat the results, run:

```
run_validation_HP_CEA_2.m
```



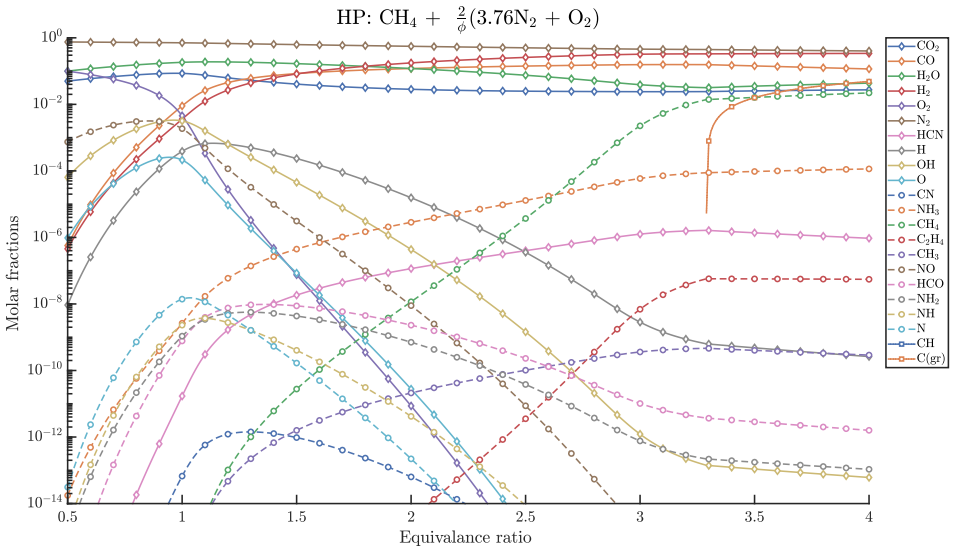

Temperature [K]
Equivalance ratio
Density [kg/m^3]
Equivalance ratio
Enthalpy [kJ/kg]
Equivalance ratio
Internal energy [kJ/kg]
Equivalance ratio
Gibbs energy [kJ/kg]
$\times 10^4$
Equivalance ratio
Specific heat pressure [kJ/kg-K]
Equivalance ratio
Entropy [kJ/kg-K]
Equivalance ratio
Adibatic index
Equivalance ratio

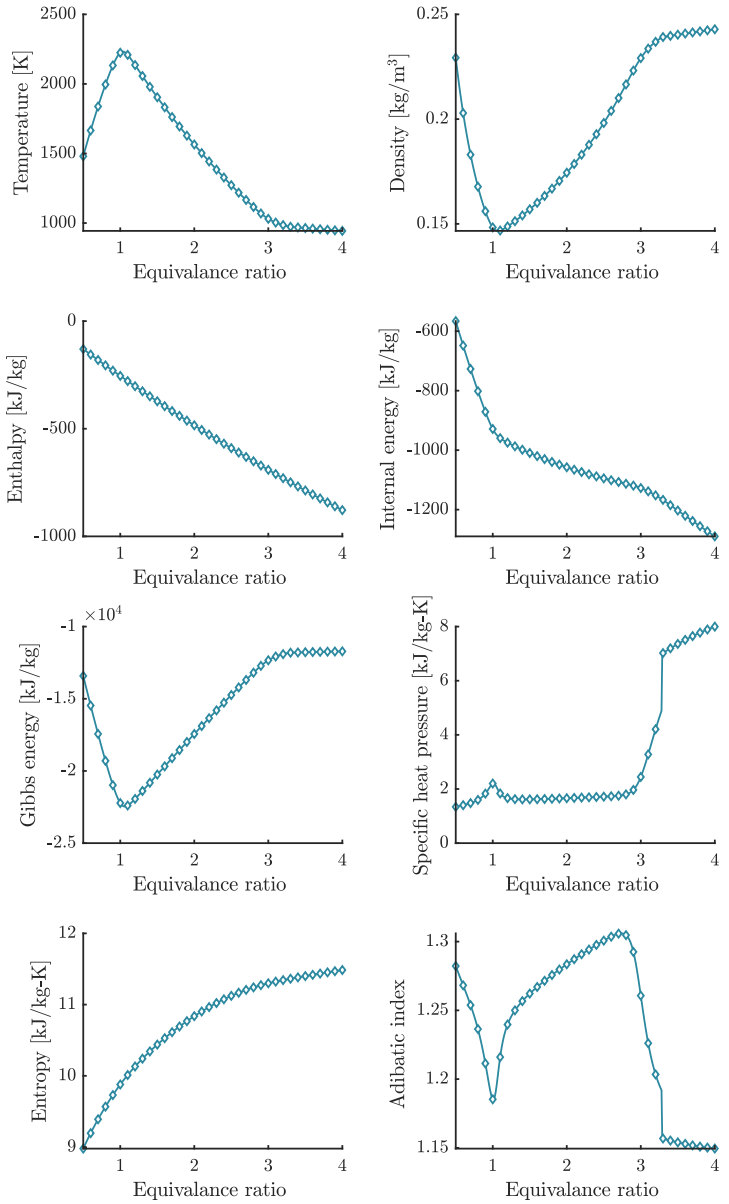
6.7 Validation HP 3

- Contrasted with: NASA's Chemical Equilibrium with Applications software
- Problem type: Adiabatic temperature and composition at constant pressure
- Temperature [K] = 300
- Pressure [bar] = 1
- Initial mixture [moles]: $CH_4 + \frac{2}{\phi}\,(3.76N_2 + O_2)$, with equivalence ratio $\phi \in [0.5, 4]$
- List of species considered = `list_species('Soot formation Extended')`
- URL Folder Results CEA: ./validations/cea/data/hp

To repeat the results, run:

```
run_validation_HP_CEA_3.m
```



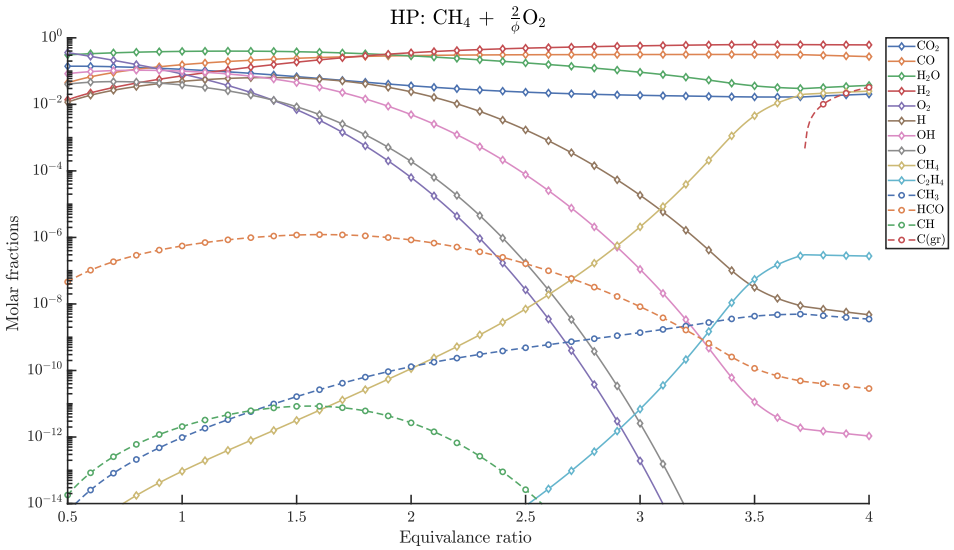

Temperature [K]
Equivalance ratio
Density [kg/m3]
Equivalance ratio
Enthalpy [kJ/kg]
Equivalance ratio
Internal energy [kJ/kg]
Equivalance ratio
Gibbs energy [kJ/kg]
×10^4
Equivalance ratio
Specific heat pressure [kJ/kg-K]
Equivalance ratio
Entropy [kJ/kg-K]
Equivalance ratio
Adibatic index
Equivalance ratio

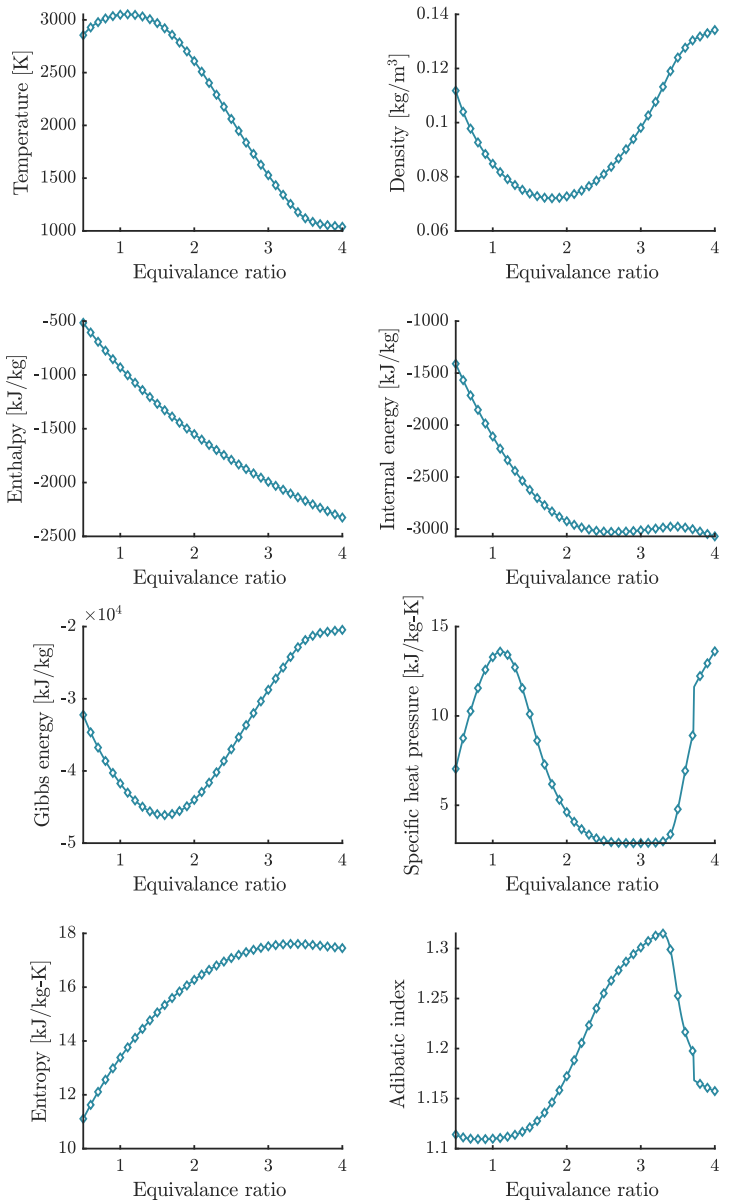
6.8 Validation HP 4

- Contrasted with: NASA's Chemical Equilibrium with Applications software
- Problem type: Adiabatic temperature and composition at constant pressure
- Temperature [K] = 300
- Pressure [bar] = 1
- Initial mixture [moles]: $CH_4 + \frac{2}{\phi}O_2$, with equivalence ratio $\phi \in [0.5, 4]$
- List of species considered = `list_species('Soot formation Extended')`
- URL Folder Results CEA: ./validations/cea/data/hp

To repeat the results, run:

```
run_validation_HP_CEA_4.m
```



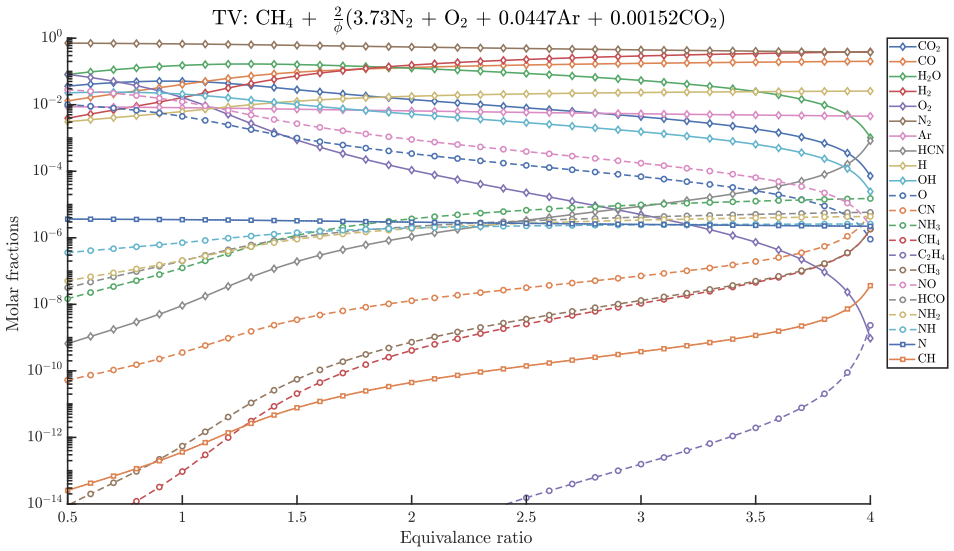

Temperature [K]
Equivalance ratio
Density [kg/m3]
Equivalance ratio
Enthalpy [kJ/kg]
Equivalance ratio
Internal energy [kJ/kg]
Equivalance ratio
×10^4
Gibbs energy [kJ/kg]
Equivalance ratio
Specific heat pressure [kJ/kg-K]
Equivalance ratio
Entropy [kJ/kg-K]
Equivalance ratio
Adibatic index
Equivalance ratio

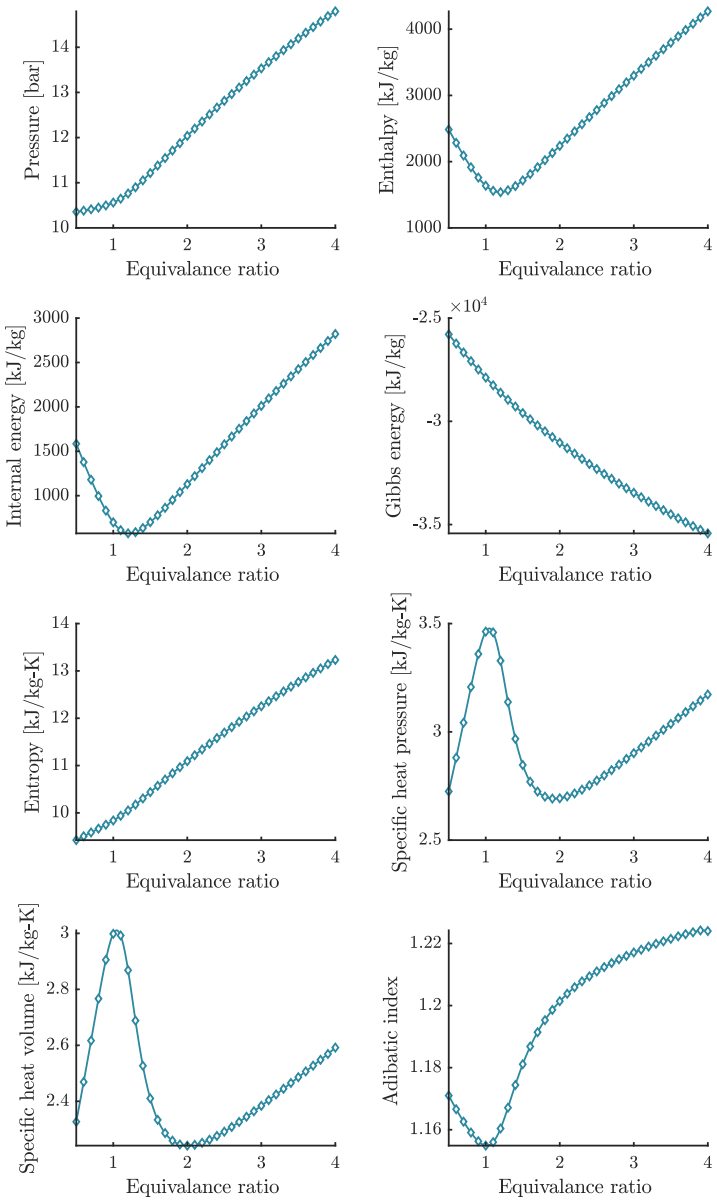
6.9 Validation TV 1

- Contrasted with: NASA's Chemical Equilibrium with Applications software
- Problem type: Equilibrium composition at defined temperature and volume
- Temperature [K] = 3000
- Pressure [bar] = 1.0132
- Initial mixture [moles]: $CH_4 + \frac{2}{\phi}\,(3.73N_2 + O_2 + 0.0447Ar + 0.00152CO_2)$, with equivalence ratio $\phi \in [0.5, 4]$
- List of species considered = `list_species('Soot formation Extended')`
- URL Folder Results CEA: ./validations/cea/data/tv

To repeat the results, run:

```
run_validation_TV_CEA_1.m
```



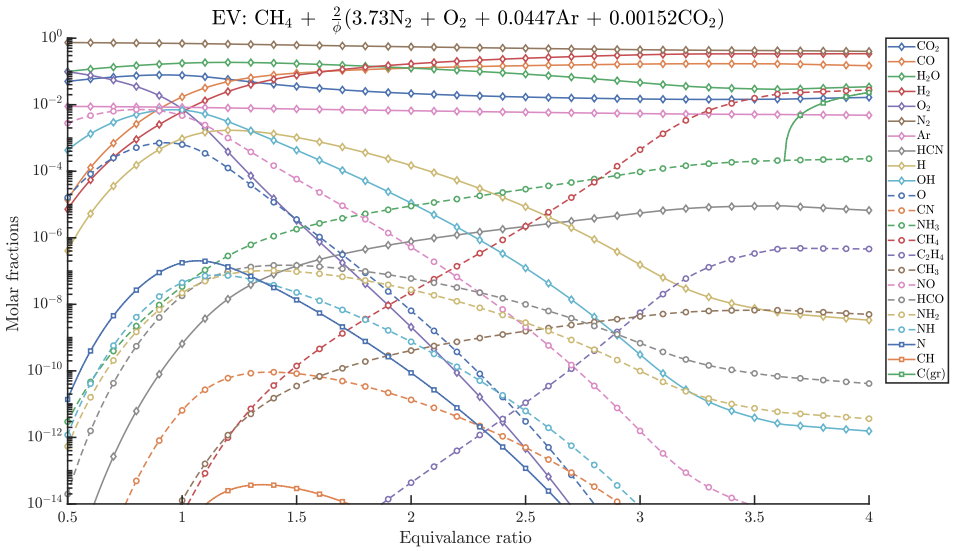

Pressure [bar]
Equivalance ratio
Enthalpy [kJ/kg]
Equivalance ratio
Internal energy [kJ/kg]
Equivalance ratio
Gibbs energy [kJ/kg]
×10^4
Equivalance ratio
Entropy [kJ/kg-K]
Equivalance ratio
Specific heat pressure [kJ/kg-K]
Equivalance ratio
Specific heat volume [kJ/kg-K]
Equivalance ratio
Adibatic index
Equivalance ratio

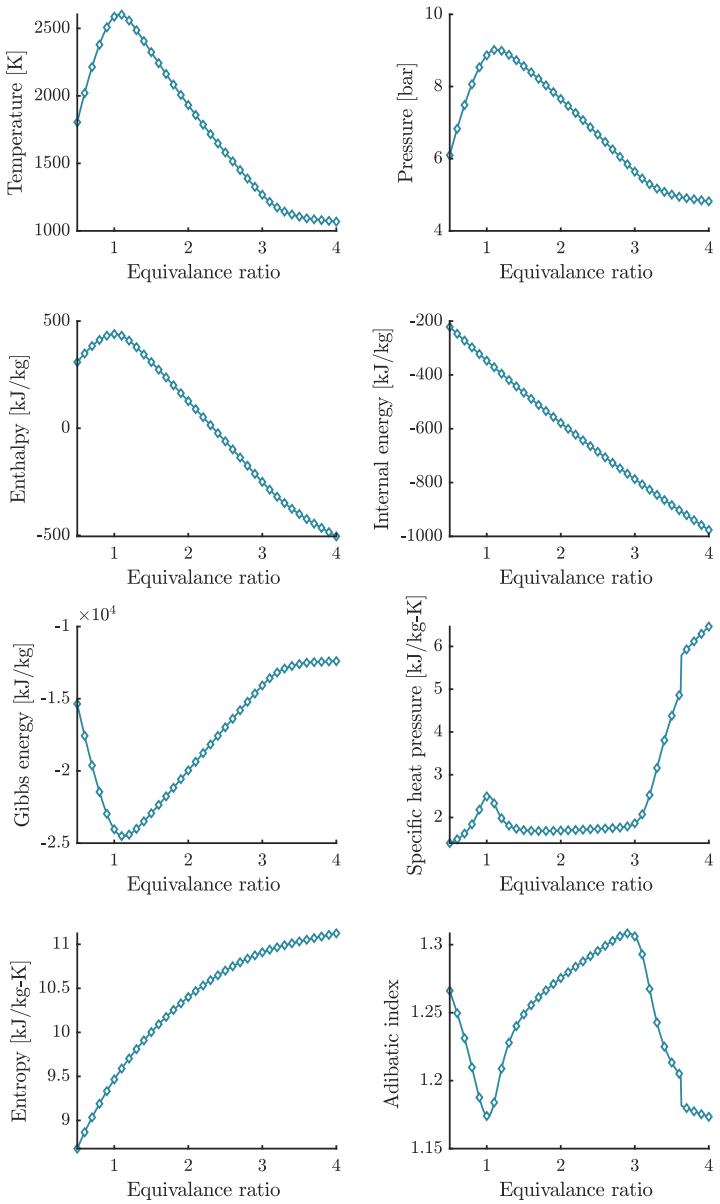
6.10 Validation EV 1

- Contrasted with: NASA's Chemical Equilibrium with Applications software
- Problem type: Adiabatic temperature and composition at constant volume
- Temperature [K] = 300
- Pressure [bar] = 1.0132
- Initial mixture [moles]: $CH_4 + \frac{2}{\phi}\,(3.73N_2 + O_2 + 0.0447Ar + 0.00152CO_2)$, with equivalence ratio $\phi \in [0.5, 4]$
- List of species considered = `list_species('Soot formation Extended')`
- URL Folder Results CEA: ./validations/cea/data/ev

To repeat the results, run:

```
run_validation_EV_CEA_1.m
```



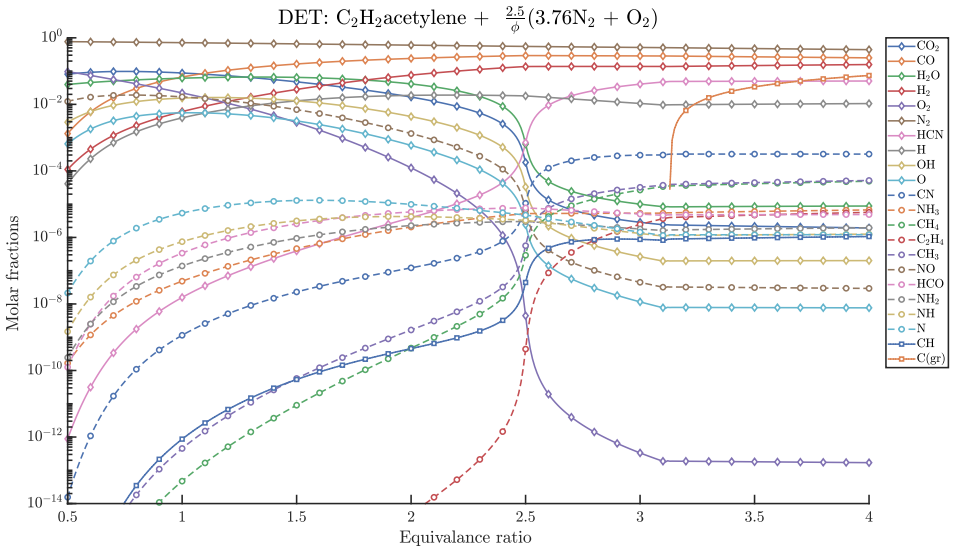

Temperature [K]
2500
2000
1500
1000
1
2
3
4
Equivalance ratio
Pressure [bar]
10
8
6
4
1
2
3
4
Equivalance ratio
Enthalpy [kJ/kg]
500
0
-500
1
2
3
4
Equivalance ratio
Internal energy [kJ/kg]
-200
-400
-600
-800
-1000
1
2
3
4
Equivalance ratio
Gibbs energy [kJ/kg]
×10^4
-1
-1.5
-2
-2.5
1
2
3
4
Equivalance ratio
Specific heat pressure [kJ/kg-K]
6
5
4
3
2
1
2
3
4
Equivalance ratio
Entropy [kJ/kg-K]
11
10.5
10
9.5
9
1
2
3
4
Equivalance ratio
Adibatic index
1.3
1.25
1.2
1.15
1
2
3
4
Equivalance ratio

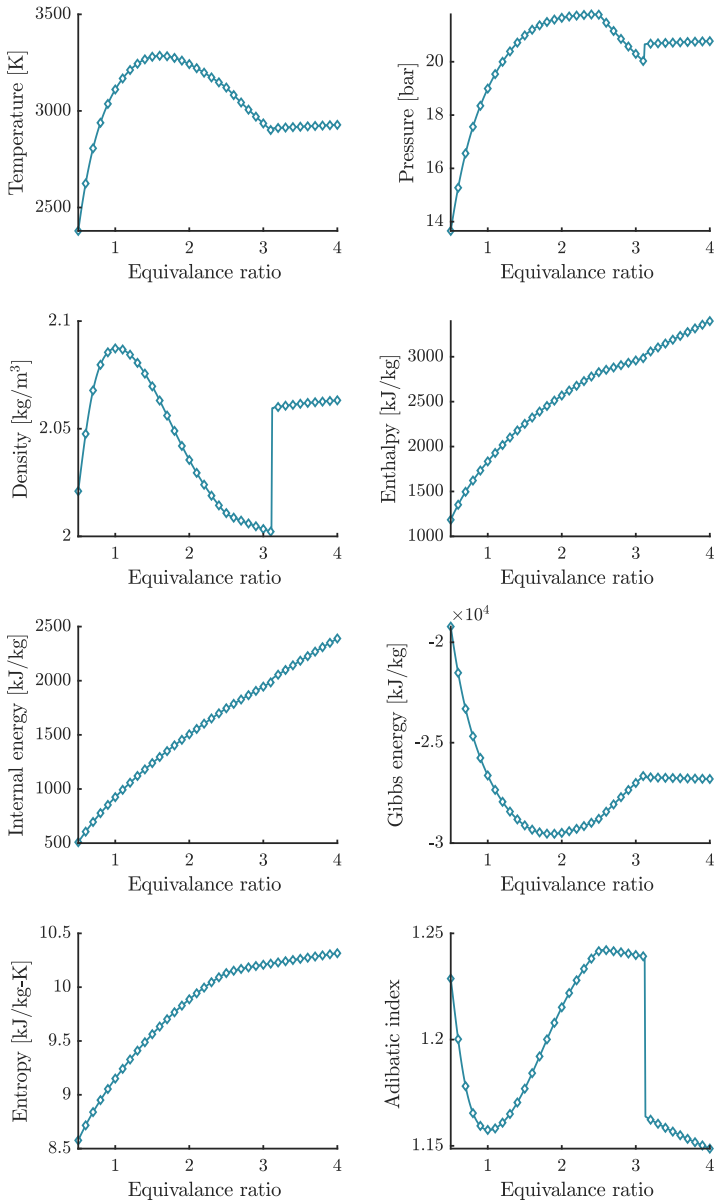
6.11 Validation DET 1

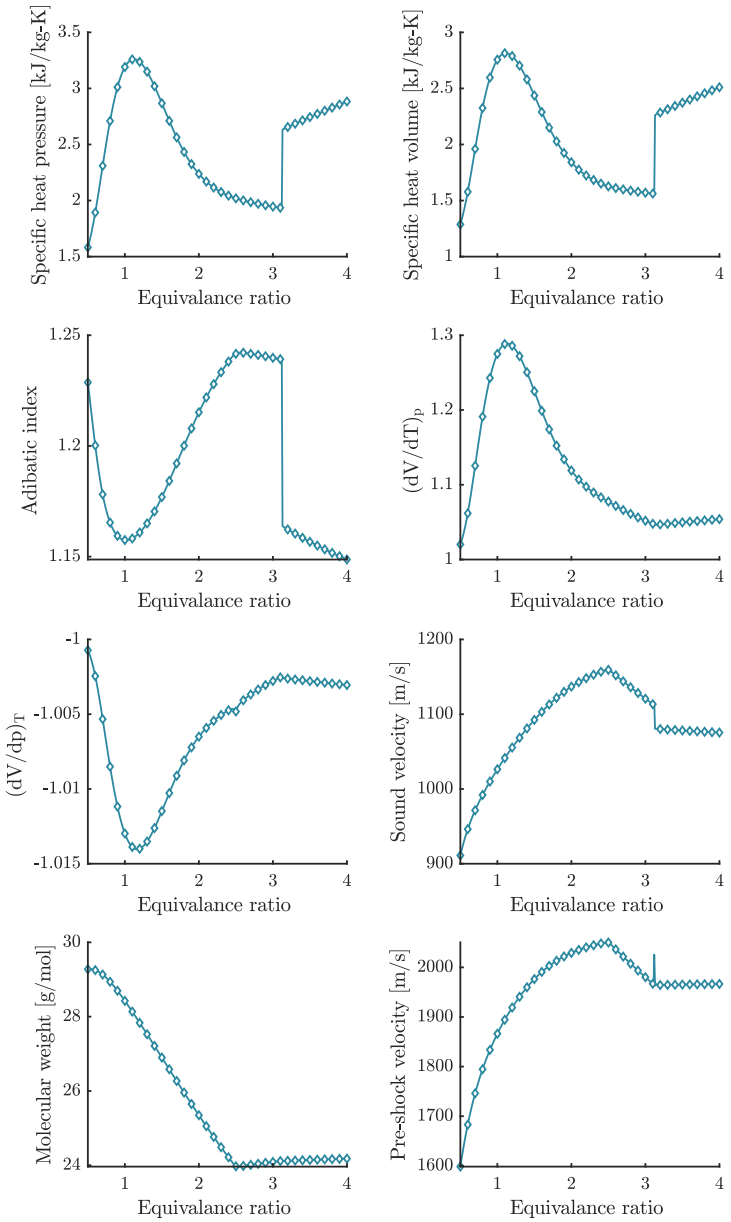
- Contrasted with: NASA's Chemical Equilibrium with Applications software
- Problem type: Chapman-Jouguet detonation
- Temperature [K] = 300
- Pressure [bar] = 1
- Initial mixture [moles]: C_2H_2acetylene $+ \frac{2.5}{\phi}\,(3.76N_2 + O_2)$, with equivalence ratio $\phi \in [0.5, 4]$
- List of species considered = `list_species('Soot formation Extended')`
- URL Folder Results CEA: ./validations/cea/data/hp

To repeat the results, run:

```
run_validation_DET_CEA_1.m
```




Temperature [K]
3500
3000
2500
Equivalance ratio
Pressure [bar]
20
18
16
14
Equivalance ratio
Density [kg/m^3]
2.1
2.05
2
Equivalance ratio
Enthalpy [kJ/kg]
3000
2500
2000
1500
1000
Equivalance ratio
Internal energy [kJ/kg]
2500
2000
1500
1000
500
Equivalance ratio
Gibbs energy [kJ/kg]
$\times 10^4$
-2
-2.5
-3
Equivalance ratio
Entropy [kJ/kg-K]
10.5
10
9.5
9
8.5
Equivalance ratio
Adibatic index
1.25
1.2
1.15
Equivalance ratio

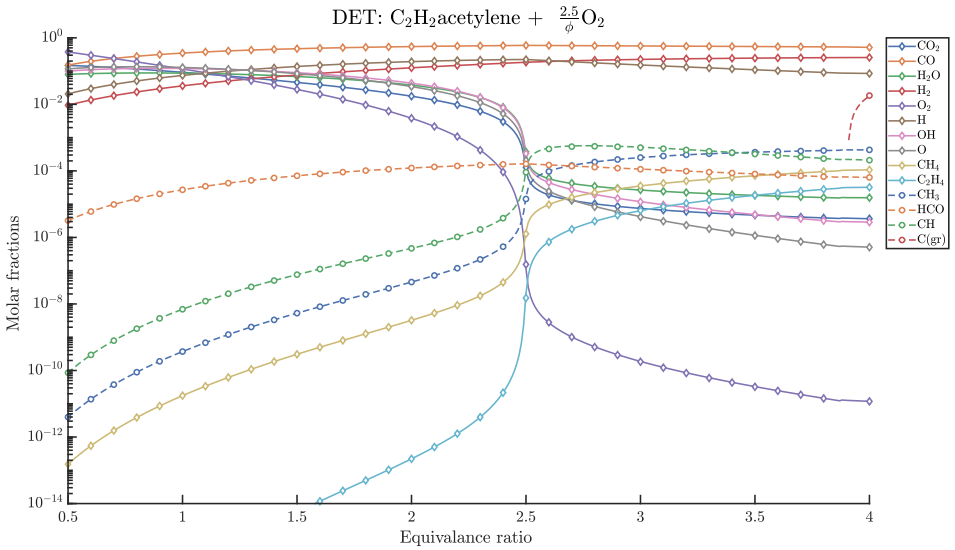

Specific heat pressure [kJ/kg-K]
Equivalance ratio
Specific heat volume [kJ/kg-K]
Equivalance ratio
Adibatic index
Equivalance ratio
(dV/dT)p
Equivalance ratio
(dV/dp)T
Equivalance ratio
Sound velocity [m/s]
Equivalance ratio
Molecular weight [g/mol]
Equivalance ratio
Pre-shock velocity [m/s]
Equivalance ratio

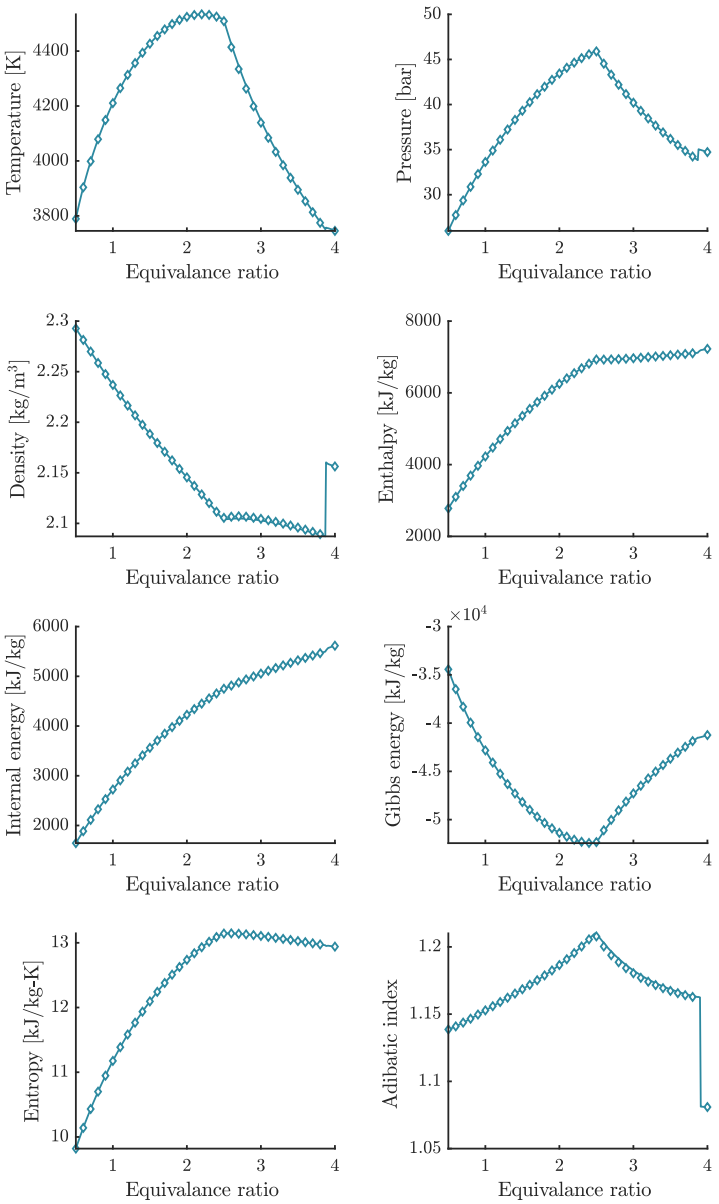
6.12 Validation DET 2

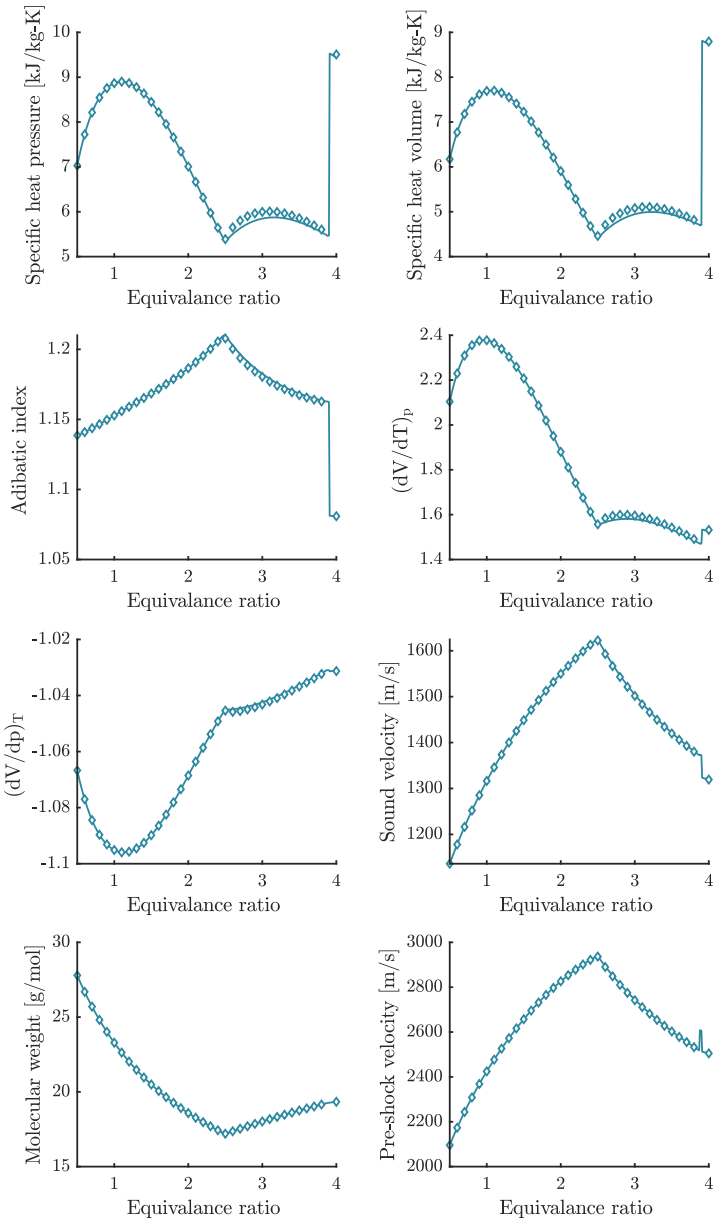
- Contrasted with: NASA's Chemical Equilibrium with Applications software
- Problem type: Chapman-Jouguet detonation
- Temperature [K] = 300
- Pressure [bar] = 1
- Initial mixture [moles]: C_2H_2acetylene $+ \frac{2.5}{\phi} O_2$, with equivalence ratio $\phi \in [0.5, 4]$
- List of species considered = `list_species('Soot formation Extended')`
- URL Folder Results CEA: ./validations/cea/data/det

To repeat the results, run:

```
run_validation_DET_CEA_2.m
```




Temperature [K]
4400
4200
4000
3800
1
2
3
4
Equivalance ratio
Pressure [bar]
50
45
40
35
30
1
2
3
4
Equivalance ratio
Density [kg/m^3]
2.3
2.25
2.2
2.15
2.1
1
2
3
4
Equivalance ratio
Enthalpy [kJ/kg]
8000
6000
4000
2000
1
2
3
4
Equivalance ratio
Internal energy [kJ/kg]
6000
5000
4000
3000
2000
1
2
3
4
Equivalance ratio
Gibbs energy [kJ/kg]
$\times 10^4$
-3
-3.5
-4
-4.5
-5
1
2
3
4
Equivalance ratio
Entropy [kJ/kg-K]
13
12
11
10
1
2
3
4
Equivalance ratio
Adibatic index
1.2
1.15
1.1
1.05
1
2
3
4
Equivalance ratio

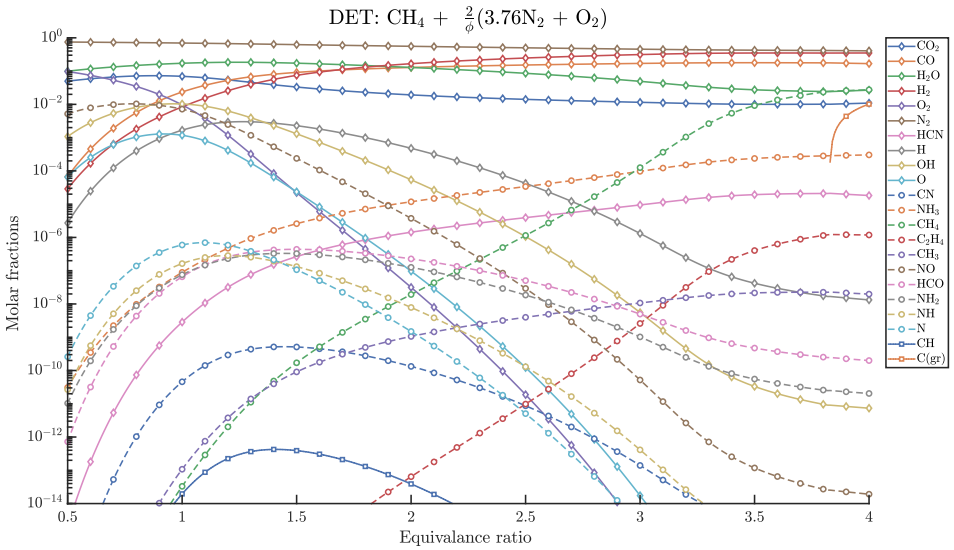

Specific heat pressure [kJ/kg-K]
Equivalance ratio
Specific heat volume [kJ/kg-K]
Equivalance ratio
Adibatic index
Equivalance ratio
(dV/dT)p
Equivalance ratio
(dV/dp)T
Equivalance ratio
Sound velocity [m/s]
Equivalance ratio
Molecular weight [g/mol]
Equivalance ratio
Pre-shock velocity [m/s]
Equivalance ratio

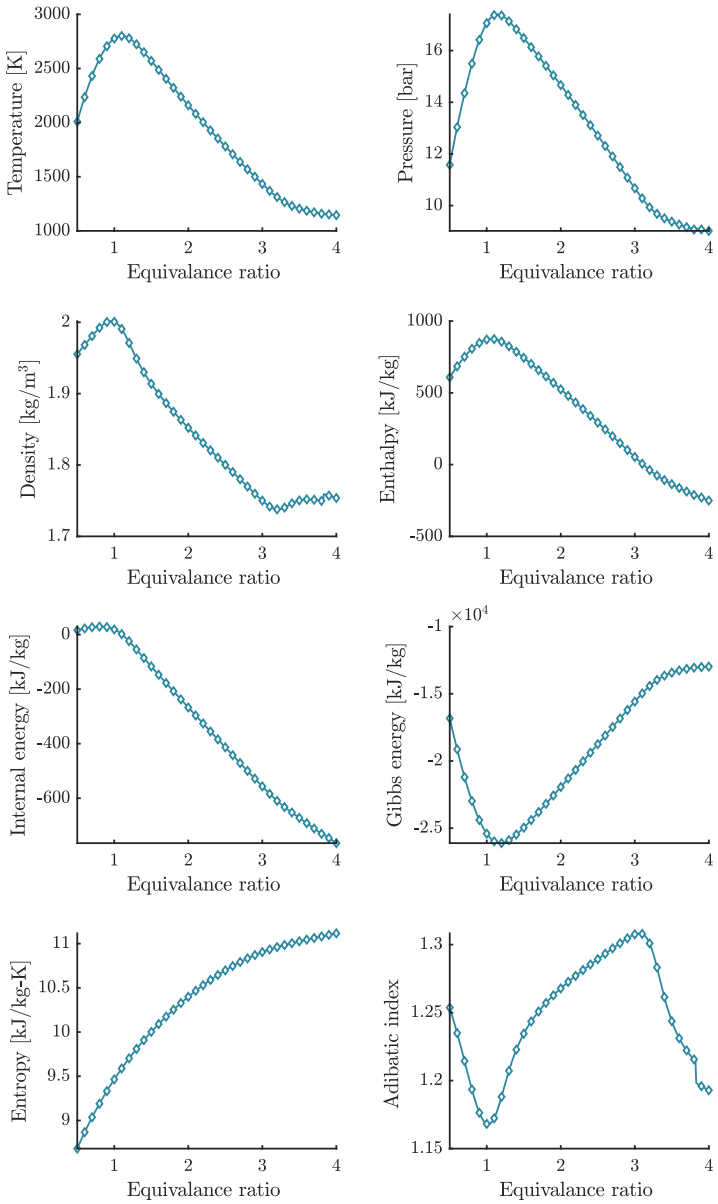
6.13 Validation DET 3

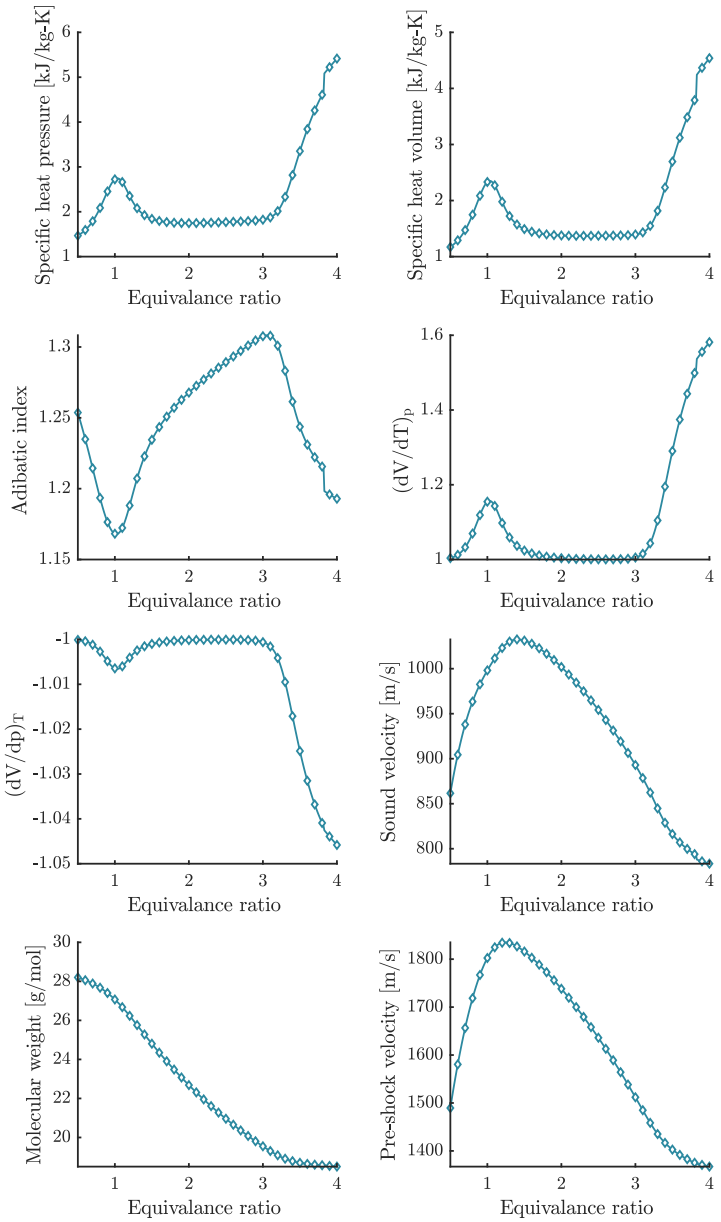
- Contrasted with: NASA's Chemical Equilibrium with Applications software
- Problem type: Chapman-Jouguet detonation
- Temperature [K] = 300
- Pressure [bar] = 1
- Initial mixture [moles]: $CH_4 + \frac{2}{\phi}\,(3.76N_2 + O_2)$, with equivalence ratio $\phi \in [0.5, 4]$
- List of species considered = `list_species('Soot formation Extended')`
- URL Folder Results CEA: ./validations/cea/data/det

To repeat the results, run:

```
run_validation_DET_CEA_3.m
```




Temperature [K]
Equivalance ratio
Pressure [bar]
Equivalance ratio
Density [kg/m^3]
Equivalance ratio
Enthalpy [kJ/kg]
Equivalance ratio
Internal energy [kJ/kg]
Equivalance ratio
Gibbs energy [kJ/kg]
$\times 10^4$
Equivalance ratio
Entropy [kJ/kg-K]
Equivalance ratio
Adibatic index
Equivalance ratio

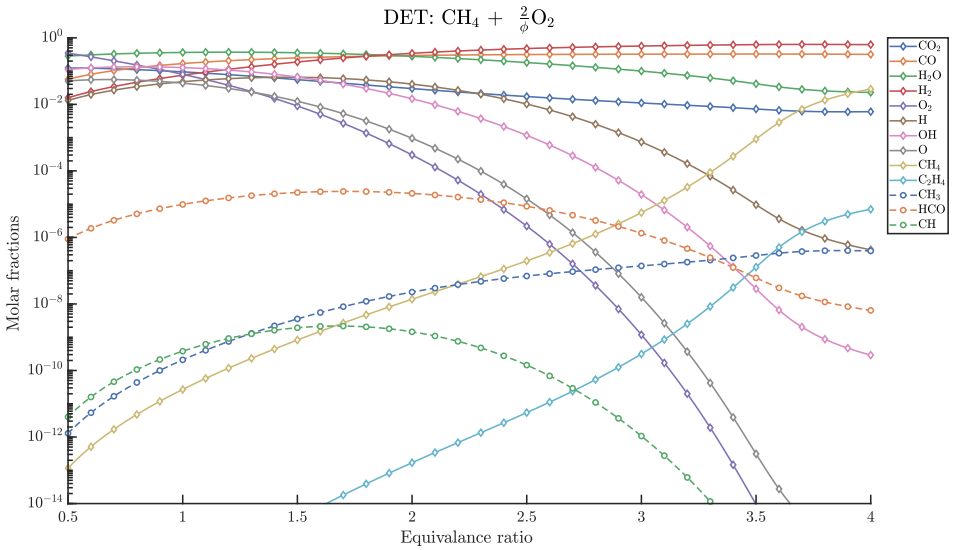

Specific heat pressure [kJ/kg-K]
Equivalance ratio
Specific heat volume [kJ/kg-K]
Equivalance ratio
Adibatic index
Equivalance ratio
(dV/dT)p
Equivalance ratio
(dV/dp)T
Equivalance ratio
Sound velocity [m/s]
Equivalance ratio
Molecular weight [g/mol]
Equivalance ratio
Pre-shock velocity [m/s]
Equivalance ratio

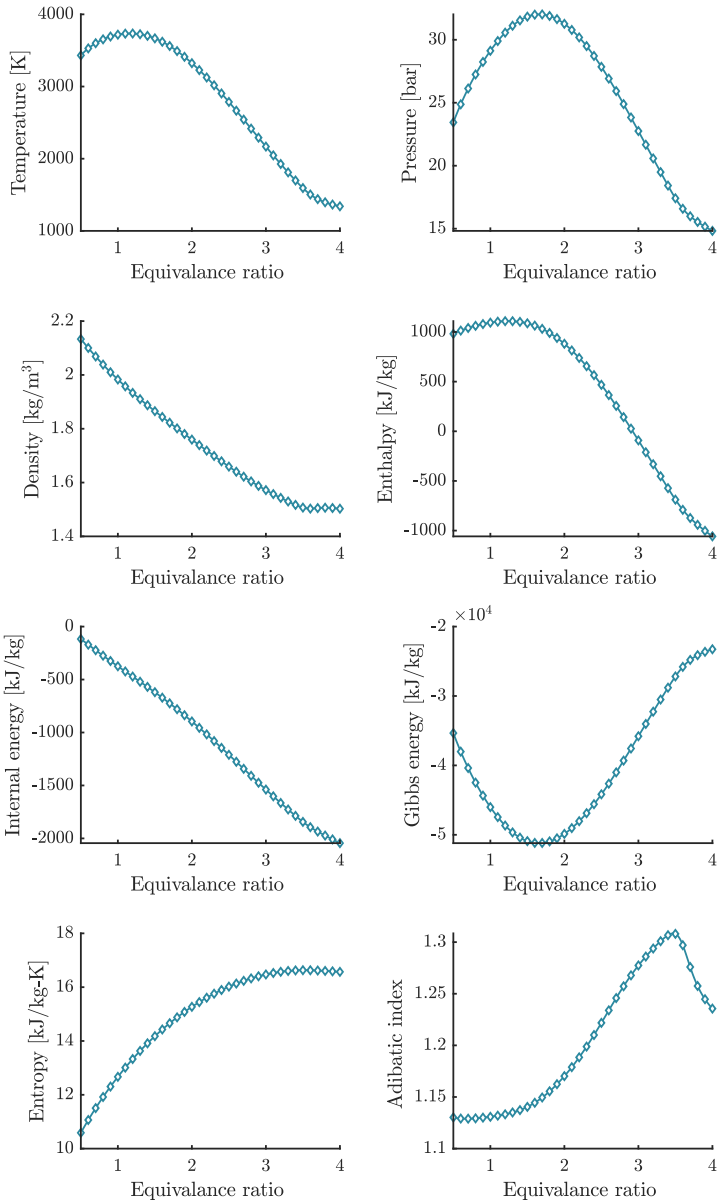
6.14 Validation DET 4

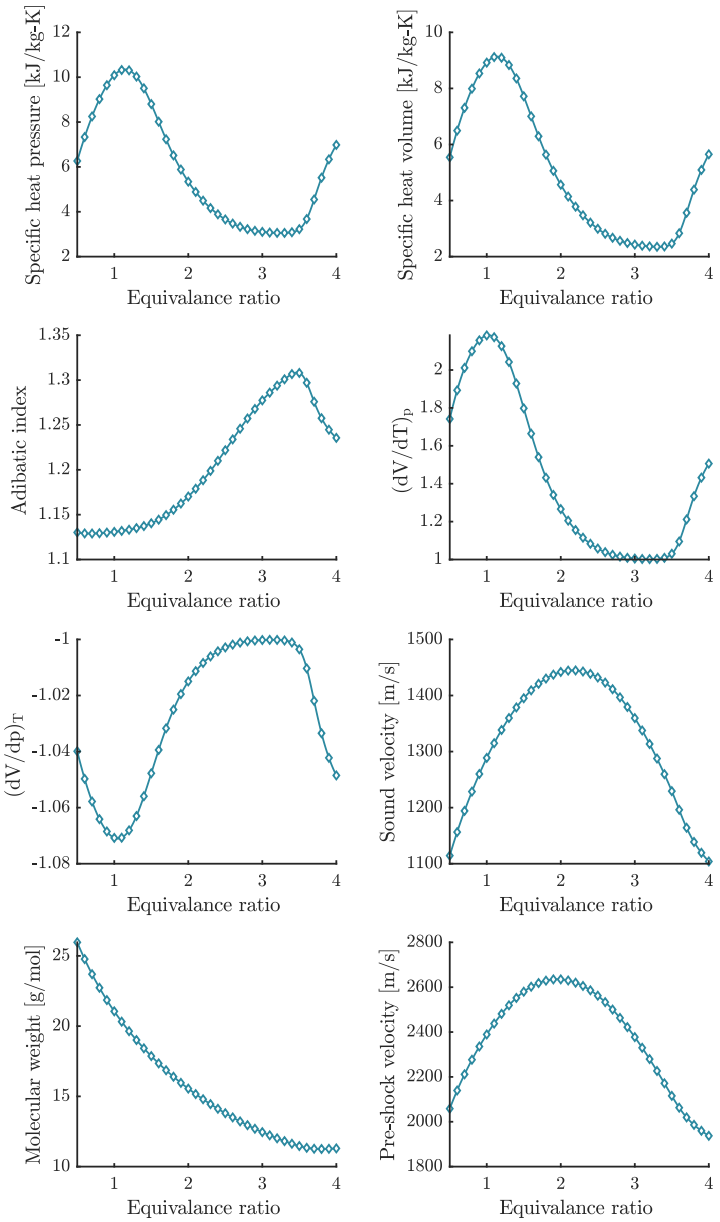
- Contrasted with: NASA's Chemical Equilibrium with Applications software
- Problem type: Chapman-Jouguet detonation
- Temperature [K] = 300
- Pressure [bar] = 1
- Initial mixture [moles]: $CH_4 + \frac{2}{\phi}O_2$, with equivalence ratio $\phi \in [0.5, 4]$
- List of species considered = `list_species('Soot formation Extended')`
- URL Folder Results CEA: ./validations/cea/data/det

To repeat the results, run:

```
run_validation_DET_CEA_4.m
```




Temperature [K]
4000
3000
2000
1000
Equivalance ratio
Pressure [bar]
30
25
20
15
Equivalance ratio
Density [kg/m^3]
2.2
2
1.8
1.6
1.4
Equivalance ratio
Enthalpy [kJ/kg]
1000
500
0
-500
-1000
Equivalance ratio
Internal energy [kJ/kg]
0
-500
-1000
-1500
-2000
Equivalance ratio
Gibbs energy [kJ/kg]
$\times 10^4$
-2
-3
-4
-5
Equivalance ratio
Entropy [kJ/kg-K]
18
16
14
12
10
Equivalance ratio
Adibatic index
1.3
1.25
1.2
1.15
1.1
Equivalance ratio

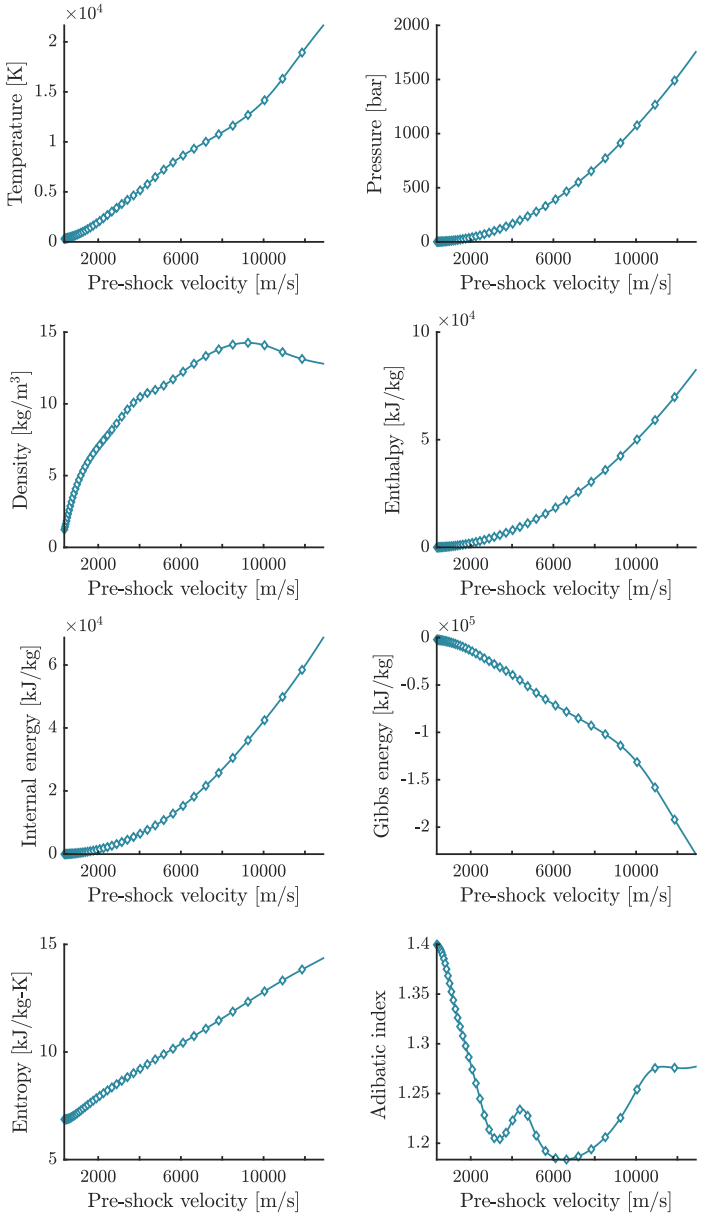

Specific heat pressure [kJ/kg-K]
Equivalance ratio
Specific heat volume [kJ/kg-K]
Equivalance ratio
Adibatic index
Equivalance ratio
(dV/dT)p
Equivalance ratio
(dV/dp)T
Equivalance ratio
Sound velocity [m/s]
Equivalance ratio
Molecular weight [g/mol]
Equivalance ratio
Pre-shock velocity [m/s]
Equivalance ratio

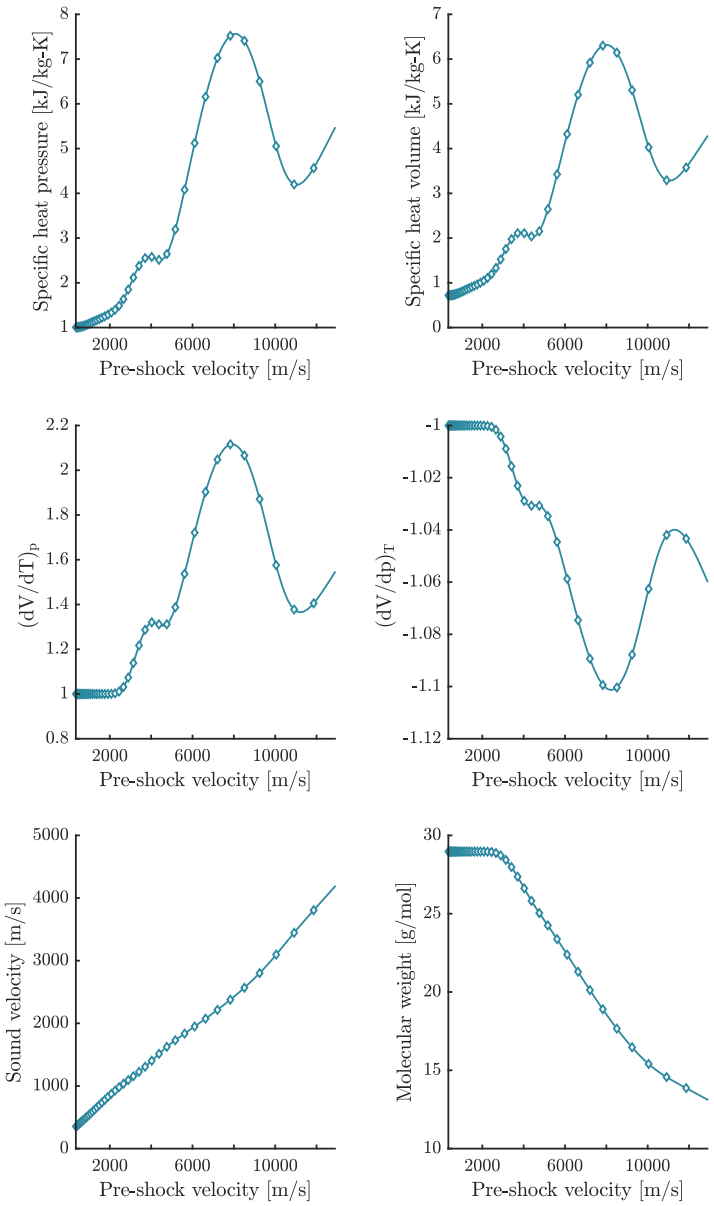
6.15 Validation SHOCK IONIZATION 1

- Contrasted with: NASA's Chemical Equilibrium with Applications software
- Problem type: Planar shock wave
- Temperature [K] = 300
- Pressure [bar] = 1
- Initial mixture [moles]: $3.7276\mathrm{N_2} + \mathrm{O_2} + 0.0447\mathrm{Ar} + 0.00152\mathrm{CO_2}$
- List of species considered = list_species('Air_ions')
- URL Folder Results CEA: ./validations/cea/data/shocks

To repeat the results, run:

```
run_validation_SHOCK_IONIZATION_CEA_1.m
```


×10^4
Temperature [K]
Pre-shock velocity [m/s]
Pressure [bar]
Pre-shock velocity [m/s]
Density [kg/m^3]
Pre-shock velocity [m/s]
×10^4
Enthalpy [kJ/kg]
Pre-shock velocity [m/s]
×10^4
Internal energy [kJ/kg]
Pre-shock velocity [m/s]
×10^5
Gibbs energy [kJ/kg]
Pre-shock velocity [m/s]
Entropy [kJ/kg-K]
Pre-shock velocity [m/s]
Adibatic index
Pre-shock velocity [m/s]

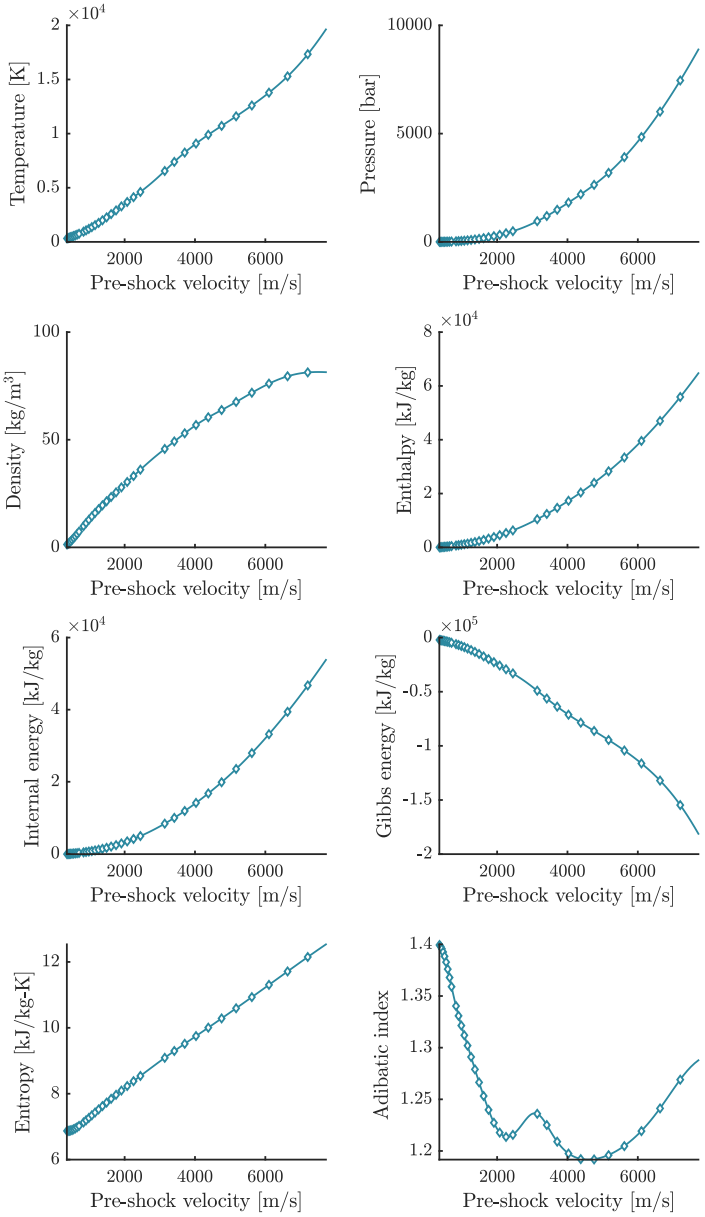

Specific heat pressure [kJ/kg-K]
Pre-shock velocity [m/s]
Specific heat volume [kJ/kg-K]
Pre-shock velocity [m/s]
(dV/dT)p
Pre-shock velocity [m/s]
(dV/dp)T
Pre-shock velocity [m/s]
Sound velocity [m/s]
Pre-shock velocity [m/s]
Molecular weight [g/mol]
Pre-shock velocity [m/s]

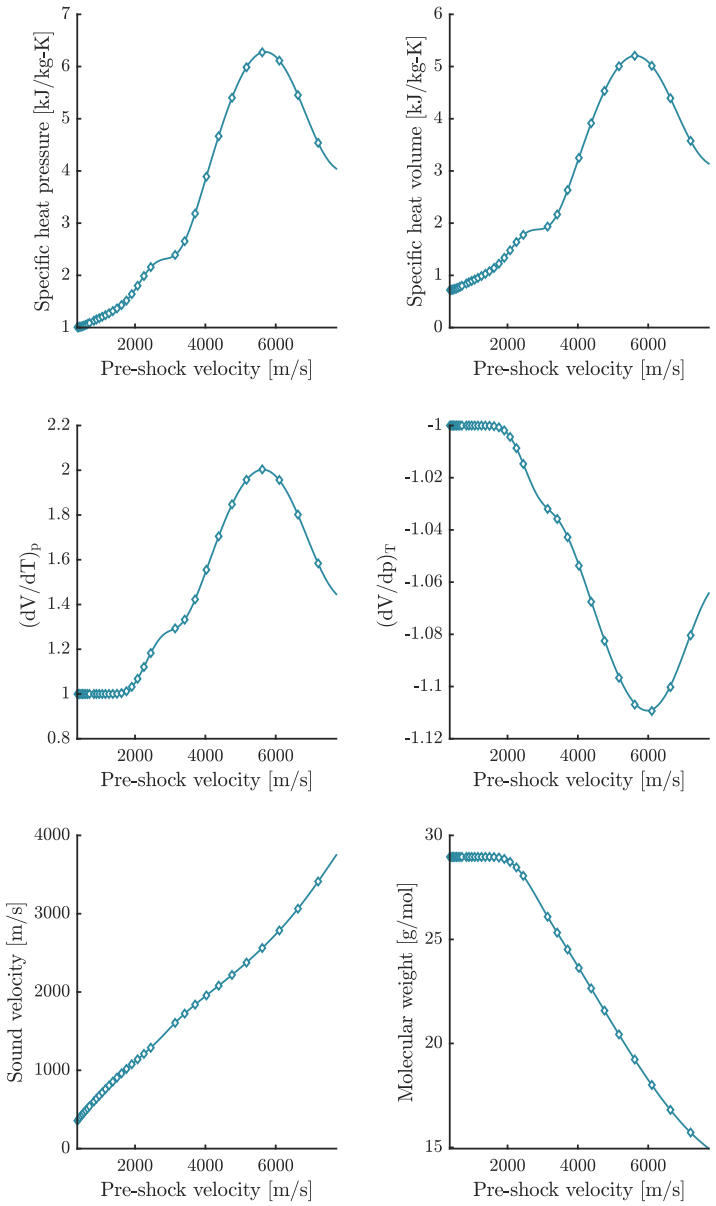
6.16 Validation SHOCK REFLECTED IONIZATION 1

- Contrasted with: NASA's Chemical Equilibrium with Applications software
- Problem type: Normal reflection of a planar shock wave from a parallel rigid wall
- Temperature [K] = 300
- Pressure [bar] = 1
- Initial mixture [moles]: $3.7276\mathrm{N_2} + \mathrm{O_2} + 0.0447\mathrm{Ar} + 0.00152\mathrm{CO_2}$
- List of species considered = `list_species('Air_ions')`
- URL Folder Results CEA: ./validations/cea/data/shocks

To repeat the results, run:

```
run_validation_SHOCK_R_IONIZATION_CEA_1.m
```


×10^4
Temperature [K]
Pre-shock velocity [m/s]
Pressure [bar]
Pre-shock velocity [m/s]
Density [kg/m^3]
Pre-shock velocity [m/s]
×10^4
Enthalpy [kJ/kg]
Pre-shock velocity [m/s]
×10^4
Internal energy [kJ/kg]
Pre-shock velocity [m/s]
×10^5
Gibbs energy [kJ/kg]
Pre-shock velocity [m/s]
Entropy [kJ/kg-K]
Pre-shock velocity [m/s]
Adibatic index
Pre-shock velocity [m/s]

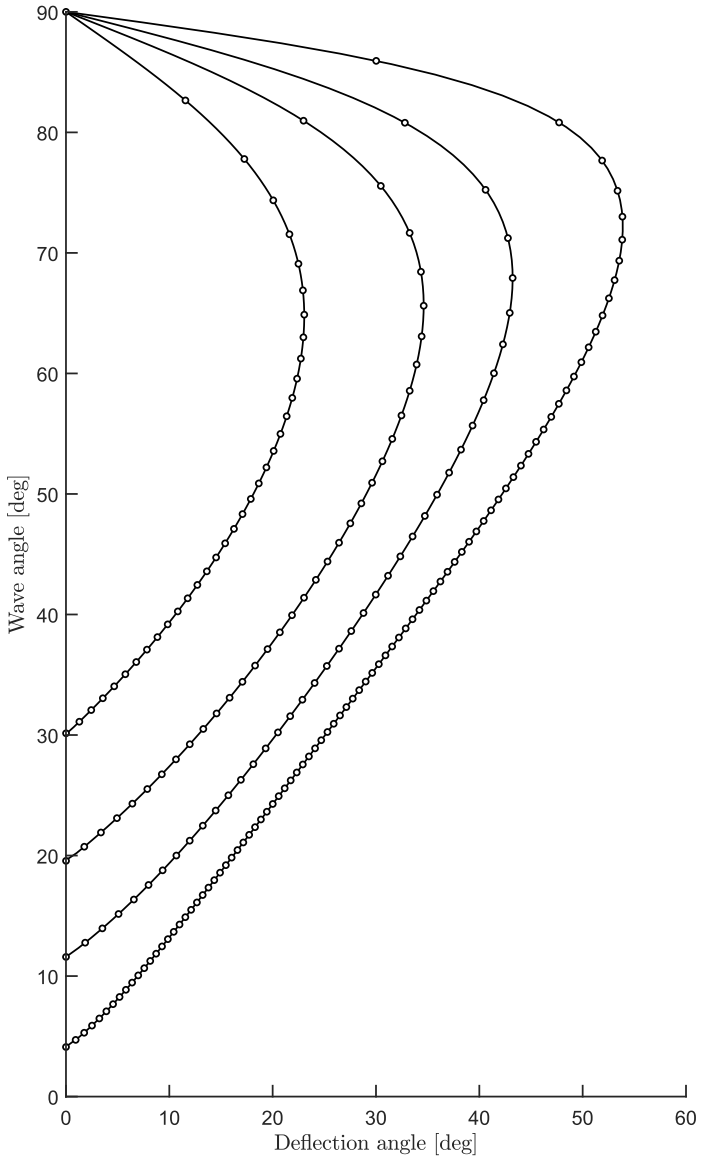

Specific heat pressure [kJ/kg-K]
Pre-shock velocity [m/s]
Specific heat volume [kJ/kg-K]
Pre-shock velocity [m/s]
(dV/dT)p
Pre-shock velocity [m/s]
(dV/dp)T
Pre-shock velocity [m/s]
Sound velocity [m/s]
Pre-shock velocity [m/s]
Molecular weight [g/mol]
Pre-shock velocity [m/s]

6.17 Validation SHOCK POLAR 1

- Contrasted with: Caltech's SD Toolbox within Cantera
- Problem type: Shock polar diagrams
- Temperature [K] = 300
- Pressure [bar] = 1.01325
- Initial mixture [moles]: $3.7619\mathrm{N_2} + \mathrm{O_2}$
- List of species considered = Frozen
- URL Folder Results SDToolbox: ./validations/sdtoolbox/data

To repeat the results, run:

```
run_validation_SHOCK_POLAR_SDToolbox_1.m
```


90
80
70
60
50
40
30
20
10
0
Wave angle [deg]
0
10
20
30
40
50
60
Deflection angle [deg]

6.18 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]: RP1 + $\frac{0.6723}{\phi}$LOX
- 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
```

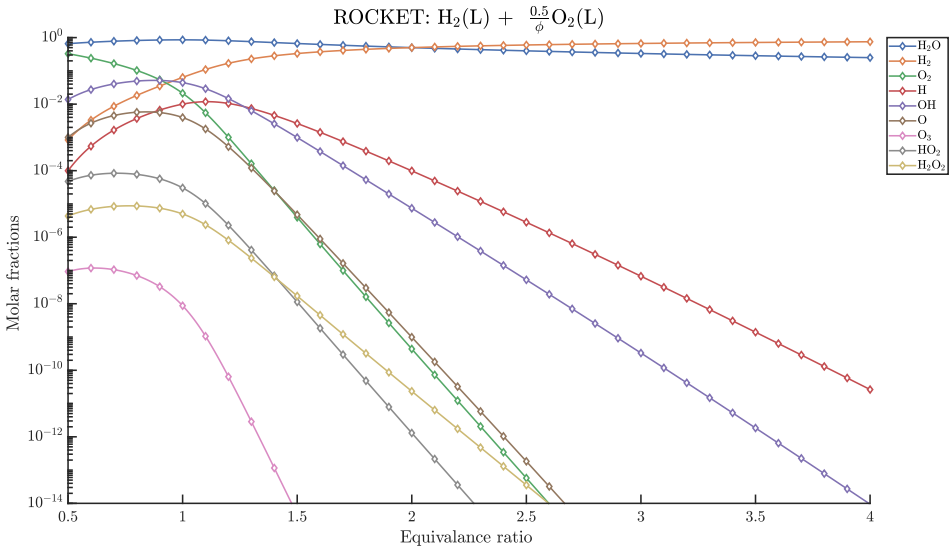
6.19 Validation ROCKET 2

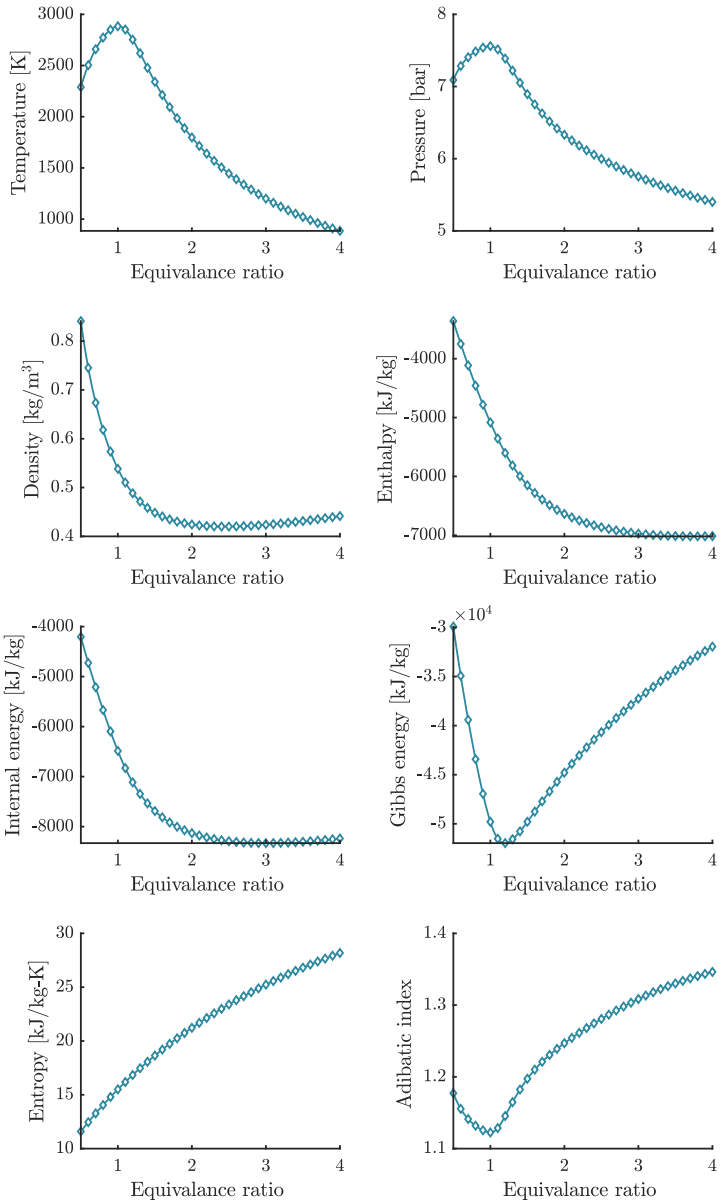
- Contrasted with: NASA's Chemical Equilibrium with Applications software
- Problem type: ROCKET
- Description: Equilibrium composition and performance parameters at exit of the rocket nozzle
- Temperature Fuel [K] = 20.27
- Temperature Oxid [K] = 90.17
- Chamber pressure [bar] = 22
- Infinite-Area-Chamber model (IAC)
- Area ratio $A_{\text{exit}}/A_{\text{throat}}$ = 3
- Initial mixture [moles]: LH2 + $\frac{0.5}{\phi}$LOX

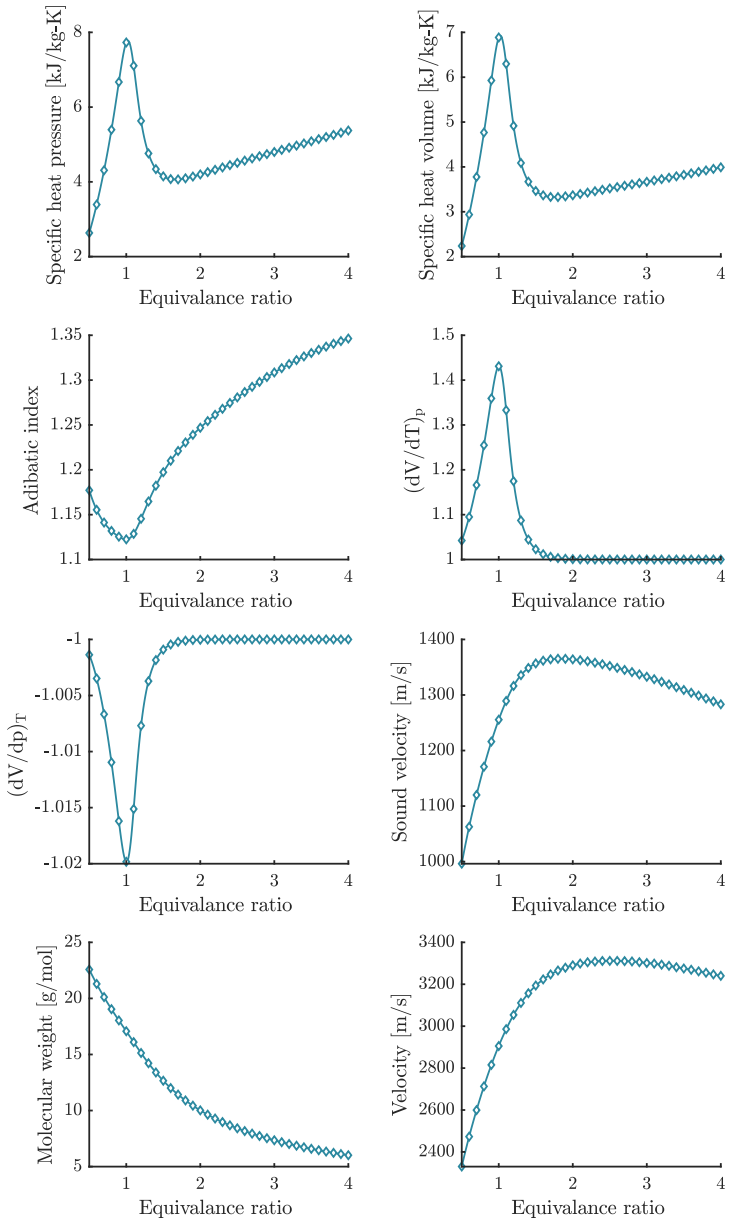
- List of species considered = HYDROGEN_L
- URL Folder Results CEA: ./validations/cea/rocket

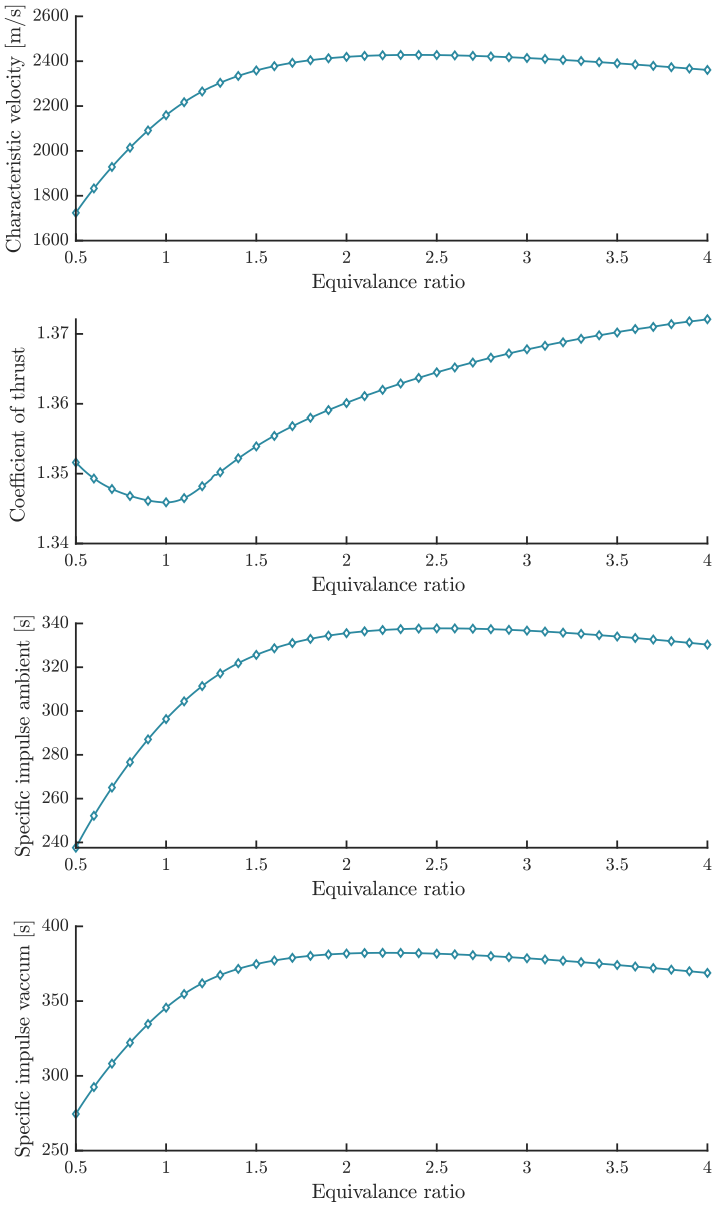
To repeat the results, run:

```
run_validation_ROCKET_CEA_17.m
```




Temperature [K]
Equivalance ratio
Pressure [bar]
Equivalance ratio
Density [kg/m^3]
Equivalance ratio
Enthalpy [kJ/kg]
Equivalance ratio
Internal energy [kJ/kg]
Equivalance ratio
Gibbs energy [kJ/kg]
$\times 10^4$
Equivalance ratio
Entropy [kJ/kg-K]
Equivalance ratio
Adibatic index
Equivalance ratio


Specific heat pressure [kJ/kg-K]
Equivalance ratio
Specific heat volume [kJ/kg-K]
Equivalance ratio
Adibatic index
Equivalance ratio
(dV/dT)p
Equivalance ratio
(dV/dp)T
Equivalance ratio
Sound velocity [m/s]
Equivalance ratio
Molecular weight [g/mol]
Equivalance ratio
Velocity [m/s]
Equivalance ratio

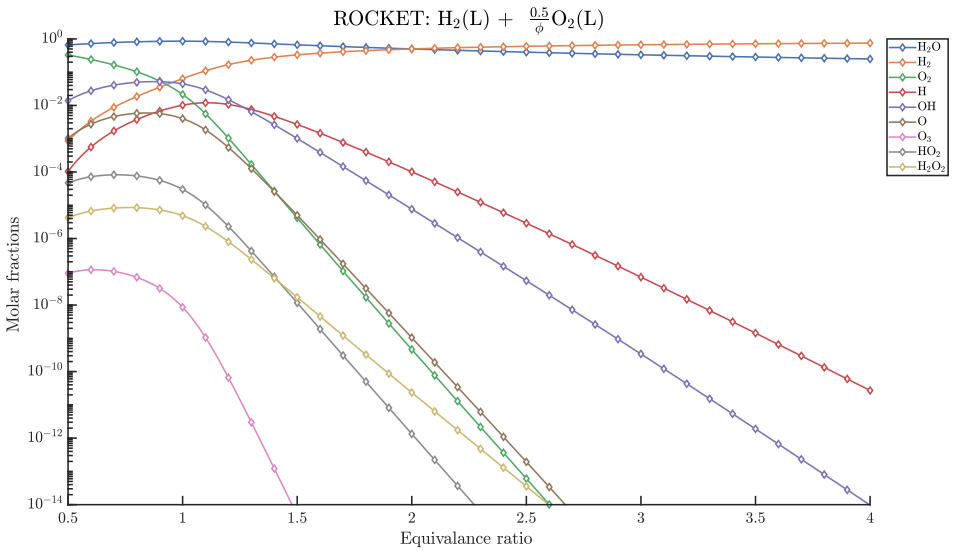

Characteristic velocity [m/s]
2600
2400
2200
2000
1800
1600
0.5
1
1.5
2
2.5
3
3.5
4
Equivalance ratio
Coefficient of thrust
1.37
1.36
1.35
1.34
Equivalance ratio
Specific impulse ambient [s]
340
320
300
280
260
240
Equivalance ratio
Specific impulse vaccum [s]
400
350
300
250
Equivalance ratio

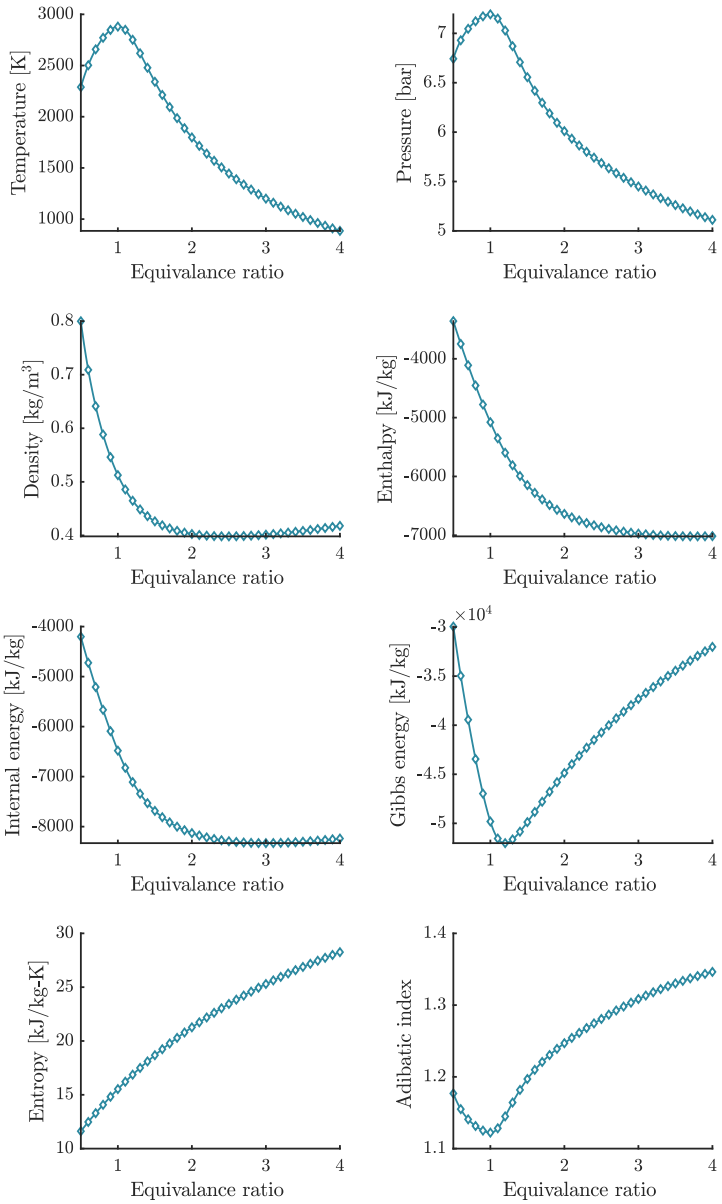
6.20 Validation ROCKET 3

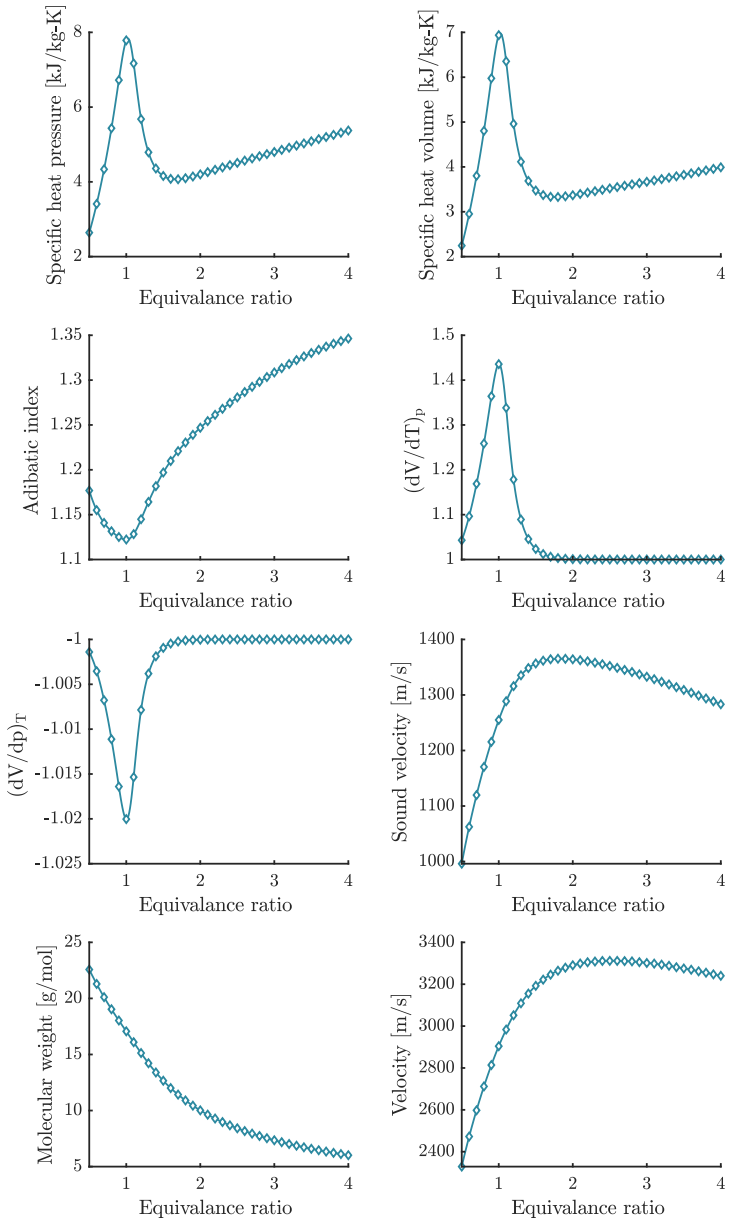
- Contrasted with: NASA's Chemical Equilibrium with Applications software
- Problem type: ROCKET
- Description: Equilibrium composition and performance parameters at exit of the rocket nozzle
- Temperature Fuel [K] = 20.27
- Temperature Oxid [K] = 90.17
- Chamber pressure [bar] = 22
- Finite-Area-Chamber model (FAC)
- Area ratio $A_{\text{chamber}}/A_{\text{throat}}$ = 2
- Area ratio $A_{\text{exit}}/A_{\text{throat}}$ = 3
- Initial mixture [moles]: LH2 + $\frac{0.5}{\phi}$LOX
- List of species considered = HYDROGEN_L
- URL Folder Results CEA: ./validations/cea/rocket

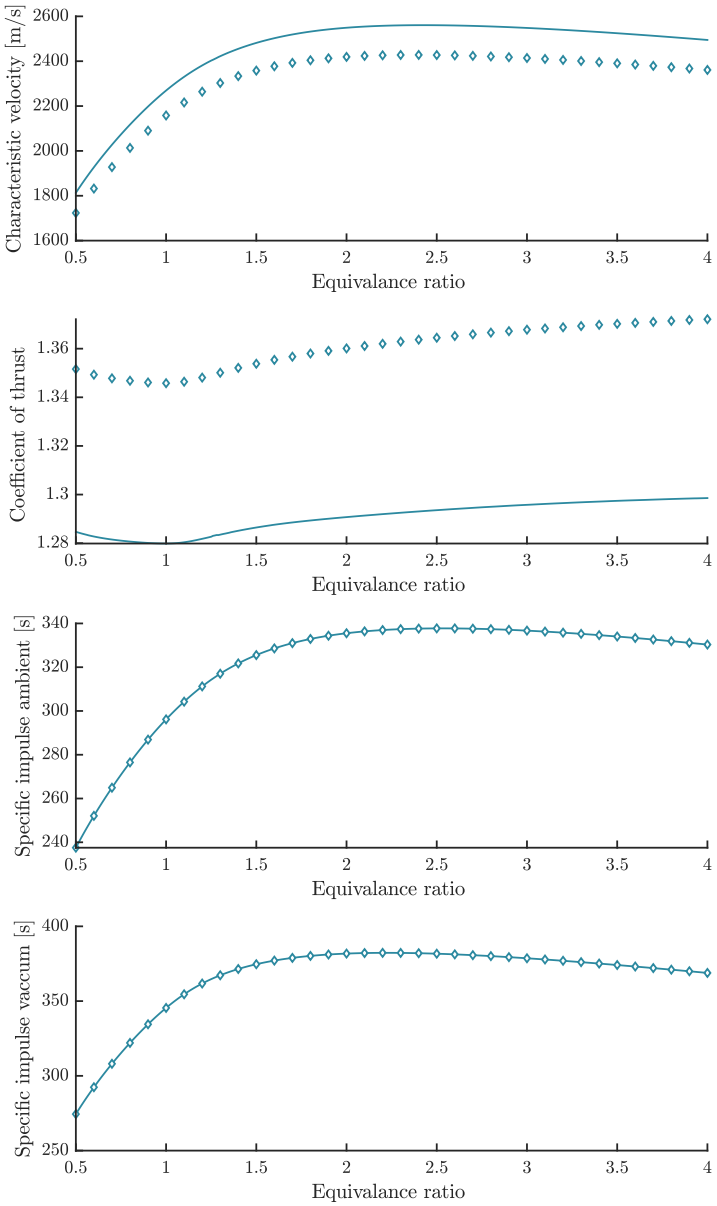
To repeat the results, run:

```
run_validation_ROCKET_CEA_18.m
```




Temperature [K]
Equivalance ratio
Pressure [bar]
Equivalance ratio
Density [kg/m^3]
Equivalance ratio
Enthalpy [kJ/kg]
Equivalance ratio
Internal energy [kJ/kg]
Equivalance ratio
Gibbs energy [kJ/kg]
$\times 10^4$
Equivalance ratio
Entropy [kJ/kg-K]
Equivalance ratio
Adibatic index
Equivalance ratio


Specific heat pressure [kJ/kg-K]
Equivalance ratio
Specific heat volume [kJ/kg-K]
Equivalance ratio
Adibatic index
Equivalance ratio
(dV/dT)p
Equivalance ratio
(dV/dp)T
Equivalance ratio
Sound velocity [m/s]
Equivalance ratio
Molecular weight [g/mol]
Equivalance ratio
Velocity [m/s]
Equivalance ratio

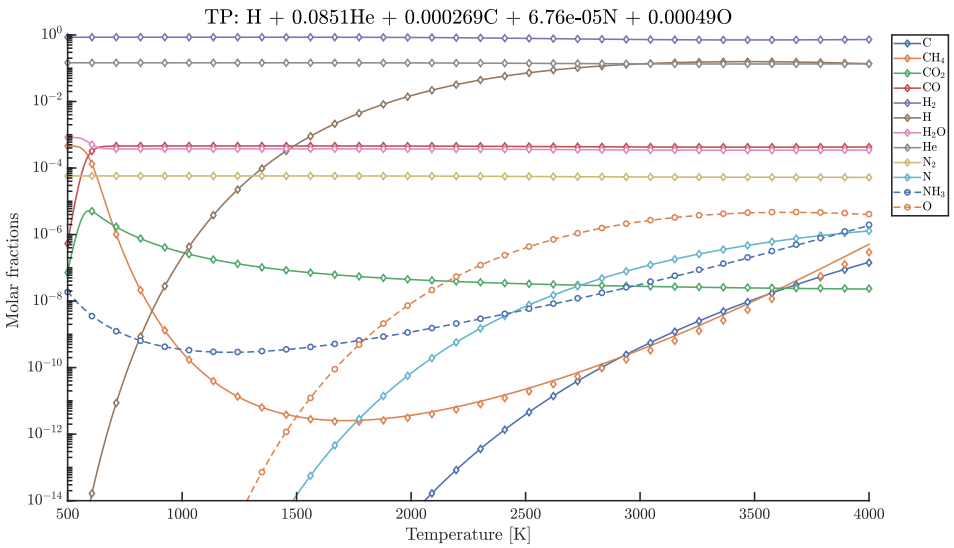

Characteristic velocity [m/s]
2600
2400
2200
2000
1800
1600
0.5
1
1.5
2
2.5
3
3.5
4
Equivalance ratio
Coefficient of thrust
1.36
1.34
1.32
1.3
1.28
0.5
1
1.5
2
2.5
3
3.5
4
Equivalance ratio
Specific impulse ambient [s]
340
320
300
280
260
240
0.5
1
1.5
2
2.5
3
3.5
4
Equivalance ratio
Specific impulse vaccum [s]
400
350
300
250
0.5
1
1.5
2
2.5
3
3.5
4
Equivalance ratio

6.21 Validation TEA 1

- Contrasted with: Thermochemical Equilibrium Abundances of chemical species software
- Problem type: Equilibrium composition at defined temperature and pressure
- 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
```

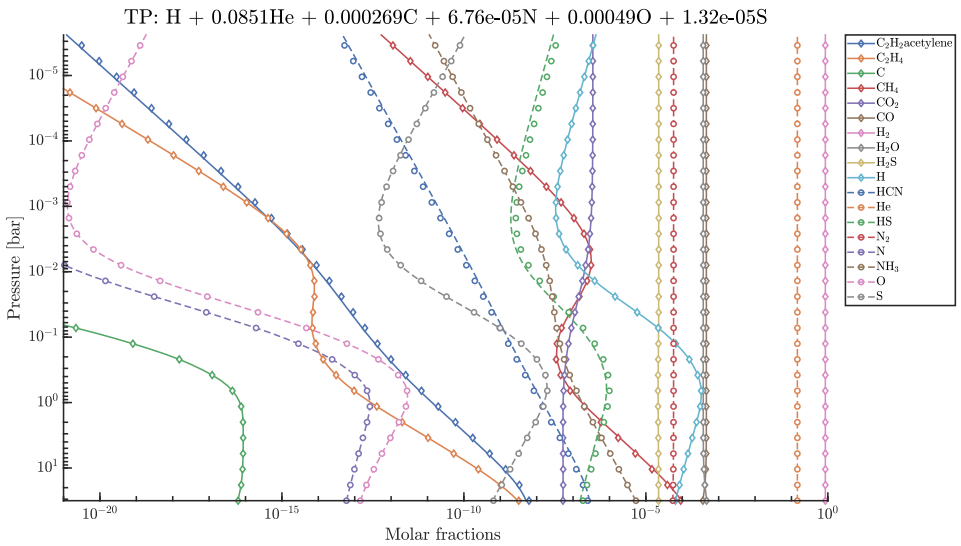


6.22 Validation TEA 2

- Contrasted with: Thermochemical Equilibrium Abundances of chemical species software
- Problem type: Equilibrium composition at defined temperature and pressure
- Description: Thermochemical equilibrium vertical distribution of WASP-43b with a metallicity $\zeta = 1$ assumming 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
```

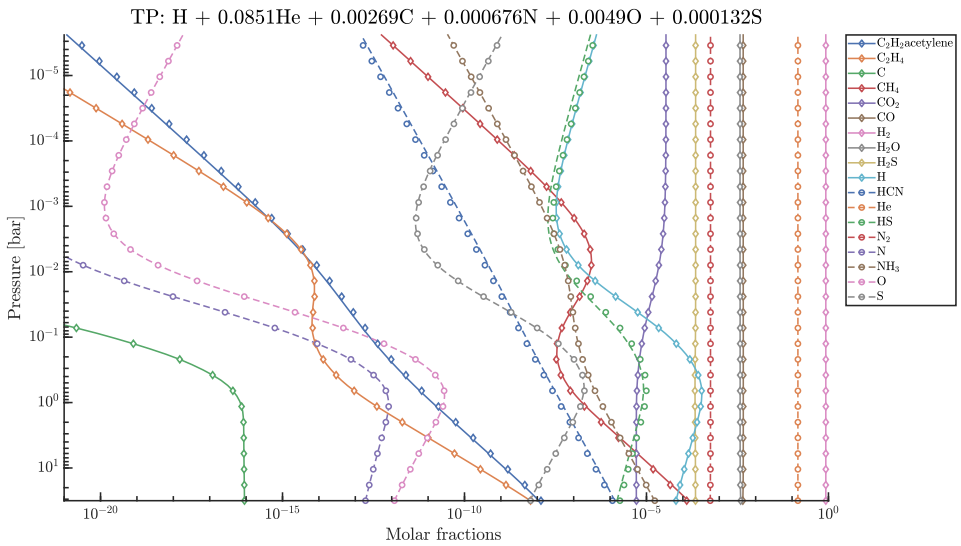


6.23 Validation TEA 3

- Contrasted with: Thermochemical Equilibrium Abundances of chemical species software
- Problem type: Equilibrium composition at defined temperature and pressure
- Description: Thermochemical equilibrium vertical distribution of WASP-43b with a metallicity $\zeta = 10$ assumming 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
```

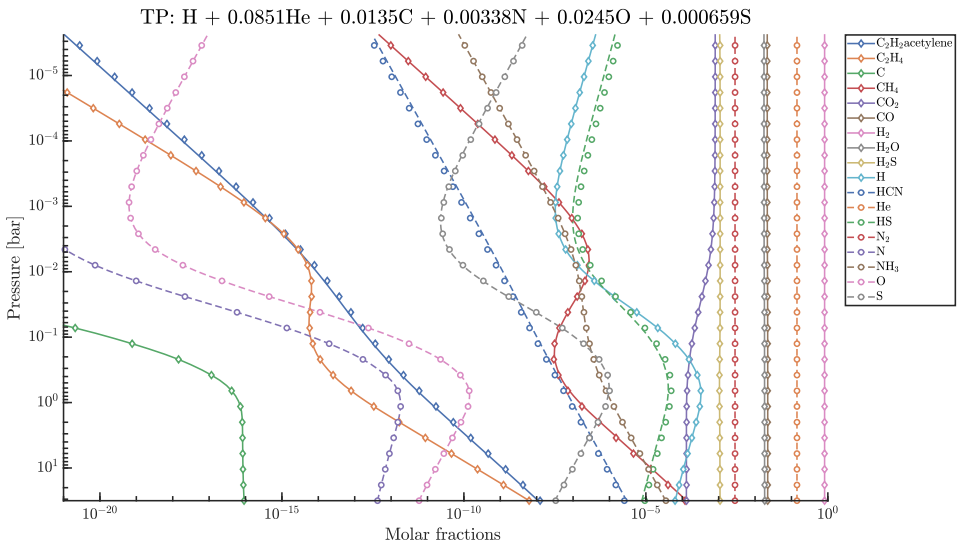


6.24 Validation TEA 4

- Contrasted with: Thermochemical Equilibrium Abundances of chemical species software
- Problem type: Equilibrium composition at defined temperature and pressure
- Description: Thermochemical equilibrium vertical distribution of WASP-43b with a metallicity $\zeta = 50$ assumming 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
```


TP: H + 0.0851He + 0.0135C + 0.00338N + 0.0245O + 0.000659S
Pressure [bar]
Molar fractions
10^{-5}
10^{-4}
10^{-3}
10^{-2}
10^{-1}
10^{0}
10^{1}
10^{-20}
10^{-15}
10^{-10}
10^{-5}
10^{0}
C_2H_2acetylene
C_2H_4
C
CH_4
CO_2
CO
H_2
H_2O
H_2S
H
HCN
He
HS
N_2
N
NH_3
O
S

CHAPTER

SEVEN

DOCUMENTATION

7.1 Databases

Combustion Toolbox generates its own databases using an up-to-date version of NASA's 9-coefficient polynomial fits [McBride, 2002] that incorporates the Third Millennium database [Burcat and Ruscic, 2005], including the available values from Active Thermochemical Tables. This fitting allows to evaluate the dimensionless thermodynamic functions of the species for the specific heat at constant pressure, enthalpy, and entropy as a 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 [McBride, 2002]. 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

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

This data is collected from the `thermo_CT.inp` file and next formatted into a more accessible structure (\textit{DB_master}) with the built-in function `generate_DB_master.m`. Then, for faster data access, we generate a new database `DB` using griddedInterpolant objects that contain piecewise cubic Hermite interpolating polynomials (PCHIP) [Fritsch and Carlson, 1980]. This allows the evaluation of the thermodynamic

functions at a given temperature with ease. 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**.

Note: For temperatures outside the bounds, we avoid the higher order terms of the polynomials by linear extrapolation, similar to Stock *et al.* [2018], extending the range of validity of the thermodynamic data available. It should be emphasized that this extension is limited to a narrow temperature range and may not apply to temperatures significantly outside of this range.

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)
```

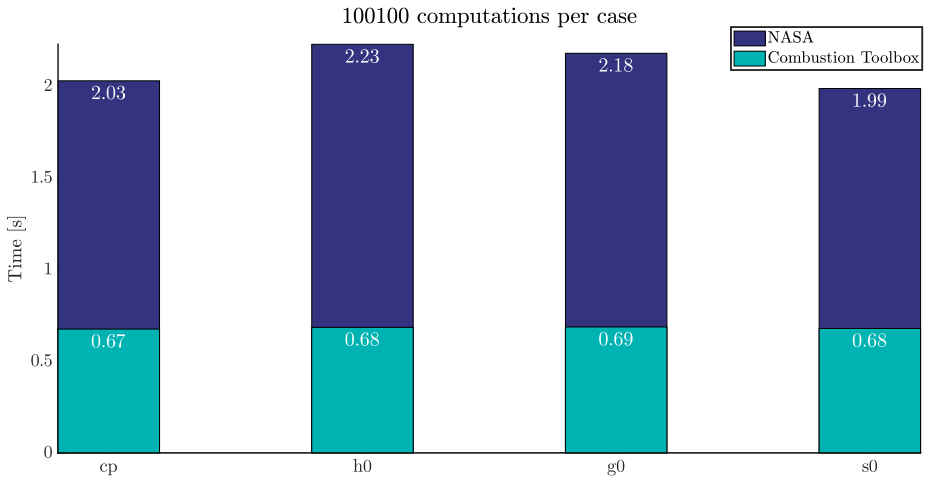


Figure 1: *Performance test, execution times for over 10^5 calculations of the specific heat at constant pressure, enthalpy, Gibbs energy, and entropy, denoted as c_p, h_0, g_0, and s_0, respectively, using the NASA's 9 coefficient polynomials (dark blue) and the piecewise cubic Hermite interpolating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

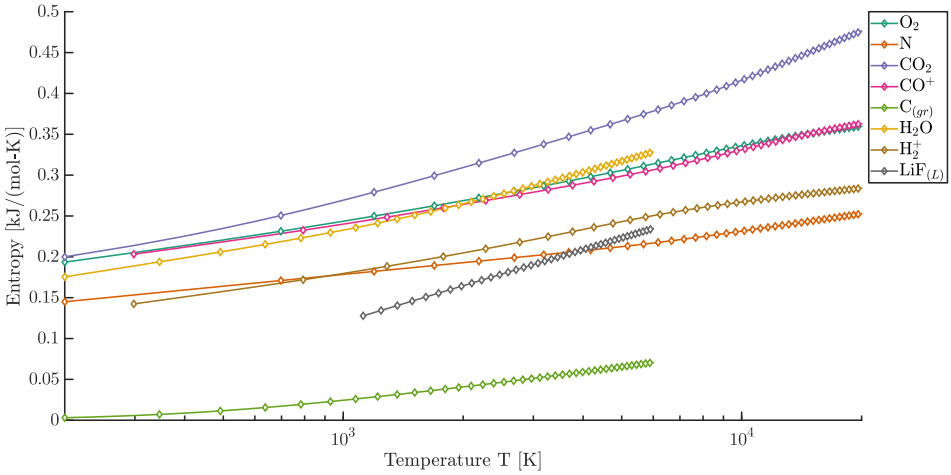


Figure 2: *Comparison of entropy [kJ/(mol-K)] as a function of temperature [K] obtained using the piecewise cubic Hermite interpolating polynomials (lines) and using the NASA's 9 coefficient polynomials (symbols) for a set of species.*

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

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

or in matricial form

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

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

$$\mathrm{H_2 + Br_2 \rightleftharpoons 2HBr} \tag{7.11}$$

which have only three species involve. The system obtained is

$$\begin{pmatrix} 0 & 2 & 1 \\ 2 & 0 & 1 \end{pmatrix} \begin{pmatrix} n_{H_2} \\ n_{Br_2} \\ n_{HBr} \end{pmatrix} - \begin{pmatrix} b^\circ_{Br} \\ b^\circ_{H} \end{pmatrix} = 0. \tag{7.12}$$

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
```

7.2 API Documentation

Note: Please note that the documentation is currently under development.

Here we can find the documentation for the routines implemented in the Combustion Toolbox (CT). The source code contains the following folders and files in the main directory:

```
.combustion_toolbox
|-- databases
|-- examples
|-- gui
|-- installer
|-- modules
|-- utils
|-- validations
|-- run_test.m
|-- CODE_OF_CONDUCT.md
|-- CONTENTS.m
|-- CONTRIBUTING.md
|-- CONTRIBUTORS.md
|-- INSTALL.m
```

(continues on next page)

(continued from previous page)

```
|-- LICENSE.md
`-- README.md
```

These top-layer folders contain the following:

- `databases`: encompasses all databases utilized by CT.
- `examples`: contains examples showcasing the usage of CT modules.
- `gui`: houses functions and assets required to generate the app (GUI), add-ons, and extend the functionality of the plain code to the GUI.
- `installer`: hosts installation files (toolbox and royalty-free stand-alone version).
- `modules`: contains functions for the different modules, including CT-EQUIL, CT-SD, and CT-ROCKET, along with routines for initializing CT.
- `utils`: accommodates utility functions with different purposes.
- `validations`: includes routines used to validate CT with the results obtained with other codes, unit testing files for ensuring correct code functionality, and all graphs generated from these verification processes.

Regarding the files in the main source folder, we have the following: the file `run_test.m` runs the unit tests of CT. The file `CONTENTS.m` is a script that briefly describes the problems that can be solved with CT. The file `INSTALL.m` is a script that installs the CT code and the GUI. The file `LICENSE.md` contains the license of CT (GNU General Public License v3.0). Finally, the file `README.md` is the official description in the GitHub repository.

7.2.1 Modules

Combustion Toolbox has implemented three different modules: `CT-EQUIL`, `CT-SD`, and `CT-ROCKET`. The core module, CT-EQUIL, minimizes the Gibbs/Helmholtz free energy of the system using Lagrange multipliers combined with a multidimensional Newton-Raphson method, upon the condition that the mixture properties are defined by two functions of state (e.g., enthalpy and pressure). CT-SD solves processes that involve strong changes in dynamic pressure, such as steady state shock and detonation waves in either normal or oblique stream configurations within the limits of regular shock reflections. Finally, CT-ROCKET estimates rocket propellant performance under ideal conditions. Additionally, this folder includes the initialization routines for the main variable `self`.

Initialization module

CT built-in functions are written to perform fast parametric studies, thus the data passed between the functions has been organized in a hierarchical tree structure (except for the GUI which is based on OOP) as shown in Fig.1, namely:

- `self (App)`: parent node; contains all the data of the code, e.g., databases, input values, and results.
- `Constants (C)`: contains constant values.
- `Elements (E)`: contains data of the chemical elements in the problem (names and indices for fast data access).
- `Species (S)`: contains data of the chemical species in the problem (names and indices for fast data access), as well as lists (cells) with the species for complete combustion.
- `Problem Description (PD)`: contains data of the problem to solve, e.g., initial mixture (composition, temperature, pressure), problem type, and its configuration.
- `Problem Solution (PS)`: contains results (mixtures).
- `Tuning Properties (TN)`: contains parameters that control the numerical error of the algorithms implemented in the different modules.
- `Miscellaneous (Misc)`: contains values that configure the auto-generated plots and export setup, as well as flags, e.g., setting FLAG_RESULTS = true (by default) the results are shown in the command window (only in the desktop environment).
- `Database master (DB_master)`: a structured thermochemical database including data from McBride [2002], Burcat and Ruscic [2005].
- `Database (DB)`: a structured thermochemical database with *griddedInterpolant* objects (see MATLAB built-in function griddedInterpolant.m) that contain piecewise cubic Hermite interpolating polynomials (PCHIP) [Fritsch and Carlson, 1980] for faster data access.

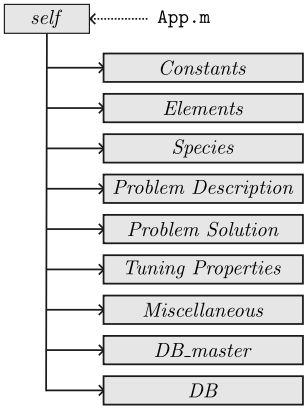


Figure 1: *Combustion Toolbox hierarchical data tree structure, where* `App.m` *is the initialization function.*

App

Routines to initialize the Combustion Toolbox.

Routines

App(*varargin*)

Generate self variable with all the data required to initialize the computations

Optional Args:

- LS (cell): List of species
- obj (class): Class combustion_toolbox_app (GUI)
- type (char): 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

---

Examples

- **self = App() % This initialization will consider all all the possible species** that can appear depending on the elements of the reactant (see routine find products.m)
- self = App('Air_ions') % Specify predefined list of species (see routine list_species.m)
- self = App({'N2', 'O2', 'NO', 'N', 'O'}) % Specify species to consider as possible products
- self = App('Complete') % Complete combustion
- self = App('fast', DB_master, DB) % Fast initialization that injects preloaded databases
- **self = App('fast', DB_master, DB, {'N2', 'O2', 'NO', 'N', 'O'}) % Fast initialization** that injects preloaded databases and considers the given list of species in the calculations
- **self_2 = App('copy', self, {'N2', 'O2', 'NO', 'N', 'O'}) % Copy previous initialization** in another variable with a different set of species in the calculation

- self = App(app) % Initialization for the GUI
- self = App(app, {'N2', 'O2', 'NO', 'N', 'O'}) % Initialization for the GUI

---

complete_initialize(*self*, *species*)

Complete initialization process

Parameters

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

Returns

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

contained_elements(*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 has 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 branch in the self variable (struct).

Routines

`Constants()`
Initialize struct with constants data

`description`
Description of the struct

Type
char

`release`
Release of the Combustion Toolbox

Type
char

`date`
Date of the release

Type
char

`R0`
Universal gas constant [J/(K mol)]

Type
float

`gravity`
Standard gravity [m/s2]

Type
float

A0

Stoichiometric Matrix

Type
struct

M0

Matrix with properties of each species

Type
struct

N_prop

Number of properties in properties_matrix

Type
struct

N0

Reduced Matrix with number of moles and phase of each species

Type
struct

MassorMolar

'mass' or 'molar'

Type
char

mintol_display

Minimum tolerance to display results

Type
float

l_phi

Length equivalence ratio vector

Type
float

composition_units

Possible values: mol, molar fraction or mass fraction

Type
char

Returns
self (struct) – Struct with constants data

Elements

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

Routines

`Elements()`
Initialize struct with elements data

`description`
Description of the struct

Type
char

`elements`
Cell with the elements in the periodic table

Type
cell

`NE`
Number of elements

Type
float

`ind_C`
Index element Carbon

Type
float

`ind_H`
Index element Hydrogen

Type
float

ind_O

Index element Oxygen

Type
float

ind_N

Index element Nytrogen

Type
float

ind_E

Index element Electron

Type
float

ind_S

Index element Sulfur

Type
float

ind_Si

Index element Silicon

Type
float

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 branch in the self variable (struct).

Routines

`Miscellaneous()`

Initialize struct with miscellaneous data

`description`

Description of the struct

Type
char

`timer_0`

Timer to measure the time of the computations (total time)

Type
float

`timer_loop`

Timer to measure the time of the computations (only calculation time)

Type
float

`config`

Struct with configuration data for plots

Type
struct

`FLAG_INITIALIZE`

Flag indicating self variable is not fully initialized

Type
bool

`FLAG_FIRST`

Flag indicating first calculation

Type
bool

FLAG_FOI

% Flag indicating that the reactant mixture has been checked

Type
bool

FLAG_ADDED_SPECIES

Flag indicating that there are added reactants species, because were not considered as products -> to recompute stochiometric matrix

Type
bool

FLAG_N_Fuel

Flag indicating that the number of moles of the fuel species are defined

Type
bool

FLAG_N_Oxidizer

Flag indicating that the number of moles of the oxidant species are defined

Type
bool

FLAG_N_Inert

Flag indicating that the number of moles of the inert species are defined

Type
bool

FLAG_WEIGHT

Flag indicating that the number of moles of the oxidizer/inert speces are defined from its weight percentage

Type
bool

FLAG_RESULTS

Flag to show results in the command window

Type
bool

FLAG_CHECK_INPUTS

Flag indicating that the algorithm has checked the input variables

Type
bool

FLAG_GUI

Flag indicating that the user is using the GUI

Type
bool

FLAG_LABELS

Flag (to be completed)

Type
bool

FLAG_PROP

Struct with flags indicanting if there are several values of the respective property (fieldname)

Type
struct

FLAG_LENGTH

Flag indicating parametric study

Type
struct

export_results

Struct with data to export results

Type
struct

index_LS_original

Vector with the indeces original list of products

Type
float

display_species

Struct with data to display species

Type
struct

i

Index of the current calculation

Type
float

Returns
self (struct) – Struct with miscellaneous data

ProblemDescription

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

Routines

`ProblemDescription()`
Initialize struct with problem description data

`description`
Description of the problem

Type
char

`ProblemType`
Type of problem (TP, HP, SP, TV, EV, SV, SHOCK_I, SHOCK_R, ...)

Type
char

`R_Fuel`
Fuel property matrix

Type
float

`R_Oxidizer`
Oxidizer property matrix

Type
float

`R_Inert`
Inert property matrix

Type
float

phi

Equivalence ratio (struct)

Type
struct

Fuel

Fuel data

Type
struct

TR

Temperature of reactants

Type
struct

pR

Pressure of reactants

Type
struct

TP

Temperature of products

Type
struct

pP

Pressure of products

Type
struct

vP_vR

Volume relation Products/Reactants

Type
struct

u1

Incident shock velocity

Type
struct

`overdriven`
Overdriven shock velocity

Type
struct

`theta`
Deflection angle - oblique shocks

Type
struct

`beta`
Wave angle - oblique shocks

Type
struct

`Aratio`
Area ratio exit/throat - rocket

Type
struct

`Aratio_c`
Area ratio combustion chamber/thoat - rocket

Type
struct

`S_Fuel`
Cell with the list of fuel species in the mixture

Type
cell

`N_Fuel`
Vector with the number of moles of the fuel species in the mixture

Type
float

`T_Fuel`
Vector with the temperature values of the fuel species in the mixture

Type
float

`S_Oxidizer`

Cell with the list of oxidizer species in the mixture

Type
cell

`N_Oxidizer`

Vector with the number of moles of the oxidizer species in the mixture

Type
float

`T_Oxidizer`

Vector with the temperature values of the oxidizer species in the mixture

Type
float

`S_Inert`

Cell with the list of inert species in the mixture

Type
cell

`N_Inert`

Vector with the number of moles of the inert species in the mixture

Type
float

`T_Inert`

Vector with the temperature values of the inert species in the mixture

Type
float

`ratio_oxidizers_O2`

Ratio oxidizers / O2 [% moles]

Type
float

`ratio_inerts_O2`

Ratio oxidizers / Inerts [% moles]

Type
float

`wt_ratio_oxidizers`

Weight ratio percentage of oxidizer species

Type
float

`wt_ratio_inerts`

Weight ratio percentage of inert species

Type
float

`EOS`

Equation of States

Type
struct

`FLAG_ION`

Flag to indicate if the system contains ionized species

Type
bool

`FLAG_TCHEM_FROZEN`

Flag to indicate if the thermodynamic properties are thermochemically frozen (calorically perfect gas)

Type
bool

`FLAG_FROZEN`

Flag to indicate if the thermodynamic properties are frozen (calorically imperfect gas with frozen chemistry)

Type
bool

`FLAG_IAC`

Flag to use IAC model for rocket computations

Type
bool

`FLAG_SUBSONIC`

Flag to indicate subsonic Area ratio (CT-ROCKET)

Type
bool

FLAG_EOS

Flag to use non-ideal Equation of States (EoS)

Type
bool

Returns
self (struct) – Struct with problem description data

ProblemSolution

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

Routines

ProblemSolution()

Initialize struct with problem solution data

Returns
self (struct) – struct with problem solution data

Species

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

Routines

Species()

Initialize struct with chemical species data

description

Description of the struct

Type
char

LS_DB

List of species in the database

Type
cell

NS_DB

Number of species in the database

Type
float

NG

Number of gaseous species in the mixture

Type
float

NS

Number of species in the mixture

Type
float

LS

List of species in the mixture

Type
cell

LS_formula

Formula of each species contained in LS

Type
cell

ind_nswt

Indeces gaseous species

Type
float

ind_swt

Indeces condensed species

Type
float

`ind_cryogenic`

Indeces cryogenic liquified species

Type
float

`ind_ox_ref`

Indeces reference oxidizer (default: O2)

Type
float

`ind_ions`

Indeces ionized species in LS

Type
float

`ind_react`

Indeces react species

Type
float

`ind_frozen`

Indeces inert/frozen species

Type
float

`LS_lean`

List of species for a lean complete combustion (equivalence ratio < 1)

Type
cell

`LS_rich`

List of species for a lean complete combustion (equivalence ratio > 1)

Type
cell

`LS_soot`

List of species for a lean complete combustion (equivalence ratio > equivalence ratio soot)

Type
cell

FLAG_COMPLETE

Flag indicating if the complete combustion is considered

Type
bool

FLAG_BURCAT

Find all the combinations of species from the database (without BURCAT's DB) that can appear as products for the given list of reactants

Type
bool

FLAG_ION

Flag indicating to include ionized species in the automatic finder of species

Type
bool

Returns
self (struct) – struct with chemical species 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

`list_species`(*varargin*)

Set list of species in the mixture (products)

Predefined list of species:

- SOOT FORMATION (default)
- COMPLETE
- HC/O2/N2 EXTENDED
- SOOT FORMATION EXTENDED
- NASA ALL
- NASA ALL CONDENSED
- NASA ALL IONS
- AIR, DISSOCIATED AIR

- AIR IONS, AIR_IONS
- IDEAL_AIR, AIR_IDEAL
- HYDROGEN
- HYDROGEN_L, HYDROGEN (L)
- HC/O2/N2 PROPELLANTS
- SI/HC/O2/N2 PROPELLANTS

Optional Args:

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

Returns

Tuple containing

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

Examples

- LS = list_species('soot formation');
- [self, LS] = list_species(self, 'soot formation');
- [self, LS] = list_species(self, 'complete', 1.5, 2.5);

TuningProperties

Routines to initialize the TuningProperties branch in the self variable (struct).

Routines

`TuningProperties()`

Initialize struct with tunning properties attributes

`FLAG_FAST`

Flag indicating use guess composition of the previous computation (default: true)

Type
bool

`FLAG_EXTRAPOLATE`

Flag indicating linear extrapolation of the polynomials fits (default: true)

Type
bool

`tolN`

Tolerance of the composition of the mixture (default: 1e-14)

Type
float

`tol_gibbs`

Tolerance of the Gibbs/Helmholtz minimization method (default: 1e-05)

Type
float

`itMax_gibbs`

Max number of iterations - Gibbs/Helmholtz minimization method (default: 70)

Type
float

`tolN_guess`

Tolerance of the molar composition of the mixture (guess) (default: 1e-06)

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

`itMax_ions`
: Max number of iterations - charge balance (ions) (default: 30)

  Type
  float

`T_ions`
: Minimum temperature [K] to consider ionized species (default: 0)

  Type
  float

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

  Type
  float

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

  Type
  float

`root_method`
: Root finding method (default: newton)

  Type
  function

`root_T0_l`
: First guess T [K] left branch - root finding method (default: 1000)

  Type
  float

`root_T0_r`
: First guess T [K] right branch - root finding method (default: 3000)

  Type
  float

`root_T0`

Guess T[K] if it's of previous range - root finding method (default: 3000)

Type
float

`tol_shocks`

Tolerance of shocks/detonations routines (default: 1e-05)

Type
float

`it_shocks`

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

Type
float

`Mach_thermo`

Pre-shock Mach number above which T2_guess will be computed considering h2 = h1 + u1^2 / 2 (default: 2)

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_limitRR`

Tolerance to calculate the limit of regular reflections (default: 1e-04)

Type
float

`it_limitRR`
Max number of iterations - limit of regular reflections (default: 10)

Type
float

`it_guess_det`

5)

Type
float

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

Type
float

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

Type
float

`tol_eos`
Tolerance of the EoS (default: 1e-04)

Type
float

`it_eos`
Max number of iterations - EoS (default: 30)

Type
float

Returns
self (struct) – struct with tunning properties data

Thermochemical equilibrium module

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 and McBride [1994].

Thermodynamic derivatives

All thermodynamic first derivatives can be obtained with any set of three independent first derivatives [McBride, 1996]. 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

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

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

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

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

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

Routines

complete_combustion(*self*, *mix*, *phi*)

Solve chemical equilibrium for CHNO mixtures assuming a complete combustion using the equilibrium constants method

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

---

Example

[moles, species] = complete_combustion(self, self.PS.strR{1}, 0.5)

`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

Example

mix2 = equilibrate(self, self.PS.strR{1}, 1.01325)

`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

Example

mix2 = equilibrate(self, self.PS.strR{1}, 1.01325, 3000)

---

equilibrate_T_tchem(*self*, *mix1*, *pP*, *TP*)

Obtain equilibrium properties and composition for the given temperature [K] and pressure [bar] assuming a calorically perfect gas

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]

Returns
mix2 (struct) – Properties of the final mixture assuming a calorically perfect gas

---

Example

mix2 = equilibrate_T_tchem(self, self.PS.strR{1}, 1.01325, 3000)

---

equilibrium_dT(*J*, *N0*, *A0*, *NE*, *ind_nswt*, *ind_swt*, *ind_elem*, *H0RT*)

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

- **J** (float) – Matrix J to solve the linear system J*x = b
- **N0** (float) – Equilibrium composition [moles]
- **A0** (float) – Stoichiometric matrix
- **NE** (float) – Temporal total number of elements
- **ind** (float) – Temporal index of species in the final mixture
- **ind_nswt** (float) – Temporal index of gaseous species in the final mixture

- **ind_swt** (float) – Temporal index of condensed species in the final mixture
- **ind_elem** (float) – Temporal index of elements in the final mixture
- **H0RT** (float) – Dimensionless enthalpy

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

---

Example

[dNi_T, dN_T] = equilibrium_dT(J, N0, A0, NE, ind, ind_nswt, ind_swt, ind_elem, H0RT)

---

equilibrium_dT_large(*self*, *moles*, *T*, *A0*, *NG*, *NS*, *NE*, *ind*, *ind_nswt*, *ind_swt*, *ind_E*)

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]
- **A0** (float) – Stoichiometric matrix
- **NG** (float) – Temporal total number of gaseous species
- **NS** (float) – Temporal total number of species
- **NE** (float) – Temporal total number of elements
- **ind** (float) – Temporal index of species in the final mixture
- **ind_nswt** (float) – Temporal index of gaseous species in the final mixture
- **ind_swt** (float) – Temporal index of condensed species in the final mixture
- **ind_E** (float) – Temporal index of elements in the final 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

Example

[dNi_T, dN_T] = equilibrium_dT_large(self, moles, T, A0, NG, NS, NE, ind, ind_nswt, ind_swt, ind_E)

equilibrium_dp(*J*, *N0*, *A0*, *NE*, *ind_nswt*, *ind_swt*, *ind_elem*)

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

- **J** (float) – Matrix J to solve the linear system J*x = b
- **N0** (float) – Equilibrium composition [moles]
- **A0** (float) – Stoichiometric matrix
- **NE** (float) – Temporal total number of elements
- **ind** (float) – Temporal index of species in the final mixture
- **ind_nswt** (float) – Temporal index of gaseous species in the final mixture
- **ind_swt** (float) – Temporal index of condensed species in the final mixture
- **ind_elem** (float) – Temporal index of elements in the final 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

Example

[dNi_p, dN_p] = equilibrium_dp(J, N0, A0, NE, ind, ind_nswt, ind_swt, ind_elem)

equilibrium_dp_large(*self*, *N0*, *A0*, *NG*, *NS*, *NE*, *ind*, *ind_nswt*, *ind_swt*, *ind_E*)

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
- **N0** (float) – Equilibrium composition [moles]
- **A0** (float) – Stoichiometric matrix
- **NG** (float) – Temporal total number of gaseous species
- **NS** (float) – Temporal total number of species
- **NE** (float) – Temporal total number of elements
- **ind** (float) – Temporal index of species in the final mixture
- **ind_nswt** (float) – Temporal index of gaseous species in the final mixture
- **ind_swt** (float) – Temporal index of condensed species in the final mixture
- **ind_E** (float) – Temporal index of elements in the final 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

---

Example

[dNi_p, dN_p] = equilibrium_dp_large(self, N0, A0, NG, NS, NE, ind, ind_nswt, ind_swt, ind_E)

---

equilibrium_gibbs(*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 temperature and pressure.

The algorithm implemented take advantage of the sparseness of the upper left submatrix obtaining a matrix J of size NE + NS - NG + 1.

This method is based on the method outlined in 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): Composition matrix [n_i, FLAG_CONDENSED_i] for the given temperature [K] and pressure [bar] at equilibrium
- 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
- 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
- ind (float): List of chemical species indices
- STOP (float): Relative error in moles of species [-]

Examples

- N0 = equilibrium_gibbs(self, 1.01325, 3000, self.PS.strR{1}, [])
- [N0, dNi_T, dN_T, dNi_p, dN_p, ind, STOP, STOP_ions] = equilibrium_gibbs(self, 1.01325, 3000, self.PS.strR{1}, [])

---

equilibrium_gibbs_eos(*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 temperature and pressure.

This method is based on the method outlined in 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]
- 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
- 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
- STOP (float): Relative error in moles of species [-]
- STOP_ions (float): Relative error in moles of ionized species [-]

Example

[N0, dNi_T, dN_T, dNi_p, dN_p, STOP, STOP_ions] = equilibrium_gibbs_eos(self, pP, TP, mix1, guess_moles)

---

equilibrium_gibbs_large(*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 temperature and pressure.

This method is based on the method outlined in 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]
- 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
- 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
- ind (float): List of chemical species indices
- STOP (float): Relative error in moles of species [-]
- STOP_ions (float): Relative error in moles of ionized species [-]

Examples

- N0 = equilibrium_gibbs_large(self, 1.01325, 3000, self.PS.strR{1}, [])
- [N0, dNi_T, dN_T, dNi_p, dN_p, ind, STOP, STOP_ions] = equilibrium_gibbs_large(self, 1.01325, 3000, self.PS.strR{1}, [])

---

equilibrium_helmholtz(*self*, *vP*, *TP*, *mix1*, *guess_moles*)

Obtain equilibrium composition [moles] for the given temperature [K] and volume [m3]. 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 temperature and volume.

The algorithm implemented take advantage of the sparseness of the upper left submatrix obtaining a matrix J of size NE + NS - NG.

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

Parameters

- **self** (struct) – Data of the mixture, conditions, and databases
- **vP** (float) – Volume [m3]
- **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]
- 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
- 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
- ind (float): List of chemical species indices
- STOP (float): Relative error in moles of species [-]
- STOP_ions (float): Relative error in moles of ionized species [-]

Examples

- N0 = equilibrium_helmholtz(self, 0.0716, 3000, self.PS.strR{1}, [])
- [N0, dNi_T, dN_T, dNi_p, dN_p, ind, STOP, STOP_ions] = equilibrium_helmholtz(self, 0.0716, 3000, self.PS.strR{1}, [])

---

equilibrium_helmholtz_eos(*self*, *vP*, *TP*, *mix1*, *guess_moles*)

Obtain equilibrium composition [moles] for the given temperature [K] and volume [m3]. 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 temperature and volume.

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

Parameters

- **self** (struct) – Data of the mixture, conditions, and databases
- **vP** (float) – Volume [m3]
- **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]
- 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
- 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
- STOP (float): Relative error in moles of species [-]
- STOP_ions (float): Relative error in moles of ionized species [-]

Example

[N0, dNi_T, dN_T, dNi_p, dN_p, STOP, STOP_ions] = equilibrium_helmholtz_eos(self, vP, TP, mix1, guess_moles)

---

equilibrium_helmholtz_large(*self*, *vP*, *TP*, *mix1*, *guess_moles*)

Obtain equilibrium composition [moles] for the given temperature [K] and volume [m3]. 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 temperature and volume.

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

Parameters

- **self** (struct) – Data of the mixture, conditions, and databases
- **vP** (float) – Volume [m3]
- **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]
- 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
- 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
- ind (float): List of chemical species indices
- STOP (float): Relative error in moles of species [-]
- STOP_ions (float): Relative error in moles of ionized species [-]

Examples

- N0 = equilibrium_helmholtz_large(self, 0.0716, 3000, self.PS.strR{1}, [])
- [N0, dNi_T, dN_T, dNi_p, dN_p, ind, STOP, STOP_ions] = equilibrium_helmholtz_large(self, 0.0716, 3000, self.PS.strR{1}, [])

Shocks and detonations module

In this section, you will find the documentation of the routines implemented to solves processes that involve strong changes in dynamic pressure, such as steady state shock and detonation waves in either normal or oblique stream configurations within the limits of regular shock reflections.

Note: The kernel of the incident, reflected, and Chapman-Jouguet detonations are based on Gordon and McBride [1994].

Routines

det_cj(*self*, *mix1*, *varargin*)

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

This method is based on the method outlined in 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

---

Examples

- [mix1, mix2] = det_cj(self, self.PS.strR{1})
- [mix1, mix2] = det_cj(self, self.PS.strR{1}, self.PS.strP{1})

---

det_compute_guess(*self*, *mix1*, *phi*, *drive_factor*)

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 [-]
- **drive_factor** (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 [-]

Example

[P, T, M1, R, Q, STOP] = det_compute_guess(self, self.PS.strR{1}, 1, 2)

---

det_compute_guess_CEA(*self*, *mix1*)

Obtain guess of the jump conditions for a Chapman-Jouguet detonation as in NASA's CEA code (see Sec. 8.3 of [1])

[1] 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

Returns

Tuple containing

- P (float): Pressure ratio [-]
- T (float): Temperature ratio [-]
- STOP (float): Relative error [-]

Example

[P, T, STOP] = det_compute_guess_CEA(self, self.PS.strR{1})

`det_oblique_beta`(*self*, *mix1*, *drive_factor*, *beta*, *varargin*)

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

Parameters

- **`self`** (`struct`) – Data of the mixture, conditions, and databases
- **`mix1`** (`struct`) – Properties of the mixture in the pre-shock state
- **`drive_factor`** (`float`) – Drive_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-detonation state
- mix2_1 (struct): Properties of the mixture in the post-detonation state - weak detonation
- mix2_2 (struct): Properties of the mixture in the post-detonation state - strong detonation

---

Examples

- [mix1, mix2_1, mix2_2] = det_oblique_beta(self, self.PS.strR{1}, 2, 60)
- [mix1, mix2_1, mix2_2] = det_oblique_beta(self, self.PS.strR{1}, 2, 60, self.PS.strP{1})

---

det_oblique_theta(*self*, *mix1*, *drive_factor*, *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
- **drive_factor** (float) – Drive 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-detonation state
- mix2_1 (struct): Properties of the mixture in the post-detonation state - weak detonation
- mix2_2 (struct): Properties of the mixture in the post-detonation state - strong detonation

---

Examples

- [mix1, mix2_1, mix2_2] = det_oblique_theta(self, self.PS.strR{1}, 2, 30)
- [mix1, mix2_1, mix2_2] = det_oblique_theta(self, self.PS.strR{1}, 2, 30, self.PS.strP{1})

---

det_overdriven(*self*, *mix1*, *drive_factor*, *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
- **drive_factor** (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

---

Examples

- [mix1, mix2] = det_overdriven(self, mix1, 1.5)
- [mix1, mix2] = det_overdriven(self, mix1, 1.5, mix2)

---

det_polar(*self*, *mix1*, *drive_factor*, *varargin*)

Compute detonation polar diagrams

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

---

Examples

- [mix1, mix2] = det_polar(self, mix1, 3000)
- [mix1, mix2] = det_polar(self, mix1, 3000, mix2)

---

det_underdriven(*self*, *mix1*, *drive_factor*, *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
- **drive_factor** (float) – Underdriven 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

---

Examples

- [mix1, mix2] = det_underdriven(self, mix1, 1.5)
- [mix1, mix2] = det_underdriven(self, mix1, 1.5, mix2)

---

shock_ideal_gas(*gamma*, *M1*)

Compute jump conditions assuming a thermochemically frozen gas (calorically perfect 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 [-]

- Gammas (float): Rankine-Hugoniot slope parameter [-]
- M1 (float): Pre-shock Mach number [-]

Example

[R, P, T, Gammas, M1] = shock_ideal_gas(1.4, 2.0)

---

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

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

This method is based on the method outlined in 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

---

Examples

- [mix1, mix2] = shock_incident(self, mix1, u1)
- [mix1, mix2] = shock_incident(self, mix1, u1, mix2)

---

shock_incident_2(*self*, *mix1*, *u1*, *varargin*)

Compute pre-shock and post-shock states of a planar incident 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 in the post-shock state

---

Examples

[mix1, mix2] = shock_incident_2(self, mix1, u1) [mix1, mix2] = shock_incident_2(self, mix1, u1, mix2)

---

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

---

Examples

- [mix1, mix2] = shock_oblique_beta(self, mix1, u1, beta)
- [mix1, mix2] = shock_oblique_beta(self, mix1, u1, beta, mix2)

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

Examples

- [mix1, mix2, mix5_1, mix5_2] = shock_oblique_reflected_theta(self, mix1, u2, theta, mix2)
- [mix1, mix2, mix5_1, mix5_2] = shock_oblique_reflected_theta(self, mix1, u2, theta, mix2, mix5_1, mix5_2)

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

Examples

- [mix1, mix2_1, mix2_2] = shock_oblique_theta(self, mix1, u1, theta)
- [mix1, mix2_1, mix2_2] = shock_oblique_theta(self, mix1, u1, theta, mix2_1, mix2_2)

---

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

Compute shock polar diagrams

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

---

Examples

- [mix1, mix2] = shock_polar(self, mix1, u1)
- [mix1, mix2] = shock_polar(self, mix1, u1, mix2)

---

shock_polar_limitRR(*self*, *mix1*, *u1*)

Obtain polar curves for the given pre-shock conditions using Broyden's method

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]

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 - polar diagrams from mix1 (incident)

- mix2_1 (struct): Properties of the mixture in the post-shock state - weak shock
- mix3 (struct): Properties of the mixture in the post-shock state - polar diagrams from mix2_1 (reflected)

Example

[mix1, mix2, mix2_1, mix3] = shock_polar_limitRR(self, mix1, u1)

---

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 the method outlined in 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

---

Examples

- [mix1, mix2, mix5] = shock_reflected(self, mix1, u1, mix2)

- [mix1, mix2, mix5] = shock_reflected(self, mix1, u1, mix2, mix5)

Rocket module

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

- IAC: Infinite-Area-Chamber,
- FAC: Finite-Area-Chamber.

Note: This module is based on Gordon and McBride [1994].

Routines

compute_FAC(*self*, *mix1*, *mix2_inj*, *mix2_c*, *mix3*)

Compute chemical equilibria at the injector, outlet of the chamber and at the throat using the Finite-Area-Chamber (FAC) model

This method is based on the method outlined in 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
- **mix2_inj** (struct) – Properties of the mixture at the injector of the chamber (previous calculation)
- **mix2_c** (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

- mix2_inj (struct): Properties of the mixture at the injector of the chamber
- mix2_c (struct): Properties of the mixture at the outlet of the chamber
- mix3 (struct): Properties of the mixture at the throat

Example

[mix2_inj, mix2_c, mix3] = compute_FAC(self, mix1, mix2_inj, mix2_c, mix3)

`compute_chamber_IAC`(*self*, *mix1*, *mix2*)

Compute chemical equilibria at the exit of the chamber (HP) using the Infinite-Area-Chamber (IAC) model

This method is based on the method outlined in 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
- **mix2** (struct) – Properties of the mixture at the outlet of the chamber (previous calculation)

Returns

mix2 (struct) – Properties of the mixture at the outlet of the chamber

Example

mix2 = compute_chamber_IAC(self, mix1, mix2)

`compute_exit`(*self*, *mix2*, *mix3*, *mix4*, *Aratio*, *varargin*)

Compute thermochemical composition for a given Aratio

This method is based on the method outlined in 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
- **mix4** (struct) – Properties of the mixture at the exit (previous calculation)
- **Aratio** (struct) – Ratio area_exit / area_throat

Optional Args:

mix2_in (struct): Properties of the mixture at the inlet of the chamber

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

Examples

- mix4 = compute_exit(self, mix2, mix3, mix4, Aratio)
- mix4 = compute_exit(self, mix2, mix3, mix4, Aratio, mix2_in)

`compute_throat_IAC`(*self*, *mix2*, *mix3*)

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

This method is based on the method outlined in 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

Example

mix3 = compute_throat_IAC(self, mix2, mix3)

`guess_pressure_IAC_model`(*mix*)

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

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

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

Returns
pressure (float) – Pressure at the throat [bar]

Example

pressure = guess_pressure_IAC_model(mix)

---

guess_pressure_exit_IAC(*mix2*, *mix3*, *Aratio*, *FLAG_SUBSONIC*)

Compute guess logarithm of the ratio pressure_inf / pressure_exit [-] for the given Area ratio [-] and indicanting if the point of interest is in the subsonic area ratios or the supersonic area ratios

This method is based on the method outlined in 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
- **Aratio** (struct) – Ratio area_exit / area_throat
- **FLAG_SUBSONIC** (bool) – Flag indicating if the Aratio refer to the subsonic region or the supersonic region

Returns

log_P (float) – Log pressure ratio [-]

Example

log_P = guess_pressure_exit_IAC(mix2, mix3, 3, false)

---

rocket_parameters(*mix2*, *mix3*, *gravity*, *varargin*)

Compute Rocket performance parameters at the throat

This method is based on the method outlined in 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
- **gravity** (float) – Gravitational acceleration [m/s2]

Returns

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

rocket_performance(*self*, *mix1*, *varargin*)

Routine that computes the propellant rocket performance

Methods implemented:

- Infinite-Area-Chamber (IAC)
- Finite-Area-Chamber (FAC)

This method is based on the method outlined in 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:

- Aratio (struct): Ratio area_exit / area_throat
- mix2_inj (struct): Properties of the mixture at the injector (previous calculation)
- mix2_c (struct): Properties of the mixture at the outlet of the chamber (previous calculation)
- mix3 (struct): Properties of the mixture at the throat (previous calculation)
- mix4 (struct): Properties of the mixture at the given exit points (previous calculation)

Returns

Tuple containing

- mix1 (struct): Properties of the initial mixture
- mix2_inj (struct): Properties of the mixture at the injector
- mix2_c (struct): Properties of the mixture at the outlet of the chamber
- mix3 (struct): Properties of the mixture at the throat
- mix4 (struct): Properties of the mixture at the given exit points

---

Examples

- [mix1, mix2_inj, mix2_c, mix3, mix4] = rocket_performance(self, mix1)
- [mix1, mix2_inj, mix2_c, mix3, mix4] = rocket_performance(self, mix1, Aratio)
- [mix1, mix2_inj, mix2_c, mix3, mix4] = rocket_performance(self, mix1, Aratio, mix2_inj)
- [mix1, mix2_inj, mix2_c, mix3, mix4] = rocket_performance(self, mix1, Aratio, mix2_inj, mix2_c)
- [mix1, mix2_inj, mix2_c, mix3, mix4] = rocket_performance(self, mix1, Aratio, mix2_inj, mix2_c, mix3)
- [mix1, mix2_inj, mix2_c, mix3, mix4] = rocket_performance(self, mix1, Aratio, mix2_inj, mix2_c, mix3, mix4)

solve_model_rocket(*self*, *mix1*, *mix2_inj*, *mix2_c*, *mix3*, *mix4*, *Aratio*)

Compute chemical equilibria at different points of the rocket depending of the model selected

Methods implemented:

- Infinite-Area-Chamber (IAC)
- Finite-Area-Chamber (FAC)

This method is based on the method outlined in 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
- **mix2_inj** (struct) – Properties of the mixture at the injector [only FAC] (previous calculation)
- **mix2_c** (struct) – Properties of the mixture at the outlet of the chamber (previous calculation)
- **mix3** (struct) – Properties of the mixture at the throat (previous calculation)
- **mix4** (struct) – Properties of the mixture at the given exit points (previous calculation)
- **Aratio** (float) – Area ratios [-]

Returns

Tuple containing

- mix2_1 (struct): Properties of the mixture at injector of the chamber (only FAC)
- mix2 (struct): Properties of the mixture at the outlet of the chamber
- mix3 (struct): Properties of the mixture at the throat
- mix4 (struct): Properties of the mixture at the given exit points

Example

[mix2_1, mix2, mix3, mix4] = solve_model_rocket(self, mix1, mix2_1, mix2, mix3, mix4, Aratio)

7.2.2 Utilities

A collection of routines with multiple purposes organized as follows:

- `unclasified`
- `databases`
- `display`
- `eos`
- `export`
- `extensions`
- `root_finding`
- `thermo`
- `validations`

Unclasified utility functions

A collection of unclasified functions necessary to perform all the calculations in CT.

Routines

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

Example

Yi_Fuel = Compute_YFuel(mix, mix_Fuel)

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

GPL()

Return Combustion Toolbox license

Returns
license_content (char) – The license text

Example

license_content = GPL()

---

abundances2moles(*elements*, *filename*, *varargin*)

Read solar abundances in log 10 scale and compute the initial molar fractions in the mixture [-]

Parameters

- **elements** (cell) – List with the given elements
- **filename** (file) – Filename with the data

Optional Args:
metallicity (float): Metallicity

Returns
moles (float) – moles relative to H of the remaining elements in the mixture

---

Examples

- moles = abundances2moles({'H', 'He', 'C', 'N', 'O', 'Ne', 'Ar', 'S', 'Cl', 'Fe'}, 'abundances.txt')
- moles = abundances2moles({'H', 'He', 'C', 'N', 'O', 'Ne', 'Ar', 'S', 'Cl', 'Fe'}, 'abundances.txt', 10)

---

append_cells(*cell1*, *cell2*, *varargin*)

Append two or more cells in one common cell

Parameters

- **cell1** (struct) – Cell 1
- **cell2** (struct) – Cell 2

Optional Args:
celli (struct): Additional cells

Returns
append_cell (struct) – Merged cell

append_structs(*s1*, *s2*, *varargin*)

Append two or more structs in one common struct

Parameters

- **s1** (struct) – Struct 1
- **s2** (struct) – Struct 2

Optional Args:
si (struct): Additional structs

Returns
append_s (struct) – Merged struct

ask_problem(*self*)

Create a list selection dialog box (deprecated)

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

Returns
PT (string) – String with the problem selected

assign_vector2cell(*cell*, *vector*, *varargin*)

Assign values of a vector into a cell

Parameters

- **cell** (cell) – Cell in which the values of the given vector are going to be included
- **vector** (any) – Vector with the values that are going to be included in the cell

Optional Args:
ind (float): List of index positions to assign specific positions to the cell

Returns
cell (cell) – Cell with the values of the given vector

cell2vector(*value*, *varargin*)

Convert values of an 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)

check_FOI(*self*, *FOI_species*)

Check that fuel species are contained in the list of products (only for initial computations)

Parameters

- **self** (`struct`) – Data of the mixture, conditions, and databases
- **FOI_species** (`bool`) – Species in the initial mixture (Fuel, Oxidizer, Inert)

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

check_inputs(*self*)

Check that all the inputs are specified

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

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

check_temperature_range(*self*, *T*, *ind*, *NS*, *FLAG*)

Remove species indeces out of the temperature range if FLAG = true, e.g., linear extrapolation of the polynomial fits is not allowed.

Parameters

- **self** (`struct`) – Data of the mixture, conditions, and databases
- **T** (`float`) – Temperature
- **ind** (`float`) – Vector with the species indeces

- **NS** (float) – Number of species
- **FLAG** (bool) – Flag indicating linear extrapolation of the polynomials fits

Returns

Tuple containing

- ind (float): Updated vector with the species indeces
- NS (float): Update number of species

check_update(*varargin*)

Check if there is a new release of the Combustion Toolbox

Optional Args:
fig (object): UIFigure class

Returns

Tuple containing

- FLAG_UPDATE (bool): FLAG indicating true (false) if there is (not) an update of the Combustion Toolbox
- message (char): Message displayed

Examples

- [FLAG_UPDATE, message] = check_update();
- [FLAG_UPDATE, message] = check_update(UIFigure);

---

compute_Gammas(*u2*, *rho2*, *p2*)

Compute slope of the Hugoniot curve

Parameters

- **u2** (float) – Post-shock velocity [m/s]
- **rho2** (float) – Post-shock density [kg/m3]
- **p2** (float) – Post-shock pressure [bar]

Returns
Gammas (float) – Slope of the Hugoniot curve [-]

compute_Gammas_frozen(*M1*, *R*, *P*)

Compute slope of the Hugoniot curve for thermochemically frozen air

Parameters

- **M1** (float) – Pre-shock Mach number [-]
- **R** (float) – Density ratio [-]
- **P** (float) – Pressure ratio [-]

Returns

Gammas (float) – Slope of the Hugoniot curve [-]

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_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 [-]

compute_properties(*self*, *properties_matrix*, *p*, *T*)

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

Parameters

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

Returns

mix (struct) – Properties of the mixture

Example

mix = compute_properties(self, properties_matrix, p, T)

---

compute_ratio_oxidizers_O2(*self*)

Compute ratio oxidizers/O2

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

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

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** (cell) – Cell array with the species of the mixture
- **moles** (float) – Moles of the species in the mixture [mol]
- **temperatures** (float) – Vector or cell array with the temperatures of each species

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

Example

T = compute_temperature_mixture(self, {'H2', 'O2'}, [1, 2], [300, 400])

---

convert_Pa_to_bar(*value*)

Convert pressure in [Pa] units to [bar]

Parameters
value (float) – Pressure value(s) in [bar]

Returns
value (float) – Pressure value(s) in [bar]

Example

value = convert_Pa_to_bar(1e5)

---

convert_atm_to_bar(*value*)

Convert pressure in [atm] units to [bar]

Parameters
value (float) – Pressure value(s) in [atm]

Returns
value (float) – Pressure value(s) in [bar]

Example

value = convert_atm_to_bar(1)

---

convert_bar_to_Pa(*value*)

Convert pressure in [bar] units to [Pa]

Parameters
value (float) – Pressure value(s) in [bar]

Returns
value (float) – Pressure value(s) in [Pa]

Example

value = convert_bar_to_Pa(1)

---

convert_bar_to_atm(*value*)

Convert pressure in [bar] units to [atm]

Parameters
value (float) – Pressure value(s) in [bar]

Returns
value (float) – Pressure value(s) in [atm]

Example

value = convert_bar_to_atm(1.01325)

---

convert_weight_percentage_to_moles(*LS*, *weight_percentage*, *DB*)

Convert weight percentage (wt%) to moles

Parameters

- **LS** (cell) – List of species
- **weight_percentage** (float) – Weight percentage of the species [%]

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

Returns
moles (float) – Number of moles [mol]

Example

moles = convert_weight_percentage_to_moles({'H2O', 'CO2'}, [50, 50], DB)

---

create_cell_ntimes(*varargin*)

Create cell array with the same item n-times

define_F(*self*)

Set Fuel of the mixture

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

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

define_FOI(*self*, *i*)

Set up mixture: fuel, oxidizer and diluent/inert species

Parameters

- **self** (struct) – Data of the mixture, conditions, and databases
- **i** (float) – Position of the evaluated problem

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

define_I(*self*)

Set Inert of the mixture

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

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

define_O(*self*)

Set Oxidizer of the mixture

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

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

docs_CT()

Open Combustion Toolbox's documentation in default web browser

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)

Example

index = find_ind({'H2O', 'CO2', 'CH4'}, {'H2O', 'CH4'})

---

get_FLAG_N(*self*)

Flag if the number of moles of fuel, oxidant and inert species is specified. If not, consider 1 mole for the fuel and calculate the remaining moles from the equivalence relation.

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

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

get_combustion_toolbox_version()

Get Combustion Toolbox version

Returns

Tuple containing

- release (char): Release version
- date (char): Release date

get_index_phase_species(*self*, *LS*)

Get index of gaseous, condensed and cryogenic species

Parameters

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

- **LS** (cell) – Name list species / list of species

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

get_latest_version_github(*user*, *repo_name*)

Get latest version from a repository from Github

Parameters

- **user** (char) – Username of the owner of the repository
- **repo_name** (char) – Name of the repository

Returns

Tuple containing

- release (char): Release tag (latest)
- git_data (struct): Body data of the request

Example

[release, git_data] = get_latest_version_github('AlbertoCuadra', 'combustion_toolbox')

---

get_monitor_positions(*varargin*)

Routine that gets the position in pixels of the monitor(s) connected to the device using Java (default) or MATLAB's routines. If no monitor is specified, the position of the main monitor is returned.

Optional Args:
monitor_id (float): Get position in pixels for the given monitor

Returns
position (float) – Position of the monitor(s)

Examples

- position = get_monitor_positions() returns the position of the main monitor
- position = getMonitorPositions(2) returns the position of the second monitor
- position = getMonitorPositions(1) returns the position of the first monitor
- position = getMonitorPositions(monitor_id) returns the position in pixels of the given monitor

---

Notes

This function first tries using Java to get the screen size values that do not take into account ui scaling, so it will return the proper screen size values. Otherwise, it gets the screen position using MATLAB's routines

---

`get_monitor_positions_MATLAB`(*varargin*)

Routine that gets the position in pixels of the monitor(s) connected to the device using MATLAB's routines. If no monitor is specified, the position of the main monitor is returned.

Optional Args:

monitor_id (float): Get position in pixels for the given monitor

Returns

position (float) – Position of the monitor(s)

Examples

- position = get_monitor_positions() returns the position of the main monitor
- position = getMonitorPositions(2) returns the position of the second monitor
- position = getMonitorPositions(1) returns the position of the first monitor
- position = getMonitorPositions(monitor_id) returns the position in pixels of the given monitor

---

`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

Example

order = get_order(0.0001)

`get_oxidizer_reference`(*self*, *varargin*)

Get oxidizer of reference for computations with the equivalence ratio

Parameters

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

Returns
self (struct) – Data of the mixture, conditions, and databases, included the oxidizer of reference which can be obtained as self.S.ind_ox_ref

`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_title`(*self*)

Get a title based on the problem type and species involved

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

Returns
titlename (char) – Title based on the problem type and species involved

`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** (char) – Fieldname in Problem Description (PD)

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

`get_typeSpecies`(*self*)

Create cell array with the type of species in the mixture

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

Returns
typeSpecies (cell) – Cell array with the type of species in the mixture

`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

mixture(*T*, *p*, *species*, *moles*, *varargin*)

Compute the properties of a mixture at a given temperature and pressure

Parameters

- **T** (float) – Temperature [K]
- **p** (float) – Pressure [bar]
- **species** (cell) – List of species
- **moles** (cell) – List of moles of each species

Optional Name-Value Pairs Args:

- phi (float): Equivalence ratio [-]

Returns

mix (struct) – Mixture properties

Examples

- mix = mixture(300, 1, {'CH4', 'O2', 'N2'}, [1, 2, 7.52])
- mix = mixture(300, 1, {'CH4', 'O2', 'N2'}, [1, 2, 7.52], 'self', self)
- mix = mixture(300, 1, {'CH4', 'O2', 'N2'}, [1, 2, 7.52], 'DB_master', DB_master, 'DB', DB)

---

post_results(*self*)

Postprocess all the results with predefined plots

Parameters

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

print_error(*ME*, *varargin*)

Print message error

Parameters

ME (object) – MException object that allows to identify the error

Optional Name-Value Pairs Args:

- type (char): Type of message (error, warning, or other)
- message_solution (char): Message solution

Returns
error_message (char) – Message error

Examples

- error_message = print_error(ME, 'Type', 'Warning')
- error_message = print_error(ME, 'Type', 'Warning', 'Solution', 'Returning an empty index value.')

`read_abundances`(*filename*)

Read solar abundances file

Format: [number element, element, abundance, name, molar mass (g/mol)]

Parameters
filename (`file`) – Filename with the data

Returns

Tuple containing

- abundances (float): Vector with the logarithmic base 10 solar abundances
- elements (cell): List with the given elements

`reorganize_index_phase_species`(*self*, *LS*)

Reorginize index of gaseous, condensed and cryogenic species

Parameters

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

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

`set_air`(*self*, *FLAG_IDEAL_AIR*)

Incluide air in the initial mixture

Parameters

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

- **FLAG_IDEAL_AIR** (bool) – Flag indicating consider ideal or non-ideal air mixture

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_react_index(*self*, *species*)

Set index of react (non-frozen) and frozen species

Parameters

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

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

set_species(*self*, *species*, *moles*, *T*, *varargin*)

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]

Optional Args:
ind (float): Vector with the indexes of the species to fill the properties matrix

Returns
properties_matrix (float) – Properties matrix

Examples

properties_matrix = set_species(self, {'N2', 'O2'}, [3.76, 1], 300) properties_matrix = set_species(self, {'N2', 'O2'}, [3.76, 1], 300, [1, 2])

`set_species_initilize`(*self*, *species*)

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

Returns
properties_matrix (float) – Properties matrix

`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

`setup_segregated_solver`(*self*, *LS*)

Get additional inputs necessary to use the segregated model

`smooth_data`(*x*, *y*, *varargin*)

Smooth data using Fourier NonlinearLeastSquares method

Parameters

- **x** (`float`) – data in the x direction
- **y** (`float`) – data in the y direction

Optional Args:
start_point (float): initial point of the fit

Returns

Tuple containing

- x (float): smooth data in the x direction
- y (float): smooth data in the y direction

`solve_problem`(*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** (`char`) – Tag of the problem to solve

Returns

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

`soundspeed_eq`(*self*, *mix*, *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
- **P0** (`float`) – Pressure [bar]
- **T0** (`float`) – Temperature [K]

Returns

sound (float) – sound speed [m/s]

`stoich_prop_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

`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

website_CT()

Open Combustion Toolbox's website in default web browser

Database functions

A collection of functions necessary to obtain generate the databases in CT.

Routines

FullName2name(*species*)

Get full name of the given species

Parameters
species (char) – Chemical species

Returns
name (char) – Full name of the given species

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

Include not defined species in database from master database (deprecated)

Parameters

- **self** (struct) – Data of the mixture, conditions, and databases
- **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): Check only the given list of species

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

Examples

- [DB, E, S, C] = check_DB(self, DB_master, DB)
- [DB, E, S, C] = check_DB(self, DB_master, DB, {'H2O', 'CO2'})

`compute_change_moles_gas_reaction`(*element_matrix*, *phase*)

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
- **phase** (`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

Example

Delta_n = compute_change_moles_gas_reaction(element_matrix, phase)

`compute_interval_NASA`(*species*, *T*, *DB*, *tRange*, *ctTInt*)

Compute interval NASA polynomials

Parameters

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

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

---

Example

index_open_parenthesis = detect_location_of_phase_specifier('C8H18(L),isooct')

---

find_products(*self*, *species*, *varargin*)

Find all the combinations of species from DB that can appear as products for the given list of reactants

Parameters

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

Optional Name-Value Pairs Args:

- ind_elements_DB (float): Matrix NS_DB x MAX_ELEMENTS with element indeces of the species contained in the database
- FLAG_BURCAT (bool): Flag indicating to look for species also in Burcat's database
- FLAG_ION (bool): Flag indicating to include ionized species
- FLAG_CONDENSED (bool): Flag indicating to include condensed species

Returns

Tuple containing

- LS (cell): List of products
- ind_elements_DB (float): Matrix NS_DB x MAX_ELEMENTS with element indeces of the species contained in the database

Examples

- [LS, ind_elements_DB] = find_products(self, {'O2', 'N', 'eminus'})
- [LS, ind_elements_DB] = find_products(self, {'O2', 'CO', 'N'}, 'flag_burcat', true)
- [LS, ind_elements_DB] = find_products(self, {'O2', 'CO', 'N'}, 'flag_burcat', true, 'flag_ion', true)
- [LS, ind_elements_DB] = find_products(self, {'O2', 'CO', 'N'}, 'flag_burcat', true, 'flag_ion', true, 'flag_condensed', true, 'ind', ind_elements_DB)
- [LS, ind_elements_DB] = find_products(self, {'O2', 'CO', 'N'}, 'flag_burcat', true, 'flag_ion', true, 'ind', ind_elements_DB)

`find_species_LS`(*LS*, *cond_with*, *type_with*, *cond_without*, *type_without*)

Find species in the given list that contain all/any elements of cond_with and that not include all/any elements of cond_without

Parameters

- **LS** (`cell`) – List of species
- **cond_with** (`cell`) – List of elements to include
- **type_with** (`char`) – Satisfy all or any of the elements in cond_with
- **cond_without** (`cell`) – List of elements to avoid
- **type_without** (`char`) – Satisfy all or any of the elements in cond_without

Returns

LS (cell) – List of species

Examples

- **LS = find_species_LS(LS, {'C','N','O','minus','plus','Ar'}, 'any',…**

  {'I', 'S', 'L', 'T', 'P', 'F', 'ab', 'W',…
  'Z','X','R','Os','Cr','H','Br','G','K',… 'U','Co','Cu','B','V','Ni','Na','Mg',…
  'Mo','Ag','Nb','Cb','Cl','D','T',… 'Ca','Cs','Ne','Cd','Mn'}, 'all')

- LS = find_species_LS(self.S.LS_DB, {}, 'any', {'_M'}, 'all')

`generate_DB`(*DB_master*, *varargin*)

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

Optional Args:
LS (cell): List of species to be included in DB

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

---

Examples

- DB = generate_DB(DB_master)
- DB = generate_DB(DB_master, {'CO2', 'H2O', 'O2', 'N2'})

---

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:
FLAG_REDUCED_DB (bool): Flag indicating reduced database (default: false) thermoFile (char): File name of NASA's thermodynamic database (default: thermo_CT.inp)

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

---

Examples

- DB_master = generate_DB_master(false)
- DB_master = generate_DB_master(false, 'thermo_CT.inp')

---

generate_DB_master_reduced(*DB_master*)

Generate Reduced Mater Database (DB_master_reduced) with the thermodynamic data of the chemical species (deprecated)

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_ind_elements(*LS*, *DB*, *elements*, *MAX_ELEMENTS*)

Get element indeces of each species contained in LS

Parameters

- **LS** (cell) – List of species
- **DB** (struct) – Database with custom thermodynamic polynomials functions generated from NASAs 9 polynomials fits
- **elements** (cell) – Elements in the periodic table
- **MAX_ELEMENTS** (float) – Maximum number of elements contained in one species

Returns
ind_elements (float) – Matrix numel(LS) x MAX_ELEMENTS with element indeces of the species contained in LS

Example

ind_elements = get_ind_elements(LS, DB, elements, MAX_ELEMENTS)

---

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** (char) – 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** (`char`) – Chemical species
- **T** (`float`) – Temperature [K]
- **MassOrMolar** (`char`) – 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)]
- hf0 (float): Enthalpy of formation [J/mol]
- h0 (float): Enthalpy [J/mol]
- ef0 (float): Internal energy of formation [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** (`char`) – Chemical species
- **T** (`float`) – Temperature

Returns
name (char) – Full name of the given species

Example

[FLAG_RE, REname] = isRefElm(reference_elements, 'O', 1000)

name_with_parenthesis(*species*)

Update the name of the given char with parenthesis. The character b if comes in pair represents parenthesis in the NASA's database

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

Returns
species_with (char) – Chemical species with parenthesis

Example

species_with = name_with_parenthesis('Cbgrb')

set_DhT(*LS*, *T*, *DB*)

Function that computes the vector of thermal enthalpy 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
DhT (float) – Thermal enthalpy in molar basis [J/mol]

Example

DhT = set_DhT({'H2O', 'CO2'}, 298.15, DB)

set_cP(*LS*, *T*, *DB*)

Function that computes the vector of specific heats at constant pressure for the given set of species [J/(mol-K)]

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

cP (float) – Specific heat at constant pressure in molar basis [J/(mol-K)]

Example

cP = set_cP({'H2O', 'CO2'}, 298.15, DB)

---

set_e0(*LS*, *T*, *DB*)

Function that computes the vector of internal 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

e0 (float) – Internal energy in molar basis [J/mol]

Example

e0 = set_e0({'H2O', 'CO2'}, 298.15, DB)

---

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 in molar basis [J/mol]

Example

g0 = set_g0({'H2O', 'CO2'}, 298.15, DB)

---

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) – Enthalpy in molar basis [J/mol]

Example

h0 = set_h0({'H2O', 'CO2'}, 298.15, DB)

---

set_prop_DB(*LS*, *property*, *DB*)

Function that gets the vector of the defined property for the given set of species

Parameters

- **LS** (cell) – List of species
- **property** (str) – Property to obtain from the database
- **DB** (struct) – Database with custom thermodynamic polynomials functions generated from NASAs 9 polynomials fits

Returns
value (float) – Property vector

Example

value = set_prop_DB({'H2O', 'CO2'}, 'hf', DB)

`set_reference_form_of_elements`

Get list with reference form of elements

Returns
Reference_form_of_elements (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
Reference_form_of_elements_with_T_intervals (cell) – List with reference form of elements and its temperature intervals

`set_s0`(*LS*, *T*, *DB*)

Function that computes the vector of entropy for the given set of species [J/(mol-K)]

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
s0 (float) – Entropy in molar basis [J/(mol-K)]

Example

s0 = set_s0({'H2O', 'CO2'}, 298.15, DB)

`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** (`char`) – 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 in molar basis [kJ/mol]

Example

DeT = species_DeT('H2O', 300, DB)

---

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** (`char`) – 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 in molar basis [kJ/mol]

Example

DeT = species_DeT_NASA('H2O', 300:100:6000, DB)

---

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** (`char`) – 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 in molar basis [kJ/mol]

Example

DhT = species_DhT('H2O', 300, DB)

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** (char) – 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 in molar basis [kJ/mol]

Example

DhT = species_DhT_NASA('H2O', 300:100:6000, DB)

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** (char) – 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 in molar basis [J/(mol-K)]

Example

cP = species_cP('H2O', 300, DB)

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** (char) – 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 in molar basis [J/(mol-K)]
- cV (float): Specific heat at constant volume in molar basis [J/(mol-K)]

Example

[cP, cV] = species_cP_NASA('H2O', 300:100:6000, DB)

---

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** (char) – 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 in molar basis [J/(mol-K)]

Example

cV = species_cV('H2O', 300, DB)

---

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** (char) – 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 in molar basis [J/(mol-K)]

Example

cV = species_cV_NASA('H2O', 300:100:6000, DB)

---

species_e0(*species*, *T*, *DB*)

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

Parameters

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

Returns

e0 (float) – Internal energy in molar basis [kJ/mol]

Example

e0 = species_e0('H2O', 300, DB)

---

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** (char) – 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 in molar basis [kJ/mol]
- DeT (float): Thermal internal energy in molar basis [kJ/mol]

Example

[e0, DeT] = species_e0_NASA('H2O', 300:100:6000, DB)

`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`** (`char`) – 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 in molar basis [kJ/mol]

Example

g0 = species_g0('H2O', 298.15, DB)

`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`** (`char`) – 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 in molar basis [kJ/mol]

Example

g0 = species_g0_NASA('H2O', 300:100:6000, DB)

`species_gamma`(*species*, *T*, *DB*)

Compute adiabatic index of the species [-] at the given temperature [K] using piecewise cubic Hermite interpolating polynomials and linear extrapolation

Parameters

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

Returns

gamma (float) – Adiabatic index [-]

Example

gamma = species_gamma('H2O', 300, DB)

`species_gamma_NASA`(*species*, *T*, *DB*)

Compute adiabatic index of the species [-] at the given temperature [K] using piecewise cubic Hermite interpolating polynomials and linear extrapolation

Parameters

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

Returns

gamma (float) – Adiabatic index [-]

Example

gamma = species_gamma_NASA('H2O', 300:100:6000, DB)

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

Returns

h0 (float) – Enthalpy in molar basis [kJ/mol]

Example

h0 = species_h0('H2O', 300, DB)

---

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** (char) – 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 in molar basis [kJ/mol]
- DhT (float): Thermal enthalpy in molar basis [kJ/mol]

Example

[h0, DhT] = species_h0_NASA('H2O', 300:100:6000, DB)

---

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

Returns

s0 (float) – Entropy in molar basis [kJ/(mol-K)]

Example

s0 = species_s0('H2O', 300, DB)

---

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** (char) – 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 in molar basis [kJ/(mol-K)]

Example

s0 = species_s0_NASA('H2O', 300:100:6000, DB)

---

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

Compute thermodynamic function using NASA's 9 polynomials

Parameters

- **species** (char) – 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 in molar basis [J/(mol-K)]
- cV (float): Specific heat at constant volume in molar basis [J/(mol-K)]
- h0 (float): Enthalpy in molar basis [kJ/mol]
- DhT (float): Thermal enthalpy in molar basis [kJ/mol]
- e0 (float): Internal energy in molar basis [kJ/mol]
- DeT (float): Thermal internal energy in molar basis [kJ/mol]
- s0 (float): Entropy in molar basis [J/(mol-K)]
- g0 (float): Gibbs energy in molar basis [kJ/mol]

Example

[cP, cV, h0, DhT, e0, DeT, s0, g0] = species_thermo_NASA('H2O', 300:100:6000, DB)

`thermo_millennium_2_thermoNASA9`(*filename*)

Read Extended Third Millennium Thermodynamic Database of New NASA Polynomials with Active Thermochemical Tables update and write a new file compatible with thermo NASA 9 format

Parameters

filename (char) – Filename of the thermo_millennium data

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

Unpack NASA's polynomials coefficients from database

Parameters

- **species** (char) – 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 (char): Chemical formula
- phase (float): 0 or 1 indicating gas or condensed phase, respectively

---

Example

[a, b, tRange, tExponents, ctTInt, txFormula, phase] = unpack_NASA_coefficients('H2O', DB)

Display functions

A collection of functions necessary to display the results (command window and plots).

Routines

displaysweepresults(*self*, *mix*, *xvar*, *varargin*)

Plot a given variable against the molar fractions of the mixture

Parameters

- **self** (struct) – Data of the mixture, conditions, and databases
- **mix** (struct) – Properties of the mixture
- **xvar** (float) – Vector with the x data

Optional Args:

- config (struct): Struct with the configuration for plots
- xscale (string): Scale fot the x-axis (linear or log)
- yscale (string): Scale fot the y-axis (linear or log)
- xdir (string): Direction of the x-axis (normal or reverse)
- ydir (string): Direction of the x-axis (normal or reverse)

Returns

ax (axes) – Axes object

`get_mixtures`(*PS*, *pattern*)

Get all non-empty mixture

Parameters

- **PS** (`struct`) – Struct with all the data of Problem Solution (PS)
- **pattern** (`str`) – Pattern/s name of the mixture

Returns

mixtures (cell) – Cell with all the non-empty mixtures

Examples

- mixtures = get_mixtures(self.PS, 'mix');
- mixtures = get_mixtures(self.PS, 'strP');

`interpreter_label`(*property*, *varargin*)

Interpreter label for properties - returns property name

Note: The 'interpreter_label.m' routine considers that the properties are in mass basis. This will be fixed in a future patch.

Parameters

property (`char`) – Property name

Optional Args:

type (char): Type of label to return. Can be 'short', 'medium' or 'long' (default: medium)

Returns

value (char) – Corresponding name of the property

`plot_figure`(*x_field*, *x_var*, *y_field*, *y_var*, *varargin*)

Plot figure with customizable settings

Note: The 'interpreter_label.m' routine considers that the properties are in mass basis. This will be fixed in a future patch.

Parameters

- **x_field** (`char`) – Field name for the x-axis data

- **x_var** (cell) – Cell array containing the x-axis data
- **y_field** (char) – Field name for the y-axis data
- **y_var** (cell) – Cell array containing the y-axis data

Optional Name-Value Pair Args:

- config (struct): Struct with the configuration for plots
- leg or legend (cell): Cell array of strings containing the legend names
- legend_location (char): Location of the legend
- ax or axes (object): Handle of the axes to plot on
- linestyle (char): Line style
- linewidth (float): Line width
- fontsize (float): Font size
- title (char): Title of the figure
- labelx, xlabel, label_x, or x_label (char): x-axis label
- labely, ylabel, label_y, or 'y_label (char): y-axis label
- label_type (char): Label type
- xscale (char): Set x-axis scale (linear or log)
- yscale (char): Set y-axis scale (linear or log)
- xdir (char): Set x-axis direction (normal or reverse)
- ydir (char): Set y-axis direction (normal or reverse)
- color (float): Line color [R, G, B]

Returns

Tuple containing

- ax (object): Handle of the axes
- dline (object): Handle of the plotted line

plot_figure_set(*range_name*, *range*, *properties*, *mix*, *varargin*)

Plot a set of properties in a tiled layout figure

Parameters

- **range_name** (char) – Variable name of the x-axis parameter

- **range** (float) – Vector x-axis values
- **properties** (cell) – Cell array of properties to plot
- **mix** (struct) – Mixture

Optional Args:

- ax (handle): Handle to the main axis
- config (struct): Configuration settings for the figure
- basis (cell): Cell array with the basis for each property

Returns

** main_ax (obj)* – Handle to the main axis

Examples

- plot_figure_set('T', 300:100:1000, {'cp', 'cv', 'h', 's'}, mix)
- plot_figure_set('T', 300:100:1000, {'cp', 'cv', 'h', 's'}, mix, 'config', config)
- plot_figure_set('T', 300:100:1000, {'cp', 'cv', 'h', 's'}, mix, 'config', config, 'basis', {'', '', 'mi', 'mi'});
- plot_figure_set('T', 300:100:1000, {'cp', 'cv', 'h', 's'}, mix, 'config', config, 'basis', {'', '', 'mi', 'mi'}, 'ax', ax);

---

plot_hugoniot(*self*, *mix1*, *mix2*, *varargin*)

Plot the Hugoniot curve for a a given pre-shock state (mix1) and post-shock state (mix2)

Parameters

- **self** (struct) – Data of the mixture, conditions, and databases
- **mix1** (struct) – Pre-shock mixture
- **mix2** (struct) – Post-shock mixture

Optional Args:

ax (object): Axis handle to plot on

Returns

ax (object) – Axis handle to plot on

Examples

- ax = plot_hugoniot(self, mix1, mix2)
- ax = plot_hugoniot(self, mix1, mix2, ax)

---

plot_molar_fractions(*self*, *x_var*, *x_field*, *y_field*, *varargin*)

Plot molar fractions againts any variable

Parameters

- **self** (struct) – Data of the mixture, conditions, and databases
- **x_var** (cell) – Properties of the mixture for all the cases
- **x_field** (char) – Field name for the x-axis data
- **y_field** (char) – Field name for the y-axis data

Optional Name-Value Pair Args:

- validation (struct): Struct that contains validations with (x_field, y_field)
- nfrec (float): Frequency points to plot validations
- mintol (float): Minimum limit i-axis with the composition of the mixture
- display_species (cell): List of species to plot
- y_var (cell): Get y-axis data from a different mixture
- config (struct): Struct with the configuration for plots
- axis_x (char): Set x-axis limits
- axis_y (char): Set y-axis limits
- xscale (char): Set x-axis scale (linear or log)
- yscale (char): Set y-axis scale (linear or log)
- xdir (char): Set x-axis direction (normal or reverse)
- ydir (char): Set y-axis direction (normal or reverse)
- ax (object): Handle of the axes to plot on

Returns

Tuple containing

- ax (object): Handle of the axes
- fig (object): Handle of the figure

Examples

- [ax, fig] = plot_molar_fractions(self, mix1, 'phi', 'Xi')
- [ax, fig] = plot_molar_fractions(self, mix1, 'phi', 'Xi', 'y_var', mix2)
- [ax, fig] = plot_molar_fractions(self, mix1, 'phi', 'Xi', 'y_var', mix2, 'validation', results_CEA)
- [ax, fig] = plot_molar_fractions(self, mix1, 'phi', 'Xi', 'y_var', mix2, 'validation', results_CEA, 'display_species', display_species)

`plot_shock_polar`(*varargin*)

Routine to obtain shock polar plots

- Plot (pressure, deflection)
- Plot (wave angle, deflection)
- Plot velocity components

`polynomial_regression`(*x*, *y*, *n*)

Obtain polynomial regression for the given dataset (x, y) and polynomial order

Parameters

- **x** (`float`) – x values
- **y** (`float`) – y values
- **n** (`float`) – polynomial order

Returns

y_poly (float) – y values of the polynomial regression

`print_mixture`(*self*, *varargin*)

Print properties and composition of the given mixtures in the command window

Parameters

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

Optional Args:

- mix1 (struct): Struct with the properties of the mixture
- mix2 (struct): Struct with the properties of the mixture
- mixi (struct): Struct with the properties of the mixture
- mixN (struct): Struct with the properties of the mixture

Examples

- print_mixture(self, mix1)
- print_mixture(self, mix1, mix2)
- print_mixture(self, mix1, mix2, mix3)
- print_mixture(self, mix1, mix2, mix3, mix4)

print_stoichiometric_matrix(*self*, *varargin*)

Print stoichiometric matrix

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

Optional args:
type (char): 'transpose'

Optional returns:

A0 (table): Stoichiometric matrix. In case type == 'transpose'
it returns the transpose of stoichiometric matrix

results(*self*, *i*)

Display results in the command window

Parameters

- **self** (struct) – Data of the mixture, conditions, and databases
- **i** (float) – index of the current iteration

set_figure(*varargin*)

Initialize figure with a standard composition

Optional Args:

- ax (axis): Figure axis
- config (struct): Struct with default plot parameters

Returns

Tuple containing

- ax (axis): Axis of the standard figure
- config (struct): Struct with default plot parameters

- fig (figure): Standard figure

set_legends(*ax*, *legend_name*, *varargin*)

Set legend to the given axes

Parameters

- **ax** (object) – Handle to the axes
- **legend_name** (cell) – Cell array of char containing the legend names

Optional Name-Value Pairs Args:

- config (struct): Struct containing the configuration parameters for the plots
- obj (object): Handle to the plotted objects (e.g. lines, patches, etc.)

set_title(*ax*, *varargin*)

Set legend to the given axes

species2latex(*species*, *varargin*)

Convert species name into LateX format

Parameters

species (char) – Species name

Optional Args:

FLAG_BURCAT (bool): If true, do not remove Burcat database prefix (default: true)

Returns

speciesLatex (char) – Species name in LateX format

Examples

- species2latex('H2ObLb')
- species2latex('Si2H6_M')
- species2latex('Si2H6_M', false)

Equation of State functions

A collection of Equation of States (EoS) implemented in CT.

Routines

eos_PengRobinson(*self*, *T*, *p*, *species*, *Xi*)

Compute molar volume of the mixture considering Peng-Robinson Equation of State (EoS)

Parameters

- **self** (struct) – Data of the mixture, conditions, and databases
- **species** (cell) – List of the species in the mixture
- **Xi** (float) – Molar fractions of the mixture
- **T** (float) – Temperature [K]
- **p** (float) – Pressure [bar]

Returns

Tuple containing

- V (float): Molar volume of the mixture [m3/mol]
- Vi (float): Molar volume of the components [m3/mol]

eos_VanderWaals(*self*, *species*, *Xi*, *T*, *p*)

Compute molar volume of the mixture considering Van der Waals Equation of State (EoS)

Parameters

- **self** (struct) – Data of the mixture, conditions, and databases
- **species** (cell) – List of the species in the mixture
- **Xi** (float) – Molar fractions of the mixture
- **T** (float) – Temperature [K]
- **p** (float) – Pressure [bar]

Returns

Tuple containing

- V (float): Molar volume of the mixture [m3/mol]
- Vi (float): Molar volume of the components [m3/mol]

eos_ideal(*self*, *T*, *p*, *varargin*)

Compute pressure considering ideal Equation of State (EoS)

Parameters

- **self** (struct) – Data of the mixture, conditions, and databases
- **T** (float) – Temperature of the mixture [K]
- **p** (float) – Pressure of the mixture [Pa]

Returns

V (float) – Molar volume of the mixture [m3/mol]

eos_ideal_p(*self*, *n*, *T*, *v*, *varargin*)

Compute pressure considering ideal Equation of State (EoS)

Parameters

- **self** (struct) – Data of the mixture, conditions, and databases
- **n** (float) – Number of moles of the mixture in gaseous phase [mol]
- **T** (float) – Temperature of the mixture [K]
- **v** (float) – Volume of the mixture [m3]

Returns

p (float) – Pressure of the mixture [Pa]

mu_ex_eos(*self*, *Xi*, *T*, *p*, *V*, *a_mix*, *b_mix*, *a*, *b*)

Compute non ideal contribution (excess) of the chemical potential assuming cubic Equation of State (EoS) [J/mol]

Parameters

- **self** (struct) – Data of the mixture, conditions, and databases
- **T** (float) – Temperature of the mixture [K]
- **p** (float) – Pressure of the mixture [bar]
- **V** (float) – Molar volume of the mixture [m3/mol]
- **a_mix** (float) – Atraction factor mixture of the cubic EoS
- **b_mix** (float) – Repulsion factor mixture of the cubic EoS
- **a** (float) – Atraction factor components of the cubic EoS
- **b** (float) – Repulsion factor components of the cubic EoS

Returns

chemical_potential_ex (float) – Chemical potential excess [J/mol]

mu_ex_ideal(*self*, *moles*, *temperature*, *volume*)

Compute non ideal contribution (excess) of the chemical potential assuming ideal Equation of State [J/mol]

Parameters

- **self** (struct) – Data of the mixture, conditions, and databases
- **moles** (float) – Number of moles of the mixture in gaseous phase [mol]
- **temperature** (float) – Temperature of the mixture [K]
- **volume** (float) – Volume of the mixture [m3]

Returns

pressure (float) – Pressure of the mixture [Pa]

mu_ex_vanderwaals(*self*, *moles*, *temperature*, *volume*)

Compute non ideal contribution (excess) of the chemical potential assuming Van der Waal's Equation of State [J/mol]

Parameters

- **self** (struct) – Data of the mixture, conditions, and databases
- **moles** (float) – Number of moles of the mixture in gaseous phase [mol]
- **temperature** (float) – Temperature of the mixture [K]
- **volume** (float) – Volume of the mixture [m3]

Returns

pressure (float) – Pressure of the mixture [Pa]

mu_ex_virial(*self*, *moles*, *temperature*, *volume*)

Compute non ideal contribution (excess) of the chemical potential assuming Virial Equation of State [J/mol]

Parameters

- **self** (struct) – Data of the mixture, conditions, and databases
- **moles** (float) – Number of moles of the mixture in gaseous phase [mol]
- **temperature** (float) – Temperature of the mixture [K]
- **volume** (float) – Volume of the mixture [m3]

Returns

pressure (float) – Pressure of the mixture [Pa]

Export functions

A collection of functions to export results.

Routines

`export_results`(*self*)

Export results of reactants (mix1) and products (mix2) into a .xls file

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

`get_excel_cell`(*mix*, *species*, *phi*)

Construct a cell with the thermodynamic data of the given mixture

Parameters

- **mix** (`struct`) – Properties of the mixture
- **species** (`cell`) – List of species
- **phi** (`float`) – Vector of equivalence ratio

Returns
excell_cell (cell) – Cell with the thermodynamic data of the given mixture

---

Example

excell_cell(self.PS.strR, self.S.LS, self.PD.phi.value)

`protected_function`

...

Extensions functions

A collection of external functions from other repositories.

- 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.

- Combustion Toolbox's splash screen is based on a routine from the following repository: Ben Tordoff (2022). SplashScreen (https://www.mathworks.com/matlabcentral/fileexchange/30508-splashscreen), MATLAB Central File Exchange. Retrieved October 15, 2022.

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

newton_2(*f*, *fprime*, *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]
- **guess_moles** (float) – Guess moles final mixture

Returns

Tuple containing

- x (float): Temperature at equilibrium [K]
- STOP (float): Relative error [-]
- guess_moles (struct): Guess moles final mixture

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 [-]

regula_guess(*self*, *mix1*, *pP*, *field*)

Find a estimate of the temperature for the set chemical equilibrium transformation using the regula falsi 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)

Returns

x0 (float) – Guess temperature [K]

newton(*self*, *mix1*, *pP*, *field*, *x0*, *guess_moles*)

Find the temperature [K] (root) for the set chemical transformation at equilibrium using the second-order 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]
- **guess_moles** (float) – Guess moles final mixture

Returns

Tuple containing

- x (float): Temperature at equilibrium [K]
- STOP (float): Relative error [-]
- guess_moles (struct): Guess moles final mixture

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

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*, *f_vector*)

Get point using the regula falsi method

Parameters

- **x_vector** (float) – Guess temperature [K]
- **f_vector** (struct) – evaluated functions [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*, *guess_moles*)

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]
- **guess_moles** (float) – Guess moles final mixture

Returns

Tuple containing

- x (float): Temperature at equilibrium [K]
- STOP (float): Relative error [-]
- guess_moles (float): Guess moles final mixture

Thermodynamic properties

Functions to obtain thermodynamic properties from a given mixture.

Routines

MolecularWeight(*mix*)

Get the molecular weight [g/mol] of the mixture

Parameters
mix (struct) – Properties of the mixture

Returns
value (float) – Molecular weight [g/mol] of the mixture

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

compressibility_factor(*mix*)

Compute compressibility factor of the mixture [-]

Parameters
mix (struct) – Properties of the mixture

Returns
Z (float) – Compressibility factor [-]

compute_heatrelease(*mix1*, *mix2*)

Compute heat release [J/kg] of the chemical transformation of the mixture 1 to mixture 2

Parameters

- **mix1** (struct) – Properties of the initial mixture
- **mix2** (struct) – Properties of the final mixture

Returns
q (float) – heat release [J/kg] == [m^2/s^2] Q (float): dimensionless heat release

compute_sound(*T*, *p*, *species*, *composition*, *varargin*)

Routine to compute sound velocity [m/s] for a given temperature-pressure profile

Parameters

- **T** (float) – Temperature [K]
- **p** (float) – Pressure [bar]
- **species** (cell) – List of species
- **composition** (float) – composition of species (mol)

Optional Name-Value Pairs Args:
self (struct): Data of the mixtures, conditions, databases

Returns
sound (float) – Sound velocity [m/s]

Examples

sound = compute_sound(300, 1, {'H2', 'O2'}, [1, 1]) sound = compute_sound(300, 1, {'H2', 'O2'}, [1, 1], 'self', self)

`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_formation_mass`(*mix*)

Get the mass specific enthalpy formation [kJ/kg] of the mixture

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

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

`enthalpy_formation_mole`(*mix*)

Get the mole specific enthalpy formation [kJ/mol] of the mixture

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

Returns
value (float) – Mole-basis specific enthalpy formation [kJ/mol] 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

humidity_specific(*T*, *p*, *humidity_relative*)

Get the specific humidity of air [kg_w/kg_da] at a given temperature, pressure, and relative humidity

Parameters

- **T** (float) – Temperature [K]
- **p** (float) – Pressure [bar]
- **humidity_relative** (float) – Relative humidity [%]

Returns
value (float) – Specific humidity of air [kg_w/kg_da]

`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

`mass`(*mix*)

Get the mass [kg] of the mixture

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

Returns
value (float) – mass [kg] 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

`molesGas`(*mix*)

Get the moles of the gases in the mixture [mol]

`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

volume(*mix*)

Get the volume [m3] of the mixture

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

Returns
value (float) – volume [m3] of the mixture

Validations functions

A collection of functions to generate the validations automatically.

Routines

compute_error_moles_CEA(*results1*, *results2*, *varname_x*, *value*, *varname_y*, *species*)

Compute max error of CT against CEA

compute_error_prop_CEA(*results1*, *results2*, *varsname_x*, *value*, *varsname_y*, *type*)

Compute max error of CT against CEA

debug_plot_error(*it*, *STOP*, *varargin*)

Debug function that plots the error per iteration along with the value of the correction factor

get_problems_solved(*varargin*)

Get problems solved based on the length of the given variable

load_struct(*filename*, *variable_name*)

Load variable from a struct saved in a file

plot_molar_fractions_validation(*results1*, *results2*, *varname_x*, *varname_y*, *species*, *varargin*)

Default values

plot_properties_validation(*results1*, *results2*, *varsname_x*, *varsname_y*, *type*, *varargin*)

Plot properties varname_y vs varname_x from CT (results1) against results obtained from other code (results2). The properties to plot are specified by varsname_x and varsname_y, which are cell arrays of strings.

plot_thermo_validation(*species*, *property*, *DB*, *varargin*)

Validation custom thermodynamic polynomials with NASA's 9 polynomials

Parameters

- **species** (`cell`) – List of species

- **property** (str) – Name of the thermodynamic property to check
- **DB** (struct) – Database with custom thermodynamic polynomials functions generated from NASAs 9 polynomials fits

Optional Args:

- nfrec (float): Points frequency for NASA values
- range (float): Temperature range [K]

Returns
ax (object) – Axes of the plotted figure

plot_validation_shock_polar_SDToolbox(*results_CT*, *results_SDToolbox*, *config*)

Plot numerical results obtained with SDToolbox, which use CANTERA as a thermochemical kernel.

- Pressure ratio with the deflection angle [deg]
- Wave angle [deg] with the deflection angle [deg]

read_CEA(*filename*)

READ DATA FROM CEA AS TXT EXTENSION fid=fopen('test_soot_acetylene.txt','r');

run_CT(*varargin*)

A generalized function to run Combustion Toolbox for a given set of inputs. Otherwise, it will run the predefined case.

set_inputs_thermo_validations(*property*)

Set corresponding thermodynamic functions for NASA and Combustion Toolbox

Parameters
property (str) – Thermodynamic property name

Returns

Tuple containing

- funname_NASA (function): Function to use NASA's polynomials
- funname_CT (function): Function to use Combustion Toolbox polynomials
- y_labelname (str): Label y axis

7.2.3 GUI

Routines to generate the app, add-ons, assets, and all the necessary functions to be compatible with the plain code and extend its functionality.

Combustion Toolbox GUI

Figure 1: *Post-process of results using the GUI of the adiabatic combustion for a lean-to-rich acetylene (C2H2)-air mixture at standard conditions (T1 = 300 K and p1 = 1 atm); labels: name of the different components of the GUI. In particular, the numerical results correspond to phi = 0.5 (selected case in the tree component) [part 1].*

Figure 2: *Post-process of results using the GUI of the adiabatic combustion for a lean-to-rich acetylene (C2H2)-air mixture at standard conditions (T1 = 300 K and p1 = 1 atm); labels: name of the different components of the GUI. In particular, the numerical results correspond to phi = 0.5 (selected case in the tree component) [part 2].*

Figure 3: *Post-process of results using the GUI of the adiabatic combustion for a lean-to-rich acetylene (C2H2)-air mixture at standard conditions (T1 = 300 K and p1 = 1 atm); labels: name of the different components of the GUI. In particular, the numerical results correspond to phi = 0.5 (selected case in the tree component) [part 3].*

Routines

class combustion_toolbox

Bases: matlab.apps.AppBase

C = None

Constants

DB = None

Reduced DataBase

DB_master = None

Master DataBase

E = None

Elements

LS = None

List of species considered (reactants + products)

LS_products = None

List of species considered as products

LS_reactants = None
List of reactants

Misc = None
Miscellaneous

NS_display = None
Number of display species (plots)

NS_products = None
Number of product species (computations)

N_flags = None
Number of flags active

PD = None
Problem Description

PP1_var_name = None
Variable name for PP1

PP1_vector = None
Condition Products 1

PP2_var_name = None
Variable name for PP2

PP2_vector = None
Condition Products 2

PR1_var_name = None
Variable name for PR1

PR1_vector = None
Condition Reactants 1

PR2_var_name = None
Variable name for PR2

PR2_vector = None
Condition Reactants 2

PR3_var_name = None
Variable name for PR3

PR3_vector = None
Condition Reactants 3

PS = None
Problem Solution

S = None
Species

TN = None
Tuning properties

color_lamp_done = '[0.5608, 0.7255, 0.6588]'
Lamp color (rgb): done

color_lamp_error = '[0.9451, 0.5059, 0.5529]'
Lamp color (rgb): error

color_lamp_nothing = '[0.8000, 0.8000, 0.8000]'
Lamp color (rgb): nothing to report

color_lamp_working = '[0.9961, 0.9804, 0.8314]'
Lamp color (rgb): working

color_splash = '[0.5098, 0.6039, 0.6745]'
Font color splash

current_history = None
Current history of commands

default = None
Struct with default values of some components in the GUI

delete(*app*)
Delete UIFigure when app is deleted

dynamic_components = None
Struct with all the dynamic components

fig = None
Auxiliary figure

flag_PP1 = None
FLAG for PP1: true-> vector

flag_PP2 = None
FLAG for PP2: true-> vector

flag_PR1 = None
FLAG for PR1: true-> vector

flag_PR2 = None
FLAG for PR2: true-> vector

flag_PR3 = None
FLAG for PR3: true-> vector

flag_phi = None
FLAG for phi: true-> vector

ind_Fuel = None
Index position Fuel species

ind_Inert = None
Index position Inert species

ind_Oxidizer = None
Index position Oxidizer species

public_ProductsValueChanged(*app*)
Update Listbox (extended settings)

public_get_current_history(*app*)
Get current history

temp_index = None
Temporal index to get current position of command history

temp_results = None
Temporal variable that contains the last parametric study

Utility functions

A collection of functions for Combustion Toolbox GUI.

Routines

gui_CalculateButtonPushed(*app*, *event*)
Solve selected problem, update GUI with the results, and generate predefined plots

Parameters

- **app** (`object`) – Combustion Toolbox app object
- **event** (`object`) – Event object

Returns
app (object) – Combustion Toolbox app object

gui_ConsoleValueChanged(*app*, *event*)

Print output of commands through GUI's command window

gui_FrozenchemistryCheckBoxValueChanged(*app*)

Set frozen chemistry and update Listbox of species

gui_ProblemTypeValueChanged(*app*)

Clear GUI results tab (except UITree) and update GUI items for the problem selected

gui_ProductsValueChanged(*app*)

Update List of species considered as Products

gui_ReactantsValueChanged(*app*, *event*)

Update values of the UITable items with: * a given predefined set of reactants * the new species added in the finder

gui_SnapshotMenuSelected(*UIFigure*)

Routine to exports the current figure to a file. This function is called when the Snapshot menu is selected.

Notes

- The file type is determined by the file extension
- The file name is determined by the user
- The file path is determined by the user

---

gui_UITable_RCellEdit(*app*, *event*)

Update values of the UITable items with the changes made

gui_add_nodes(*parent_node*, *results*)

Function that generate nodes in a UITree and save data on them

Parameters

- **parent_node** (`object`) – Parent node of the UITree
- **results** (`struct`) – struct with the data to save in the nodes

gui_add_nodes_validations(*app*, *code_validation_name*)

Add nodes with the name of the validations routines into the corresponding tree in the UIValidation app

Parameters

- **app** (object) – UIValidation app object
- **code_validation_name** (char) – Name of the validation code

gui_check_temperature_reactants(*app*, *DB*, *species*, *temperature*, *Nspecies*)

Check if is a condensed species with a fixed temperature

gui_clear_results(*app*)

Function that clears the result tab panel, setting them to 0

Parameters

app (object) – Combustion Toolbox app object

gui_compute_mach_or_velocity(*app*, *inputname*)

Function that computes the pre-shock Mach number of the mixture from a given pre-shock velocity or viceversa.

gui_compute_propReactants(*app*, *self*)

Function that compute fundamental properties (e.g., equivalence ratio, molar fractions, ...) of the mixture

Parameters

- **app** (object) – Combustion Toolbox app object
- **self** (struct) – Data of the mixture, conditions, and databases

Returns

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

gui_create_temp_app(*app*, *event*, *FLAG_COMPUTE_FROM_PHI*)

Function that creates a self struct required for preliminary calculations

Parameters

- **app** (object) – Combustion Toolbox app object
- **event** (object) – Event object
- **FLAG_COMPUTE_FROM_PHI** (bool) – Flag to compute properties from the equivalence ratio

Returns

self (struct) – Struct containing the properties of the mixture and the databases

gui_display_splash(*varargin*)

Display splash using SplashScreen [1]

Optional Name-Value Pairs Args:

- app (object): Combustion Toolbox app object

- color (float): Normalized RGB color (values from 0 to 1)
- pause (float): Time to pause before deleting the splash [seconds]

Returns
splash_obj (object) – Splash object

References

[1] Ben Tordoff (2022). SplashScreen (https://www.mathworks.com/matlabcentral/fileexchange/30508-splashscreen), MATLAB Central File Exchange.

gui_edit_phiValueChanged(*app*, *event*)

Update moles and mole fractions of the reactant UITables for the given equivalence ratio

gui_empty_UITables(*app*)

Clear data UITables and set to default the value of the equivalence ratio (-)

gui_fontcolor_error(*app*, *item*, *MAX_ERROR*)

Change fontcolor of the given item if the value is greater than error_max

gui_generate_panel_base(*app*)

Routine that generates all the components to read the thermodynamic properties at different stages of the rocket

gui_generate_panel_rocket(*app*)

Routine that generates all the components to read the thermodynamic properties at different stages of the rocket

gui_get_parameters(*app*)

GET EQUIVALENCE RATIO

gui_get_reactants(*app*, *event*, *self*, *varargin*)

Get the species and the number of moles of the current data in UITable_R

gui_get_tolerances(*app*)

Get tolerance from GUI and update values

gui_keep_last_entry(*app*, *entry_app*)

Function that keeps only the last entry, i.e., if the order is entry1 and entry2, entry1 will be set to empty value

gui_plot_custom_figures(*app*)

Function that plot custom figures based on the parameters of the GUI's custom figures tab

gui_save_results(*app*, *format*)

Save results as the given format (.xls or .mat)

Parameters

- **app** (obj) – class with all the data of the app
- **format** (char) – format extension/name

gui_seeker_exact_value(*app*, *event*, *ListValues*)

Return the value that match with the value introduced in the finder

gui_seeker_value(*app*, *event*, *ListValues*)

Return the value that match with the value introduced in the finder

gui_update_UITable_R(*app*, *self*)

Update data in the UITable_R with the next order: Inert -> Oxidizer -> Fuel

gui_update_from_uitree(*app*, *selectedNodes*)

Update GUI with the data that correspond with the selected node from the UITree

gui_update_frozen(*app*)

Set frozen chemistry and update Listbox of species

Parameters

app (object) – Combustion Toolbox app object

gui_update_phi(*app*, *self*)

Update GUI: equivalence ratio, O/F, and percentage Fuel

Parameters

- **app** (object) – Combustion Toolbox app object
- **self** (object) – Data of the mixture, conditions, and databases

gui_update_terminal(*app*, *self*, *type*)

Update GUI command window depending on the type of message

Parameters

- **app** (object) – Combustion Toolbox app object
- **self** (struct) – Data of the mixture, conditions, and databases
- **type** (char) – Type of message to be displayed

gui_value2list(*app*, *value*, *LS*, *action*)

Add/remove (action) selected value to/from the list of species (LS)

Parameters

- **app** (object) – Combustion Toolbox app object
- **value** (cell) – Cell array of char containing the selected species
- **LS** (cell) – Cell array of char containing the list of species
- **action** (char) – 'add' or 'remove'

Returns

LS (cell) – Cell array of char containing the list of species after the action

---

Note: If value is empty, the function returns LS without any change

---

gui_write_results(*app*, *results*, *i*, *varargin*)

Update GUI with the results of the ith case solved

Add-ons

The Add-ons included in CT are:

- `uielements`: to select and analyse the species in the database. The species from Third Millenium Database [Burcat and Ruscic, 2005] are denoted with suffix _M (see Fig. 1).
- `uipreferences`: to set all the preferences of Combustion Toolbox (see Fig. 2).
- `uifeedback`: to report bug/inquiries of Combustion Toolbox (see Fig. 3).
- `uivalidations`: to reproduce all the validations of Combustion Toolbox (see Fig. 4).
- `uiabout`: to know who are the developers and get useful links (see Fig. 5).

Figure 1: *Add-on uielements.*

Routines

class uielements(*varargin*)

Bases: `matlab.apps.AppBase`

C = None

Constants

DB = None

Reduced DataBase

DB_master = None

Master DataBase

E = None
Elements

FLAG_CALLER = None
Flag app is called from other app

LE_omit = None
List of elements to omit

LE_selected = None
List of elements selected

Misc = None
Miscelaneous

PD = None
Problem Description

PS = None
Problem Solution

S = None
Species

TN = None
Tunning properties

delete(*app*)
Delete UIFigure when app is deleted

Figure 2: *Add-on uipreferences.*

Routines

class uipreferences(*varargin*)
Bases: `matlab.apps.AppBase`

C = None
Constants

Misc = None
Miscellaneous properties

TN = None
Tuning properties

background_color = '[0.9098 0.9098 0.8902]'
Backgound color of the app

caller_app = None
Handle to caller app

delete(*app*)
Delete UIFigure when app is deleted

delta_x = '9'
Left margin in the right panel

delta_y = '-12'
Top margin in the right panel

dynamic_components = None
Struct with all the dynamic components

height_box = '22'
Default box height;

height_panel_0 = '38'
Default height of the right panel [pixels]

height_text = '14'
Default box height;

width_box = '60'
Default box width;

width_right = '521'
Default width right panel

x0_panel_right = '206'
Initial positition of right panel in the x-axis [pixels]

y0_panel_right = '428'
Initial positition of right panel in the y-axis [pixels]

Figure 3: *Add-on uifeedback.*

Figure 4: *Add-on uivalidations.*

Routines

class uivalidations

Bases: matlab.apps.AppBase

delete(*app*)

Delete UIFigure when app is deleted

Figure 5: *Add-on uiabout.*

Routines

class uiabout

Bases: matlab.apps.AppBase

delete(*app*)

Delete UIFigure when app is deleted

7.3 References

BIBLIOGRAPHY

[1] A. Cuadra, C. Huete, and M. Vera. Combustion Toolbox: A MATLAB-GUI based open-source tool for solving combustion problems. 2024. Version 1.0.5. doi:10.5281/zenodo.5554911.

[2] S. Gordon and B. J. McBride. Computer program for calculation of complex chemical equilibrium compositions and applications. Part 1: Analysis. *No. NAS 1.61:1311*, 1994.

[3] B. J. McBride. *NASA Glenn coefficients for calculating thermodynamic properties of individual species*. National Aeronautics and Space Administration, Glenn Research Center, 2002.

[4] A. Burcat and B. Ruscic. Third millenium ideal gas and condensed phase thermochemical database for combustion (with update from active thermochemical tables). Technical Report, Argonne National Lab. (ANL), Argonne, IL (United States), 2005. doi:10.2172/925269.

[5] B. Ruscic, R. E. Pinzon, G. Von Laszewski, D. Kodeboyina, A. Burcat, D. Leahy, D. Montoy, and A. F. Wagner. Active Thermochemical Tables: thermochemistry for the 21st century. In *Journal of Physics: Conference Series*, volume 16, 078. IOP Publishing, 2005. doi:10.1088/1742-6596/16/1/078.

[6] J. Sánchez-Monreal, A. Cuadra, C. Huete, and M. Vera. SimEx: A Tool for the Rapid Evaluation of the Effects of Explosions. *Applied Sciences*, 2022. doi:10.3390/app12189101.

[7] A. Cuadra, C. Huete, and M. Vera. Effect of equivalence ratio fluctuations on planar detonation discontinuities. *Journal of Fluid Mechanics*, 903:A30 1–39, 2020. doi:10.1017/jfm.2020.651.

[8] C. Huete, A. Cuadra, M. Vera, and J. Urzay. Thermochemical effects on hypersonic shock waves interacting with weak turbulence. *Physics of Fluids*, 33(8):086111, 2021. doi:10.1063/5.0059948.

[9] A. Cuadra, M. Vera, M. Di Renzo, and César Huete. Linear theory of hypersonic shocks interacting with turbulence in air. In *AIAA SciTech 2023 Forum, AIAA paper 2023–0075*. 2023. doi:10.2514/6.2023-0075.

[10] D. G. Goodwin, R. L. Speth, H. K. Moffat, and B. W. Weber. Cantera: an object-oriented software toolkit for chemical kinetics, thermodynamics, and transport processes. https://www.cantera.org, 2021. Version 2.5.1. doi:10.5281/zenodo.4527812.

[11] S. Browne, J. Ziegler, N. Bitter, B. Schmidt, J. Lawson, and J. E. Shepherd. SDToolbox - Numerical Tools for Shock and Detonation Wave Modeling. *GALCIT Technical Report FM2018.001 Revised January 2021, California Institute of Technology, Pasadena, CA*, 2008. , https://shepherd.caltech.edu/EDL/PublicResources/sdt.

[12] S. Browne, J. Ziegler, and J. E. Shepherd. Numerical solution methods for shock and detonation jump conditions. *GALCIT report FM2006*, 6:1–90, 2008.

[13] F. N. Fritsch and R. E. Carlson. Monotone piecewise cubic interpolation. *SIAM Journal on Numerical Analysis*, 17(2):238–246, 1980. doi:10.1137/0717021.

[14] J. Blecic, J. Harrington, and M. O. Bowman. TEA: A code calculating thermochemical equilibrium abundances. *The Astrophysical Journal Supplement Series*, 225(1):4 1–14, 2016. doi:10.3847/0067-0049/225/1/4.

[15] A. Cuadra. *Development of a wide-spectrum thermochemical code with application to planar reacting and non-reacting shocks*. PhD thesis, Universidad Carlos III de Madrid, Madrid, Spain, May 2023. Available at http://hdl.handle.net/10016/38179.

[16] J. W. Stock, D. Kitzmann, A. B. C. Patzer, and E. Sedlmayr. FastChem: A computer program for efficient complex chemical equilibrium calculations in the neutral/ionized gas phase with applications to stellar and planetary atmospheres. *Monthly Notices of the Royal Astronomical Society*, 479(1):865–874, 2018. doi:10.1093/mnras/sty1531.

[17] B. J. McBride. *Computer program for Calculation of Complex Chemical Equilibrium Compositions and Applications*. Volume 2. NASA Lewis Research Center, 1996.

INDEX

D

E

F

G

H

I

L

M

N

O

P

R

S

T

U

V

W

X

Y