MS Thermo

COOP Team

Jun 29, 2022

CONTENTS:

CONTENTS:

CHAPTER

ONE

INSTALLATION

This package is available on Python Package Index. Install with `pip install ms_thermo`.

This is a small package from Cerfacs dedicated to multispecies thermodynamics operations. Some short command line tools come with the installtion of the package

CHAPTER

TWO

COMMAND LINE TOOLS

- `tadia_table` : gives the final adiabatic temperature of a kerosene mixture. Inputs are the initial temperature, pressure and equivalence ratio.

```
>tadia_table 300 102000 0.7

The adiabatic flame temperature of a mix C10H22-air from tables is : 1904.29 K.

Species     |    Mass fraction
------------------------------
N2          |       0.732
fuel        |       0.000
O2          |       0.067
CO2         |       0.143
H2O         |       0.058
```

- `tadia_cantera` : same as previous, but using Cantera. (Require Cantera pre-installed). See our dedicated site for the original input and much mmore http://www.cerfacs.fr/cantera/.
- `fresh_gas` : a no-brainer again, conversion from primitive vriables (T, P, phi) to conservative variables(rho, rhoE, rhoYk). The inputs are the mixture temperature, pressure and equivalence ratio.

```
>fresh_gas 300. 101325 0.

rho       |  1.172 kg/m3
rhoE      |  253179.098 J.kg/m3
rhoYk     |
 N2       |  0.899 mol.kg/m3
 O2       |  0.273 mol.kg/m3
 KERO     |  0.000 mol.kg/m3
------------------------------
Yk        |
 N2       |  0.767 [-]
 O2       |  0.233 [-]
 KERO     |  0.000 [-]
```

- `yk_from_phi` : a no-brainer conversion from equivalence ratio to mass fraction. Inputs are the equivalence ratio and the numbers of C and H atoms in your fuel.

```
>yk_from_phi 0.7 1 4

Species     |    Mass fraction
```

(continues on next page)

(continued from previous page)

```
------------------------------
fuel        |       0.078
N2          |       0.707
O2          |       0.215
```

CHAPTER

THREE

GENERIC MODULE

There is also the `state` module, which allows to move from conservative to primitive variables and back. State consider a field of several locations, usually the nodes of a mesh. In the followng example, we create a 10 points field at 600K, then change a part of the field to a different temperature.

```
import numpy as np
from ms_thermo import State

  print("\nInitialize a 600K air mixture on 10 locations")
  state = State(
      temperature=600. * np.ones(10),
      pressure=100000.* np.ones(10),
      mass_fractions_dict={
          'O2': 0.2325 * np.ones(10),
          'N2': 0.7675 * np.ones(10)})
  print(state)
  print("\nSet half of the field to 1200K.")
  state.temperature = [600., 600., 600., 600., 600., 1200., 1200., 1200., 1200., 1200.]
  print(state)
```

```
Initialize a 600K air mixture on 10 locations

Current primitive state of the mixture
                | Most Common |    Min    |    Max
-------------------------------------------------
            rho| 5.78297e-01 | 5.783e-01 | 5.783e-01
         energy| 4.38546e+05 | 4.385e+05 | 4.385e+05
    temperature| 6.00000e+02 | 6.000e+02 | 6.000e+02
       pressure| 1.00000e+05 | 1.000e+05 | 1.000e+05
           Y_O2| 2.32500e-01 | 2.325e-01 | 2.325e-01
           Y_N2| 7.67500e-01 | 7.675e-01 | 7.675e-01


Set half of the field to 1200K.

Current primitive state of the mixture
                | Most Common |    Min    |    Max
-------------------------------------------------
            rho| 2.89148e-01 | 2.891e-01 | 5.783e-01
         energy| 4.38546e+05 | 4.385e+05 | 9.411e+05
    temperature| 6.00000e+02 | 6.000e+02 | 1.200e+03
```

(continues on next page)

(continued from previous page)

```
        pressure| 1.00000e+05 | 1.000e+05 | 1.000e+05
            Y_O2| 2.32500e-01 | 2.325e-01 | 2.325e-01
            Y_N2| 7.67500e-01 | 7.675e-01 | 7.675e-01
```

CHAPTER

FOUR

CONTRIBUTORS

This Python package is currently being developped by the CERFACS team COOP, with a non exhaustive list of the main contributors as of June 2022: Antoine Dauptain, Aimad Er-Raiy, Matthieu Rossi, Théo Defontaine, Thibault Gioud, Thibault Duranton, Elsa Gullaud, Victor Xing

4.1 ms_thermo package

4.1.1 Subpackages

ms_thermo.INPUT package

Batch examples Package

4.1.2 Submodules

4.1.3 ms_thermo.cli module

cli.py Command line interface for tools in ms_thermo

ms_thermo.cli.**add_version**(*f*)

Add the version of the tool to the help heading. :param f: function to decorate :return: decorated function

ms_thermo.cli.**redirect_fresh_gas**()

redicrection of former command

ms_thermo.cli.**redirect_gasout**()

redicrection of former command

ms_thermo.cli.**redirect_tadia_cantera**()

redicrection of former command

ms_thermo.cli.**redirect_tadia_table**()

redicrection of former command

ms_thermo.cli.**redirect_yk_from_phi**()

redicrection of former command

4.1.4 ms_thermo.constants module

Module Holding thermodynamic constants.

ms_thermo.constants.**GAS_CST = 8.3143**

ATOMIC_WEIGHTS used only in the case of a Chemkin or a Cantera Database, to compute molecular weight of a species *k* from its elemental composition as

$$W_k = \sum_{i}^{N_{elements}} b_{i,k} w_i$$

where:

- **Wk** : species *k* molecular weight
- **b(i,k)** : the number of element *i* atoms in species *k*
- **wi** : atomic weight of atom *i*

4.1.5 ms_thermo.example_gasout module

Module to show how a tool can vbe created from ms thermo.

Here we create a simple CLI for the Gasout tool

ms_thermo.example_gasout.**build_data_from_avbp**(*mesh*, *sol*)

Buid the data set from

ms_thermo.example_gasout.**gasout_tool**(*inputfile*)

Main call

ms_thermo.example_gasout.**load_mesh_and_solution**(*fname*, *mname*)

Read HDF5 mesh and solution files

ms_thermo.example_gasout.**save_data_for_avbp**(*state*, *sol*, *fname*)

Update the full solution with the state parameters

4.1.6 ms_thermo.flame_params module

Tool Flame param to get laminar premixed flame parameters such as laminar flame thickness and laminar flame speed from a table, with pressure, fresh gas temperature and equivalence ratio as entries.

class ms_thermo.flame_params.**FlameTable**

Bases: `object`

FlameTable class includes:

Parameters

- **phi_list** – list of equivalence ratios
- **temperature_list** – list of fresh gases temperatures
- **pressure_list** – list of pressures
- **datanames_list** – list of the stored data for the flames
- **data_dict** – dictionnary with all the stored data for the flames

check_bounds(*equivalence_ratio*, *temperature*, *pressure*)

Check that the input variables temperature, pressure and equivalence ratio are within the bounds of the stored data base

get_params(*equivalence_ratio=1.0*, *temperature=600*, *pressure=101325*)

Get the laminar flame params (flame speed, thickness and omega) from the pressure, the temperature and the equivalence ratio. It involves a trilinear interpolation in the table.

print_interpolated_data()

Print interpolated data

print_param_space()

Pretty print param space

read_table_hdf(*hdf_filename*)

Read the flame data table from an hdf format

Parameters

- **hdf_filename** – name of the input hdf file
- **verbosity** – display infos

4.1.7 ms_thermo.fresh_gas module

ms_thermo.fresh_gas.**fresh_gas**(*temperature*, *pressure*, *phi*, *fuel='KERO'*)

Return the the conservative values of the fresh gas from the primitive values.

Parameters

- **temperature** (*float*) – the fresh gas temperature
- **pressure** (*float*) – pressure of the fresh gas
- **phi** (*float*) – equivalence ratio of the air-fuel mixture
- **fuel** (*string*) – fuel

Returns

- **rho** - Density
- **rhoE** - Conservative energy
- **rhoyk** - Dict of conservative mass fractions

4.1.8 ms_thermo.gasout module

Module with functions helpers to create gasout for CFD solvers

ms_thermo.gasout.**alter_state**(*state*, *alpha*, *temp_new=None*, *press_new=None*, *y_new=None*, *verbose=False*)

Alter the state field rho;rhoE, rhoY

Parameters

- **state** – a ms_thermo State object od size n points.
- **alpha** – a mask of alteration shape(n) btw 0 (no alteration) to 1 (full alteration)

- **press_new** – (optionnal) the new mass pressure to apply either float or numpy array of shape (n)
- **temp_new** – (optionnal) the new temperature to apply either float or numpy array of shape (n)
- **y_new** – (optionnal) the new mass fractions to apply dict of species {"O2": 0.1, "N2", 0.9} values either floats or numpy arrays of shape (n)

ms_thermo.gasout.**directional_linear_mask**(*coor*, *axis=0*, *startstop=None*)

Compute a linear transition mask

Parameters

- **coor** – the n points coordinates a numpy array of shape (n, ndim)
- **axis** – 0 x, 1 y , 2 z
- **startstop** – start and stop on the direction [m]

Returns
alpha, a mask of floats shape (n) 0 if no mask 1 if masked

ms_thermo.gasout.**fraction_z_mask**(*state*, *specfuel*, *zmin=0.3*, *zmax=0.7*, *fuel_mass_fracs=None*, *oxyd_mass_fracs=None*, *atom_ref=None*, *verbose=False*)

Compute a mask found by a transition in Z farction

Parameters

- **state** – a ms_thermo state object
- **z_min** – mask disabled (0) below this value
- **z_max** – mask disabled (0) over this value

Specfuel
string the fuel species name

Returns
alpha, a mask of floats shape (n) 0 if no mask 1 if masked

ms_thermo.gasout.**gasout_dump_default_input**(*fname='gasout_in.yml'*)

Dump a Yamlf file adapted to control gasout_with_input

Note: i Stored this with a string to include the comments in the YAML. Not my best code…

ms_thermo.gasout.**gasout_with_input**(*coor*, *state*, *in_nob*)

Gasout function based upon a nest obeject input

Parameters

- **coor** – the n points coordinates a numpy array of shape (n, ndim)
- **state** – a ms_thermo State object od size n points.
- **in_nob** – a nested object for input, see gasout_dump_default_input fro reference

ms_thermo.gasout.**spherical_tanh_mask**(*coor*, *center=None*, *radius=None*, *delta=None*, *verbose=False*)

Compute a spherical mask.

Parameters

- **coor** – the n points coordinates a numpy array of shape (n, ndim)
- **center** – center coordinates of gasout [m] a list/tuple/nparray of size ndim

- **radius** – the radius of gasout [m]
- **delta** – the transition of gasout [m]

Returns
alpha, a mask of floats shape (n) 0 if no mask 1 if masked

4.1.9 ms_thermo.mixture_state module

Module for species and mixture

class ms_thermo.mixture_state.**MixtureState**(*species_dict*, *fuel*, *stream_update=None*, *convert_sp=None*)

Bases: object

Class managing mixture state

::

/

—-/————————
– m_ox —>

————| ————|

– m_fuel ->

—-————————
_ Stream 1

Attributes

- **_stream_dict** - Dict[‘O2’, ‘N2’, . . .] of dict[‘stream’, ‘mass_frac’]
- **_species** - List of SpeciesState object
- **_mixture_fraction** - Mixture fraction of the mixture based on Bilger’s
- **_far** - Fuel Air Ratio of the mixture
- **_far_st** - Stoechiometric Fuel Air Ratio of the mixture
- **_phi** - Equivalence ratio of the mixture

property afr

return Air Fuel Ratio

elem_mass_frac(*atom*, *stream=None*)

Compute elemental mass fraction of atom j in mixture

For each species i, get the elemental mass fraction of the atom j.

a_i,j * M_j * Y_i

Y_j = sum_i (————————)
M_i

with :

- **a_i,j** : Number of atom j in species i
- **M_j** : Molar mass of atom j
- **M_i** : Molar mass of species i
- **Y_i** : Mass fraction of species i

If stream is not None, the mass_fraction is defined as the mass_fraction of the species i in a the stream s :

- s = 1 : Fuel stream
- s = 2 : Oxydizer stream

$$Y_{j,s} = \sum_i \left(\frac{a_{i,j} * M_j * Y_{i,s}}{M_i} \right)$$

with :

- **Y_i,s** : Mass fraction of species i in stream s

property equivalence_ratio

returns equivalence ratio

property far

return Fuel Air Ratio

property far_st

return stoechiometric Fuel Air Ratio

property mixture_fraction

Return mixture fraction

property species

return list of SpeciesState object

species_by_name(*name*)

Gets SpeciesState by name

Parameters
name (*str*) – Name of the species

Returns
SpeciesState object matching with name

property species_name

Return list of species names

class ms_thermo.mixture_state.**SpeciesState**(*name*, *mass_fraction*, *stream=None*, *mass_fraction_stream=0.0*)

Bases: object

Class managing species

Attributes

- **_name** - Name of the species
- **_atoms** - Dict['CHON'] of atom numbers
- **_molar_mass** - Molar mass of the species

- **_mass_fraction** - Mass fraction of the species
- **_stream** - Input stream of the species
- **_mass_fraction_stream** - Mass fraction streamwise

property atoms

returns dict['CHON'] of number of atoms

mass()

Compute the mass of the species

Returns
Mass of the species

mass_fraction(*stream=None*)

Returns either mass fraction or stream-wize mass fraction

Returns
Mass Fraction of the species

property molar_mass

returns species molar mass

property name

returns species name

set_stream_data(*stream*, *mass_frac*)

Set stream of the species and streamwise mass fraction

Parameters

- **stream** (*int*) – Input stream number of the species (1 for fuel, 2 for oxydizer)
- **mass_frac** (*float*) – Streamsize mass fraction of the species:

4.1.10 ms_thermo.species module

Module to build thermodynamic properties of species.

ms_thermo.species.**build_thermo_from_avbp**(*database_file*)

Reading all AVBP database species and storing in a dict( ) whose keys are species names

Parameters
database_file (*str*) – Full path to database file

Returns
species - A dict( ) of Species objects whose keys are species names

ms_thermo.species.**build_thermo_from_cantera**(*database_file*)

Reading all **CANTERA** *database species and storing in a dict( ) whose keys are species names*

Parameters
database_file (*str*) – Full path to database file

Returns
species - A dict( ) of Species objects whose keys are species names

ms_thermo.species.**build_thermo_from_chemkin**(*database_file*)

Reading all **CHEMKIN** *database species and storing in a dict( ) whose keys are species names.*

Parameters
database_file (*str*) – Full path to database file

Returns
species - A dict( ) of Species objects whose keys are species names.

4.1.11 ms_thermo.state module

`State` is an object handling the internal state of a gaseous mixture, namely:

- Density
- Total energy
- Species mass fractions

Limitations of the `State` object

- Velocity is not a property of a `State` and must be treated separately.
- The spatial aspects, i.e. the position of the points, or the mesh, must be handled separately.

Warning: `State` variables are represented as structured arrays that must have the *same* shape

Initializing a `State`

A `State` object can be initialized in two ways:

- From the temperature, pressure, and species mass fractions (T, P, Y_k) through the default constructor:

```
state = State(T, P, Yk)
```

- From conservative variables $(\rho, \rho E, \rho Y_k)$ through the `from_cons` constructor:

```
state = State.from_cons(rho, rhoE, rhoYk)
```

The constructor arguments (T, P, Y_k) or $(\rho, \rho E, \rho Y_k)$ can be scalars or multidimensional arrays.

Warning: When initializing from conservative variables, T is determined by a Newton-Raphson method to ensure that the mixture energy matches the input total energy. This is an **expensive** step that may take a long time for large inputs.

Transforming a State

After a State has been initialized, T, P and Y_k can independently be set to new values (*e.g.* myState.temperature = newTemperature) and the other state variables are modified accordingly:

- When setting a new value for T, the other state variables are modified assuming an **isobaric and iso-composition** transformation from the previous state.
- When setting a new value for P, the other state variables are modified assuming an **isothermal and iso-composition** transformation from the previous state.
- When setting a new value for Y_k, the other state variables are modified assuming an **isothermal and isobaric** transformation from the previous state.

State transformations always satisfy the perfect gas equation of state

$$P = \rho \frac{R}{W_{mix}} T$$

class ms_thermo.state.**State**(*species_db=None*, *temperature=300.0*, *pressure=101325.0*, *mass_fractions_dict=None*)

Bases: object

Container class to handle mixtures.

property c_p

Get the mixture-averaged heat capacity at constant pressure

> **Warning:** C_p is computed like in AVBP as $C_v + P/(\rho T)$, *not* as the weighted average of species $C_{p,k}$

Returns
Cp - Heat capacity at constant pressure

Return type
ndarray or scalar

property c_v

Get the mixture-averaged heat capacity at constant volume

Returns
Cv - Heat capacity at constant volume

Return type
ndarray or scalar

compute_z_frac(*specfuel*, *fuel_mass_fracs=None*, *oxyd_mass_fracs=None*, *atom_ref='C'*, *verbose=False*)

Compute the Z mixture fraction.

0 oxidizer, 1 fuel

Parameters

- **specfuel** (*str*) – Fuel species
- **fuel_mass_fracs** (*dict, optional*) – Fuel mass fractions, defaults to composition at peak fuel concentration
- **oxyd_mass_fracs** (*dict, optional*) – Oxydizer mass fractions, defaults to air

- **atom_ref** (*str, optional*) – Reference atom, defaults to C
- **verbose** (*bool, optional*) – Verbosity, defaults to False

Returns
Z Mixture fraction

Return type
ndarray or scalar

property csound

Get the speed of sound

Returns
csound - Speed of sound

Return type
ndarray or scalar

classmethod from_cons(*rho*, *rho_e*, *rho_y*, *species_db=None*)

Class constructor from conservative values.

Parameters

- **rho** (*ndarray or scalar*) – Density
- **rho_e** (*ndarray or scalar*) – Total energy (conservative form)
- **rho_y** (*dict[str, ndarray or scalar]*) – Species mass fractions (conservative form)
- **species_db** (*Species, optional*) – Species database, defaults to AVBP species database

property gamma

Get the heat capacity ratio

Returns
gamma - Heat capacity ratio

Return type
ndarray or scalar

property list_spec

Get the names of the species

Returns
species_names - List of species names

Return type
list[str]

list_species()

Return primitives species names.

Returns
species_names - A list( ) of primitives species names

mach(*velocity*)

Compute the Mach number

Parameters
velocity (*ndarray or scalar*) – Velocity

Returns
M - Mach number

Return type
ndarray or scalar

property mass_fracs

Getter or setter for the species mass fractions. Setting the mass fractions will modify the density assuming an isothermal and isobaric transformation.

Return type
dict[str, ndarray or scalar]

mix_energy(*temperature*)

Compute the mixture total energy:

$$e = \sum_{k=1}^{N_{sp}} Y_k e_k$$

Parameters
temperature (*ndarray or scalar*) – Temperature

Returns
mix_energy - Mixture total energy

Return type
ndarray or scalar

mix_enthalpy(*temperature*)

Get mixture total enthalpy:

$$h = \sum_{k=1}^{N_{sp}} Y_k h_k$$

Parameters
temperature (*ndarray or scalar*) – Temperature

Returns
mix_enthalpy - Mixture total enthalpy

Return type
ndarray or scalar

mix_molecular_weight()

Compute mixture molecular weight following the formula :

$$W_{mix} = \left[\sum_{k=1}^{N_{sp}} \frac{Y_k}{W_k} \right]^{-1}$$

Returns
mix_mw (float) - Mixture molecular weight

property mix_w

Compute the mixture molecular weight:

$$W_{mix} = \left[\sum_{k=1}^{N_{sp}} \frac{Y_k}{W_k} \right]^{-1}$$

Returns
mix_mw - Mixture molecular weight

Return type
ndarray or scalar

`property pressure`

Getter or setter for the pressure. Setting the pressure will modify the density assuming an isothermal transformation.

Return type
ndarray or scalar

`pressure_total`(*velocity*)

Compute the total pressure:

$$P_t = P\left[1 + \frac{\gamma - 1}{2}M^2\right]^{\frac{\gamma}{\gamma-1}}$$

where M is the Mach number derived from the input velocity. This assumes an isentropic flow and constant gamma.

Parameters
velocity (*ndarray or scalar*) – Velocity

Returns
press_total - Total pressure

Return type
ndarray or scalar

`property temperature`

Getter or setter for the temperature. Setting the temperature will recompute the total energy and modify the density assuming an isobaric transformation.

Return type
ndarray or scalar

`temperature_total`(*velocity*)

Compute the total temperature:

$$T_t = T\left[1 + \frac{\gamma - 1}{2}M^2\right]$$

where M is the Mach number derived from the input velocity. This assumes an isentropic flow and constant gamma.

Parameters
velocity (*ndarray or scalar*) – Velocity

Returns
temp_total - Total temperature

Return type
ndarray or scalar

`update_state`(*temperature=None*, *pressure=None*, *mass_fracs=None*)

Compute density from temperature, pressure and mass fractions by assuming the following transformations:

1) Isobaric and isothermal transformation, i.e (*P=cst*, *T=cst* and only **composition** is varying)

2) Isobaric and iso-composition transformation, i.e (*P=cst*, *Y=cst* and only **temperature** is varying)

3) Isothermal and iso-composition transformation, i.e (*T=cst*, *Y=cst* and only **pressure** is varying)

Parameters

- **temperature** (*ndarray or scalar, optional*) – Temperature to set, defaults to None
- **pressure** (*ndarray or scalar, optional*) – Pressure to set, defaults to None
- **mass_fracs** (*dict[str, ndarray or scalar], optional*) – Mass fractions to set, defaults to None

4.1.12 ms_thermo.tadia module

ms_thermo.tadia.**tadia_cantera**(*t_fresh_gases*, *p_fresh_gases*, *phi*, *fuel='CH4'*, *cti_file='gri30.cti'*)

Compute the adiabatic flame temperature of a premixed fuel/air mixture from Cantera

Parameters

- **t_fresh_gases** (*float*) – Temperature of the fresh gases [K]
- **p_fresh_gases** (*float*) – Pressure of the fresh gases [Pa]
- **phi** (*float*) – Equivalence ratio [-]
- **fuel** (*string*) – Fuel considered
- **cti_file** (*string*) – Path to the cti file to consider

Returns

- **t_burnt_gases** - Temperature of the burnt gases
- **yk_burnt** - Dict of mass fractions of burnt gases

Note: **Warning**: This function may not be available if you do not have cantera in your environment

---

ms_thermo.tadia.**tadia_table**(*t_fresh_gases*, *p_fresh_gases*, *phi*, *fuel=None*, *c_x=10*, *h_y=20*)

Compute the adiabatic flame temperature of a premixed kero/air mixture from tables

Parameters

- **t_fresh_gases** (*float*) – Temperature of the fresh gases [K]
- **p_fresh_gases** (*float*) – Pressure of the fresh gases [Pa]
- **phi** (*float*) – Equivalence ratio [-]
- **fuel** (*str*) – path to flame table AVBP
- **c_x** (*float*) – nb. of C atoms
- **h_y** (*float*) – nb. of H atoms

Returns

- **t_burnt_gases** - Temperature of the burnt gases

- **yk_burnt** - Dict of mass fractions of burnt gases

4.1.13 ms_thermo.yk_from_phi module

This script calculate mass_fraction of species from a Phi

ms_thermo.yk_from_phi.**phi_from_far**(*far*, *c_x*, *h_y*)

Return phi coefficient with the fuel air ratio coeff + fuel composition

Parameters

- **far** (*float*) – the air-fuel ratio
- **c_x** (*float*) – stoechio coeff of Carbone
- **h_y** (*float*) – stoechio coeff of hydrogene

Returns

- **phi** - Equivalence ratio

ms_thermo.yk_from_phi.**yk_from_phi**(*phi*, *c_x*, *h_y*)

Return the mass fraction of species from a fuel aspect ratio and stoechio element coeff

Parameters

- **phi** (*float*) – the air-fuel aspect ratio
- **c_x** (*float*) – stoechio coeff of Carbone
- **h_y** (*float*) – stoechio coeff of hydrogene

Returns

- **y_k** - A dict of mass fractions

4.2 States in ms-thermo

The `State` object provides a full picture of the themochemical state of a gaseous mixture. It can be used to extract missing thermochemical quantities, or to keep a meaningful thermochemical state while modifying individual state variables. A `State` does not contain any velocity or numerical information.

4.2.1 Initializing a `State`

A `State` object can be initialized in two ways:

- From the temperature, pressure, and species mass fractions (T, P, Y_k) through the default constructor:

```
state = State(T, P, Yk)
```

- From conservative variables $(\rho, \rho E, \rho Y_k)$ through the `from_cons` constructor:

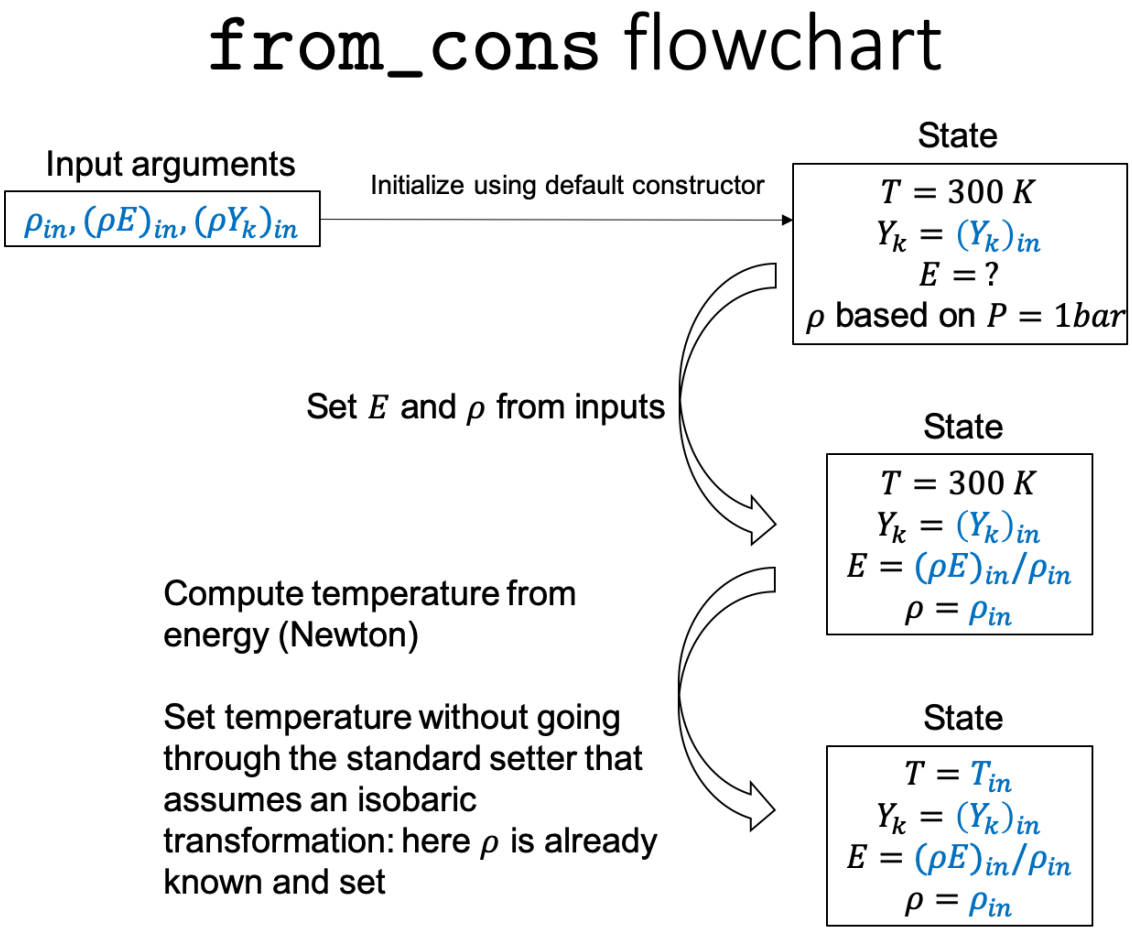
```
state = State.from_cons(rho, rhoE, rhoYk)
```

E is the total energy defined as the sum of the sensible and chemical (formation) energies ([TNC] p. 3)

The constructor arguments (T, P, Y_k) or $(\rho, \rho E, \rho Y_k)$ can be scalars or multidimensional arrays.

Warning: When initializing from conservative variables, T is determined by a Newton-Raphson method to ensure that the mixture energy matches the input energy. This is an **expensive** step that may take a long time for large inputs.

The following flowchart details how `from_cons` works



4.2.2 Transforming a `State`

After a `State` has been initialized, T, P and Y_k can independently be set to new values (*e.g.* `myState.temperature = newTemperature`) and the other state variables are modified accordingly:

- When setting a new value for T, the other state variables are modified assuming an **isobaric and iso-composition** transformation from the previous state.
- When setting a new value for P, the other state variables are modified assuming an **isothermal and iso-composition** transformation from the previous state.
- When setting a new value for Y_k, the other state variables are modified assuming an **isothermal and isobaric** transformation from the previous state.

State transformations always satisfy the perfect gas equation of state

$$P = \rho \frac{R}{W_{mix}} T$$

CHAPTER

FIVE

INDICES AND TABLES

- genindex
- modindex
- search

BIBLIOGRAPHY

[TNC] Poinsot, T.; Veynante, D. Theoretical and Numerical Combustion, 3rd ed.; 2011.

PYTHON MODULE INDEX

m

INDEX

N

P

R

S

T

U

Y