MS Thermo

COOP Team

Jun 29, 2022

CONTENTS:

This is a small package from Cerfacs dedicated to multispecies thermodynamics operations. It offers a simple way to manipulate reactive multispecies mixtures through *State objects*. *Command line tools* are also available for a few common tasks. *Tutorials* and more advanced *how-to guides* provide sample use cases for pre-processing and post-processing typical CFD multispecies mixtures.

CHAPTER

ONE

INSTALLATION

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

CHAPTER

TWO

CONTRIBUTORS

This Python package is currently being developped by the COOP team at Cerfacs, 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

2.1 Features

Presentation of the key features of ms_thermo.

2.1.1 Representing mixtures with State objects

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.

`State` initialization

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:

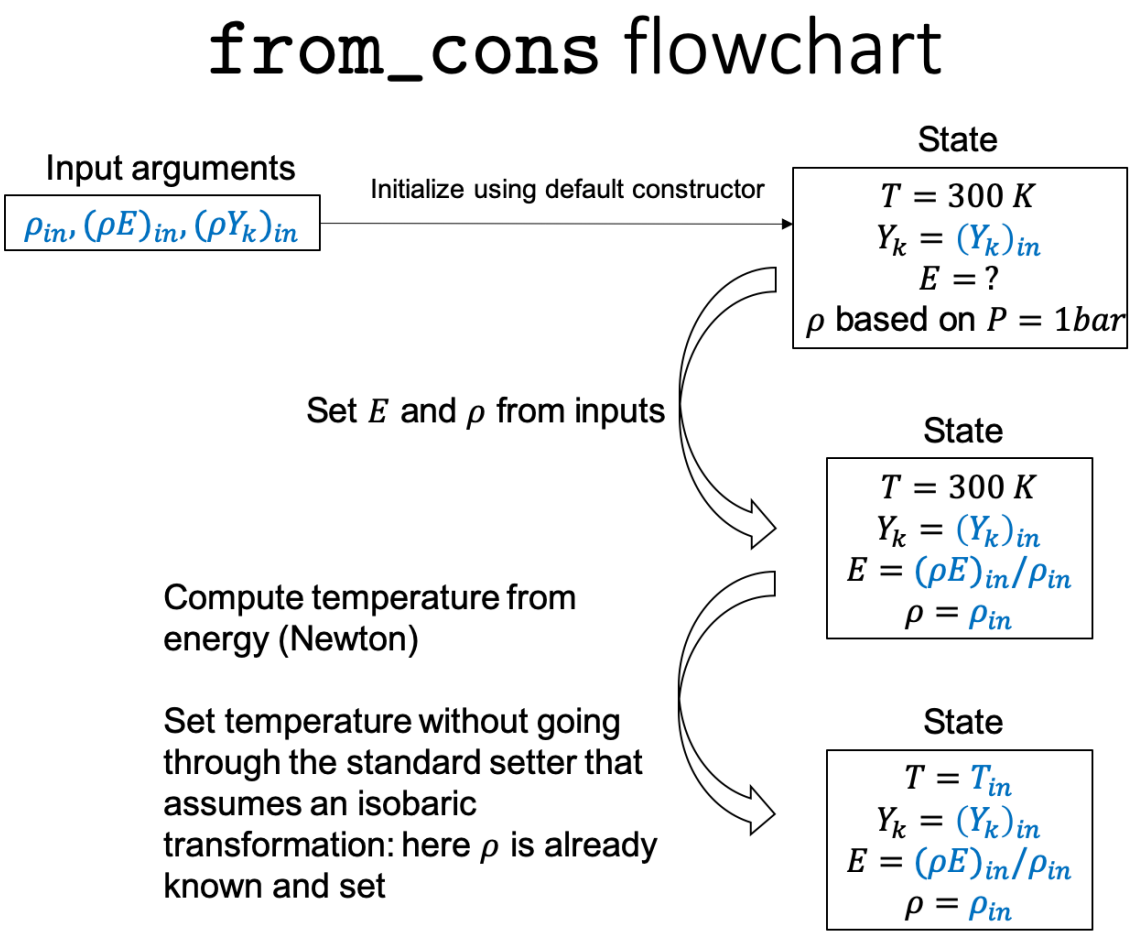
```
state = State.from_cons(rho, rhoE, rhoYk)
```

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


from_cons flowchart
Input arguments
$\rho_{in}, (\rho E)_{in}, (\rho Y_k)_{in}$
Initialize using default constructor
State
$T = 300\ K$
$Y_k = (Y_k)_{in}$
$E = ?$
ρ based on $P = 1bar$
Set E and ρ from inputs
State
$T = 300\ K$
$Y_k = (Y_k)_{in}$
$E = (\rho E)_{in}/\rho_{in}$
$\rho = \rho_{in}$
Compute temperature from energy (Newton)
Set temperature without going through the standard setter that assumes an isobaric transformation: here ρ is already known and set
State
$T = T_{in}$
$Y_k = (Y_k)_{in}$
$E = (\rho E)_{in}/\rho_{in}$
$\rho = \rho_{in}$

`State` transformations

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

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

2.1.2 Command line tools

Terminal commands are available for common use cases.

`gasout`

This command is a pre-processing tool built on top of the ms-thermo package. It modifies the state of a mixture in specific regions of the domain. This is the default input file demonstrating the actions that are available.

```yaml
# Input files
inst_solut: ./solut_0003.sol.h5
mesh: ./solut_0003.mesh.h5
inst_solut_output: ./gasouted.sol.h5

# Actions to take
actions:
- type: directional_linear_mask
  direction: "x"           # Direction x, y, or z
  transition_start: 0.1
  transition_end: 0.11
  new_pressure: null       # Use null if you don't want to edit this field
  new_temperature: 2000.0  # Temperature in K
  new_yk: null

- type: spherical_tanh_mask
  center: [0.1, 0.1, 0]    # Center of the sphere
  radius: 0.01             # Radius of the sphere [m]
  delta: 0.05              # Transition thickness [m]
  new_pressure: null
  new_temperature: 1600.
  new_yk:                  # Dictionary of species Yk
    N2: 0.8                # Order does not matter
    O2: 0.2                # Sum MUST be 1!

- type: fraction_z_mask
```

(continues on next page)

(continued from previous page)

```
  specfuel: KERO_LUCHE     # Fuel species name
  atom_ref: C              # Reference atom for Z computation (default: 'C')
  oxyd_mass_fracs:         # Mixture oxidizer for Z computation (default: AIR)
    O2: 0.233
    N2: 0.767
  fuel_mass_fracs:  None   # Mixture fuel (default: mixture at peak fuel concentration)
  zmax: 0.02
  zmin: 0.01
  new_pressure: 103000.0   # Pressure in Pa
  new_temperature: null
  new_yk: null

- type: fraction_z_mask    # You can repeat several treatments
  specfuel: KERO_LUCHE
  atom_ref: C
  oxyd_mass_fracs:
    O2: 0.233
    N2: 0.767
  fuel_mass_fracs:  None
  zmax: 0.05
  zmin: 0.04
  new_pressure: 103000.0
  new_temperature: null
  new_yk: null
```

This default input file is generated when calling `ms_thermo gasout --new foo.yml`. From an input YAML file, run the tool with `ms_thermo gasout foo.yml`.

hp-equil

Compute the adiabatic flame temperature and mass fractions using a CANTERA *.cti* file.

```
>ms_thermo hp-equil 300 101325 1 NC10H22 10 22 src/ms_thermo/INPUT/Luche1.cti

The adiabatic flame temperature of a mix NC10H22-air from cantera is : 2277.42 K.

Species     |    Mass fraction
------------------------------
CO          |       0.014
CO2         |       0.172
H2O         |       0.084
N2          |       0.718
NO          |       0.003
O2          |       0.007
OH          |       0.002
+ 82 others |       0.000
```

Warning: This command requires the Cantera package.

kero-prim2cons

Compute the conservative variables of a kerosene-air mixture from primitive variables T, P and phi (equivalence ratio).

```
>ms_thermo kero-prim2cons 300 101325 1

rho        |  1.232 kg/m3
rhoE       |  266054.682 J.kg/m3
rhoYk      |
 KERO      |  0.077 mol.kg/m3
 N2        |  0.886 mol.kg/m3
 O2        |  0.269 mol.kg/m3
------------------------------
Yk         |
 KERO      |  0.063 [-]
 N2        |  0.719 [-]
 O2        |  0.218 [-]
```

kero-tadia

Compute the final adiabatic temperature of a kerosene-air mixture from T, P and phi (equivalence ratio) It is based on tabulation created using Cantera.

```
>ms_thermo kero-tadia 300 101325 0.7

The adiabatic flame temperature of a mix C10H22-air from tables is : 1904.30 K.

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

yk-from-phi

Compute species mass fractions of a hydrocarbon fuel-air mixture. Inputs are the equivalence ratio and the numbers of C and H atoms in the fuel.

```
>ms_thermo yk-from-phi 0.7 1 4 CH4

Species     |    Mass fraction
------------------------------
CH4         |       0.078
N2          |       0.707
O2          |       0.215
```

2.2 Tutorials

Step-by-step guides for new users of ms_thermo.

2.2.1 Using a `State` object

Basic usage

In ms-thermo, `State` objects are used to represent the full thermodynamic state of a gas mixture. In the following example, we create a State from 10 values of temperature, pressure and species mass fractions, then modify the temperature in part of the field.

```
import numpy as np
from ms_thermo.state import State

print("\nInitialize a 600 K 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 1200 K.")
state.temperature = [600., 600., 600., 600., 600., 1200., 1200., 1200., 1200., 1200.]
print(state)
```

```
Initialize a 600 K 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 1200 K.

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
       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
```

Note that the modification of the maximum temperature has also affected the minimum density. Setting new values

for the temperature also affects the density to ensure that the equation of state is satisfied. Explanations on the inner workings of a `State` are given *here*.

Computing thermodynamic quantities in post-processing

`States` can be useful to easily recover any thermodynamic quantity from the base set of primitive or conservative variables. In the following example, the temperature and heat capacity at constant pressure in an AVBP solution are computed and saved using ms-thermo.

```
from h5cross import hdfdict
import h5py
import numpy as np
from ms_thermo.state import State

sol_path = '1DFLAME/solut_00000050.h5'

with h5py.File(sol_path, "r") as fin:
    sol = hdfdict.load(fin, lazy=False)
state = State.from_cons(
    sol["GaseousPhase"]["rho"], sol["GaseousPhase"]["rhoE"], sol["RhoSpecies"]
)
np.save('T.npy', state.temperature)
np.save('Cp.npy', state.c_p)
```

2.3 How-to guides

Detailed guides on practical use cases for ms_thermo.

2.3.1 Extending `gas_out` for a custom application

This script was written by Antony Cellier for a custom use case of `gas_out` in a cylindrical geometry.

First, two helper functions to convert an AVBP solution to a `State` are defined:

```
import copy as cp
import h5py as h5
import numpy as np
import hdfdict as hd
from ms_thermo.state import State
from ms_thermo.species import build_thermo_from_avbp


def load_mesh_and_solution(fname, mname):
    """Read HDF5 mesh and solution files"""

    print("Reading solution in ", fname)
    sol_h5 = h5.File(fname, 'r')

    sol = dict()
    for key1 in sol_h5.keys():
```

(continues on next page)

(continued from previous page)

```
        sol_int=dict()
        for key2 in sol_h5[key1].keys():
            ARR_tot = np.array(sol_h5[key1][key2])
            sol_int[key2] = ARR_tot
        sol[key1] = sol_int

    print("Reading mesh in ", mname)
    mesh_h5 = h5.File(mname, 'r')

    mesh = dict()
    for key1 in mesh_h5.keys():
        mesh_int=dict()
        for key2 in mesh_h5[key1].keys():
            ARR_tot = np.array(mesh_h5[key1][key2])
            mesh_int[key2] = ARR_tot
        mesh[key1] = mesh_int

    return mesh, sol

def build_data_from_avbp(mesh, sol, species_db=None):
    """Build coordinates and state from AVBP"""

    state = State.from_cons(
        sol["GaseousPhase"]["rho"],
        sol["GaseousPhase"]["rhoE"],
        sol["RhoSpecies"], species_db=species_db)

    x_coor = mesh["Coordinates"]["x"]
    y_coor = mesh["Coordinates"]["y"]

    try:
        z_coor = mesh["Coordinates"]["z"]
        coor = np.stack((x_coor, y_coor, z_coor), axis=-1)
    except KeyError:
        coor = np.stack((x_coor, y_coor, np.zeros(x_coor.shape)), axis=-1)

    return coor, state
```

The ms-thermo `State` is created:

```
sname = 'path/to/species/db'
fname = 'path/to/solution'
mname = 'path/to/mesh'
species_db = build_thermo_from_avbp(sname)
MESH, INIT = ms_to.load_mesh_and_solution(fname, mname)
COOR, STATE = ms_to.build_data_from_avbp(MESH, INIT, species_db=species_db)
```

Then, a custom alteration mask adapted to the geometry is introduced:

```
def where_alter_cyl(COOR, axis, l_min, l_max, r_cyl):
    """Give the list of points to alter in the solution
    Avoid applying alteration to the entire domain"""
```

(continues on next page)

(continued from previous page)

```
    ALTER = []

    for k in range(COOR.shape[0]):
        x = COOR[k, axis]
        d = [0, 1, 2]
        d.remove(axis)
        r = np.sqrt(COOR[k, d[0]]**2 + COOR[k, d[1]]**2)

        # Lower Cylinder
        if x >= l_min and x <= l_max and r < r_cyl :
            ALTER += [k]

    return ALTER

axis = ...
x_0_min = ...
x_0_max = ...
r_0 = ...
ALTER = where_alter_cyl(COOR, axis, x_0_min, x_0_max, r_0)
```

Temperature, pressure, species mass fractions are generated from Riemann solutions and altered with `gas_out`. For this case, velocity vectors are also generated and altered using the same mask:

```
T, P, Y, U = ...

def create_new_vectors(STATE, COOR, ALTER, T, P, Y, U):
    ''' Apply the local changes on the points to alter
        Build the array on the entire initial solution '''

    #Empty
    T_new = STATE.temperature
    P_new = STATE.pressure
    U_new = np.zeros((COOR.shape[0],))

    Y_new = STATE.mass_fracs

    #Fill
    for k,I in enumerate(ALTER):

        T_new[I] = T[k]
        P_new[I] = P[k]
        U_new[I] = U[k]

        for spec in Y:
            Y_new[spec][I] = Y[spec][k]

    return T_new, P_new, Y_new, U_new

T_new, P_new, Y_new, U_new = create_new_vectors(STATE, COOR, ALTER, T, P, Y, U)
```

The `State` is updated with the new values:

```
def alter_state_riemann(state, temp_new=None, press_new=None, y_new=None, verbose=False):
    ''' Fill the initial state with Riemann solutions'''

    log = str()

    if y_new is not None:

        state_species = list(state.mass_fracs.keys())

        for spec in y_new:
            if spec not in state_species:
                msgerr = "\n\n Species mismatch:"
                msgerr += "\n"+ str(spec) + " is not part of the mixture:\n"
                msgerr +=  "/".join(state_species)
                msgerr +=  "\nCheck your input file pretty please..."
                raise RuntimeWarning(msgerr)

        for spec in state.mass_fracs:
            if spec in y_new:
                state.mass_fracs[spec] = y_new[spec]

    if temp_new is not None:
        state.temperature = temp_new

    if press_new is not None:
        state.pressure = press_new

    if verbose:
        log += ("Filling of the Riemann solution DONE")

    return log, state

    _, STATE_FILL = alter_state(STATE, temp_new=T_new, press_new=P_new, y_new=Y_new)
```

Finally, the AVBP solution is updated with the new state. The kinetic energy from the velocity vector is added to the total energy.

```
def save_data_for_avbp(state, sol, fname, movement=False, U=None, V=None, W=None):
    """Update the full solution with the state parameters"""

    sol_new = cp.deepcopy(sol)
    sol_new["GaseousPhase"]["rho"] = state.rho

    if movement:
        if W is None:
            KIN = 1/2 * (np.power(U, 2) + np.power(V, 2))
        else:
            KIN = 1/2 * (np.power(U, 2) + np.power(V, 2) + np.power(W, 2))

        sol_new["GaseousPhase"]["rhoE"] = state.rho * (state.energy + KIN)
        sol_new["GaseousPhase"]["rhou"] = state.rho * U
        sol_new["GaseousPhase"]["rhov"] = state.rho * V
```

(continues on next page)

(continued from previous page)

```
    else:
        sol_new["GaseousPhase"]["rhoE"] = state.rho * state.energy

    for spec in sol_new["RhoSpecies"]:
        sol_new["RhoSpecies"][spec] = state.rho * state.mass_fracs[spec]

    try:
        sol_new["Additionals"]["temperature"] = state.temperature
        sol_new["Additionals"]["pressure"] = state.pressure
    except KeyError:
        print("-Additionals- group is not part of the solution.")

    print("Saving solution in ", fname)
    with h5.File(fname, 'w') as fout:
        hd.dump(sol_new, fout)

V_new = np.zeros(U_new.shape)
save_data_for_avbp(STATE_FILL, INIT, fname, movement=True, U=U_new, V=V_new)
```

2.4 API reference

This section is an exhaustive list of all the interfaces of ms_thermo.

2.4.1 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

2.4.2 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*

2.4.3 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

2.4.4 ms_thermo.kero_prim2cons module

ms_thermo.kero_prim2cons.**kero_prim2cons**(*temperature*, *pressure*, *phi*)

Compute conservative variables from primitive variables in a kerosene-air mixture.

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

2.4.5 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*)

Apply gasout alterations to a State.

Parameters

- **state** (State) – State before alterations
- **alpha** (*ndarray or scalar*) – Alteration mask with values from 0 (no alteration) to 1 (full alteration)
- **temp_new** (*ndarray or scalar, optional*) – New temperature to apply, defaults to None
- **press_new** (*ndarray or scalar, optional*) – New temperature to apply, defaults to None
- **y_new** (*dict[str, ndarray or scalar], optional*) – New mass fractions to apply, defaults to None
- **verbose** (*bool, optional*) – Verbosity, defaults to False

Returns

log, debug log

Return type

str

ms_thermo.gasout.**directional_linear_mask**(*coor*, *axis*, *transition_start*, *transition_end*)

Define a directional mask aligned with the *axis* coordinate axis.

If *transition_end* > *transition_start*, the mask is 0 before *transition_start* and 1 after *transition_end. If `transition_start* > *transition_end*, the mask is 1 before *transition_end* and 0 after *transition_start. A linear transition is imposed between `transition_start* and *transition_end*.

Parameters

- **coor** (*ndarray*) – Array of spatial coordinates (shape (n, ndim))
- **axis** (*int*) – Coordinate axis of the mask (0 x, 1 y, 2 z)
- **transition_start** (*float*) – Start point of the linear mask
- **transition_end** (*float*) – End point of the linear mask

Returns
alpha, alteration mask

Return type
ndarray

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

Compute a mask based on the mixture fraction Z.

The mask is 1 between *zmin* and *zmax* and 0 outside.

Parameters

- **state** (State) – ms_thermo State object
- **specfuel** (*str*) – Fuel species name
- **zmin** (*float*) – mask disabled (0) below this value
- **zmax** (*float*) – mask disabled (0) over this value
- **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
alpha, alteration mask

Return type
ndarray

ms_thermo.gasout.**gasout_dump_default_input**(*fname*)

Dump the default gasout input file

ms_thermo.gasout.**gasout_tool**(*inputfile*)

Main call

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

Update a State with gasout actions.

Parameters

- **coor** (*ndarray*) – Array of spatial coordinates (shape (n, ndim))

- **state** (State) – ms_thermo State object
- **in_nob** (*dict*) – Contents of the input file

Returns
Tuple containing the updated State and a log

Return type
tuple

ms_thermo.gasout.**load_mesh_and_solution**(*fname*, *mname*)
Load a mesh and solution into HDF5 objects

Parameters

- **fname** (*str*) – Filename of the solution
- **mname** (*str*) – Filename of the mesh

ms_thermo.gasout.**save_data_for_avbp**(*state*, *sol*, *fname*)
Update the full solution with the state parameters

ms_thermo.gasout.**spherical_tanh_mask**(*coor*, *center*, *radius*, *delta*)
Define a spherical mask. 0 inside the sphere, 1 outside, with a tanh transition.

Parameters

- **coor** (*ndarray*) – Array of spatial coordinates (shape (n, ndim))
- **center** (*list, tuple or ndarray*) – Array of sphere center coordinates (shape (ndim,))
- **radius** (*float*) – Radius of the sphere
- **delta** (*float*) – Transition thickness at the edge of the sphere

Returns
alpha, alteration mask

Return type
ndarray

2.4.6 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.

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

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:

2.4.7 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.

2.4.8 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

2.4.9 ms_thermo.tadia module

ms_thermo.tadia.**tadia_cantera**(*t_fresh_gases*, *p_fresh_gases*, *phi*, *c_x*, *h_y*, *fuel_name*, *cti_file*)

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 [-]
- **c_x** (*float*) – nb. of C atoms
- **h_y** (*float*) – nb. of H atoms
- **fuel_name** (*string*) – Name of the fuel species
- **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*, *fuel_name='KERO'*, *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
- **fuel_name** (*str*) – Name of the fuel species
- **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

2.4.10 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*, *fuel_name*)

Return the species mass fractions in a fresh fuel-air mixture

Parameters

- **phi** (*float*) – equivalence ratio
- **c_x** (*float*) – stoechio coeff of Carbone
- **h_y** (*float*) – stoechio coeff of hydrogene
- **fuel_name** (*str*) – Name of the fuel

Returns

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

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