PyCSP `v1.1`

Software documentation

February 2021

Contents

1 Introduction

PyCSP is a collection of libraries for the analysis, model reduction and ODE integration of reacting systems using the computational singular perturbation (CSP) method and its extensions, including the Tangential Stretching Rate.

1.1 Copyright

Copyright ©2021 Riccardo Malpica Galassi

Permission is hereby granted, free of charge, to any person obtaining a copy of this software and associated documentation files (the "Software"), to deal in the Software without restriction, including without limitation the rights to use, copy, modify, merge, publish, distribute, sublicense, and/or sell copies of the Software, and to permit persons to whom the Software is furnished to do so, subject to the following conditions:

The above copyright notice and this permission notice shall be included in all copies or substantial portions of the Software.

THE SOFTWARE IS PROVIDED "AS IS", WITHOUT WARRANTY OF ANY KIND, EXPRESS OR IMPLIED, INCLUDING BUT NOT LIMITED TO THE WARRANTIES OF MERCHANTABILITY, FITNESS FOR A PARTICULAR PURPOSE AND NONINFRINGEMENT. IN NO EVENT SHALL THE AUTHORS OR COPYRIGHT HOLDERS BE LIABLE FOR ANY CLAIM, DAMAGES OR OTHER LIABILITY, WHETHER IN AN ACTION OF CONTRACT, TORT OR OTHERWISE, ARISING FROM, OUT OF OR IN CONNECTION WITH THE SOFTWARE OR THE USE OR OTHER DEALINGS IN THE SOFTWARE.

1.2 Contacts

Riccardo Malpica Galassi:
`riccardo.malpicagalassi@uniroma1.it`, `riccardo.malpica.galassi@ulb.be`

1.3 Changes with respect to previous version (v1.0)

- Added extended TSR analysis
- minor bug-fixes

1.4 Folder structure

```
PyCSP
├── PyCSP
│   ├── ThermoKinetics.py
│   ├── Functions.py
│   ├── Solver.py
│   ├── Simplify.py
│   └── utils.py
├── tests
│   ├── test_rhs.py
│   ├── test_rhs_constV.py
│   └── test_kernel.py
└── examples
    ├── cspAnalysis
    │   ├── exhaustedModes.py
    │   └── indices.py
    ├── dataImportExport
    │   └── export_import.py
    ├── cspSolver
    │   ├── cspsolver_const_p.py
    │   └── cspsolver_const_v.py
    ├── cspSimplify
    │   ├── simp.py
    │   └── simp_TSR.py
    └── tsrAnalysis
        ├── TSR.py
        └── TSRext.py
```

2 Installation

2.1 Prerequisites

PyCSP relies on Cantera 2.4.0, numpy and matplotlib. The current version of PyCSP has been tested with Python 3.6.10. The Python version must comply with Cantera. Presently, versions higher than 3.6.10 are not supported by Cantera 2.4.0. In February 2021, Cantera's authors released thier version 2.5.0. PyCSP compliance with such version will be part of a future release.

2.2 Installation

Installation using Anaconda is recommended. In such case, proceed as follows via command line:

$ `git clone https://github.com/rmalpica/PyCSP.git`

$ `cd $PATH_TO_PyCSP_MAIN_FOLDER`

$ `conda create --name py36 python=3.6 anaconda --file requirements.txt --channel default --channel anaconda --channel cantera`

$ `conda activate py36`

$ `pip install $PATH_TO_PyCSP_MAIN_FOLDER` (e.g. /Users/rmalpica/PyCSP)

Alternatively, once the prerequisite libraries are installed, the installation may be performed via command line as follows:

$ `git clone https://github.com/rmalpica/PyCSP.git`

$ `pip install $PATH_TO_PyCSP_MAIN_FOLDER` (e.g. /Users/rmalpica/PyCSP)

2.3 Testing

To test the installation, proceed as follows via command line:

$ `cd tests`

$ `python test_kernel.py`

The test should outcome two images: first, the time evolution of temperature, OH, H and H_2 in a constant pressure reactor run with Cantera, second, the time evolution of the system's eigenvalues.

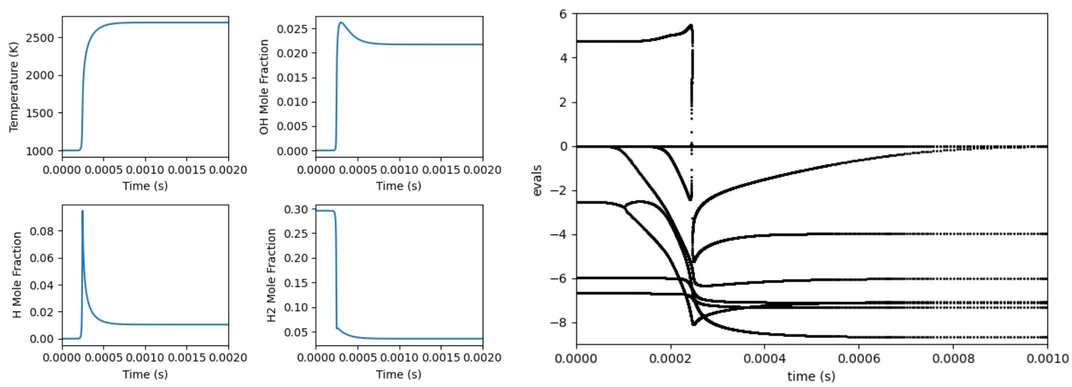

Figure 1: Testing outcomes

3 Classes, attributes and methods

3.1 `ThermoKinetics.py`

3.1.1 class `CanteraThermoKinetics`

This class inherits attributes and methods from Cantera's `Solution` class and extends its capabilities by adding thermodynamics and kinetics-related class methods. Hence, the constructor for this class is equivalent to the constructor of Cantera's `Solution` class. The proper way to instantiate `CanteraThermoKinetics` objects is by using species and reactions defined in an input file, e.g. `hydrogen.cti`:

```python
import PyCSP.ThermoKinetics as csp
gas = csp.CanteraThermoKinetics('hydrogen.cti')
```

Different formulations exist for the majority of this class methods depending on whether the reactor is considered having a constant pressure or a constant volume (density). Hence, it is mandatory to choose between constant pressure and constant density after instantiating an object of the `CanteraThermoKinetics` class and before accessing its attributes. The choice is made by defining either the pressure value or the density value through the attributes `constP` or `constRho` (see below). A lack of this choice will raise an exception when accessing the class attributes having multiple formulations.

Instance attributes:

- `constP`
  Type (*float*).
  Set/get the thermodynamic pressure [Pa] and activates the constant pressure version of all the class methods (`problemtype == const_p`).

- `constRho`
  Type (*float*).
  Set/get the thermodynamic density [Kg/m^3] and activates the constant volume version of all the class methods (`problemtype == const_v`).

- `problemtype`
  Type (*string, read-only*).
  Get the active definition of `problemtype`, either `const_p` or `const_v`.

- `nv`
  Type (*int*).
  Set/get the number of variables. Defaults to `n_species` + 1. The only alternative value is `n_species`, e.g., to exclude temperature from the list of variables and get appropriately sized objects from the class methods.

- `source`
  Type ([N_s+1] *array of float, read-only*).
  Get the chemical source term [$\dot{\omega}_1$, $\dot{\omega}_2$, ..., $\dot{\omega}_{N_s}$, $\dot{\omega}_T$] for the species mass fractions and temperature, based on either the constant pressure or constant volume formulation. Depending on the value of `nv`, the last component (temperature source term) is returned or not.

- `jacobian`
  Type ([$N_s+1 \times N_s+1$] *array of float, read-only*).
  Get the numerical Jacobian of the chemical source term, based on either the constant pressure or constant volume formulation.

- `generalized_Stoich_matrix`
  Type ([$N_v \times 2N_r$] *array of float, read-only*).
  Get the generalized stoichiometric matrix, based on either the constant pressure or constant volume formulation. Depending on the value of `nv`, the last row (temperature coefficients) is returned or not.

- `R_vector`
  Type ([$2N_r$] *array of float, read-only*).
  Get the reaction rates vector.

Class methods:

- `set_stateYT(state)`.
  Set the thermochemical state of the mixture. This is an alternative to separately setting Cantera's attributes `Y` and either `TP` or `TD`. Pressure and density are automatically set based on the values of `constP` or `constRho`.
  Input:

`state` (array of float). Array containing the mass fractions Y_i of all the species and the temperature: [Y_1, Y_2, ..., Y_N, T]. The order of the Y_i values must comply with the species ordering set at the class instantiation. This can be retrieved with the method `species()`.

- `stateYT()`.
  Returns the thermochemical state of the mixture [Y_1, Y_2, ..., Y_N, T].

- `rhs_const_p()`.
  Returns the constant-pressure chemical source term of the mixture $[\dot{\omega}_1, \dot{\omega}_2, \ldots, \dot{\omega}_{N_s}, \dot{\omega}_T]_P$, where:

$$\dot{\omega}_i = \frac{W_i}{\rho} \sum_{k=1}^{2N_r} \nu_{ik} r^k \qquad i = 1, \ldots, N_s$$

$$\dot{\omega}_T = -\frac{1}{c_p} \sum_{i=1}^{N_s} \dot{\omega}_i h_i$$

  where W_i is the molecular mass of species i, ρ is the mixture density, ν_{ik} is the molar stoichiometric coefficient of species i in reaction k, r^k is the molar rate of progress of reaction k, c_p is the mixture constant pressure specific heat and h_i is the molar enthalpy of species i.

- `rhs_const_v()`.
  Returns the constant-volume chemical source term of the mixture $[\dot{\omega}_1, \dot{\omega}_2, \ldots, \dot{\omega}_{N_s}, \dot{\omega}_T]_v$, where:

$$\dot{\omega}_i = \frac{W_i}{\rho} \sum_{k=1}^{2N_r} \nu_{ik} r^k \qquad i = 1, \ldots, N_s$$

$$\dot{\omega}_T = -\frac{1}{c_v} \sum_{i=1}^{N_s} \dot{\omega}_i h_i + \frac{RT}{c_v} \sum_{i=1}^{N_s} \dot{\omega}_i$$

- `generalized_Stoich_matrix_const_p()`.
  Returns the constant-pressure generalized stoichiometric matrix $\mathcal{S}$, defined as:

$$\mathcal{S}_{i,k} = \frac{W_i}{\rho} \nu_{ik} \qquad i = 1, \ldots, N_s \qquad k = 1, \ldots, 2N_r$$

$$\mathcal{S}_{N_s+1,k} = -\frac{1}{\rho c_p} \sum_{i=1}^{N_s} W_i \, \nu_{ik} \, h_i \qquad k = 1, \ldots, 2N_r$$

- `generalized_Stoich_matrix_const_v()`.
  Returns the constant-volume generalized stoichiometric matrix $\mathcal{S}$, defined as:

$$\mathcal{S}_{i,k} = \frac{W_i}{\rho} \nu_{ik} \qquad i = 1, \ldots, N_s \qquad k = 1, \ldots, 2N_r$$

$$\mathcal{S}_{N_s+1,k} = -\frac{1}{\rho c_v} \sum_{i=1}^{N_s} W_i \nu_{ik} \, h_i + \frac{RT}{\rho \, c_v} \sum_{i=1}^{N_s} W_i \, \nu_{ik} \qquad k = 1, \ldots, 2N_r$$

- `Rates_vector()`.
  Returns the molar rates of progress r^k vector of the mixture.

- `jacobian_const_p()`.
  Returns the constant-pressure numerically-evaluated Jacobian of the chemical source term

of the mixture:

$$J_{\dot{\omega}} = \begin{bmatrix} \left.\frac{\partial \dot{\omega}_1}{\partial Y_1}\right|_P & \left.\frac{\partial \dot{\omega}_1}{\partial Y_2}\right|_P & \cdots & \left.\frac{\partial \dot{\omega}_1}{\partial Y_{N_s}}\right|_P & \left.\frac{\partial \dot{\omega}_1}{\partial T}\right|_P \\ \vdots & \ddots & & & \\ \vdots & & & & \\ \vdots & & & \ddots & \\ \left.\frac{\partial \dot{\omega}_{N_s}}{\partial Y_1}\right|_P & & & & \\ \left.\frac{\partial \dot{\omega}_T}{\partial Y_1}\right|_P & \cdots & & \cdots & \left.\frac{\partial \dot{\omega}_T}{\partial T}\right|_P \end{bmatrix}$$

- `jacobian_const_v()`.
  Returns the constant-volume numerically-evaluated Jacobian of the chemical source term of the mixture:

$$J_{\dot{\omega}} = \begin{bmatrix} \left.\frac{\partial \dot{\omega}_1}{\partial Y_1}\right|_v & \left.\frac{\partial \dot{\omega}_1}{\partial Y_2}\right|_v & \cdots & \left.\frac{\partial \dot{\omega}_1}{\partial Y_{N_s}}\right|_v & \left.\frac{\partial \dot{\omega}_1}{\partial T}\right|_v \\ \vdots & \ddots & & & \\ \vdots & & & & \\ \vdots & & & \ddots & \\ \left.\frac{\partial \dot{\omega}_{N_s}}{\partial Y_1}\right|_v & & & & \\ \left.\frac{\partial \dot{\omega}_T}{\partial Y_1}\right|_v & \cdots & & \cdots & \left.\frac{\partial \dot{\omega}_T}{\partial T}\right|_v \end{bmatrix}$$

- `jac_contribution()`.
  Returns a $2\mathrm{N}_r$-long array of $[\mathrm{N}_v \times \mathrm{N}_v]$ Jacobian matrices, each one representing the contribution of reaction k to the Jacobian:

$$J_{\dot{\omega}_k} = \frac{\partial \mathcal{S}_{ik}\, r^k}{\partial Y_j} \qquad i = 1, \ldots N_s \qquad j = 1, \ldots, N_s$$

- `jacKinetic()`.
  Returns the $N_s \times N_s$ Jacobian, excluding the temperature row/column.

- `reaction_names()`.
  Returns the names of the chemical reactions (forward + backward).

3.2 `Functions.py`

3.2.1 class `CanteraCSP`

This class inherits attributes and methods from the `CanteraThermoKinetics` class, which is in turn a derived class of Cantera's `Solution` class (multilevel inheritance) and extends their capabilities by adding computational singular perturbation (CSP)-related class methods. Hence, the constructor for this class is equivalent to the constructor of Cantera's `Solution` class. The proper way to instantiate `CanteraCSP` objects is by using species and reactions defined in an input file, e.g. `hydrogen.cti`:

```python
import PyCSP.Functions as csp
gas = csp.CanteraCSP('hydrogen.cti')
```

Instance attributes:

- `jacobiantype`
  Type (*string*).
  Set/get the Jacobian type. Available choices are: `full`, `kinetic`. Defaults to `full`. Note that this choice affects the dimension of the system ($N_v = N_s$+1 or N_s), and in turn the dimension of the CSP-related attributes (eigenvalues, eigenvectors, timescales, amplitudes).

- `rtol`
  Type (*float*).
  Set/get the CSP relative tolerance for the exhausted modes calculation. Defaults to `1.0e-2`.

- `atol`
  Type (*float*).
  Set/get the CSP absolute tolerance for the exhausted modes calculation. Defaults to `1.0e-8`.

- `rhs`
  Type ([N_v] *array of float, read-only*).
  Get the chemical source term used for the CSP calculations.

- `jac`
  Type ([$N_v \times N_v$] *array of float, read-only*).
  Get the Jacobian of the chemical source term used for the CSP calculations.

- `evals`
  Type ([N_v] *array of complex, read-only*).
  Get the Jacobian eigenvalues.

- `Revec`
  Type ([$N_v \times N_v$] *array of float, read-only*).
  Get the matrix of the Jacobian right eigenvectors. The eigenvectors are assembled as row vectors, hence the i-th eigenvector can be accessed using `Revec[i]`. Note that the properly defined right eigenvectors matrix is the transpose of `Revec`.

- `Levec`
  Type ([$N_v \times N_v$] *array of float, read-only*).
  Get the matrix of the Jacobian left eigenvectors, obtained by inversion of the right eigenvectors matrix. The eigenvectors are assembled as row vectors, hence the i-th eigenvector can be accessed using `Levec[i]`.

- `f`
  Type ([N_v] *array of positive float, read-only*).
  Get the CSP mode amplitudes.

- `tau`
  Type ([N_v] *array of float, read-only*).
  Get the modes timescales.

- `nUpdates`
  Type (*int, read-only*).
  Get the number of kernel updates, starting from the instantiation of the object.

Class methods:

- `update_kernel()`.
  Updates the CSP kernel (`evals, Revec, Levec, f, tau`), based on the current thermochemical state, employing the Jacobian formulation specified by setting either `constP` or `constRho`, and the Jacobian type specified with `jacobyantype`. Does not return any output.

- `get_kernel()`.
  Returns the content of the `[evals, Revec, Levec, f]` attributes. If any attributes of the instance are changed with respect to the previous kernel calculation, the kernel is recomputed and returned.

- `calc_exhausted_modes()`.
  Computes and returns the number of exhausted modes (*int*).
  Optional keyword arguments:
  `rtol =` *float*. Relative tolerance. If not specified, instance attribute is employed.

`atol` = *float*. Absolute tolerance. If not specified, instance attribute is employed.

- `calc_TSR()`.
  Computes number of exhausted modes and Tangential Stretching Rate. Returns the Tangential Stretching Rate (*float*).
  Optional keyword arguments:

  `rtol` = *float*. Relative tolerance for the exhausted modes calculation. If not specified, instance attribute is employed.

  `atol` = *float*. Absolute tolerance for the exhausted modes calculation. If not specified, instance attribute is employed.

  `getM` = *boolean*. If `True`, the number of exhausted modes is returned as well.

- `calc_CSPindices()`.
  Computes number of exhausted modes and returns the CSP amplitude (API), timescale (TPI) participation indices, the CSP fast importance indices, slow importance indices, the species classification (fast/slow/trace): `[API, TPI, Ifast, Islow, species_type]` ([N$_v$× 2N$_r$] *array of float*,[N$_v$× 2N$_r$] *array of float*, [N$_v$× 2N$_r$] *array of float*, [N$_v$× 2N$_r$] *array of float*, [N$_v$] *array of strings*)
  Optional keyword arguments:

  `rtol` = *float*. Relative tolerance for the exhausted modes calculation.

  `atol` = *float*. Absolute tolerance for the exhausted modes calculation.

  `getM` = *boolean*. If `True`, the number of exhausted modes is returned as well.

  `getAPI` = *boolean*. If `False`, does not calculate the API.

  `getTPI` = *boolean*. If `False`, does not calculate the TPI.

  `getImpo` = *boolean*. If `False`, does not calculate the Importance indices.

  `getspeciestype` = *boolean*. If `False`, does not calculate the species classification.

- `calc_TSRindices()`.
  Computes and returns the number of exhausted modes, the Tangential Stretching Rate and the Tangential Stretching Rate amplitude (TSR-API) or timescale participation indices (TSR-TPI) ([2N$_r$]*array of float*).
  Optional keyword arguments:

  `rtol` = *float*. Relative tolerance for the exhausted modes calculation.

  `atol` = *float*. Absolute tolerance for the exhausted modes calculation.

  `getM` = *boolean*. If `True`, the number of exhausted modes is returned as well.

  `type` = *'string'*. If `amplitude`, the TSR-API are returned. If `timescale`, the TSR-TPI are returned. Defaults to TSR-API.

- `calc_extended_TSR()`.
  Computes number of extended exhausted modes (M_{ext}) and extended Tangential Stretching Rate. Returns the extended Tangential Stretching Rate. (*float*).
  Optional keyword arguments:

  `rhs_diffYT` = [N$_v$] *array of float, read-only*. Diffusion right hand side. Array containing the diffusion flux of the mass fractions of all the species and the temperature: [D_1, D_2, ..., D_N, D_T]. The unit of measure must be [$1/s$] and [K/s]. The order of the D_i values must comply with the species ordering set at the class instantiation.

  `rhs_convYT` = [N$_v$] *array of float, read-only*. Convection right hand side. Array containing the convection flux of the mass fractions of all the species and the temperature: [C_1, C_2, ..., C_N, C_T]. The unit of measure must be [$1/s$] and [K/s]. The order of the C_i values must comply with the species ordering set at the class instantiation.

  `rtol` = *float*. Relative tolerance for the exhausted modes calculation. If not specified, instance attribute is employed.

  `atol` = *float*. Absolute tolerance for the exhausted modes calculation. If not specified, instance attribute is employed.

getMext = *boolean*. If True, the extended number of exhausted modes is returned as well.

- calc_extended_TSRindices().
  Computes and returns the extended number of exhausted modes, the extended Tangential Stretching Rate and the extended Tangential Stretching Rate amplitude (TSR-API) ([$2N_r+N_v+N_v$]*array of float*).
  Optional keyword arguments:

  rtol = *float*. Relative tolerance for the exhausted modes calculation.

  atol = *float*. Absolute tolerance for the exhausted modes calculation.

  rhs_diffYT = [N_v] *array of float, read-only*. Diffusion right hand side. Array containing the diffusion flux of the mass fractions of all the species and the temperature: [D_1, D_2, ..., D_N, D_T]. The unit of measure must be [$1/s$] and [K/s]. The order of the D_i values must comply with the species ordering set at the class instantiation.

  rhs_convYT = [N_v] *array of float, read-only*. Convection right hand side. Array containing the convection flux of the mass fractions of all the species and the temperature: [C_1, C_2, ..., C_N, C_T]. The unit of measure must be [$1/s$] and [K/s]. The order of the C_i values must comply with the species ordering set at the class instantiation.

  getMext = *boolean*. If True, the number of exhausted modes is returned as well.

3.3 Solver.py

3.3.1 class CSPsolver

This class is an ODE system integrator class based on the CSP solver. The class takes as input an instance of the CanteraThermoKinetics class.

Instance attributes:

- jacobiantype
  Type (*string*).
  Set/get the Jacobian type. Available choices are: full. Defaults to full.

- csprtol
  Type (*float*).
  Set/get the CSP relative tolerance for the exhausted modes calculation. Defaults to 1.0e-2.

- cspatol
  Type (*float*).
  Set/get the CSP absolute tolerance for the exhausted modes calculation. Defaults to 1.0e-8.

- factor
  Type (*float*).
  Set/get the timestep safety factor. Defaults to 0.2

- M
  Type (*int, read-only*).
  Get the current number of exhausted modes.

- y
  Type (*array of float, read-only*).
  Get the current system state in the form [$Y_1, \ldots, Y_{N_s}$, T].

- t
  Type (*float, read-only*).
  Get the current system time.

- `dt`
  Type (*float, read-only*).
  Get the current system timestep size.

- `Qs`
  Type ([$N_s + 1 \times N_s + 1$] *float, read-only*).
  Get the current projection matrix.

- `Rc`
  Type ([$N_s + 1$] *float, read-only*).
  Get the current radical correction vector.

Class methods:

- `set_integrator()`.
  Set the integrator parameters. Does not return any output.
  Optional keyword arguments:
    `cspRtol =` *float*. Relative tolerance for the exhausted modes calculation.
    `cspAtol =` *float*. Absolute tolerance for the exhausted modes calculation.
    `factor =` *float*. Timestep safety factor.
    `jacobinatype =` *'string'*. Jacobian type.

- `set_initial_value(y0,t0)`.
  Set the integrator initial condition. Does not return any output.
  Mandatory positional arguments:
    `y0` ([$N_s + 1$]*array of float*). Initial state in the form [$Y_1, \ldots, Y_{N_s}$, T].
    `t0` (*float*). Initial time.

- `integrate()`.
  Advance in time by one timestep. The timestep is calculated according to the $M + 1$-th eigenvalue, with a safety factor of `factor`.

3.4 `Simplify.py`

3.4.1 class `CSPsimplify`

This class is simplified kinetic mechanism generation class based on CSP. The class takes as input an instance of the `CanteraThermoKinetics` class and a dataset. The instance should be initialized with the detailed mechanism. The dataset must be a 2D array, each row containing species mass fractions (in the order of the detailed mechanism), temperature and pressure: `dataset=[`Y_1`, `Y_2`, ..., `Y_N`, T, P ]`.

Instance attributes:

- `csprtol`
  Type (*float*).
  Set/get the CSP relative tolerance for the exhausted modes calculation. Defaults to `1.0e-2`.

- `cspatol`
  Type (*float*).
  Set/get the CSP absolute tolerance for the exhausted modes calculation. Defaults to `1.0e-8`.

- `scaled`
  Type (*boolean*).
  Set/get the *scaled* option for the importance indexes. When active, indexes are normalized with respect to their maximum value. Defaults to `True`.

- `targetset`
  Type (*string, dictionary*).
  Set/Get the list of target species. Example: `{ 'CH4', 'H2O', 'CO2' }`.

- `TSRtargetset`
  Type (*boolean*).
  Set/get the *TSRtargetset* option for the target set. When active, the user-defined `targetset`, which may also be empty, is augmented with the local TSR-relevant species (each state will have its own set of targets based on TSR-API indexes). Defaults to `False`.

- `tsrtol`
  Type (*float*).
  Set/get the TSR-API tolerance for including species in the `TSRtargetset`. A species is included if it is involved in a reaction which is part of the subset of reactions contributing to the `tsrtol`·100% of the cumulative participation to the TSR. The higher the value of `tsrtol`, the more species will be included. Defaults to `0.5`.

- `y`
  Type (*problemtype*).
  Type (*string*).
  Set/Get the problem type, among `constP` and `constRho`.

Class methods:

- `dataset_info()`.
  Outputs info on `dataset`.

- `process_dataset()`.
  Computes the fast/slow importance indexes and the species fast/slow classification for each state in `dataset`.

- `simplify_mechanism(threshold)`.
  Returns the sets of retained species and reactions, based on the value of `threshold`, which must be between 0 and 1.

4 Examples

Several examples are available in the `examples` folder to exploit the functionalities related to:

- exhausted modes (M)
- tangential stretching rate (TSR)
- CSP and TSR indices (importance indices, amplitude and timescale participation indices, TSR amplitude/timescale participation indices)
- CSP solver
- CSP simplify

The reader is referred to [4, 1, 2, 3] for the theoretical details of the methods employed.

4.1 CSP analysis

It is hereby reported an example of how to analyze with PyCSP a set of thermochemical data stored in an external file called `testdata.dat`, containing N lines, each one being a full thermochemical state of the kind [T, P, Y_1, ..., Y_{N_s}]. The kinetic mechanism is `hydrogen.cti`, containing 9 species and 19 reversible reactions. The dataset is the time evolution of a stoichiometric H_2/Air constant pressure homogeneous reactor, with initial temperature of 1000 K and pressure of one atmosphere.

```python
import numpy as np
import PyCSP.Functions as csp
import PyCSP.utils as utils
import matplotlib.pyplot as plt

#import dataset
data = np.loadtxt('testdata.dat')
time = data[:,0]
Temp = data[:,1]
Pressure = data[:,2]
Y =  data[:,3:]

#create an instance of the CanteraCSP class
gas = csp.CanteraCSP('hydrogen.cti')

#set jacobiantype
gas.jacobiantype = 'full'

#set CSP tolerances
gas.rtol = 1.0e-2
gas.atol = 1.0e-8

evals = []
Revec = []
Levec = []
fvec = []
M = []

for step in range(len(time)):
    gas.constP = Pressure[step]
    state = np.append(Y[step],Temp[step])
    gas.set_stateYT(state)
    lam,R,L,f = gas.get_kernel()
    NofDM = gas.calc_exhausted_modes()

    evals.append(lam)
    Revec.append(R)
    Levec.append(L)
    fvec.append(f)
    M.append(NofDM)

evals = np.array(evals)
Revec = np.array(Revec)
Levec = np.array(Levec)
fvec = np.array(fvec)
M = np.array(M)

#plot eigenvalues and lambda_M+1
evalM = utils.select_eval(evals,M)
logevals = np.clip(np.log10(1.0+np.abs(evals)),0,100)*np.sign(evals.real)
logevalM = np.clip(np.log10(1.0+np.abs(evalM)),0,100)*np.sign(evalM.real)
fig, ax = plt.subplots(figsize=(6,4))
for idx in range(evals.shape[1]):
    ax.plot(states.t, logevals[:,idx], color='black', marker='.', markersize = 5,
    linestyle = 'None')
ax.plot(states.t, logevalM, color='orange', marker='.', markersize = 4,linestyle =
    'None', label='lam(M+1) rtol e-2; atol e-8')
ax.set_xlabel('time (s)')
ax.set_ylabel('evals')
ax.set_ylim([-9, 6])
ax.set_xlim([0., 0.001])
ax.grid(False)
ax.legend()
plt.show()

#plot exhausted modes
print('plotting exhausted modes...')
fig, ax = plt.subplots(figsize=(6,4))
#ax.plot(states.t, M, color='black')
ax.plot(M, color='orange', label='rtol e-2; atol e-8')
```

```
#ax.set_xlabel('time (s)')
ax.set_xlabel('# timestep')
ax.set_ylabel('M')
ax.set_ylim([0,10])
#ax.set_xlim([0., 0.001])
ax.grid(False)
ax.legend()
plt.show()
```

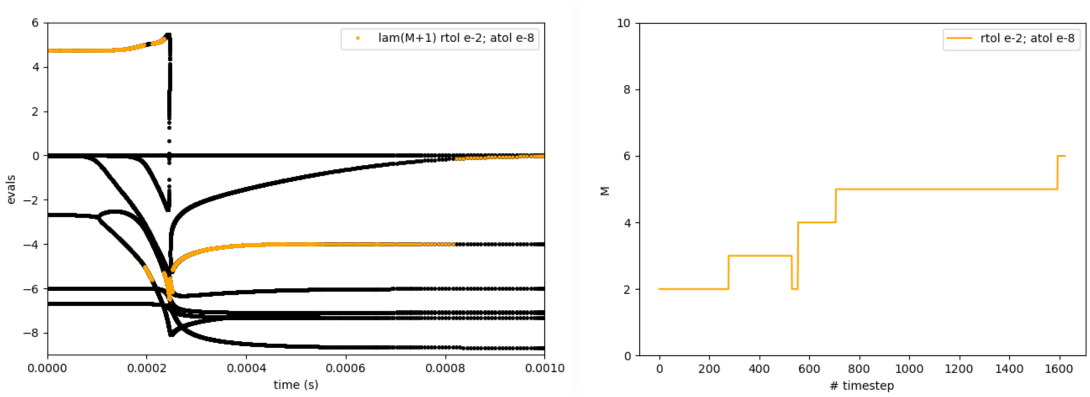

Figure 2: Example outcomes. Left: eigenvalues (black) and M+1-th eigenvalue (orange). Right: number of exhausted modes.

4.2 CSP solver

This example shows how to integrate a constant volume homogeneous reactor with the CSPsolver class.

```
import PyCSP.Functions as cspF
import PyCSP.Solver as cspS

#create an instance of the CanteraCSP class
gas = cspF.CanteraCSP('hydrogen.cti')

#set the gas state using Cantera's methods ( as an alternative to set_stateYT() )
T = 1000
P = ct.one_atm
gas.TP = T, P
gas.set_equivalence_ratio(1.0, 'H2', 'O2:1, N2:3.76')

#push density
rho = gas.density
gas.constRho = rho

#initial condition
y0 = np.hstack((gas.Y,gas.T))
t0 = 0.0

t_end = 1e-2

#create an instance of CSPsolver
solver = cspS.CSPsolver(gas)
solver.set_integrator(cspRtol=1e-2,cspAtol=1e-8,factor=0.2,jacobiantype='full')
solver.set_initial_value(y0,t0)

states = ct.SolutionArray(gas, 1, extra={'t': [0.0], 'M': 0})

#integrate ODE with CSP solver
while solver.t < t_end:
    solver.integrate()
    states.append(gas.state, t=solver.t, M=solver.M)
```

```python
    print('%10.3e %10.3f %10.3e %10.3e %10.3e %2i' % (solver.t, solver.y[-1],
    solver.dt, gas.P, gas.density, solver.M))
```

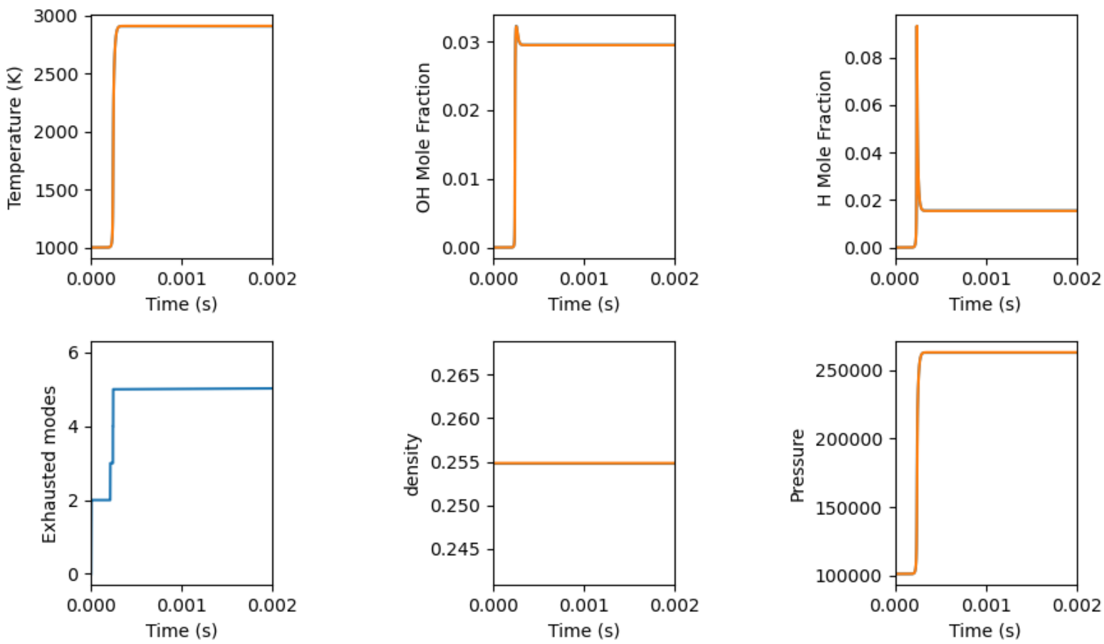

Figure 3: Example outcomes.

4.3 CSP simplify

This example shows how to generate a set of simplified kinetic mechanisms, based on a dataset produced with a detailed mechanism, with the CSPsimplify class.

```python
#-------CREATE DATASET---------
#create gas from original mechanism file hydrogen.cti
dtl_mech = csp.CanteraCSP('gri30.cti')

#set the gas state
T = 1000
P = ct.one_atm
dtl_mech.TP = T, P
dtl_mech.constP = P
dtl_mech.set_equivalence_ratio(1.0, 'CH4', 'O2:1, N2:3.76')


#integrate ODE with detailed mech
r = ct.IdealGasConstPressureReactor(dtl_mech)
sim = ct.ReactorNet([r])
states = ct.SolutionArray(dtl_mech, 1, extra={'t': [0.0], 'rhsT': [0.0]})

sim.set_initial_time(0.0)
while sim.time < 1000:
    sim.step()
    states.append(r.thermo.state, t=sim.time, rhsT=dtl_mech.source[-1])

dataset = np.concatenate((states.Y,states.T[:,np.newaxis],states.P[:,np.newaxis]),
    axis=1)
print('Dataset created, start processing...')

#coarsen dataset (to speed up)
dataset = dataset[::64]

```

```python
#-------SIMPLIFY---------
#init simplifier
simplifier = simp.CSPsimplify(dtl_mech,dataset)

#simplifier settings
simplifier.targetset = {'CH4','O2','N2','HCO','H2O','CO2'}
simplifier.problemtype = 'constP'
simplifier.scaled = True
simplifier.csprtol = 1.0e-2
simplifier.cspatol = 1.0e-8

#process dataset
simplifier.process_dataset()

print('Done processing')

print('Start pruning...may take a while')
#loop over thresholds
thr = np.arange(0.01, 1, 0.01)

simp_mech = []
prev_species = dtl_mech.species_names
for i in range(len(thr)):
    species, reactions = simplifier.simplify_mechanism(thr[i])
    simp = csp.CanteraCSP(thermo='IdealGas', kinetics='GasKinetics', species=species, reactions=reactions)
    if simp.species_names != prev_species:  #append only if different from previous one
        simp_mech.append(simp)
        prev_species = simp.species_names

nmech = len(simp_mech)
print('%i mechanisms found' % (nmech))

#-------RE-RUN---------
# Re-run the ignition problem with the simplified mechanisms
all_states_simp = []
for i in range(nmech):
    simp_mech[i].TP = T,P
    simp_mech[i].constP = P
    simp_mech[i].set_equivalence_ratio(1.0, 'CH4', 'O2:1, N2:3.76')
    r = ct.IdealGasConstPressureReactor(simp_mech[i])
    sim = ct.ReactorNet([r])
    states_simp = ct.SolutionArray(simp_mech[i], 1, extra={'t': [0.0], 'rhsT': [0.0]})

    sim.set_initial_time(0.0)
    while sim.time < 1000:
        sim.step()
        states_simp.append(r.thermo.state, t=sim.time, rhsT=simp_mech[i].source[-1])
    all_states_simp.append(states_simp)

#-------ERRORS---------
dtl_idt = states.t[states.rhsT.argmax()]
simp_idt = [all_states_simp[i].t[all_states_simp[i].rhsT.argmax()] for i in range(nmech)]
err_idt = abs((simp_idt-dtl_idt)/dtl_idt)*100

fig, ax = plt.subplots(figsize=(6,4))
ax.plot([simp_mech[i].n_species for i in range(nmech)], err_idt, color='red', marker='.')
ax.set_xlabel('# of species')
ax.set_ylabel('ignition delay time relative Error [%]')
ax.set_yscale('log')
ax.set_ylim([1e-3, 1e2])
plt.savefig('ign_delay_error.png', dpi=800, transparent=False)
```

Figure 4 shows the percent relative error on the ignition delay time versus the number of retained species. A significant improvement is given by the TSR kernel set, which replaces the

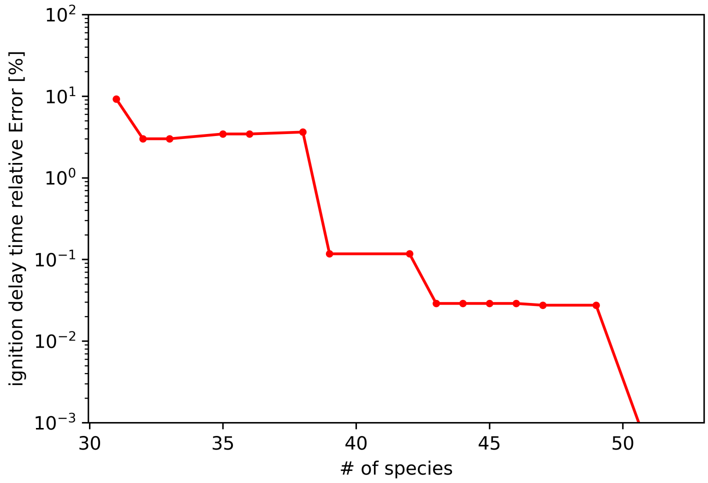

Figure 4: Example outcomes of the mechanism simplification.

user-specified target species set (see [1]). With the following settings:

```
simplifier.TSRtargetset = True
simplifier.TSRtol = 0.5
simplifier.targetset = {'N2'}
```

the previous example delivers the ignition delay relative errors shown in Fig.5. The manual addition of N_2 to the TSR kernel set is needed to force the presence of N_2 in the skeletal mechanisms.

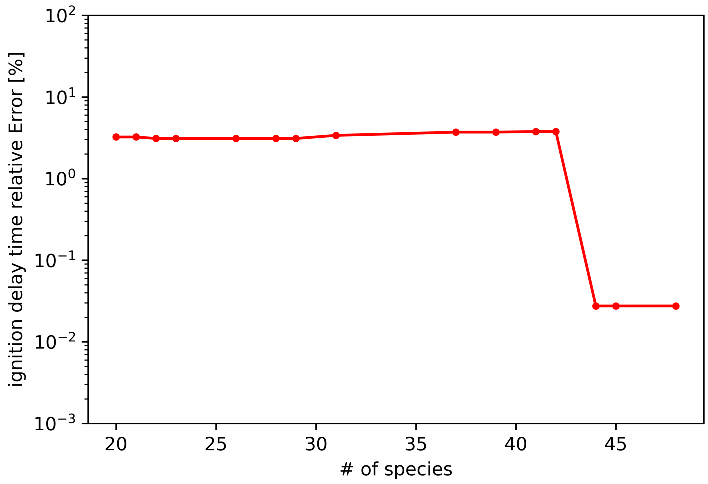

Figure 5: Example outcomes of the mechanism simplification with TSR kernel set.

4.4 TSR analysis

4.4.1 Chemical TSR

This example shows how to compute the TSR of data points produced in a homogeneous reactor auto-ignition.

```python
#create gas from original mechanism file hydrogen.cti
gas = csp.CanteraCSP('hydrogen.cti')

#set the gas state
T = 1000
P = ct.one_atm
gas.TP = T, P
gas.set_equivalence_ratio(1.0, 'H2', 'O2:1, N2:3.76')
gas.constP = P
gas.jacobiantype = 'full'

#integrate ODE
r = ct.IdealGasConstPressureReactor(gas)
sim = ct.ReactorNet([r])
time = 0.0
states = ct.SolutionArray(gas, extra=['t'])

evals = []
Revec = []
Levec = []
fvec = []
M = []
tsr = []

sim.set_initial_time(0.0)
while sim.time < 10:
    sim.step()
    states.append(r.thermo.state, t=sim.time)
    print('%10.3e %10.3f %10.3f %14.6e' % (sim.time, r.T, r.thermo.P, r.thermo.u))
    lam,R,L,f = gas.get_kernel()
    omegatau, NofDM = gas.calc_TSR(getM=True,rtol=1.0e-3,atol=1.0e-10)
    evals.append(lam)
    Revec.append(R)
    Levec.append(L)
    fvec.append(f)
    M.append(NofDM)
    tsr.append(omegatau)


evals = np.array(evals)
Revec = np.array(Revec)
Levec = np.array(Levec)
fvec = np.array(fvec)
M = np.array(M)
tsr = np.array(tsr)

#plot eigenvalues and lambda_M+1
evalM = utils.select_eval(evals,M)
logevals = np.clip(np.log10(1.0+np.abs(evals.real)),0,100)*np.sign(evals.real)
logevalM = np.clip(np.log10(1.0+np.abs(evalM.real)),0,100)*np.sign(evalM.real)
logTSR = np.clip(np.log10(1.0+np.abs(tsr)),0,100)*np.sign(tsr)
print('plotting eigenvalues...')
fig, ax = plt.subplots(figsize=(6,4))
for idx in range(evals.shape[1]):
    #color = next(ax._get_lines.prop_cycler)['color']
    ax.plot(states.t, logevals[:,idx], color='black', marker='.', markersize = 5, linestyle = 'None')
ax.plot(states.t, logevalM, color='orange', marker='.', markersize = 4,linestyle = 'None', label='lam(M+1)')
ax.plot(states.t, logTSR, color='green', marker='.', markersize = 2,linestyle = 'None', label='TSR')
ax.set_xlabel('time (s)')
ax.set_ylabel('evals')
ax.set_ylim([-9, 6])
ax.set_xlim([0., 0.001])
ax.grid(False)
ax.legend()
plt.show(block = False)
plt.savefig('TSR.png', dpi=500, transparent=False)
```

Figure 6 shows the TSR compared to the system's eigenvalues.

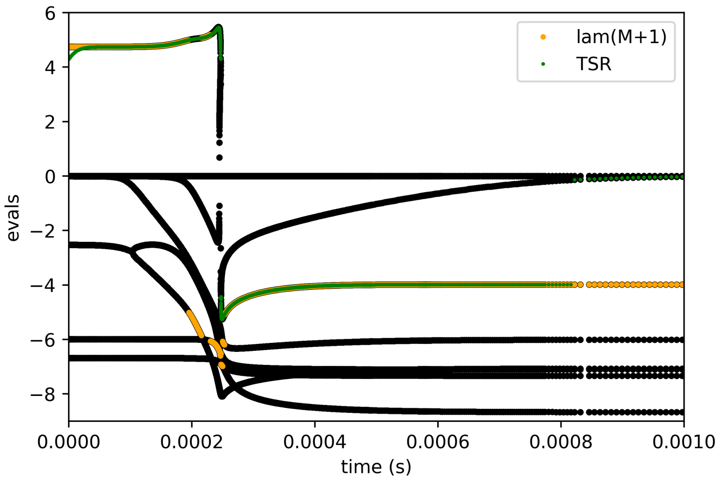

Figure 6: Example outcomes of the TSR analysis.

4.4.2 Extended TSR

It is hereby reported an example of how to make an extended TSR analysis of pre-computed thermochemical data stored in external files called `flamelet_state.dat`, containing N lines, each one being a full thermochemical state of the kind [`time`, `counter`, `Z`, `P`, `T`, `Y`$_1$, ..., `Y`$_{N_s}$], and `flamelet_rhsDiff.dat`, containing the diffusive fluxes in the form [`time`, `counter`, `Z`, `diff-T`, `diff-Y`$_1$, ..., `diff-Y`$_{N_s}$]. This analysis can be performed supplying either the diffusive fluxes, the convective fluxes, or both simultaneously. The kinetic mechanism is `chaos12.cti`, containing 12 species and 33 reversible reactions. The dataset and the results are described in [4].

```
#create gas from original mechanism file hydrogen.cti
gas = csp.CanteraCSP('chaos12.cti')

#-------IMPORT DATA---------
#read data from file
state = np.loadtxt('flamelet_state.dat')
time = state[:,0]
counter = state[:,1]
zeta = state[:,2]
Pressure = state[:,3]
Temp = state[:,4]
Y =  state[:,5:]


rhsD = np.loadtxt('flamelet_rhsDiff.dat')
diffTemp = rhsD[:,3]
diffY =  rhsD[:,4:]

#set jacobiantype
gas.jacobiantype = 'full'

evals = []
M = []
tsr = []
Mext = []
tsrext = []


print('Analyzing %i data points, may take a while...' %len(state))

for step in range(time.shape[0]):
#for step in range(begin,end):
    gas.constP = Pressure[step]
```

```
    stateYT = np.append(Y[step],Temp[step])
    rhsdiffYT = np.append(diffY[step],diffTemp[step])
    gas.set_stateYT(stateYT)
    gas.update_kernel()
    omegatau, NofDM = gas.calc_TSR(getM=True)
    omegatauext, NofDMext = gas.calc_extended_TSR(getMext=True,diff=rhsdiffYT)

    evals.append(lam)
    M.append(NofDM)
    tsr.append(omegatau)
    Mext.append(NofDMext)
    tsrext.append(omegatauext)

evals = np.array(evals)
Mext = np.array(Mext)
tsrext = np.array(tsrext)
M = np.array(M)
tsr = np.array(tsr)
```

Figure 7 shows the contour plots of the chemical and extended TSR.

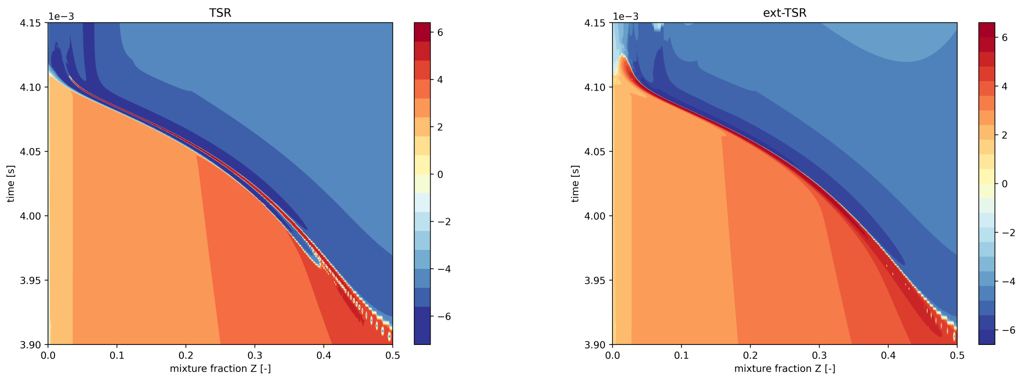

Figure 7: Example outcomes. Left: contour plot of the chemical TSR. Right: contour plot of the extended TSR.

The amplitude participation indices (API) to the extended TSR can be obtained as:

```
omegatauext, api = gas.calc_extended_TSRindices(diff=rhsdiffYT)
```

Figure 8 shows the TSR-API in a slice of the field.

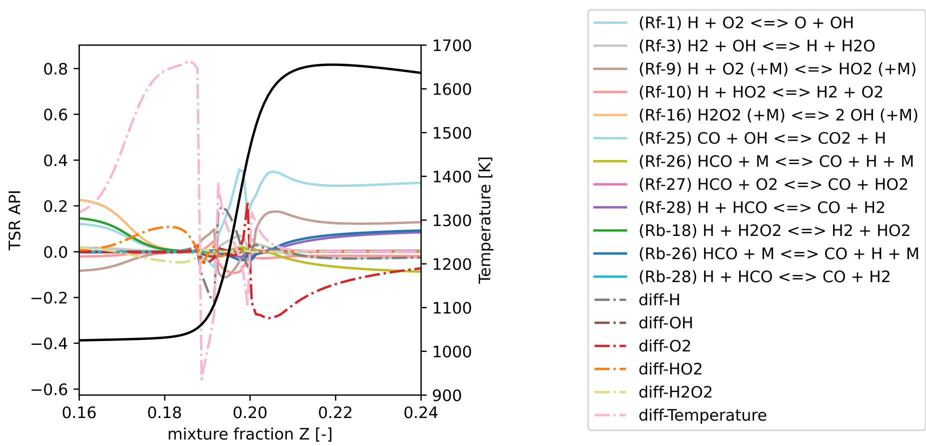

Figure 8: Example outcomes. Amplitude Participation Indices to the extended TSR.

References

[1] Riccardo Malpica Galassi, Pietro P. Ciottoli, Subram Mani Sarathy, Hong G. Im, Samuel Paolucci, and Mauro Valorani. Automated chemical kinetic mechanism simplification with minimal user expertise. *Combustion and Flame*, 197:439–448, 2018.

[2] M. Valorani, F. Creta, P. P. Ciottoli, R. Malpica Galassi, D. A. Goussis, H. N. Najm, S. Paolucci, H. G. Im, E.-A. Tingas, D. M. Manias, A. Parente, Z. Li, and T. Grenga. *Computational Singular Perturbation Method and Tangential Stretching Rate Analysis of Large Scale Simulations of Reactive Flows: Feature Tracking, Time Scale Characterization, and Cause/-Effect Identification. Part 1, Basic Concepts*, pages 43–64. Springer International Publishing, Cham, 2020.

[3] M. Valorani, F. Creta, P. P. Ciottoli, R. Malpica Galassi, D. A. Goussis, H. N. Najm, S. Paolucci, H. G. Im, E.-A. Tingas, D. M. Manias, A. Parente, Z. Li, and T. Grenga. *Computational Singular Perturbation Method and Tangential Stretching Rate Analysis of Large Scale Simulations of Reactive Flows: Feature Tracking, Time Scale Characterization, and Cause/Effect Identification. Part 2, Analyses of Ignition Systems, Laminar and Turbulent Flames*, pages 65–88. Springer International Publishing, Cham, 2020.

[4] Mauro Valorani, Pietro Paolo Ciottoli, and Riccardo Malpica Galassi. Tangential stretching rate (TSR) analysis of non premixed reactive flows. *Proceedings of the Combustion Institute*, 36(1):1357–1367, 2017.