PyChemkin

ANSYS, Inc. <ansys.support@ansys.com>

Dec 11, 2025

CONTENTS

PyChemkin provides Pythonic access to the Ansys Chemkin API for CFD (computational fluid dynamics)models. It facilitates programmatic customization of Chemkin simulation workflows within the Python ecosystem and permits access to Chemkin property and rate utilities as well as selected reactor models. With PyChemkin, you can perform these tasks:

- Process Chemkin-compatible gas-phase mechanisms.
- Evaluate species and mixture thermodynamic and transport properties.
- Compute reaction rate of progress and species rate of production.
- Combine gas mixtures isothermally or adiabatically.
- Find the equilibrium state of a gas mixture.
- Run gas-phase batch, plug-flow, and PSR (perfectly-stirred reactor) models.
- Calculate the laminar flame speed of a combustible mixture.
- Create and solve steady-state reactor networks.

Getting started Learn how to install PyChemkin.

Getting started User guide Understand key concepts for using PyChemkin. Also learn how to set up and run a basic reactor model and where you can find supporting information.

User guide API reference Understand how to use Python to interact programmatically with PyChemkin.

API reference Examples Explore examples that show how to use PyChemkin utilities and reactor models to perform different types of simulations.

Examples Contribute Learn how to contribute to the PyChemkin codebase or documentation.

Contribute

CHAPTER

ONE

INSTALL PREREQUISITES

- Ansys Chemkin 2025 R2 or later with a valid license
- Python 3.9 or later
- NumPy 1.14.0 or later
- PyYAML 6.0 or later
- Matplotlib to run examples

Note

Using the latest Ansys Chemkin version is highly recommended.

CHAPTER

TWO

INSTALL PYCHEMKIN

1. Install PyChemkin.

   Download the `ansys-chemkin` package from the PyAnsys GitHub repository. Build the wheelhouse locally using Flit:

   ```
   python -m build
   ```

   Install the package using pip:

   ```
   pip install dist\ansys_chemkin-*.whl
   ```

2. Verify the installation.

   Open the Python interpreter from the Windows command prompt and import the `ansys-chemkin` package:

   ```
   >>> import ansys.chemkin
   ```

   If PyChemkin is installed correctly, Python displays a statement like this:

   ```
   Chemkin version number = xxx
   ```

   PyChemkin is probably not installed locally if Python displays nothing:

   ```
   >>>
   ```

   If Python displays the following statement, update the local Ansys Chemkin installation to 2025 R2 or later:

   ```
   PyChemkin does not support Chemkin versions older than 2025 R2
   ```

Note

You must have a valid Ansys license to run PyChemkin after installation.

CHAPTER

THREE

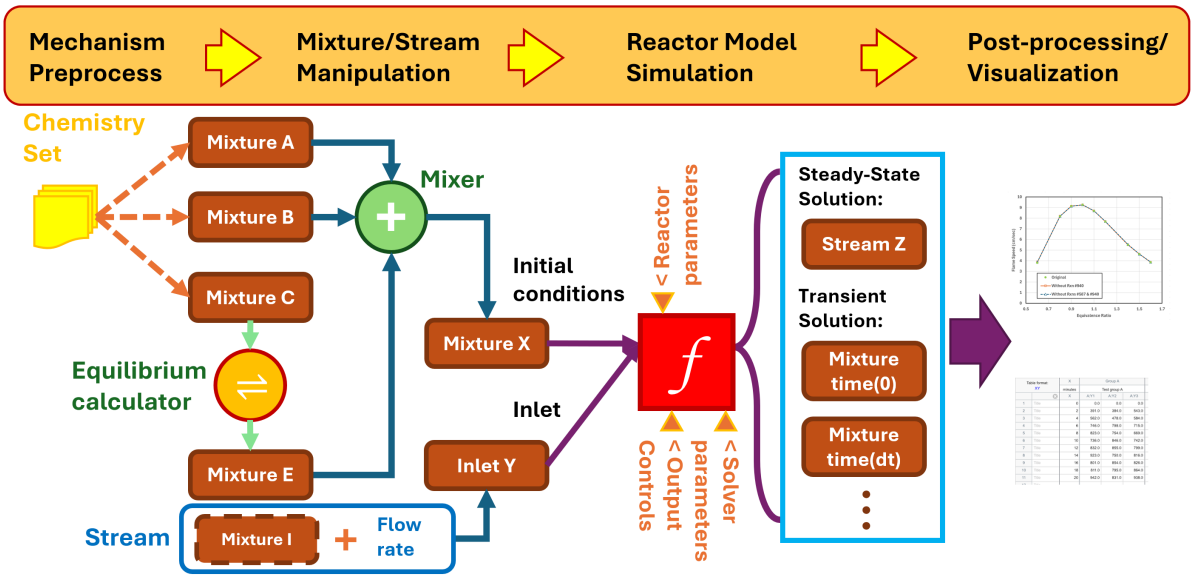
USER GUIDE

3.1 Introduction

PyChemkin inherits all Ansys Chemkin functionalities, providing Pythonic interfaces to Chemkin utilities and reactor models. In addition, PyChemkin introduces a hierarchy of four key objects to enhance user experiences within the Python framework:

- **Chemistry set:** A collection of utilities that handle chemistry and species properties.
- **Mixture:** A basic element representing a gas mixture that can be manipulated and transformed.
- **Stream:** A mixture object with an additional property of mass flow rate.
- **Reactor:** An instance of a Chemkin reactor model that transforms the initial mixture into another mixture.

The following figure shows PyChemkin's mixture-centric concept. It illustrates the basic types of operations applicable to a mixture, which inherits all properties of a chemistry set. A mixture can be combined with a constraint, equilibrium (with constraints), or a process (by a reactor model).



3.2 Workflow for a basic reactor model

The workflow for setting up and running a basic reactor model in PyChemkin is the same as in the Ansys Chemkin GUI:

1. Create a chemistry set, which includes the mechanism, thermodynamic data, and/or transport data.

2. Preprocess the chemistry set.
3. Create a mixture or a stream based on the chemistry set.
4. Specify mixture or stream properties, such as temperature, pressure, volume (optional), and species composition.
5. Instantiate the reactor using the created mixture.
6. Set up the simulation:
    - Specify reactor properties not provided by the initial mixture or stream, such as heat loss rate to the surroundings and end time.
    - Specify solver parameters, such as tolerances and solver timestep size.
    - Specify saving controls, such as the solution saving interval and adaptive solution saving.
7. Run the simulation.
8. Process the solution.
9. Customize the output if necessary and generate plots.

3.3 Code example

This code example computes the density of a mixture named `air`:

```
import os

# import PyChemkin
import ansys.chemkin as chemkin

# create a chemistry set for the GRI 3.0 mechanism in the data directory
mech_dir = os.path.join(chemkin.ansys_dir, "reaction", "data")
# set up mechanism file names
mech_file = os.path.join(mech_dir, "grimech30_chem.inp")
therm_file = os.path.join(mech_dir, "grimech30_thermo.dat")
tran_file = os.path.join(mech_dir, "grimech30_transport.dat")
# instantiate a chemistry set named 'GasMech'
GasMech = chemkin.Chemistry(chem=mech_file, therm=therm_file,  tran=tran_file,  label=
↪'GRI 3.0')
# preprocess the chemistry set
status = GasMech.preprocess()
# check preprocess status
if status != 0:
    # failed
    print(f'Preprocessing: Error encountered. Code = {status:d}.')
    print(f'See the summary file {GasMech.summaryfile} for details.')
    exit()
# create a mixture named 'air' based on the 'GasMech' chemistry set
air = chemkin.Mixture(GasMech)
# set 'air' condition
# mixture pressure in [dynes/cm2]
air.pressure = 1.0 * chemkin.Patm
# mixture temperature in [K]
air.temperature = 300.0
# mixture composition in mole fractions
air.X = [('O2', 0.21), ('N2', 0.79)]
```

(continues on next page)

(continued from previous page)

```
#
print(f"Pressure    = {air.pressure/chemkin.Patm} [atm].")
print(f"Temperature = {air.temperature} [K].")
# print the 'air' composition in mass fractions
air.list_composition(mode='mass')
# get 'air' mixture density [g/cm3]
print(f"Mixture density = {air.RHO} [g/cm3].")
```

For more examples, see *Examples*.

3.4 Support

3.4.1 Pythonic methods

To get help within PyChemkin, use these pythonic methods:

- `ansys.chemkin.help()`: Get general information about a topic, such as `ignition`.
- `ansys.chemkin.keywordhints()`: Get the description and syntax of a reactor keyword.
- `ansys.chemkin.phrasehints()`: Get a list of reactor keywords related to a phrase.

3.4.2 Product training and documentation

For product training and documentation, see these Ansys resources:

- Ansys Learning Hub
- Chemkin product documentation

CHAPTER

FOUR

EXAMPLES

This section shows how to use PyChemkin utilities and reactor models to perform different types of simulations. Early examples demonstrate how to perform essential operations, such as loading and preprocessing the chemistry set. Later examples explain how to tackle more complex simulations, such as running an ignition delay parameter study. Thus, you should go through the examples in the order listed.

CHAPTER

FIVE

CHEMISTRY

PyChemkin examples to demonstrate basic operations for instantiating *Chemistry Set* from the gas-phase mechanism and the thermodynamic/transport data files.

The examples walk through the steps about how to pre-process the *Chemistry Set*, retrieve mechanism information and the species properties, as well as how to handle multiple *Chemistry Sets* in a **Python Chemkin** project.

CHAPTER

SIX

MIXTURE

Mixture is the "core" token in **PyChemkin**. It represents a gas-phase mixture by storing its pressure, temperature, and species composition. *Stream* is an extended "Class" of *Mixture* with the additional attribute of "mass/volumetric flow rate". *Mixture* and *Stream* include utilities that can be used to evaluate the mixture properties (density, mixture enthalpy, mixture viscosity, . . .) and the reaction rates. There are also methods to manipulate the *Mixture*/*Stream* such as merging two mixtures adiabatically.

The examples demonstrate how to instantiate the *Mixture*/*Stream* objects, how to extract information and properties of a *Mixture*/*Stream*, and how to manipulate *Mixture*/*Stream* objects.

CHAPTER

SEVEN

BATCH REACTOR

The *batch reactor* is an idealized 0-D transient closed reactor model in which the gas inside the reactor is assumed to be homogeneous. When the pressure of the *batch reactor* is constrained, the reactor volume would vary to conserve the total gas mass. Likewise, the reactor pressure might change when the reactor volume is "given". The gas temperature can be "given" by piecewise linear profile data or solved by the energy equation.

The examples consist of projects that utilize the *batch reactor* model to perform simulations of various complexities, from tracking the evolution of the mixture properties to the A-factor sensitivity analysis.

CHAPTER

EIGHT

0-D ENGINE MODELS

PyChemkin offers two types of 0-D engine models: the *Homogeneous Charged Compression Ignition (HCCI)* engine model and the *Spark Ignition (SI)* engine model. These 0-D engine models simulate the combustion process (or the lack of, in case of a misfire) inside an engine cylinder between the *Intake Valve Close (IVC)* and the *Exhaust Valve Open (EVO)* when the cylinder is considered as a closed system. The Chemkin *Theory* manual has detailed descriptions of these 0-D engine models.

The examples show the steps of setting up simulations with different *Chemkin* engine models.

CHAPTER

NINE

PERFECTLY STIRRED REACTOR

The *perfectly Stirred reactor (PSR)* model is a 0-D steady-state reactor. This ideal reactor model assumes the gas mixture inside is uniform and the properties of the outlet stream are exactly the same as the gas properties inside the reactor. A PSR can have either its pressure or its volume "specified", and the reactor temperature can either be solved from the energy equation or be "given" as an input parameter.

The examples will show the procedures of setting up single PSR simulations with different constraints and with multiple inlet streams.

CHAPTER

TEN

PLUG-FLOW REACTOR

Plug-Flow Reactor (PFR) model is an idealized 1-D steady-state flow reactor model with single inlet stream. The reactor pressure is assumed to be given, and the mass (flow rate) conservation is maintained by the velocity change. The gas temperature along the reactor can be either specified or solved by the energy equation.

CHAPTER

ELEVEN

PSR NETWORK

PyChemkin can be used to create an *Equivalence Reactor Network (ERN)* as a reduced-order (in geometry) model for complex combustion applications such as gas turbine combustor. Each reactor in the network represents a sub-region (zone) in the actual equipment. Normally these zones are created according to specific criteria such as temperature, equivalence ratio, or location; and the gas flow (mean, turbulent, and diffusive) between the zones becomes the internal streams between the reactors.

There are several ways to build an *ERN* in **PyChemkin**. The first method is to create and solve the reactors one-by-one starting from the most upstream (first) reactor. Adjust/manipulate the external and outlet streams from connected reactors (that is, solutions from the reactors) using the *Mixture*/*Stream* utilities and apply the desired stream as the inlet for the downstream reactors. The second method takes advantage of the *hybridreactornetwork* "model" of **PyChemkin**. Simply defined the reactors with the associated external inlets and add the reactors to the *hybridreactornetwork*. If there are *recycling* streams from the downstream reactor to the upstream ones, the *hybridreactornetwork* will solve the entire *ERN* iteratively. In this case, a *tear point* must be explicitly specified. The last method, the coupled method, is to create the network and its connectivity first, and the entire *ERN* will be solved in a coupled manner. This method is the default method in Chemkin GUI and is available as the *PSRCluster* "model" in **PyChemkin**.

Chemkin *ERN* has a few limitations:

- The first reactor of the reactor network must have at least one external inlet.
- when using the *coupled* model, the entire reactor network can have only one outlet to the surroundings, and the outlet must be attached to the last reactor in the network.
- PSR is the only reactor model allowed in the *PSRCluster* and the *hybridreactornetwork* reactor networks.

The *PSRChain_xxx* examples show the two methods to build and run a series of linked PSRs (no stream recycling) can be modeled in **PyChemkin**. The *PSRnetwork* example goes over the steps of creating and running an *ERN* with recycling streams by using the *hybridreactornetwork* method. The *PSRnetwork_coupled* example solves the same *ERN* as the *PSRnetwork* example but utilizes the *coupled* method. Because the sole outlet from the *coupled ERN* must be attached to the last reactor, the order of the downstream zones is different from the *PSRnetwork* example.

CHAPTER

TWELVE

PREMIXED FLAME MODELS

The *Premixed Flame* model is a 1-D steady-state application, and it contains two sub-models: *"flame speed calculator"* (or the freely propagating flame model) and *"flat flame"* model. The *flame speed calculator* is commonly used to calculate the laminar flame speed of a premixed fuel-oxidizer mixture. The predicted/computed flame speeds can be used to validate a combustion mechanism or can serve as a prre-processor to construct a "flame speed table" or a "combustion progress table" for other applications. The *flat flame* burner-stablized model is primarily used to study the flame chemistry in conjunction with the flat flame experiments.

The examples show different ways to perform the "premixed flame" calculations.

12.1 Chemistry

PyChemkin examples to demonstrate basic operations for instantiating *Chemistry Set* from the gas-phase mechanism and the thermodynamic/transport data files.

The examples walk through the steps about how to pre-process the *Chemistry Set*, retrieve mechanism information and the species properties, as well as how to handle multiple *Chemistry Sets* in a **Python Chemkin** project.

12.1.1 Preprocess a gas-phase mechanism

Before you can run a Chemkin simulation, you must load and preprocess the reaction mechanism data. The mechanism data consists of a reaction mechanism input file, a thermodynamic data file, and an optional transport data file.

This example shows how to instantiate and preprocess a chemistry set, which is always the first task in running any PyChemkin simulation.

PyChemkin allows several chemistry sets to coexist in the same Python project. However, only one chemistry set can be active at a time.

Import PyChemkin packages and start the logger

```
import os

# import PyChemkin packages
import ansys.chemkin as ck
from ansys.chemkin import Color
from ansys.chemkin.logger import logger

# check the working directory
current_dir = os.getcwd()
```

(continues on next page)

(continued from previous page)

```
logger.debug("working directory: " + current_dir)

# set PyChemkin verbose mode
ck.set_verbose(True)
```

Create a chemistry set

The first mechanism to load is the GRI 3.0 mechanism for methane combustion. This mechanism and its associated data files come with the standard Ansys Chemkin installation in the /reaction/data directory.

```
# set mechanism directory (the default Chemkin mechanism data directory)
data_dir = os.path.join(ck.ansys_dir, "reaction", "data")
mechanism_dir = data_dir

# set mechanism input files
# including the full file path is recommended
# the gas-phase reaction mechanism file (GRI 3.0)
chemfile = os.path.join(mechanism_dir, "grimech30_chem.inp")
# the thermodynamic data file
thermfile = os.path.join(mechanism_dir, "grimech30_thermo.dat")
# the transport data file
tranfile = os.path.join(mechanism_dir, "grimech30_transport.dat")

# create a chemistry set based on the GRI 3.0 methane combustion mechanism
MyGasMech = ck.Chemistry(chem=chemfile, therm=thermfile, tran=tranfile, label="GRI 3.0")
```

Preprocess the chemistry set

```
# preprocess the mechanism files
iError = MyGasMech.preprocess()

# display the preprocess status
print()
if iError != 0:
    # When a non-zero value is returned from the process, check the text output files,
    # "chem.out," "tran.out," and "summary.out," for potential error messages about the␣
↪mechanism data.
    print(f"Preprocessing error encountered. Code = {iError:d}.")
    print(f"See the summary file {MyGasMech.summaryfile} for details.")
    exit()
else:
    Color.ckprint("OK", ["Preprocessing succeeded.", "!!!"])
    print("Mechanism information:")
    print(f"Number of elements = {MyGasMech.number_elements:d}.")
    print(f"Number of gas species = {MyGasMech.number_species:d}.")
    print(f"Number of gas reactions = {MyGasMech.number_gas_reactions:d}.")
```

Display the basic mechanism information

```
print(f"\nElement and species information of mechanism {MyGasMech.label}")
print("=" * 50.0)

# extract element symbols as a list
elelist = MyGasMech.element_symbols
# get atomic masses as numpy 1D double array
AWT = MyGasMech.atomic_weight
# print element information
for k in range(len(elelist)):
    print(f"Element number {k+1:3d}: {elelist[k]:16}. Mass = {AWT[k]:f}.")

print("=" * 50)

# extract gas species symbols as a list
specieslist = MyGasMech.species_symbols
# get species molecular masses as numpy 1D double array
WT = MyGasMech.species_molar_weight
# print gas species information
for k in range(len(specieslist)):
    print(f"Species number {k+1:3d}: {specieslist[k]:16}. Mass = {WT[k]:f}.")
print("=" * 50)
```

Create a second chemistry set

The second mechanism to load into the project is the C2-NOx mechanism for the combustion of C1-C2 hydrocarbons. This mechanism differs from the GRI mechanism in the sense that it is self-contained, that is, the thermodynamic and transport data of all species is included in the `C2_NOx_SRK.inp` mechanism input file, which sits in the same reaction data directory as the GRI mechanism input files.

Use the same steps that were used to instantiate the first chemistry set to process any set of reaction mechanism files. Here, use a slightly different procedure to instantiate the second chemistry set. After you have created it, specify the reaction mechanism files one by one. The reaction mechanism file in this case contains all the necessary thermodynamic and transport data. Thus, there is no need to specify thermodynamic and transport data files. However, an additional step is required to instruct the preprocessor to include the transport data.

```
# set the second mechanism directory (the default Chemkin mechanism data directory)
mechanism_dir = data_dir

# create a chemistry set based on C2_NOx using an alternative method
My2ndMech = ck.Chemistry(label="C2 NOx")

# set mechanism input files individually
# this mechanism file contains all the necessary thermodynamic and transport data
# thus, there is no need to specify thermodynamic and transport data files
My2ndMech.chemfile = os.path.join(mechanism_dir, "C2_NOx_SRK.inp")

# instruct the preprocessor to include the transport properties
# only when the mechanism file contains all the transport data
My2ndMech.preprocess_transportdata()
```

Preprocess the second chemistry set

```
# preprocess the second mechanism file
iError = My2ndMech.preprocess()

# display the preprocess status
print()
if iError != 0:
    # When a non-zero value is returned from the process, check the text output files,
    # "chem.out," "tran.out," and "summary.out," for potential error messages about the␣
↪mechanism data.
    print(f"Preprocessing error encountered. Code = {iError:d}.")
    print(f"See the summary file {My2ndMech.summaryfile} for details.")
    exit()
else:
    Color.ckprint("OK", ["Preprocessing succeeded.", "!!!"])
    print("Mechanism information:")
    print(f"Number of elements = {My2ndMech.MM:d}.")
    print(f"Number of gas species = {My2ndMech.KK:d}.")
    print(f"Number of gas reactions = {My2ndMech.IIGas:d}.")
```

Display the basic mechanism information

```
print(f"\nElement and species information of mechanism {My2ndMech.label}")
print("=" * 50.0)

# extract element symbols as a list
elelist = My2ndMech.element_symbols
# get atomic masses as numpy 1D double array
AWT = My2ndMech.AWT
# print element information
for k in range(len(elelist)):
    print(f"Element # {k+1:3d}: {elelist[k]:16}. Mass = {AWT[k]:f}.")

print("=" * 50)

# extract gas species symbols as a list
specieslist = My2ndMech.species_symbols
# get species molecular masses as numpy 1D double array
WT = My2ndMech.WT
# print gas species information
for k in range(len(specieslist)):
    print(f"Species # {k+1:3d}: {specieslist[k]:16}. Mass = {WT[k]:f}.")

print("=" * 50)
```

12.1.2 Work with multiple mechanisms

PyChemkin can facilitate multiple mechanisms in one project. However, only one chemistry set can be active at a time. This example shows how to use the `activate()` method to switch between multiple chemistry sets (mechanisms) in the same Python project. You can use this method to compare results from two different mechanisms, such as the *base* and *reduced* mechanisms.

```
```

Import PyChemkin packages and start the logger

```
import os

import ansys.chemkin as ck  # Chemkin
from ansys.chemkin import Color
from ansys.chemkin.logger import logger

# check working directory
current_dir = os.getcwd()
logger.debug("working directory: " + current_dir)
# set verbose mode
ck.set_verbose(True)
```

Create the first chemistry set

The first mechanism to load is the GRI 3.0 mechanism for methane combustion. This mechanism and its associated data files come with the standard Ansys Chemkin installation in the `/reaction/data` directory.

```
# set mechanism directory (the default Chemkin mechanism data directory)
data_dir = os.path.join(ck.ansys_dir, "reaction", "data")
mechanism_dir = data_dir

# specify the mechanism input files
# including the full file path is recommended
chemfile = os.path.join(mechanism_dir, "grimech30_chem.inp")
thermfile = os.path.join(mechanism_dir, "grimech30_thermo.dat")
tranfile = os.path.join(mechanism_dir, "grimech30_transport.dat")
# create a chemistry set based on GRI 3.0
My1stMech = ck.Chemistry(chem=chemfile, therm=thermfile, tran=tranfile, label="GRI 3.0")
```

Preprocess the first chemistry set

```
# preprocess the mechanism files
iError = My1stMech.preprocess()
print()
if iError != 0:
    # encountered error during preprocessing
    print(f"Preprocessing error encountered. Code = {iError:d}.")
    print(f"See the summary file {My1stMech.summaryfile} for details.")
    exit()
else:
    # Display the basic mechanism information
    print(Color.GREEN + "Preprocessing succeeded.", end=Color.END)
    print("Mechanism information:")
    print(f"Number of elements = {My1stMech.MM:d}.")
    print(f"Number of gas species = {My1stMech.KK:d}.")
    print(f"Number of gas reactions = {My1stMech.IIGas:d}.")
```

Set up a gas mixture

Set up a gas mixture based on the species in this chemistry set. Create a gas mixture named `mymixture1` based on the `My1stMech` chemistry set. The species mole fractions/ratios are given in the `recipe` format. The `X` property is used because the mole fractions are given.

```
mymixture1 = ck.Mixture(My1stMech)
# set mixture temperature [K]
mymixture1.temperature = 1000.0
# set mixture pressure [dynes/cm2]
mymixture1.pressure = ck.Patm
# use the "X" property to specify the molar compositions of the mixture
mymixture1.X = [("CH4", 0.1), ("O2", 0.21), ("N2", 0.79)]
```

Perform an equilibrium calculation

The equilibrium state is stored as a mixture named `equil_mix1_HP`. You can get the equilibrium temperature from the `temperature` property of this mixture. You can learn more about the PyChemkin `equilibrium()` method by either typing `ck.help("equilibrium")` at the Python prompt or uncommenting the following line.

```
# ck.help("equilibrium")

# find the constrained H-P equilibrium state of ``mymixture1``
equil_mix1_HP = ck.equilibrium(mymixture1, opt=5)
# print the equilibrium temperature
print(f"Equilibrium temperature of mymixture1: {equil_mix1_HP.temperature} [K]")
```

Create the second chemistry set

The second mechanism is the `C2 NOx` mechanism in the `/reaction/data` directory. Here, a different process for setting up a chemistry set is used. The chemistry set instance is created before the mechanism files are specified. You can make changes to the files to include in the chemistry set before running the preprocessing step.

The `C2 NOx` mechanism file, in addition to the reactions, contains the thermodynamic and transport data of all species in the mechanism. Thus, you must only specify the mechanism file, that is, `chemfile`. If your simulation requires transport properties, you must use the `preprocess_transportdata()` method to tell the preprocessor to also include the transport data.

```
# set the second mechanism directory (the default Chemkin mechanism data directory)
mechanism_dir = data_dir
# create a chemistry set based on "C2_NOx" using an alternative method
My2ndMech = ck.Chemistry(label="C2 NOx")
# set mechanism input files individually
# this mechanism file contains all necessary thermodynamic and transport data
# thus, there is no need to specify thermodynamic and transport data files
My2ndMech.chemfile = os.path.join(mechanism_dir, "C2_NOx_SRK.inp")

# direct the preprocessor to include the transport properties
# only when the mechanism file contains all the transport data
My2ndMech.preprocess_transportdata()
```

Preprocess the second chemistry set

The `C2 NOx` mechanism also includes information about the *Soave* cubic Equation of State (EOS) for real-gas applications. The PyChemkin preprocessor indicates the availability of the real-gas model in the chemistry set processed. For example, during preprocessing, this is printed: `real-gas cubic EOS 'Soave' is available`. As soon as the second chemistry set is preprocessed successfully, it becomes the active chemistry set of the project. The first chemistry set, `My1stMech`, is pushed to the background.

```
# preprocess the second mechanism files
iError = My2ndMech.preprocess()
print()
if iError != 0:
    # encountered error during preprocessing
    print(f"Preprocessing error encountered. Code = {iError:d}.")
    print(f"See the summary file {My2ndMech.summaryfile} for details.")
    exit()
else:
    # Display the basic mechanism information
    print(Color.GREEN + "Preprocessing succeeded.", end=Color.END)
    print("Mechanism information:")
    print(f"Number of elements = {My2ndMech.MM:d}.")
    print(f"Number of gas species = {My2ndMech.KK:d}.")
    print(f"Number of gas reactions = {My2ndMech.IIGas:d}.")
```

Set up the second gas mixture based on the species in the second chemistry set

Create a gas mixture named `mymixture2` based on the `My2ndMech` chemistry set.

```
mymixture2 = ck.Mixture(My2ndMech)
# set mixture temperature [K]
mymixture2.temperature = 500.0
# set mixture pressure [dynes/cm2]
mymixture2.pressure = 2.0 * ck.Patm
# set mixture molar composition
mymixture2.X = [("H2", 0.02), ("O2", 0.2), ("N2", 0.8)]
```

Run the detonation calculation

You can now compute the detonation wave speed with the `mymixture2` gas mixture. The `CJ_mix2` represents the mixture at the Chapman-Jouguet state. The speed of sound and the detonation wave speed are returned in the `speeds_mix2` tuple.

```
speeds_mix2, CJ_mix2 = ck.detonation(mymixture2)
#  print the detonation calculation results
print(f"Detonation mymixture2 temperature: {CJ_mix2.temperature} [K]")
print(f"Detonation wave speed = {speeds_mix2[1]/100.0} [m/sec]")
```

Switch to the first chemistry set and gas mixture

Use the `activate()` method to reactivate the `My1stMech` chemistry set and the `mymixture1` gas mixture.

```
My1stMech.activate()
```

Run the detonation calculation

You can now compute the detonation wave speed with the `mymixture1` gas mixture. The `CJ_mix1` represents the mixture at the Chapman-Jouguet state. The speed of sound and the detonation wave speed are returned in the `speeds_mix1` tuple.

Note

The `mymixture1` and `mymixture2` gas mixtures have different initial conditions.

```
speeds_mix1, CJ_mix1 = ck.detonation(mymixture1)
# print the detonation calculation results
print(f"Detonation 'mymixture1' temperature: {CJ_mix1.temperature} [K]")
print(f"Detonation wave speed = {speeds_mix1[1]/100.0} [m/sec]")
```

12.1.3 Set up a PyChemkin project

This example shows how to set up a PyChemkin project and start to use PyChemkin features. You begin by importing the PyChemkin package, which is named `ansys.chemkin`, and optionally the PyChemkin logger package, which is named `ansys.chemkin.logger`.

Next, you use the `Chemistry()` method to instantiate and preprocess a chemistry set. This requires specifying the file names (with full file paths) of the mechanism file, thermodynamic data file, and optional transport data file.

Once the chemistry set instance is instantiated, you use the `preprocess()` method to preprocess the mechanism data. You can then start to use PyChemkin features to build your simulation.

Note

Running PyChemkin requires Ansys 2025 R2 or later.

```

```

Import PyChemkin packages

Import the `ansys.chemkin` package to start using PyChemkin in the project. Importing the `ansys.chemkin.logger` package is optional.

```
import os

import ansys.chemkin  # import PyChemkin
from ansys.chemkin.logger import logger
```

Set up the file paths of the mechanism and data files

The `ansys_dir` variable on your local computer specifies the location of your Ansys installation.

Use `os.path` methods to construct the file names with the full paths.

Assign the files to the corresponding chemistry set arguments:

- `mech_file`: Mechanism input file
- `therm_file`: Thermodynamic data file

- `tran_file`: Transport data file (optional)

```
# create GRI 3.0 mechanism from the data directory
mechanism_dir = os.path.join(ansys.chemkin.ansys_dir, "reaction", "data")
# set up mechanism file names
mech_file = os.path.join(mechanism_dir, "grimech30_chem.inp")
therm_file = os.path.join(mechanism_dir, "grimech30_thermo.dat")
tran_file = os.path.join(mechanism_dir, "grimech30_transport.dat")
```

Instantiate the chemistry set

Use the `Chemistry()` method to instantiate a chemistry set named `GasMech`.

```
GasMech = ansys.chemkin.Chemistry(
    chem=mech_file, therm=therm_file, tran=tran_file, label="GRI 3.0"
)
```

Preprocess the chemistry set

Preprocess the GRI 3.0 chemistry set.

```
status = GasMech.preprocess()
```

Check the preprocessing status

```
if status != 0:
    # failed
    print(f"Preprocessing: Error encountered. Code = {status:d}.")
    print(f"See the summary file {GasMech.summaryfile} for details.")
    logger.error("Preprocessing failed.")
    exit()
```

Start to use PyChemkin features

Start to use PyChemkin features to build your simulation. For example, using the `Mixture()` method, create an `air` mixture based on the `GasMech` chemistry set.

```
# create 'air' mixture based on 'GasMech' chemistry set
air = ansys.chemkin.Mixture(GasMech)
# set 'air' condition
# mixture pressure in [dynes/cm2]
air.pressure = 1.0 * ansys.chemkin.Patm
# mixture temperature in [K]
air.temperature = 300.0
# mixture composition in mole fractions
air.X = [("O2", 0.21), ("N2", 0.79)]
```

Print the properties of the air mixture for verification

Note

- The default units of temperature and pressure are [K] and [dynes/cm^2], respectively.

- The constant `Patm` is a conversion multiplier for pressure.
- Transport property methods such as `mixture_viscosity()` require transport data. You must include the transport data file when creating the chemistry set.

```
# print pressure and temperature of the `air` mixture
print(f"Pressure    = {air.pressure/ansys.chemkin.Patm} [atm]")
print(f"Temperature = {air.temperature} [K]")
# print the 'air' composition in mass fractions
air.list_composition(mode="mass")
# get 'air' mixture density [g/cm3]
print(f"Mixture density   = {air.RHO} [g/cm3]")
# get 'air' mixture viscosity [g/cm-sec] or [poise]
print(f"Mixture viscosity = {air.mixture_viscosity()*100.0} [cP]")
```

12.1.4 Evaluate gas species properties

PyChemkin inherits many Ansys Chemkin utilities for evaluating species properties in a mechanism. Most tools for calculating the species properties are part of the PyChemkin `Chemistry()` methods. This example shows how use some of these methods to evaluate species thermodynamic and transport properties.

```

```

Import PyChemkin packages and start the logger

```
import os

import ansys.chemkin as ck  # Chemkin
from ansys.chemkin.logger import logger
import matplotlib.pyplot as plt  # plotting
import numpy as np  # number crunching

# check working directory
current_dir = os.getcwd()
logger.debug("working directory: " + current_dir)
# set verbose mode
ck.set_verbose(True)
# set interactive mode for plotting the results
# interactive = True: display plot
# interactive = False: save plot as a PNG file
global interactive
interactive = True
```

Create a chemistry set

The mechanism to load is the GRI 3.0 mechanism for methane combustion. This mechanism and its associated data files come with the standard Ansys Chemkin installation in the `/reaction/data` directory.

```
# set mechanism directory (the default Chemkin mechanism data directory)
data_dir = os.path.join(ck.ansys_dir, "reaction", "data")
mechanism_dir = data_dir

```

(continues on next page)

(continued from previous page)

```
# create a chemistry set based on GRI 3.0
MyGasMech = ck.Chemistry(label="GRI 3.0")
# set mechanism input files
# inclusding the full file path is recommended
MyGasMech.chemfile = os.path.join(mechanism_dir, "grimech30_chem.inp")
MyGasMech.thermfile = os.path.join(mechanism_dir, "grimech30_thermo.dat")
MyGasMech.tranfile = os.path.join(mechanism_dir, "grimech30_transport.dat")
```

Preprocess the chemistry set

```
# preprocess the mechanism files
iError = MyGasMech.preprocess()
```

Display the basic mechanism information

Display the element and species information from the GRI 3.0 mechanism. You can get the entire list of the elements and the gas species in the mechanism by using the `element_symbols` and `species_symbols` lists, respectively.

Use the `get_specindex()` method to get the species index of a species in the mechanism. Use the `SpeciesComposition()` method to check the elemental composition of a gas species.

Note

Both the species name/symbol and element name/symbol are case-sensitive.

```
# extract element symbols as a list
elelist = MyGasMech.element_symbols
# extract gas species symbols as a list
specieslist = MyGasMech.species_symbols
# list of gas species of interest
plotspeclist = ["CH4", "O2", "N2"]
# find elemental compositions of selected species
print(" ")
for s in plotspeclist:
    speciesID = MyGasMech.get_specindex(s)
    print("species " + specieslist[speciesID])
    print("elemental composition")
    for elemID in range(MyGasMech.MM):
        num_elem = MyGasMech.SpeciesComposition(elemID, speciesID)
        print(f"    {elelist[elemID]:>4}: {num_elem:2d}")
    print("=" * 10)
print()
```

Plot selected species properties

Calculate and plot the properties of CH_4, O_2, and N_2 against the temperature. Use these property methods:

- `SpeciesCv`: Species specific heat capacity at constant volume [erg/mol-K]
- `SpeciesCond`: Species thermal conductivity [erg/cm-K-sec]
- `SpeciesDiffusionCoeffs`: Binary diffusion coefficients between species pairs [cm^2/sec]

```
# plot Cv and thermal conductivity values at different temperatures for selected gas␣
↪species
#
plt.figure(figsize=(12, 6))
# temperature increment
dTemp = 20.0
# number of property data points
points = 100
# curve attributes
curvelist = ["g", "b--", "r:"]
```

Prepare the species Cv data for plotting

```
# create arrays
# specify specific heat capacity at constant volume data
Cv = np.zeros(points, dtype=np.double)
# temperature data
T = np.zeros(points, dtype=np.double)
# start plotting loop #1
k = 0
# loop over selected gas species
for s in plotspeclist:
    # starting temperature at 300 [K]
    Temp = 300.0
    # loop over temperature data points
    for i in range(points):
        HeatCapacity = MyGasMech.SpeciesCv(Temp)
        ID = MyGasMech.get_specindex(s)
        T[i] = Temp
        # convert ergs to joules
        Cv[i] = HeatCapacity[ID] / ck.ergs_per_joule
        Temp += dTemp
    plt.subplot(121)
    plt.plot(T, Cv, curvelist[k])
    k += 1
# plot Cv versus temperature
plt.xlabel("Temperature [K]")
plt.ylabel("Cv [J/mol-K]")
plt.legend(plotspeclist, loc="upper left")
```

Prepare the species thermal conductivity data for plotting

```
# create arrays
# specify conductivity
kappa = np.zeros(points, dtype=np.double)
# start plotting loop #2
k = 0
# loop over selected gas species
for s in plotspeclist:
    # starting temperature at 300 [K]
    Temp = 300.0
    # loop over temperature data points
```

(continues on next page)

(continued from previous page)

```
    for i in range(points):
        conductivity = MyGasMech.SpeciesCond(Temp)
        ID = MyGasMech.get_specindex(s)
        T[i] = Temp
        # convert ergs to joules
        kappa[i] = conductivity[ID] / ck.ergs_per_joule
        Temp += dTemp
    plt.subplot(122)
    plt.plot(T, kappa, curvelist[k])
    k += 1
# plot conductivity versus temperature
plt.xlabel("Temperature [K]")
plt.ylabel("Conductivity [J/cm-K-sec]")
plt.legend(plotspeclist, loc="upper left")
```

Evaluate the binary diffusion coefficients between different gas species

Use the `SpeciesDiffusionCoeffs()` method to calculate the binary diffusion coefficients between pairs of gas species. Here, the binary diffusion coefficients are evaluated at 2 [atm] and 500 [K]. The binary diffusion coefficient between CH_4 and O_2 is shown.

```
diffcoef = MyGasMech.SpeciesDiffusionCoeffs(2.0 * ck.Patm, 500.0)
ID1 = MyGasMech.get_specindex(plotspeclist[0])
ID2 = MyGasMech.get_specindex(plotspeclist[1])
c = diffcoef[ID1][ID2]
print(
    f"Diffusion coefficient for {plotspeclist[0]} against {plotspeclist[1]} is {c:e}␣
↪[cm^2/sec]."
)

# plot results
if interactive:
    plt.show()
else:
    plt.savefig("plot_species_properties.png", bbox_inches="tight")
```

12.2 Mixture

Mixture is the "core" token in **PyChemkin**. It represents a gas-phase mixture by storing its pressure, temperature, and species composition. *Stream* is an extended "Class" of *Mixture* with the additional attribute of "mass/volumetric flow rate". *Mixture* and *Stream* include utilities that can be used to evaluate the mixture properties (density, mixture enthalpy, mixture viscosity, ...) and the reaction rates. There are also methods to manipulate the *Mixture*/*Stream* such as merging two mixtures adiabatically.

The examples demonstrate how to instantiate the *Mixture*/*Stream* objects, how to extract information and properties of a *Mixture*/*Stream*, and how to manipulate *Mixture*/*Stream* objects.

12.2.1 Estimate the adiabatic flame temperature of a gas mixture

This example shows how to find the equilibrium state of a mixture. It uses the `equilibrium()` method with the `constant pressure and enthalpy` option to estimate the adiabatic flame temperature of a methane-oxygen mixture. This example also explores the influence of the equivalence ratio on the predicted adiabatic flame temperature.

Import PyChemkin packages and start the logger

```
import os

import ansys.chemkin as ck  # Chemkin
from ansys.chemkin.logger import logger
import matplotlib.pyplot as plt  # plotting
import numpy as np  # number crunching

# check working directory
current_dir = os.getcwd()
logger.debug("working directory: " + current_dir)
# set verbose mode
ck.set_verbose(True)
# set interactive mode for plotting the results
# interactive = True: display plot
# interactive = False: save plot as a PNG file
global interactive
interactive = True
```

Create a chemistry set

The first mechanism to load is the GRI 3.0 mechanism for methane combustion. This mechanism and its associated data files come with the standard Ansys Chemkin installation in the `/reaction/data` directory.

```
# set mechanism directory (the default Chemkin mechanism data directory)
data_dir = os.path.join(ck.ansys_dir, "reaction", "data")
mechanism_dir = data_dir

# create a chemistry set based on the GRI 3.0 methane combustion mechanism
MyGasMech = ck.Chemistry(label="GRI 3.0")
# set mechanism input files
# including the full file path is recommended
MyGasMech.chemfile = os.path.join(mechanism_dir, "grimech30_chem.inp")
MyGasMech.thermfile = os.path.join(mechanism_dir, "grimech30_thermo.dat")
# skip the transport data file where is not needed by this example
```

Preprocess the chemistry set

```
# preprocess the mechanism files
iError = MyGasMech.preprocess()
```

Set up gas mixtures

Set up gas mixtures based on the species in this chemistry set. PyChemkin has a few methods for creating a gas mixture. Here, the equivalence ratio method is used to set up the combustible mixture so that you can easily change the mixture composition by assigning a different equivalence ratio value.

```
# create an "oxid" mixture associated with the 'MyGasMech' chemistry set
oxid = ck.Mixture(MyGasMech)
# use a "recipe" to set the mole fractions of the mixture
# the "oxid" mixture consists of 100% O2
oxid.X = [("O2", 1.0)]
oxid.temperature = 295.15  # [K]
oxid.pressure = ck.Patm  # 1 atm

# create the "fuel" mixture
fuel = ck.Mixture(MyGasMech)
# set the "fuel" molar composition to 100% CH4
fuel.X = [("CH4", 1.0)]
fuel.temperature = oxid.temperature
fuel.pressure = oxid.pressure

# create the final fuel-oxidizer mixture
mixture = ck.Mixture(MyGasMech)
mixture.pressure = oxid.pressure
mixture.temperature = oxid.temperature

# the use of equivalence ratio requires the definition of the complete combustion
# products of the fuel-oxidizer pair:
# CH4 + 2O2 => CO2 + 2H2O
products = ["CO2", "H2O"]

# create an array to specify the composition of the additives to the fuel-oxidizer␣
↪mixture
# For example, a diluent such as argon or helium might be added to the fuel-oxidizer␣
↪mixture
# Use an all-zero array if there is no additive
add_frac = np.zeros(MyGasMech.KK, dtype=np.double)
```

Set up the parameter study

Set up a parameter study to find out the impact of the equivalence ratio on the adiabatic flame temperature of the fuel-oxidizer mixture. The equivalence ratio varies from 0.5 to 1.6 with an increment of 0.1.

```
points = 12
deq = 0.1
equiv_ini = 0.5

# create the solution arrays as double arrays
T = np.zeros(points, dtype=np.double)
equiv = np.zeros_like(T, dtype=np.double)
```

Run the parameter study

Use the `equilibrium()` method to estimate the adiabatic flame temperature of the fuel-oxidizer mixture. Choose the option corresponding to constant pressure and constant enthalpy for the equilibrium calculation because you are finding the adiabatic temperature.

The `equilibrium()` method returns a mixture object representing the mixture at the equilibrium state. Use the `temperature()` method to get the equilibrium temperature. To see all available options for this method, use the `ck.help(topic="equilibrium")` method.

This example uses the `X_by_Equivalence_Ratio()` method to set the fuel-oxidizer composition with the given equivalence ratios because the composition of both the fuel and oxidizer mixtures is specified in mole fractions.

```
for i in range(points):
    # set the current mixture equivalence ratio
    equiv_current = equiv_ini

    # create the fuel-oxidizer mixture with the given equivalence ratio
    iError = mixture.X_by_Equivalence_Ratio(
        MyGasMech, fuel.X, oxid.X, add_frac, products, equivalenceratio=equiv_current
    )
    # check fuel-oxidizer mixture creation status
    if iError != 0:
        print("Error: Failed to create the fuel-oxidizer mixture.")
        print(f"       Equivalence ratio = {equiv_current}.")
        exit()

    # use "equilibrium()" method to calculate the gas mixture at the equilibrium state
    # Option #5 ("opt=5") corresponds to the constant pressure and constant-enthalpy
    # constraints of the equilibrium state
    # "EQ_mixture" is the gas mixture at the equilibrium state
    EQ_mixture = ck.equilibrium(mixture, opt=5)

    # save the results to the solution arrays
    # use "temperature" to obtain the temperature of the equilibrium state
    T[i] = EQ_mixture.temperature
    equiv[i] = equiv_current
    equiv_ini = equiv_ini + deq
```

Plot the result from the parameter study

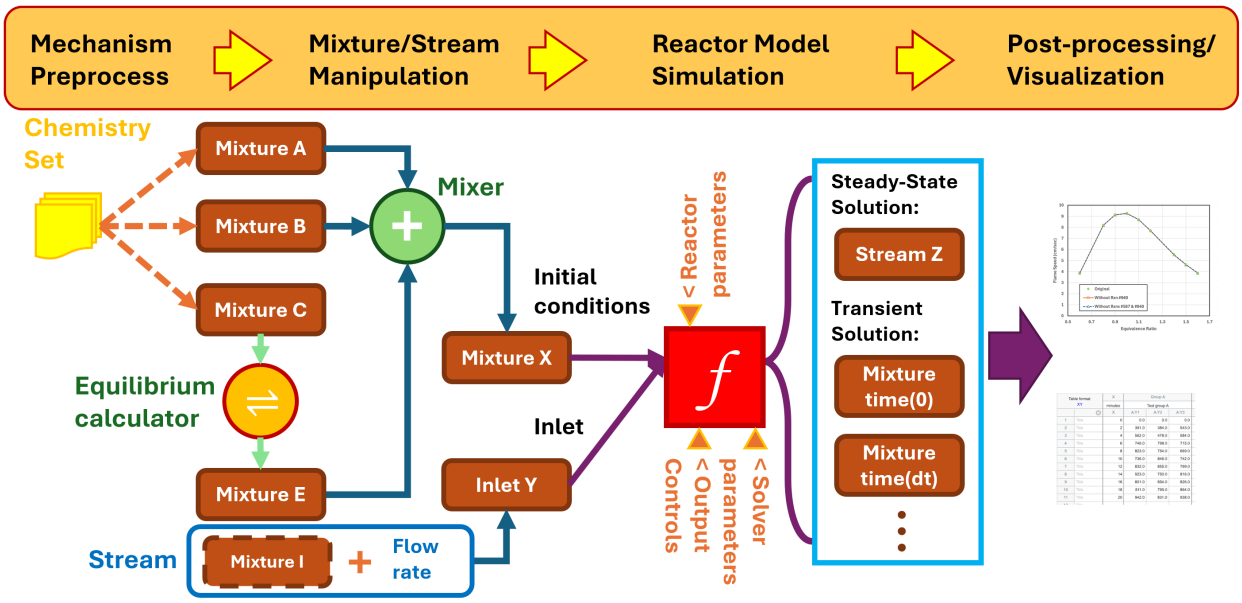
When you plot the result from the parameter study, the adiabatic flame temperature should exhibit a peak at around the stoichiometric, that is, equivalence ratio = 1.

```
# plot equilibrium/adiabatic temperatures against mixture equivalence ratios
plt.plot(equiv, T, "bs--")
# set up axis labels
plt.xlabel("Equivalence ratio")
plt.ylabel("Temperature [K]")
# display or save the plot
if interactive:
    plt.show()
else:
    plt.savefig("plot_adiabatic_flame_temperature.png", bbox_inches="tight")
```

12.2.2 Create a mixture

A *mixture* is a core component of the PyChemkin framework. In addition to getting mixture thermodynamic and transport properties, such as density, heat capacity, and viscosity, you can combine two mixtures, find the equilibrium state of a mixture, or use a mixture to define the initial state of a reactor. A PyChemkin *reactor model* is a black box that transforms a mixture from its initial state to a new one.

The following schematic shows the basic operations available for a mixture in PyChemkin: **create**, **combine/mix**, and **transform** (by a reactor model).



This example shows different ways to create a mixture in PyChemkin. The use of the composition *recipe* lets you provide just the non-zero species components with a list of species-fraction pairing tuples. Alternatively, the NumPy array lets you use a *full-size* (equal to the number of species) mole/mass fraction array to specify the mixture composition. The `equivalence ratio()` method creates a new mixture from predefined fuel and oxidizer mixtures by assigning an equivalence ratio value.

Import PyChemkin packages and start the logger

```
import copy
import os

import ansys.chemkin as ck  # Chemkin
from ansys.chemkin.logger import logger
import matplotlib.pyplot as plt  # plotting
import numpy as np  # number crunching

# check working directory
current_dir = os.getcwd()
logger.debug("working directory: " + current_dir)
# set verbose mode
ck.set_verbose(True)
# set interactive mode for plotting the results
```

(continues on next page)

(continued from previous page)

```
# interactive = True: display plot
# interactive = False: save plot as a PNG file
global interactive
interactive = True
```

Create a chemistry set

The mechanism to load is the C2 NOx mechanism. This mechanism file and the associated data files come with the standard Ansys Chemkin installation in the `/reaction/data` directory. The C2 NOx mechanism file, in addition to the reactions, contains the thermodynamic and transport data of all species in the mechanism. In this case, you only need to specify the mechanism file, that is, `chemfile`. If your simulation requires the transport properties, you must use the `preprocess_transportdata()` method to tell the PyChemkin preprocessor to also include the transport data. The preprocessor does not include the transport data by default.

```
# set mechanism directory (the default Chemkin mechanism data directory)
data_dir = os.path.join(ck.ansys_dir, "reaction", "data")
mechanism_dir = data_dir
# create a chemistry set based on the C2 NOx mechanism
MyGasMech = ck.Chemistry(label="C2 NOx")
# set mechanism input files
# this mechanism file contains all the necessary thermodynamic and transport data
# thus, there is no need to specify thermodynamic and the transport data files
MyGasMech.chemfile = os.path.join(mechanism_dir, "C2_NOx_SRK.inp")

# direct the preprocessor to include the transport properties (when the tran data file␣
↪is not provided)
MyGasMech.preprocess_transportdata()
```

Preprocess the C2 NOx chemistry set

The C2 NOx mechanism includes information about the *Soave* cubic Equation of State (EOS) for real-gas applications. The preprocessor indicates the availability of the real-gas model in the chemistry set processed. For example, during preprocessing, you see this print out: `real-gas cubic EOS 'Soave' is available`.

```
# preprocess the mechanism files
iError = MyGasMech.preprocess()
```

Set up gas mixtures based on the species in the C2 NOx chemistry set

Create a gas mixture instance named `premixed` based on the `My2ndMech` chemistry set.

```
premixed = ck.Mixture(MyGasMech)
# set mixture pressure [dynes/cm2]
mixpressure = 2.0  # given in atm
# convert to dynes/cm2
premixed.pressure = mixpressure * ck.Patm
# set mixture temperature [K]
premixed.temperature = 500.0
# create a recipe for the molar composition of the mixture
mixture_recipe = [("CH4", 0.08), ("N2", 0.6), ("O2", 0.2), ("H2O", 0.12)]
# set mixture mole fractions
premixed.X = mixture_recipe
```

Find mixture mean molecular mass

Use the `WTM()` method to get the mean molar mass of the gas mixture.

```
print(f"Mean molecular mass = {premixed.WTM:f} gm/mole")
print("=" * 40.0)
```

List the mixture composition

Use the `Y()` method to automatically convert the mole fractions to mass fractions and vice versa. Use the `list_composition()` method to display only the non-zero components of the gas mixture.

```
print("mixture mass fractions (raw data):")
print(str(premixed.Y))
# switch back to mole fractions
print("\nmixture mole fractions (raw data):")
print(str(premixed.X))
# beautify the composition list
print("\nformatted mixture composition output:")
print("=" * 40)
premixed.list_composition(mode="mole")
print("=" * 40)
```

Create a hard copy of a mixture

Create a hard copy of the premixed mixture. (Soft copying, that is, using `anotherpremixed = premixed`, also works.)

```
anotherpremixed = copy.deepcopy(premixed)
# display the molar composition of the new mixture
print("\nFormatted mixture composition of the copied mixture")
anotherpremixed.list_composition(mode="mole")
print("=" * 40.0)
```

Compute and plot mixture properties

The PyChemkin `Mixture` module offers many basic methods to compute the thermodynamic and transport properties of a species and a gas mixture. The following code plots selected mixture properties as a function of the mixture temperature. It uses the `RHO()` and `HML()` methods to get the mixture density and mixture enthalpy, respectively. Use the `mixture_viscosity()` method to get the mixture transport property, `viscosity`. Use the `mixture_diffusion_coeffs()` method to get the mixture-averaged diffusion coefficient of CH_4. The temperature and pressure are required to compute the properties.

```
plt.subplots(2, 2, sharex="col", figsize=(9, 9))
dTemp = 20.0
points = 100
# curve attributes
curvelist = ["g", "b--", "r:"]
# list of pressure values for the plot
press = [1.0, 5.0, 10.0]  # given in atm
# create arrays for the plot
# mixture density
rho = np.zeros(points, dtype=np.double)
# mixture enthalpy
```

(continues on next page)

(continued from previous page)

```
enthalpy = np.zeros_like(rho, dtype=np.double)
# mixture viscosity
visc = np.zeros_like(rho, dtype=np.double)
# mixture averaged diffusion coefficient of CH4
diff_CH4 = np.zeros_like(rho, dtype=np.double)
# species index of CH4
CH4_index = MyGasMech.get_specindex("CH4")
# temperature data
T = np.zeros_like(rho, dtype=np.double)
# start the plotting of loop #1
k = 0
# loop over the pressure values
for j in range(len(press)):
    # starting temperature at 300K
    temp = 300.0
    # loop over temperature data points
    for i in range(points):
        # set mixture pressure [dynes/cm2]
        premixed.pressure = press[j] * ck.Patm
        # set mixture temperature [K]
        premixed.temperature = temp
        # get mixture density [gm/cm3]
        rho[i] = premixed.RHO
        # get mixture enthalpy [ergs/mol] and convert it to [kJ/mol]
        enthalpy[i] = premixed.HML() * 1.0e-3 / ck.ergs_per_joule
        # get mixture viscosity [gm/cm-sec]
        visc[i] = premixed.mixture_viscosity()
        # get mixture-averaged diffusion coefficient of CH4 [cm2/sec]
        diffcoeffs = premixed.mixture_diffusion_coeffs()
        diff_CH4[i] = diffcoeffs[CH4_index]
        T[i] = temp
        temp += dTemp

    # create subplots
    # plot mixture density versus temperature
    plt.subplot(221)
    plt.plot(T, rho, curvelist[k])
    plt.ylabel("Density [g/cm3]")
    plt.legend(("1 atm", "5 atm", "10 atm"), loc="upper right")
    # plot mixture enthalpy versus temperature
    plt.subplot(222)
    plt.plot(T, enthalpy, curvelist[k])
    plt.ylabel("Enthalpy [kJ/mole]")
    plt.legend(("1 atm", "5 atm", "10 atm"), loc="upper left")
    # plot mixture viscosity versus temperature
    plt.subplot(223)
    plt.plot(T, visc, curvelist[k])
    plt.xlabel("Temperature [K]")
    plt.ylabel("Viscosity [g/cm-sec]")
    plt.legend(("1 atm", "5 atm", "10 atm"), loc="lower right")
    # plot mixture averaged CH4 diffusion coefficient versus temperature
    plt.subplot(224)
```

(continues on next page)

(continued from previous page)

```
    plt.plot(T, diff_CH4, curvelist[k])
    plt.xlabel("Temperature [K]")
    plt.ylabel(r"$D_{CH_4}$ [cm2/sec]")
    plt.legend(("1 atm", "5 atm", "10 atm"), loc="upper left")
    k += 1

# plot legends
plt.legend(("1 atm", "5 atm", "10 atm"), loc="upper left")
# plot results
if interactive:
    plt.show()
else:
    plt.savefig("plot_create_mixture.png", bbox_inches="tight")
```

12.2.3 Calculate the detonation wave speed of a real-gas mixture

Use the `detonation()` method on a combustible mixture to compute the Chapman-Jouguet (C-J) state and detonation wave speed. This example shows how to predict the detonation wave speeds of a natural gas-air mixture at various initial pressures and compare the *ideal-gas* and the *real-gas* results against the experimental data.

Import PyChemkin packages and start the logger

```
import os

import ansys.chemkin as ck  # Chemkin
from ansys.chemkin.logger import logger
import matplotlib.pyplot as plt  # plotting
import numpy as np  # number crunching

# check working directory
current_dir = os.getcwd()
logger.debug("working directory: " + current_dir)
# set verbose mode
ck.set_verbose(True)
# set interactive mode for plotting the results
# interactive = True: display plot
# interactive = False: save plot as a PNG file
global interactive
interactive = True
```

Create a chemistry set

The `C2 NOx` mechanism comes with the standard Ansys Chemkin installation in the `/reaction/data` directory. This mechanism includes information about the *Soave* cubic Equation of State (EOS) for the real-gas applications. The PyChemkin preprocessor indicates the availability of the real-gas model in the chemistry set processed.

```
# set mechanism directory (the default Chemkin mechanism data directory)
data_dir = os.path.join(ck.ansys_dir, "reaction", "data")
mechanism_dir = data_dir
```

(continues on next page)

(continued from previous page)

```
# create a chemistry set based on C2 NOx using an alternative method
MyMech = ck.Chemistry(label="C2 NOx")
# set mechanism input files individually
# Because this mechanism file contains all the necessary thermodynamic and transport␣
↪data,
# you do not need to specify thermodynamic and transport data files.
MyMech.chemfile = os.path.join(mechanism_dir, "C2_NOx_SRK.inp")
```

Preprocess the C2 NOx chemistry set

During preprocessing, you should see this printed: `real-gas cubic EOS 'Soave' is available`. Because no transport data file is provided and the `preprocess_transportdata()` method is not used, transport property methods are not available in this project.

```
# preprocess the mechanism files
iError = MyMech.preprocess()
```

Set up gas mixtures based on the species in the C2 NOx chemistry set

Create gas mixtures named `fuel` (natural gas) and `air` based on `MyMech`. Then, use these two mixtures to form the combustible `premixed` mixture for the detonation calculations. Use the `X_by_Equivalence_Ratio()` method to set the equivalence ratio of the fuel-air mixture to 1.

```
# create the fuel mixture
fuel = ck.Mixture(MyMech)
# set mole fraction
fuel.X = [("CH4", 0.8), ("C2H6", 0.2)]
fuel.temperature = 290.0
fuel.pressure = 40.0 * ck.Patm
# create the air mixture
air = ck.Mixture(MyMech)
# set mass fraction
air.X = [("O2", 0.21), ("N2", 0.79)]
air.temperature = fuel.temperature
air.pressure = fuel.pressure
# create the initial mixture
# create the premixed mixture to be defined by equivalence ratio
premixed = ck.Mixture(MyMech)
# products from the complete combustion of the fuel mixture and air
products = ["CO2", "H2O", "N2"]
# species mole fractions of added/inert mixture. Can also create an additives mixture␣
↪here.
add_frac = np.zeros(MyMech.KK, dtype=np.double)  # no additives: all zeros

iError = premixed.X_by_Equivalence_Ratio(
    MyMech, fuel.X, air.X, add_frac, products, equivalenceratio=1.0
)
# check fuel-oxidizer mixture creation status
if iError != 0:
    print("Error: Failed to create the premixed mixture.")
    exit()
```

Display the molar composition of the premixed mixture

List the composition of the premixed mixture for verification.

```
premixed.list_composition(mode="mole")
```

Run the detonation calculation

Use the `detonation()` method to find the C-J state and detonation wave speed of the fuel-air mixture with the initial mixture pressure increasing from 40 to 80 [atm]. The initial mixture temperature is kept at 290 [K].

The `detonation()` method returns two objects: a `speed` tuple containing the speed of sound and the detonation wave speed at the C-J state. The `CJState` mixture contains the mixture properties at the C-J state. For instance, you can use `CJState.pressure` to get the mixture pressure at the C-J state.

Note

- By default, Chemkin variables are in cgs units.
- You can enter the `ansys.chemkin.help("equilibrium")` command at the Python prompt to see the input and output parameters of the `detonation()` method.

Run the parameter study

Set up the parameter study of the detonation wave speed with respect to the initial pressure. The predicted detonation wave speed values are saved in the `Det` array. The experimental data is stored in the `Det_data` array. By default, the *ideal-gas law* is assumed. You can use the `use_realgas_cubicEOS()` method to turn on the *real-gas model* if the mechanism contains the real-gas parameters in the EOS block. Use the `use_idealgas_law()` method to reactivate the ideal-gas law assumption.

```
points = 5
dpres = 10.0 * ck.Patm
pres = fuel.pressure
P = np.zeros(points, dtype=np.double)
Det = np.zeros_like(P, dtype=np.double)
premixed.pressure = pres
premixed.temperature = fuel.temperature

# start of pressure loop
for i in range(points):
    # compute the C-J state corresponding to the initial mixture
    speed, CJstate = ck.detonation(premixed)
    # update plot data
    # convert pressure to atm
    P[i] = pres / ck.Patm
    # convert speed to m/sec
    Det[i] = speed[1] / 1.0e2
    # update pressure value
    pres += dpres
    premixed.pressure = pres

# create plot for ideal-gas results
plt.plot(P, Det, "bo--", label="ideal gas", markersize=5, fillstyle="none")
```

Switch to the real-gas EOS model

Use the `use_realgas_cubicEOS()` method to turn on the real-gas EOS model. You can enter `ansys.chemkin.help("real gas")` to see usage information on real-gas models or `ansys.chemkin.help("manuals")` to access the online *Chemkin Theory* manual for descriptions of the real-gas EOS models.

Note

By default, the *Van der Waals* mixing rule is applied to evaluate thermodynamic properties of a real-gas mixture. You can use the `set_realgas_mixing_rule()` method to switch to a different mixing rule.

```
# turn on real-gas cubic equation of state
premixed.use_realgas_cubicEOS()
# set mixture mixing rule to Van der Waals (default)
# premixed.set_realgas_mixing_rule(rule=0)
# restart the calculation with real-gas EOS
premixed.pressure = fuel.pressure
pres = fuel.pressure
P[:] = 0.0e0
Det[:] = 0.0e0
# set verbose mode to false to turn off extra printouts
ck.set_verbose(False)
# start of pressure loop
for i in range(points):
    # compute the C-J state corresponding to the initial mixture
    speed, CJstate = ck.detonation(premixed)
    # update plot data
    P[i] = pres / ck.Patm
    Det[i] = speed[1] / 1.0e2
    # update pressure value
    pres += dpres
    premixed.pressure = pres

# stop Chemkin
ck.done()
# create plot for real-gas results
plt.plot(P, Det, "r^-", label="real gas", markersize=5, fillstyle="none")
# plot data
P_data = [44.1, 50.6, 67.2, 80.8]
Det_data = [1950.0, 1970.0, 2000.0, 2020.0]
plt.plot(P_data, Det_data, "gD:", label="data", markersize=4)
```

Plot the result from the parameter study

You should see that the ideal-gas assumption fails to show any noticeable pressure influence on the detonation wave speeds. Because of the relatively high pressures in this study, you can observe significant differences in the predicted detonation wave speeds between the ideal-gas and real-gas models.

```
plt.legend(loc="upper left")
plt.xlabel("Pressure [atm]")
plt.ylabel("Detonation wave speed [m/sec]")
plt.suptitle("Natural Gas/Air Detonation", fontsize=16)
```

(continues on next page)

(continued from previous page)

```
# plot results
if interactive:
    plt.show()
else:
    plt.savefig("plot_detonation.png", bbox_inches="tight")
```

12.2.4 Estimate the steady-state NO emission level of a complete burned fuel-air mixture

This example shows how to quickly estimate the steady-state NO level that is formed by the combustion of a fuel-air mixture at a given temperature. Without needing any reaction, the NO concentration gets to its steady state (or the maximum level) when the product mixture from the fuel-air combustion reaches the equilibrium state at the given temperature. To find the equilibrium state of the fresh fuel-air mixture, the `equilibrium()` method is used with the `fixed pressure` and `fixed temperature` options.

This example explores the influence of temperature on the predicted adiabatic flame temperature. Knowing that nitrogen oxides (NOx) are stable at high temperatures (> 2000 [K]), you can expect that the steady-state NO level should increase sharply when the temperature gets high enough.

```

```

Import PyChemkin packages and start the logger

```
import os

import ansys.chemkin as ck  # Chemkin
from ansys.chemkin.logger import logger
import matplotlib.pyplot as plt  # plotting
import numpy as np  # number crunching

# check working directory
current_dir = os.getcwd()
logger.debug("working directory: " + current_dir)
# set verbose mode
ck.set_verbose(True)
# set interactive mode for plotting the results
# interactive = True: display plot
# interactive = False: save plot as a PNG file
global interactive
interactive = True
```

Create a chemistry set

The first mechanism to load is the GRI 3.0 mechanism for methane combustion. This mechanism and its associated data files come with the standard Ansys Chemkin installation in the `/reaction/data` directory.

```
# set mechanism directory (the default Chemkin mechanism data directory)
data_dir = os.path.join(ck.ansys_dir, "reaction", "data")
mechanism_dir = data_dir
# create a chemistry set based on GRI 3.0
MyGasMech = ck.Chemistry(label="GRI 3.0")
# set mechanism input files
```

(continues on next page)

(continued from previous page)

```
# including the full file path is recommended
MyGasMech.chemfile = os.path.join(mechanism_dir, "grimech30_chem.inp")
MyGasMech.thermfile = os.path.join(mechanism_dir, "grimech30_thermo.dat")
# transport data is not needed
```

Preprocess the chemistry set

```
# preprocess the mechanism files
iError = MyGasMech.preprocess()
```

Set up gas mixtures

Set up gas mixtures based on the species in this chemistry set. PyChemkin provides a few methods for creating a gas mixture. Here, the `isothermal_mixing()` method is used to create a `fuel-air` mixture by mixing the `fuel` and `air` mixtures with a predetermined air/fuel rate.

Create a fuel mixture

Create a fuel mixture of 80% methane and 20% hydrogen.

```
fuel = ck.Mixture(MyGasMech)
# set mole fraction
fuel.X = [("CH4", 0.8), ("H2", 0.2)]
fuel.temperature = 300.0
fuel.pressure = ck.Patm  # 1 atm
```

Create an air mixture

Create an air mixture of oxygen and nitrogen.

```
air = ck.Mixture(MyGasMech)
# set mass fraction
air.Y = [("O2", 0.23), ("N2", 0.77)]
air.temperature = 300.0
air.pressure = ck.Patm  # 1 atm
```

Create a fuel-air mixture by mixing

Use the `isothermal_mixing()` method to mix the `fuel` and `air` mixtures created earlier. Define the mixing formula using `mixture_recipe` with this mass ratio: `fuel:air=1.00:17.19`. Use the `finaltemperature` parameter to set the temperature of the new `premixed` mixture to 300 [K]. Set `mode="mass"` because the ratios given in `mixture_recipe` are mass ratios.

```
mixture_recipe = [(fuel, 1.0), (air, 17.19)]
# create the new mixture (the air-fuel ratio is by mass)
premixed = ck.isothermal_mixing(
    recipe=mixture_recipe, mode="mass", finaltemperature=300.0
)
```

Find the equilibrium composition at different temperatures

Perform the equilibrium calculation with fixed pressure and fixed temperature. The NO mole fraction of the equilibrium state at each temperature is stored in an array. The gas temperature is increased from 500 to 2480 [K].

```
# NO species index
NO_index = MyGasMech.get_specindex("NO")

# set up plotting temperatures
Temp = 500.0
dTemp = 20.0
points = 100
# curve
T = np.zeros(points, dtype=np.double)
NO = np.zeros_like(T, dtype=np.double)
# start the temperature loop
for k in range(points):
    # reset mixture temperature
    premixed.temperature = Temp
    # find the equilibrium state mixture at the given mixture temperature and pressure
    eqstate = ck.equilibrium(premixed, opt=1)
    #
    NO[k] = eqstate.X[NO_index] * 1.0e6  # convert to ppm
    T[k] = Temp
    Temp += dTemp


##################
# Plot the results
# ================
# Plot the equilibrium NO mole fractions versus the temperatures
# of the fuel-air mixtures.

plt.plot(T, NO, "bs--", markersize=3, markevery=4)
plt.xlabel("Temperature [K]")
plt.ylabel("NO [ppm]")
# plot results
if interactive:
    plt.show()
else:
    plt.savefig("plot_equilibrium_composition.png", bbox_inches="tight")
```

12.2.5 Calculate the heating values of fuel mixtures

One of the advantages of PyChemkin is flexibility. You can establish your own workflow with PyChemkin to meet your simulation goals. This tutorial is an example of building a specialized algorithm to calculate the *heating values*, the lower heating value (LHV) and the higher heating value (HHV), of fuel mixtures.

The **heating value** is the net heat release from the complete combustion of a hydrocarbon (HC) in oxygen at the standard state condition (298.15 [K] and 1 [atm]). The combustion products are mainly CO_2 and H_2O, and are assumed to be cooled back to the standard state condition in the heating value calculation. The difference between the lower heating value and the higher heating value is the final form of the H_2O; the water is in *gas phase* for the *lower* heating value, and in *liquid phase* for the *higher* heating value. You can see that the higher heating value can be obtained by adding the water *heat of vaporization* to the lower heating value. Thus, the workflow for calculating the heating values of any HC fuels consists of two main steps:

1. Calculating the water heat of vaporization at the standard state condition: this can be done by getting the value from a well trusted database or by using the **chemkin** trick described below.

2. Calculating the heat of combustion of the fuel mixture in pure oxygen: here you will use the `Find_Equilibrium` method with the *fixed pressure* and *fixed temperature* option (the default setting).

The lower heating value is the enthalpy different between the fresh fuel and oxygen mixture and the final product mixture. Adding the heat of vaporization to the lower heating value, and you get the higher heating value.

In this tutorial, you will compute the heating values of some pure fuel species such as methane and n-butane as well as some fuel mixtures such as PRF RON 80 and biodiesel. You can compare the values you get here with the known values from a trusty database.

Import PyChemkin package and start the logger

```
import os

import ansys.chemkin as ck
from ansys.chemkin import Color
from ansys.chemkin.logger import logger
from ansys.chemkin.utilities import find_file
import numpy as np  # number crunching

# check working directory
current_dir = os.getcwd()
logger.debug("working directory: " + current_dir)
# set mechanism directory (the default Chemkin mechanism data directory)
data_dir = os.path.join(ck.ansys_dir, "reaction", "data")
logger.debug("data directory: " + data_dir)
# set pressure & temperature condition of the standard state
thispressure = ck.Patm
thistemperature = 298.15
```

Calculate the heat of vaporization

Because the thermodynamic data (mainly the enthalpy) of both the vapor and the liquid water are available in the standard thermodynamic data file `therm.dat` that comes with the Ansys Chemkin in the *reaction/data* directory, you can, in theory, compute the water heat of vaporization at a given temperature by finding the enthalpy difference between the water vapor and its liquid counterpart. In the `getwaterheatofvaporization` method, a mechanism with just the water vapor `H2O` and the liquid water `H2O(L)` is created in situ. Simply find and return the enthalpy difference of the two species by using the `SpecisH` method after preprocessing the `WaterMech` `Chemistry Set`.

```
def getwaterheatofvaporization(temp: float) -> float:
    """
    Compute water heat of vaporization [erg/g-water] at the given temperature
    Use the enthalpy difference between water vapor and liquid water at the temperature
    Enthalpy data depend on temperature only
    There are empirical formulas for heat of vaporization, for example, DIPPR EQ.

    Parameters
    ----------
        temp: double scalar
```

(continues on next page)

(continued from previous page)

```
        water temperature [K]

    Returns
    -------
        enthalpy: double
            water enthalpy of vaporization [erg/g-water]
    """
    # compute water heat of vaporization
    # create a chemistry set object
    WaterMech = ck.Chemistry(label="Water Only")
    #
    # create a new mechanism input file
    #
    waterfile = os.path.join(current_dir, "water_chem.inp")
    w = open(waterfile, "w")
    # the water mechanism contains two species:
    # water vapor (H2O) and liquid water (H2O(L))
    # decalre elements
    w.write("ELEMENT H O END\n")
    w.write("SPECIES\n")
    w.write("H2O   H2O(L)\n")
    w.write("END\n")
    # no reaction is needed for thermodynamic property calculation
    w.write("REACTION\n")
    w.write("END\n")
    # close the mechnaism file
    w.close()
    # set mechanism input files
    # including the full file path is recommended
    WaterMech.chemfile = waterfile
    WaterMech.thermfile = os.path.join(
        data_dir,
        "therm.dat",
    )
    # pre-process
    iError = WaterMech.preprocess()
    if iError != 0:
        return 0.0
    # get species enthalpies [erg/mol] at 298.15 [K]
    waterenthalpies = WaterMech.SpeciesH(thistemperature)
    # compute heat of vaporization of water [erg/g-water]
    # H_water_vapor - H_liquid_water
    heatvaporization = (waterenthalpies[0] - waterenthalpies[1]) / WaterMech.WT[0]
    # remove the temporary water mechanism file
    os.remove(waterfile)
    return heatvaporization
```

Create your own fuel library

You can create a fuel "library" mechanism that contains typical fuel species, oxygen, and the complete combustion products o(carbon dioxide and water *vapor*) only. No reaction is needed for this purpose as you will perform equilibrium calculation to get the complete combustion product mixture.

Warning

It is recommended **not** to include any intermediate species such as CH3 or OH in the fuel library as these species might interfere with the equilibrium calculation used to determine the complete combustion products.

Note

The thermodynamic data file `Gasoline-Diesel-Biodiesel_PAH_NOx_therm_MFL2024.dat` should have the property data of most commonly seen fuel species. This encrypted data file is developed under the *Model Fuel Library* project and is available in the *reaction/data/ModelFuelLibrary*.

Instantiate the fuel `Chemistry Set`

Create the chemistry set object `MyGasMech`. The mechanism input file `fuel_chem.inp` will be created below.

Note

You can keep this file for later uses, for example, by adding more fuel species. (remember to remove the``remove(mymechfile))`` command at the end of this project).

```
MyGasMech = ck.Chemistry(label="EQ")
```

Create the fuel mechanism file

create a new mechanism input file

```
mymechfile = os.path.join(current_dir, "fuels_chem.inp")
m = open(mymechfile, "w")
# the mechanism contains only the necessary species (fuel, oxygen, and major combustion␣
↪products)
# decalre elements
m.write("ELEMENT c h o END\n")
# declare species
# ch4: Methane                  c4h10: n-Butane
# nc5h12: n-Pentane             nc7h16: n-Heptane
# ic8h18: iso-Octane            nc9h20: n-Nonane
# nc10h22: n-Decane             hmn: Heptamethylnonane
# c6h5ch3: Toluene              c6h5c2h5: Ethylbenzene
# chx: Cyclohexane              mch: Methylcyclohexane
# decalin: Decalin              ch3oh: Methanol
# c2h5oh: Alcohol               ch3och3: Dimethyl ether (DME)
# nc4h9oh: n-Butanol            mb: Mythyl butanoate
# md: Methyl decanoate          mhd: Methyl stearate
m.write("SPECIES\n")
m.write("ch4 c4h10 nc5h12 nc7h16 ic8h18 nc9h20 nc10h22\n")
m.write("hmn c6h5ch3 c6h5c2h5 chx mch decalin\n")
m.write("ch3oh c2h5oh ch3och3 nc4h9oh\n")
m.write("mb md mhd\n")
m.write("o2 co2 h2o\n")
m.write("END\n")
```

(continues on next page)

(continued from previous page)

```
# no reaction is needed for equilibrium calculation
m.write("REACTION\n")
m.write("END\n")
# close the mechnaism file
m.close()
#
# set mechanism input files
# including the full file path is recommended
# note that the "year" in thermodynamic data file name could be different.
# it's MFL2023 for Ansys Chemkin 2025R1, MFL2024 for 2025R2, ...
MyGasMech.chemfile = mymechfile
therm_dir = os.path.join(
    data_dir,
    "ModelFuelLibrary",
    "Full",
)
MyGasMech.thermfile = find_file(
    therm_dir,
    "Gasoline-Diesel-Biodiesel_PAH_NOx_therm_MFL",
    "dat",
)
```

Pre-process the fuel `Chemistry Set`

preprocess the fuel mechanism "MyGasMech`` just created.

```
iError = MyGasMech.preprocess()
if iError == 0:
    print(Color.GREEN + ">>> preprocess OK", end=Color.END)
else:
    print(Color.RED + ">>> preprocess failed!", end=Color.END)
    exit()
```

Find the water heat of vaporization

Set the fresh fuel-oygen mixture pressure & temperature to the standard state condition for the heating value calculations. Since the difference between the *lower heating value (LHV)* and the *higher heating value (HHV)* is the final form of the water. You will calculate the LHV first and get the HHV by adding the water heat of vaporization to the LHV. Use the `get_specindex` method to find the species index of water `MyGasMech`.

Note

The water heat of vaporization is computed by the `getwaterheatofvaporization` method you created earlier in this project. Reactivate `MyGasMech` (`MyGasMech.activate`) after calling `getwaterheatofvaporization` because another `Chemistry Set` `WaterMech` is used there.

```
# find the index for water vapor
watervaporID = MyGasMech.get_specindex("h2o")

# water heat of vaporization [erg/g-water] at 298.15 [K]
# either call this method before creating the current Chemistry Set
```

(continues on next page)

(continued from previous page)

```
# or use the activate method to switch back to the current Chemistry Set after the call
heatvaporization = getwaterheatofvaporization(thistemperature)

# switch back to MyGasMech
MyGasMech.activate()
```

Set up the unburned fuel-oxygen mixture

Set up the *unburned* "fuel-oxygen" mixture for the heating value calculations. Here the `X_by_Equivalence_Ratio` method with $\phi = 1$ is employed to generate a stoichiometric fuel-oxygen mixture. The advantage of using this method is that you do not need to calculate the "fuel-oxygen" composition for every fuel mixture. You can use a *lean* "fuel-oxygen" mixture, too, because the standard heat of formation of O_2 is zero by definition.

Here you will compute the heating values of four fuel mixtures: CH_4,C_4H_{10}, PRF 80 (20% C_7H_{16} + 80% C_8H_{18}), and a mock-up biodiesel mixture (90% $C_{19}H_{38}O_2$ + 10% CH_3OH).

```
# prepare the fuel mixtures
fuel = ck.Mixture(MyGasMech)
fuel.pressure = thispressure
fuel.temperature = thistemperature
# list of fuel compositions (mole/volume fractions) of which the heating values will be␣
↪computed
# [Methane, n-Butane, PRF RON 80, biodiesel]
fuels = [
    [("ch4", 1.0)],
    [("c4h10", 1.0)],
    [("nc7h16", 0.2), ("ic8h18", 0.8)],
    [("mhd", 0.9), ("ch3oh", 0.1)],
]

# specify oxidizers = pure oxygen
oxid = ck.Mixture(MyGasMech)
oxid.X = [("o2", 1.0)]
oxid.pressure = thispressure
oxid.temperature = thistemperature
# get o2 index
oxyID = MyGasMech.get_specindex("o2")

# specify the complete combustion product species
products = ["co2", "h2o"]
# no added species
add_frac = np.zeros(MyGasMech.KK, dtype=np.double)

# instantiate the unburned fuel-oxygen mixture
unburned = ck.Mixture(MyGasMech)
unburned.pressure = thispressure
unburned.temperature = thistemperature
```

Compute the fuel heating value of the fuel mixtures

Now compute the heating values: LHV and HHV, from the enthalpy different between the unburned stoichiometric fuel-oxygen mixture `unburned` and the complete combustion product mixture `burned`. The complete combustion product mixture is found by using the `Find_Equilibrium` method with the default *fixed pressure* and *fixed temperature* option, that is, the product mixture is also at the standard state condition.

```
LHV = np.zeros(len(fuels), dtype=np.double)
HHV = np.zeros_like(LHV, dtype=np.double)
fuelcount = 0
for f in fuels:
    # re-set the fuel composition (mole/volume fractions)
    fuel.X = f
    # create a soichiometric fuel-oxygen mixture
    iError = unburned.X_by_Equivalence_Ratio(
        MyGasMech, fuel.X, oxid.X, add_frac, products, equivalenceratio=1.0
    )
    # get the mixture enthalpy of the initial mixture [erg/g]
    Hunburned = unburned.HML() / unburned.WTM

    # compute the complete combustion state (fixed temperature and pressure)
    # this step mimics the complete burning of the initial fuel-oxygen mixture at␣
↪constant pressure
    # and the subsequent cooling of the combustion prodcts back to the original␣
↪temperature
    burned = unburned.Find_Equilibrium()
    # get the mixture enthalpy of the final mixture [erg/g]
    Hburned = burned.HML() / burned.WTM

    # get total fuel mass fraction
    fmass = 0.0e0
    bmassfrac = unburned.Y
    for i in range(MyGasMech.KK):
        if i != oxyID:
            fmass += bmassfrac[i]
    # water vapor mass fraction in the urned mixture
    wmass = burned.Y[watervaporID]
    #
    if np.isclose(fmass, 0.0, atol=1.0e-10):
        # no fuel species exists in the unburned mixture
        print(f">>> error no fuel species in the unburned mixture {f}")
        exit()

    # compute the heating values [erg/g-fuel]
    LHV[fuelcount] = -(Hburned - Hunburned) / fmass
    HHV[fuelcount] = -(Hburned - (Hunburned + heatvaporization * wmass)) / fmass
    fuelcount += 1
```

Display the heating values

List the fuel mixtures and their LHV and HHV. The heating values are converted from the cgs units [erg/g] to [kJ/g].

```
print(f"Fuel Heating Values at {thistemperature} [K] and {thispressure*1.0e-6} [bar]\n")
for i in range(len(fuels)):
```

(continues on next page)

(continued from previous page)

```
    print(f"fuel composition:  {fuels[i]}")
    print(f" LHV [kJ/g-fuel]:  {LHV[i] / ck.ergs_per_joule / 1.0e3}")
    print(f" HHV [kJ/g-fuel]:  {HHV[i] / ck.ergs_per_joule / 1.0e3}\n")

##########
# Clean up
# ========
# Delete arrays, mixture objects, and temporary files no longer needed.
del HHV, LHV
del fuel, oxid, unburned, burned
# delete the local mechanism file just created
os.remove(mymechfile)
```

12.2.6 Combine gas mixtures

PyChemkin provides a set of basic mixture utilities enabling the creation of new mixtures from existing ones. The mixing methods let you combine two mixtures at constant pressure with specific constraints.

- The `adiabatic_mixing()` method combines two mixtures while keeping the overall enthalpy constant. The temperature of the combined mixture is determined by the conservation of the overall enthalpy of the two parent mixtures.
- The `isothermal_mixing()` method simply combines two mixtures and determines the composition of the final mixture according to the mole or mass ratios specified. Because this method does not consider the enthalpy conservation, you must assign the temperature value of the combined mixture.

This example shows how to use these two mixing methods and understand the differences in the temperatures of their combined mixtures. It first creates a fuel (CH_4) mixture and an air (O_2+N_2) mixture and then makes a fuel-air mixture by mixing them *isothermally* (that is, without considering the enthalpy conservation) with a given air-to-fuel mass ratio. The example then dilutes the fuel-air mixture with argon (AR) *adiabatically* with the molar/volumetric ratio specified.

```

```

Import PyChemkin packages and start the logger

Import the PyChemkin packages, check the working directory, and start the logger in verbose mode.

```
import os

import ansys.chemkin as ck  # Chemkin
from ansys.chemkin.logger import logger

# check working directory
current_dir = os.getcwd()
logger.debug("working directory: " + current_dir)
# set verbose mode
ck.set_verbose(True)
```

Create a chemistry set

Load the GRI 3.0 mechanism and its associated data files, which come with the standard Ansys Chemkin installation in the `/reaction/data` directory.

```
# set mechanism directory (the default Chemkin mechanism data directory)
data_dir = os.path.join(ck.ansys_dir, "reaction", "data")
mechanism_dir = data_dir
# create a chemistry set based on GRI 3.0
MyGasMech = ck.Chemistry(label="GRI 3.0")
# set mechanism input files
# including the full file path is recommended
MyGasMech.chemfile = os.path.join(mechanism_dir, "grimech30_chem.inp")
MyGasMech.thermfile = os.path.join(mechanism_dir, "grimech30_thermo.dat")
# transport data is not needed
```

Preprocess the chemistry set

Preprocess the mechanism files to prepare the chemistry set.

```
# preprocess the mechanism files
iError = MyGasMech.preprocess()
```

Set up gas mixtures

Set up gas mixtures based on the species in this chemistry set. Use the equivalence ratio method to set up the combustible mixture so that you can easily change the mixture composition by assigning a different equivalence ratio value.

Create a fuel mixture

Create a fuel mixture of 100% methane.

Note

Mixture pressures are not specified here because they are not required by the calculations. The mixing process assumes a fixed pressure, meaning that the mixtures are at the same pressure.

```
fuel = ck.Mixture(MyGasMech)
# set mole fraction
fuel.X = [("CH4", 1.0)]
fuel.temperature = 300.0
```

Create an air mixture

Create an air mixture of oxygen and nitrogen.

```
air = ck.Mixture(MyGasMech)
# set mole fraction
air.X = [("O2", 0.21), ("N2", 0.79)]
air.temperature = 300.0
```

Create a fuel-air mixture by mixing

Use the `isothermal_mixing()` method to mix the `fuel` and `air` mixtures created earlier. Define the mixing formula using `mixture_recipe` with this mass ratio: `fuel:air=1.00:17.19`. Use the `finaltemperature` parameter to set the temperature of the new `premixed` mixture to 300 [K]. Set `mode="mass"` because the ratios given in `mixture_recipe` are mass ratios.

```
# mix the fuel and the air with an air-fuel ratio of 17.19 (almost stoichiometric)
mixture_recipe = [(fuel, 1.0), (air, 17.19)]
# create the new mixture (the air-fuel ratio is by mass)
premixed = ck.isothermal_mixing(
    recipe=mixture_recipe, mode="mass", finaltemperature=300.0
)
```

Display the molar composition of the premixed mixture

Use the `list_composition()` method with `mode="mole"` to list the mole fractions. The molar composition should resemble the stoichiometric methane-air mixture.

```
# list the molar composition
premixed.list_composition(mode="mole")
print()
```

Create a diluent mixture with pure argon

Create an argon mixture to dilute the premixed mixture later. Set the mixture temperature to a temperature of 600 [K].

```
ar = ck.Mixture(MyGasMech)
# species composition
ar.X = [("AR", 1.0)]
# mixture temperature
ar.temperature = 600.0
```

Dilute the fuel-air mixture with argon by adiabatic mixing

Dilute the premixed mixture by mixing 30% argon by volume adiabatically. The `adiabatic_mixing()` method determines the final mixture temperature based on the enthalpy conservation. Set `mode="mole"` to indicate that the ratios in `dilute_recipe` are molar ratios.

```
# create the mixing recipe
dilute_recipe = [(premixed, 0.7), (ar, 0.3)]
# create the diluted mixture
diluted = ck.adiabatic_mixing(recipe=dilute_recipe, mode="mole")
```

Display information for the diluted mixture

Use the `list_composition()` method with `mode="mole"` to display the molar composition. Also display the temperatures of the three mixtures involved in the adiabatic mixing process for verification. The temperature of the diluted mixture should sit in between those of the `premixed` and `ar` mixtures.

```
# list molar composition
diluted.list_composition(mode="mole")
# show the mixture temperatures
print(f"The diluted mixture temperature is  {diluted.temperature:f} [K].")
print(f"The ar mixture temperature is       {ar.temperature:f} [K].")
print(f"The premixed mixture temperature is {premixed.temperature:f} [K].")
```

12.2.7 Rank reaction rates

When you have a mixture in PyChemkin, you can not only obtain its properties but also extract the rate information. The mixture can be any one of the following:

- Set up from scratch.
- Obtained from certain mixture operations.
- Based on a point (time or grid) solution of a reactor simulation.

The mixture rate utilities let you access the net production rate of species (ROP), forward and reverse reaction rates per reaction (RR), and net chemical heat release rate (HRR).

This example shows how to use PyChemkin mixture rate tools to derive useful information from the raw data, such as isolating dominant reactions at different mixture conditions.

Import PyChemkin packages and start the logger

```
import os

import ansys.chemkin as ck  # Chemkin
from ansys.chemkin.logger import logger
import matplotlib.pyplot as plt  # plotting
import numpy as np  # number crunching

# check working directory
current_dir = os.getcwd()
logger.debug("working directory: " + current_dir)
# set verbose mode
ck.set_verbose(True)
# set interactive mode for plotting the results
# interactive = True: display plot
# interactive = False: save plot as a PNG file
global interactive
interactive = True
```

Create a chemistry set

The first mechanism to load is the GRI 3.0 mechanism for methane combustion. This mechanism and its associated data files come with the standard Ansys Chemkin installation in the `/reaction/data` directory.

```
# set mechanism directory (the default Chemkin mechanism data directory)
data_dir = os.path.join(ck.ansys_dir, "reaction", "data")
mechanism_dir = data_dir
# create a chemistry set based on GRI 3.0
MyGasMech = ck.Chemistry(label="GRI 3.0")
# set mechanism input files
# including the full file path is recommended
MyGasMech.chemfile = os.path.join(mechanism_dir, "grimech30_chem.inp")
MyGasMech.thermfile = os.path.join(mechanism_dir, "grimech30_thermo.dat")
MyGasMech.tranfile = os.path.join(mechanism_dir, "grimech30_transport.dat")
```

Preprocess the chemistry set

```
# preprocess the mechanism files
iError = MyGasMech.preprocess()
```

Set up gas mixtures based on the species in this chemistry set

Before you can create a premixed fuel-oxidizer mixture by the equivalence ratio, you must first create fuel and the oxidizer mixtures.

Create the fuel mixture

The fuel mixture consists of 100% methane.

```
fuelmixture = ck.Mixture(MyGasMech)
# set fuel composition
fuelmixture.X = [("CH4", 1.0)]
# setting pressure and temperature is not required in this case
fuelmixture.pressure = 5.0 * ck.Patm
fuelmixture.temperature = 1500.0
```

Create the air mixture

The air mixture consists of oxygen and nitrogen. The temperature and the pressure of this mixture are set to the values of the fuel mixture.

```
air = ck.Mixture(MyGasMech)
air.X = [("O2", 0.21), ("N2", 0.79)]
# setting pressure and temperature is not required in this case
air.pressure = 5.0 * ck.Patm
air.temperature = 1500.0
```

Define the combustion products and additives

To use the equivalence ratio method, you must define the complete combustion products and the composition of the additives.

```
# products from the complete combustion of the fuel and air mixtures
products = ["CO2", "H2O", "N2"]
# Specify mole fractions of the added/inert mixture. You can also create an additives␣
↪mixture here.
add_frac = np.zeros(MyGasMech.KK, dtype=np.double)  # no additives: all zeros
```

Create a placeholder for the fuel-air mixture

You can create an empty mixture object that is fully defined later. The composition of the `premixed` mixture is determined when calling one of the mixture composition methods:

- `X()`
- `Y()`
- `X_byEquivalence_Ratio()`
- `Y_byEquivalence_Ratio()`

When calling the `X_byEquivalence_Ratio()` or `Y_byEquivalence_Ratio()` method, the equivalence ratio is specified by this parameter: `equivalenceratio=1.0`.

```
# create the premixed mixture to define
premixed = ck.Mixture(MyGasMech)
# define the actual composition by the equivalence ratio
iError = premixed.X_by_Equivalence_Ratio(
    MyGasMech, fuelmixture.X, air.X, add_frac, products, equivalenceratio=1.0
)
if iError != 0:
    # check fuel-air mixture creation status
    print("Error: Failed to create the fuel-air mixture.")
    exit()
```

Display the molar composition of the premixed mixture

List the composition of the premixed mixture for verification.

```
premixed.list_composition(mode="mole")
```

Evaluate reaction rates at a given mixture temperature and pressure

Compute and rank the *net* reaction rates in the `MyGasMech` chemistry set at 1600 [K] and 5 [atm]. The *net* rate of a reaction is computed by summing the forward rate `kf` and the reverse rate `kr`. The `RxnRates` rate utility returns both the forward and the reverse rates of each reaction in the mechanism.

Get the net production rates of *all* species by using the `ROP` method. The ROP values obtained are in [mole/cm^3-sec]. Alternatively, you can use the `massROP` method to get mass-unit ROP values in [g/cm^3-sec].

Use the optional `threshold` parameter to get only the ROP values with absolute value above the given threshold value. For example, `premixed.ROP(threshold=1.0e-8)` returns a `rop` array in which any raw ROP value with an absolute value less than 1.0e-8 is set to zero. By default, `threshold=0.0`, in which case all raw ROP values are returned.

Note

Temperature and pressure are required to compute the reaction rates.

```
# set the temperature and the pressure of the premixed mixture
premixed.pressure = 5.0 * ck.Patm
premixed.temperature = 1600.0
```

Obtain and sort the rates of production of all species

Use the `ROP()` method to get the rates of production (ROP) of all species in the `MyGasMech` chemistry set. You can use the `list_ROP()` method to list and sort the ROP values. `spec_rate_order` contains the sorted species indices in descending order. `species_rates` contains the sorted ROP values.

```
rop = premixed.ROP()
# list the non-zero rates in descending order
print()
spec_rate_order, species_rates = premixed.list_ROP()
```

Evaluate and sort the rates of production of all species

Use the `RxnRates()` method to get the forward rate (`kf`) and reverse rate (`kr`) of each reaction. The `list_reaction_rates()` method computes the net rate of each reaction and sorts them in ascending or descending order. The net reaction rate is computed by summing the forward and reverse rates of each reaction. `rxn_order` is a list of the ranked reaction indices. `net_rxn_rates` contains the net reaction rates in the same order.

Note

The species and the reactions indexes returned by the `list_ROP()` and `list_reaction_rates()` methods are 0-based indexes. Increment the returned index by 1 to get the actual 1-based index.

```
kf, kr = premixed.RxnRates()
print()
print(f"Reverse reaction rates: (raw values of all {MyGasMech.IIGas:d} reactions)")
print(str(kr))
print("=" * 40)
# list the non-zero net reaction rates
rxn_order, net_rxn_rates = premixed.list_reaction_rates()
```

Plot the sorted reaction rates at 1600 [K]

Display the top net reaction rates and their reaction strings in the commonly used horizontal bar plot. Use the `get_gas_reaction_string` utility of the chemistry set to get the actual reaction for the Y-axis label.

```
# create a rate plot
plt.rcParams.update({"figure.autolayout": True})
plt.subplots(2, 1, sharex="col", figsize=(10, 5))
# convert reaction # from integers to strings
rxnstring = []
for i in range(len(rxn_order)):
    # the array index starting from 0 so the actual reaction # = index + 1
    rxnstring.append(MyGasMech.get_gas_reaction_string(rxn_order[i] + 1))
# use horizontal bar chart
plt.subplot(211)
plt.barh(rxnstring, net_rxn_rates, color="blue", height=0.4)
# use log scale on x axis
plt.xscale("symlog")
# plt.ylabel('reaction')
plt.text(-3.0e-4, 0.5, "T = 1600K", fontsize=10)
```

Evaluate reaction rates at a given mixture temperature and pressure

Compute and rank the net reaction rates in the `MyGasMech` chemistry set at 1800 [K] and 5 [atm].

```
# change the mixture temperature
premixed.temperature = 1800.0
```

Evaluate and sort the rates of production of all species at 1800 [K]

Get the list of non-zero net reaction rates at the new temperature.

```
rxn_order, net_rxn_rates = premixed.list_reaction_rates()
```

Plot the sorted reaction rates at 1800 [K]

Display the top net reaction rates and their reaction strings in the commonly used horizontal bar plot.

```
plt.subplot(212)
# convert reaction # from integers to strings
rxnstring.clear()
for i in range(len(rxn_order)):
    # the array index starting from 0 so the actual reaction # = index + 1
    rxnstring.append(MyGasMech.get_gas_reaction_string(rxn_order[i] + 1))
plt.barh(rxnstring, net_rxn_rates, color="orange", height=0.4)
plt.xlabel("reaction rate [mole/cm3-sec]")
# plt.ylabel('reaction')
plt.text(-3.0e-4, 0.5, "T = 1800K", fontsize=10)
# use log scale on x axis
plt.xscale("symlog")

# plot both ranking results
if interactive:
    plt.show()
else:
    plt.savefig("plot_reaction_rates.png", bbox_inches="tight")
```

12.3 Batch reactor

The *batch reactor* is an idealized 0-D transient closed reactor model in which the gas inside the reactor is assumed to be homogeneous. When the pressure of the *batch reactor* is constrained, the reactor volume would vary to conserve the total gas mass. Likewise, the reactor pressure might change when the reactor volume is "given". The gas temperature can be "given" by piecewise linear profile data or solved by the energy equation.

The examples consist of projects that utilize the *batch reactor* model to perform simulations of various complexities, from tracking the evolution of the mixture properties to the A-factor sensitivity analysis.

12.3.1 Simulate hydrogen combustion in a constant-pressure reactor

Ansys Chemkin offers some idealized reactor models commonly used for studying chemical processes and for developing reaction mechanisms. The batch reactor is a transient 0-D numerical portrayal of the closed homogeneous/perfectly mixed gas-phase reactor. There are two basic types of batch reactor models:

- **constrained-pressure**
- **constrained-volume**

You can choose either to specify the reactor temperature (as a fixed value or by a piecewise-linear profile) or to solve the energy conservation equation for each reactor type. In total, you get four variations out of the base batch reactor model.

This example models the ignition of a stoichiometric hydrogen-air mixture ($\phi = 1$) in a balloon (constant-pressure) reactor. The reactor is created as an instance of the `GivenPressureBatchReactor_EnergyConservation` object. The initial gas mixture in the batch reactor is set by the properties of the hydrogen-air mixture. You get the ignition delay

time (if auto-ignition of the hydrogen-air mixture occurs during the simulation time) and plot the predicted profiles of gas temperature, gas density, H_2O mole fraction, and the net production rate of H_2O.

```
```

Import PyChemkin packages and start the logger

```
import os

import ansys.chemkin as ck  # Chemkin
from ansys.chemkin import Color

# chemkin batch reactor models (transient)
from ansys.chemkin.batchreactors.batchreactor import (
    GivenPressureBatchReactor_EnergyConservation,
)
from ansys.chemkin.logger import logger
import matplotlib.pyplot as plt  # plotting
import numpy as np  # number crunching

# check working directory
current_dir = os.getcwd()
logger.debug("working directory: " + current_dir)
# set verbose mode
ck.set_verbose(True)
# set interactive mode for plotting the results
# interactive = True: display plot
# interactive = False: save plot as a PNG file
global interactive
interactive = True
```

Create a chemistry set

The mechanism to load is the GRI 3.0 mechanism for methane combustion. This mechanism and its associated data files come with the standard Ansys Chemkin installation in the `/reaction/data` directory.

```
# set mechanism directory (the default Chemkin mechanism data directory)
data_dir = os.path.join(ck.ansys_dir, "reaction", "data")
mechanism_dir = data_dir
# create a chemistry set based on the diesel 14-components mechanism
MyGasMech = ck.Chemistry(label="GRI 3.0")
# set mechanism input files
# including the full file path is recommended
MyGasMech.chemfile = os.path.join(mechanism_dir, "grimech30_chem.inp")
MyGasMech.thermfile = os.path.join(mechanism_dir, "grimech30_thermo.dat")
MyGasMech.tranfile = os.path.join(mechanism_dir, "grimech30_transport.dat")
```

Preprocess the chemistry set

```
# preprocess the mechanism files
iError = MyGasMech.preprocess()
```

Set up gas mixtures based on the species in this chemistry set

Compose the species molar ratios of the fuel-air mixture in a recipe list and use the `fuelmixture.X()` method to set the mixture composition (mole fractions) directly.

Since you are going to use the `fuelmixture` mixture to instantiate the reactor object later, setting the mixture pressure and temperature is equivalent to setting the initial temperature and pressure of the batch reactor.

```
# create the fuel mixture
fuelmixture = ck.Mixture(MyGasMech)
# set fuel composition (mole ratios)
fuelmixture.X = [("H2", 2.0), ("N2", 3.76), ("O2", 1.0)]
# setting the mixture pressure and temperature is equivalent to setting
# the initial temperature and pressure of the reactor in this case
fuelmixture.pressure = ck.Patm
fuelmixture.temperature = 1000

# list the composition of the premixed mixture
# this serves as the baseline for verification later
fuelmixture.list_composition(mode="mole")
```

Set up a constant-pressure batch reactor (with energy equation)

Create the constant-pressure batch reactor as an instance of the `GivenPressureBatchReactor_EnergyConservation` object because the reactor pressure is kept constant (or assigned as a function of time). The batch reactors must be associated with a mixture, which implicitly links the chemistry set (gas-phase mechanism and properties) to the batch reactor. Additionally, it defines the initial conditions (pressure, temperature, volume, and gas composition) of the batch reactor.

```
MyCONP = GivenPressureBatchReactor_EnergyConservation(fuelmixture, label="tran")
```

List the mixture composition

List the initial gas composition inside the reactor for verification. You can use the composition printout from the `fuelmixture` mixture to verify that the initial gas composition inside `MyCONP` is set correctly, that is, `MyCONP` has been initialized by the `fuelmixture` mixture.

```
MyCONP.list_composition(mode="mole")
```

Set up additional reactor model parameters

Before you can run the simulation, you must provide reactor parameters, solver controls, and output instructions. For a batch reactor, the initial volume and the simulation end time are required inputs.

```
# set other reactor properties
# reactor volume [cm3]
MyCONP.volume = 1
MyCONP.temperature = 1000
# simulation end time [sec]
MyCONP.time = 0.0005
```

Set output options

You can turn on adaptive solution saving to resolve the steep variations in the solution profile. Here, additional solution data points are saved for every 20 internal solver steps. You must include the `set_ignition_delay()` method for the reactor model to report the ignition delay times after the simulation is done. If `method="T_rise"` is set, the reactor model considers the gas is auto-ignited when the predicted gas temperature goes above the initial temperature by the amount indicated by the parameter `val=400`. You can choose a different auto-ignition definition.

Note

- Type `ansys.chemkin.show_ignition_definitions()` to get the list of all available ignition delay time definitions in Chemkin.
- By default, time intervals for both print and save solution are 1/100 of the simulation end time, which in this example is $dt = time/100 = 0.001$. You can change them to different values.

```
# turn on adaptive solution saving
MyCONP.adaptive_solution_saving(mode=True, steps=20)
# specify the ignition definitions
ck.show_ignition_definitions()
MyCONP.set_ignition_delay(method="T_rise", val=400)
```

Set solver controls

You can overwrite the default solver controls by using solver-related methods, such as those for tolerances.

```
# set tolerances in tuple: (absolute tolerance, relative tolerance)
MyCONP.tolerances = (1.0e-20, 1.0e-8)

# get solver parameters
ATOL, RTOL = MyCONP.tolerances
print(f"Default absolute tolerance = {ATOL}.")
print(f"Default relative tolerance = {RTOL}.")
# turn on the force non-negative solutions option in the solver
MyCONP.force_nonnegative = True
# show solver option
print(f"Timestep between solution printing: {MyCONP.timestep_for_printing_solution}")
# show timestep between printing solution
print(f"Forced non-negative solution values: {MyCONP.force_nonnegative}")
```

Display the added parameters (keywords)

Use the `showkeywordinputlines()` method to verify that the preceding parameters are correctly assigned to the reactor model.

```
MyCONP.showkeywordinputlines()
```

Run the simulation

Use the `run()` method to start the batch reactor simulation.

```
runstatus = MyCONP.run()
```

(continues on next page)

(continued from previous page)

```
# check run status
if runstatus != 0:
    # Run failed.
    print(Color.RED + ">>> Run failed. <<<", end="\n" + Color.END)
    exit()
# Run succeeded.
print(Color.GREEN + ">>> Run completed. <<<", end="\n" + Color.END)
```

Get the ignition delay time from the solution

Use the `get_ignition_delay()` method to extract the ignition delay time after the run is completed.

```
delaytime = MyCONP.get_ignition_delay()
print(f"Ignition delay time = {delaytime} [msec].")
```

Postprocess the solution

The postprocessing step parses the solution and packages the solution values at each time point into a mixture object. There are two ways to access the solution profiles:

- The raw solution profiles (value as a function of time) are available for time, temperature, pressure, volume, and species mass fractions.
- The mixture objects that permit the use of all property and rate utilities to extract information such as viscosity, density, species production rates, and mole fractions.

To obtain the raw solution profiles, you can use the `get_solution_variable_profile()` method. To obtain the solution mixture objects, you can use either the `get_solution_mixture_at_index()` method for the solution mixture at a given time point or the `get_solution_mixture()` method for the solution mixture at a given time. (In this case, the mixture is constructed by # interpolation.)

Note

Use the `getnumbersolutionpoints()` method to get the size of the solution profiles before creating the arrays.

```
MyCONP.process_solution()
# get the number of solution time points
solutionpoints = MyCONP.getnumbersolutionpoints()
print(f"Number of solution points = {solutionpoints}.")

# get the time profile
timeprofile = MyCONP.get_solution_variable_profile("time")
# get the temperature profile
tempprofile = MyCONP.get_solution_variable_profile("temperature")
# more involving postprocessing by using mixtures
# create arrays for H2O mole fraction, H2O ROP, and mixture density
H2Oprofile = np.zeros_like(timeprofile, dtype=np.double)
H2OROPprofile = np.zeros_like(timeprofile, dtype=np.double)
denprofile = np.zeros_like(timeprofile, dtype=np.double)
CurrentROP = np.zeros(MyGasMech.KK, dtype=np.double)
# find H2O species index
H2O_index = MyGasMech.get_specindex("H2O")
```

(continues on next page)

(continued from previous page)

```
# loop over all solution time points
for i in range(solutionpoints):
    # get the mixture at the time point
    solutionmixture = MyCONP.get_solution_mixture_at_index(solution_index=i)
    # get gas density [g/cm3]
    denprofile[i] = solutionmixture.RHO
    # reactor mass [g]
    # get H2O mole fraction profile
    H2Oprofile[i] = solutionmixture.X[H2O_index]
    # get H2O ROP profile
    currentROP = solutionmixture.ROP()
    H2OROPprofile[i] = currentROP[H2O_index]
```

Plot the solution profiles

```
# plot the profiles
plt.subplots(2, 2, sharex="col", figsize=(12, 6))
plt.subplot(221)
plt.plot(timeprofile, tempprofile, "r-")
plt.ylabel("Temperature [K]")
plt.subplot(222)
plt.plot(timeprofile, H2Oprofile, "b-")
plt.ylabel("H2O Mole Fraction")
plt.subplot(223)
plt.plot(timeprofile, denprofile, "m-")
plt.xlabel("time [sec]")
plt.ylabel("Mixture Density [g/cm3]")
plt.subplot(224)
plt.plot(timeprofile, H2OROPprofile, "g-")
plt.xlabel("time [sec]")
plt.ylabel("H2O Production Rate [mol/cm3-sec]")
# display the plots
if interactive:
    plt.show()
else:
    plt.savefig("plot_close_homogeneous.png", bbox_inches="tight")
```

12.3.2 Predict the ignition delay time of a combustible mixture

One of the important properties of hydrocarbon fuels is the *ignition delay time*. For automobiles, the gasoline, a mixtures of many fuel species, is graded by the *octane number* which is closely related to the fuel mixture's ignition characteristics. Higher octane number fuel tends to ignite more easily. However, this does not mean you should blindly put the highest octane number fuel into your car. Car engines are designed for specific operating conditions, and you should always use the gasoline grades recommended by the car/engine manufacturer to prevent damages to the engine.

This tutorial employs the **constant-pressure batch reactor model** (with the energy equation **ON**) to predict the *auto-ignition delay time* of a **Primary Reference Fuel (PRF)**. PRF is created for automobile fuel researches and consists of two major components: n-heptane C_7H_{16} and iso-octane C_8H_{18}. By definition, n-heptane is assigned with an octane number of 0, and an octane number of 100 for iso-octane. Thus, a PRF of 20% n-heptane and 80% iso-octane has an octane number of 80. In this tutorial, a PRF 60 fuel is used to show the **negative temperature coefficient (NTC)** behavior in the ignition delay curve. The ignition delay curve is constructed by collecting the predicted auto-ignition delay times with various initial gas conditions. Here, the initial gas temperature is increased from 700 to 1080 [K].

Import PyChemkin package and start the logger

```
import os
import time

import ansys.chemkin as ck  # chemkin
from ansys.chemkin import Color

# chemkin batch reactor models (transient)
from ansys.chemkin.batchreactors.batchreactor import (
    GivenPressureBatchReactor_EnergyConservation,
)
from ansys.chemkin.logger import logger
import matplotlib.pyplot as plt  # plotting
import numpy as np  # number crunching

# check working directory
current_dir = os.getcwd()
logger.debug("working directory: " + current_dir)
# set verbose mode
ck.set_verbose(False)
# set interactive mode for plotting the results
# interactive = True: display plot
# interactive = False: save plot as a PNG file
global interactive
interactive = True
```

Create a chemistry set

For PRF, the encrypted 14-component gasoline mechanism, `gasoline_14comp_WBencrypted.inp`, is used. `gasoline` is the name given to this `Chemistry Set`.

Note

This gasoline mechanism does not come with any transport data so you do not need to provide the transport data file.

```
# set mechanism directory (the default Chemkin mechanism data directory)
data_dir = os.path.join(ck.ansys_dir, "reaction", "data")
mechanism_dir = data_dir
# create a chemistry set based on GRI 3.0
gasoline = ck.Chemistry(label="gasoline 14comp")
# set mechanism input files
# including the full file path is recommended
gasoline.chemfile = os.path.join(mechanism_dir, "gasoline_14comp_WBencrypt.inp")
```

Pre-process the gasoline `Chemistry Set`

```
# preprocess the mechanism files
iError = gasoline.preprocess()
```

Set up the stoichiometric gasoline-air mixture

You need to set up the stoichiometric gasoline-air mixture for the subsequent ignition delay time calculations. Here the `X_by_Equivalence_Ratio``method is used. You create the ``fuel` and the `air` mixtures first. Then define the *complete combustion product species* and provide the *additives* composition if applicable. And finally you can simply set `equivalenceratio=1` to create the stoichiometric gasoline-air mixture.

For PRF 60 gasoline, the `recipe` is `[("ic8h18", 0.6), ("nc7h16", 0.4)]`.

```
# create the fuel mixture
fuelmixture = ck.Mixture(gasoline)
# set fuel = composition of PRF 60
fuelmixture.X = [("ic8h18", 0.6), ("nc7h16", 0.4)]
# setting pressure and temperature
fuelmixture.pressure = 5.0 * ck.Patm
fuelmixture.temperature = 1500.0

# create the oxidizer mixture: air
air = ck.Mixture(gasoline)
air.X = [("o2", 0.21), ("n2", 0.79)]
# setting pressure and temperature
air.pressure = 5.0 * ck.Patm
air.temperature = 1500.0

# products from the complete combustion of the fuel mixture and air
products = ["co2", "h2o", "n2"]
# species mole fractions of added/inert mixture. can also create an additives mixture␣
↪here
add_frac = np.zeros(gasoline.KK, dtype=np.double)  # no additives: all zeros

# create the premixed mixture to be defined
premixed = ck.Mixture(gasoline)
# define the composition by the equivalence ratio
iError = premixed.X_by_Equivalence_Ratio(
    gasoline, fuelmixture.X, air.X, add_frac, products, equivalenceratio=1.0
)
# check fuel-oxidizer mixture creation status
if iError != 0:
    print("Error: Failed to create the fuel-oxidizer mixture.")
    exit()
```

List the mixture composition

list the composition of the premixed mixture for verification

```
premixed.list_composition(mode="mole")
# set mixture temperature and pressure (equivalent to setting the initial temperature␣
↪and pressure of the reactor)
```

(continues on next page)

(continued from previous page)

```
premixed.pressure = 40.0 * ck.Patm
premixed.temperature = 700.0
```

Create the reactor object for ignition delay time calculations

Use the `GivenPressureBatchReactor_EnergyConservation` method to instantiate a *constant pressure batch reactor that also includes the energy equation*. This `ReactorModel` method requires a `Mixture` object as an input parameter. This `Mixture` input serves two purposes: passing the `Chemistry Set` to be used by the reactor model and setting the *initial condition* (pressure, temperature, and gas composition) of the simulation. You should use the `premixed` mixture you just created.

```
MyCONP = GivenPressureBatchReactor_EnergyConservation(premixed, label="CONP")
```

Set up additional reactor model parameters

Reactor parameters, *solver controls*, and *output instructions* need to be provided before running the simulations. For a batch reactor, the *initial volume* and the *simulation end time* are required inputs.

```
# verify initial gas composition inside the reactor
MyCONP.list_composition(mode="mole")

# set other reactor parameters
# initial reactor volume [cm3]
MyCONP.volume = 10.0
# simulation end time [sec]
MyCONP.time = 1.0
```

Set output options

You can turn on the *adaptive solution saving* to resolve the steep variations in the solution profile. Here additional solution data point will be saved for every **100 [K]** change in gas **temperature**. The `set_ignition_delay` method must be included for the reactor model to report the *ignition delay times* after the simulation is done. If `method="T_inflection"` is set, the reactor model will treat the *inflection points* in the predicted gas temperature profile as the indication of an auto-ignition. You can choose a different auto-ignition definition.

Note

Type `ansys.chemkin.show_ignition_definitions()` to get the list of all available ignition delay time definitions in Chemkin.

Note

By default, time intervals for both print and save solution are **1/100** of the *simulation end time*. In this case $dt = time/100 = 0.001$. You can change them to different values.

```
# set timestep between saving solution
MyCONP.timestep_for_saving_solution = 0.001
# change timestep between saving solution
MyCONP.timestep_for_saving_solution = 0.01
```

(continues on next page)

(continued from previous page)

```
# turn ON adaptive solution saving
MyCONP.adaptive_solution_saving(mode=True, value_change=100, target="TEMPERATURE")
# set ignition delay
MyCONP.set_ignition_delay(method="T_inflection")
```

Set solver controls

You can overwrite the default solver controls by using solver related methods, for example, `tolerances`.

```
# tolerances are given in tuple: (absolute tolerance, relative tolerance)
MyCONP.tolerances = (1.0e-10, 1.0e-8)
# stop after ignition is detected (not recommended for ignition delay time calculations)
# MyCONP.stop_after_ignition()
# show solver option
print(f"timestep between solution printing: {MyCONP.timestep_for_printing_solution}")
# show timestep between printing solution
print(f"forced non-negative solution values: {MyCONP.force_nonnegative}")
```

Run the parameter study

Construct the ignition delay curve by varying the initial reactor temperature in 20 [K] increments from 700 to 1080 [K]. Use the `get_ignition_delay` method to extract the ignition delay times after finishing each run.

```
# loop over initial reactor temperature to create an ignition delay time plot
npoints = 20
delta_temp = 20.0
init_temp = premixed.temperature
delaytime = np.zeros(npoints, dtype=np.double)
temp_inv = np.zeros_like(delaytime, dtype=np.double)
# set the start wall time
start_time = time.time()
# loop over all cases with different initial gas/reactor temperatures
for i in range(npoints):
    # update the initial reactor temperature
    MyCONP.temperature = init_temp  # K
    # show the additional keywords given by user (verify inputs before running the␣
↪simulation)
    # MyCONP.showkeywordinputlines()
    # run the reactor model
    runstatus = MyCONP.run()
    #
    if runstatus == 0:
        # plot 1/T instead of T
        temp_inv[i] = 1.0e0 / init_temp
        # get ignition delay time
        delaytime[i] = MyCONP.get_ignition_delay()
        print(f"ignition delay time = {delaytime[i]} [msec]")
    else:
        # if get this, most likely the END time is too short
        print(Color.RED + ">>> Run failed. <<<", end=Color.END)
    init_temp += delta_temp
```

(continues on next page)

(continued from previous page)

```
# compute the total runtime
runtime = time.time() - start_time
print(f"total simulation duration: {runtime} [sec] for {npoints} cases")
```

Plot the ignition delay curve

The ignition delay curve is customarily plotted against the inverse temperature. The corresponding temperature values are shown on the top axis.

Intuitively, you expect that the reactivity increases as the temperature increases. However, as you can see, in this case, the ignition delay curve does not vary monotonically with the temperature. Between 850 and 950 [K], the ignition delay time actually increases slightly (because the reactivity decreases). This is regarded as the **Negative Temperature Coefficient (NTC)** behavior which is often observed during low-temperature oxidation of large hydrocarbon fuels.

```
# create an ignition delay versus 1/T plot for the PRF fuel (should exhibit the NTC␣
↪region)
plt.rcParams.update({"figure.autolayout": True})
fig, ax1 = plt.subplots()
ax1.semilogy(temp_inv, delaytime, "bs--")
ax1.set_xlabel("1/T [1/K]")
ax1.set_ylabel("Ignition delay time [msec]")


# Create a secondary x-axis for T (=1/(1/T))
def one_over(x):
    """Vectorized 1/x, treating x==0 manually"""
    x = np.array(x, float)
    near_zero = np.isclose(x, 0)
    x[near_zero] = np.inf
    x[~near_zero] = 1 / x[~near_zero]
    return x


# the function "1/x" is its own inverse
inverse = one_over
ax2 = ax1.secondary_xaxis("top", functions=(one_over, inverse))
ax2.set_xlabel("T [K]")
# plot results
if interactive:
    plt.show()
else:
    plt.savefig("plot_ignition_delay.png", bbox_inches="tight")
```

12.3.3 Simulate a rapid compression machine

Ansys Chemkin offers some idealized reactor models commonly used for studying chemical processes and for developing reaction mechanisms. The *batch reactor* is a transient 0-D numerical portrayal of the *closed homogeneous/perfectly mixed* gas-phase reactor. There are two basic types of batch reactor models:

- **constrained-pressure**
- **constrained-volume**

You can choose either to specify the reactor temperature (as a fixed value or by a piecewise-linear profile) or to solve

the energy conservation equation for each reactor type. In total, you get four variations out of the base batch reactor model.

Rapid Compression Machine (RCM) is often employed to study fuel auto-ignition at high temperature and high-pressure conditions that are compatible to the engine-operating environments. The fuel-air mixture inside the RCM chamber is at relatively low pressure and temperature initially. The gas mixture is then suddenly compressed causing both the pressure and the temperature of the mixture to rise rapidly. The reactor/chamber pressure is monitored to identify the onset of auto-ignition after the compression stopped. This example models the RCM as a `GivenVolumeBatchReactor_EnergyConservation`, and the compression process is simulated by a predetermined time-volume profile.

Import PyChemkin package and start the logger

```
import os

import ansys.chemkin as ck  # Chemkin
from ansys.chemkin import Color

# chemkin batch reactor models (transient)
from ansys.chemkin.batchreactors.batchreactor import (
    GivenVolumeBatchReactor_EnergyConservation,
)
from ansys.chemkin.logger import logger
import matplotlib.pyplot as plt  # plotting
import numpy as np  # number crunching

# check working directory
current_dir = os.getcwd()
logger.debug("working directory: " + current_dir)
# set verbose mode
ck.set_verbose(True)
# set interactive mode for plotting the results
# interactive = True: display plot
# interactive = False: save plot as a PNG file
global interactive
interactive = True
```

Create a chemistry set

The mechanism to load is the GRI 3.0 mechanism for methane combustion. This mechanism and its associated data files come with the standard Ansys Chemkin installation in the `/reaction/data` directory.

```
# set mechanism directory (the default Chemkin mechanism data directory)
data_dir = os.path.join(ck.ansys_dir, "reaction", "data")
mechanism_dir = data_dir
# create a chemistry set based on the GRI 3.0 mechanism
MyGasMech = ck.Chemistry(label="GRI 3.0")
# set mechanism input files
# including the full file path is recommended
MyGasMech.chemfile = os.path.join(mechanism_dir, "grimech30_chem.inp")
MyGasMech.thermfile = os.path.join(mechanism_dir, "grimech30_thermo.dat")
MyGasMech.tranfile = os.path.join(mechanism_dir, "grimech30_transport.dat")
```

Preprocess the chemistry set

```
# preprocess the mechanism files
iError = MyGasMech.preprocess()
```

Set up gas mixtures based on the species in this chemistry set

Use the *equivalence ratio method* so that you can easily set up the premixed fuel-oxidizer mixture composition by assigning an equivalence ratio value.

```
# create the fuel mixture
fuelmixture = ck.Mixture(MyGasMech)
# set fuel composition
fuelmixture.X = [("CH4", 1.0)]
# setting pressure and temperature is not required in this case
fuelmixture.pressure = 5.0 * ck.Patm
fuelmixture.temperature = 1500.0

# create the oxidizer mixture: air
air = ck.Mixture(MyGasMech)
air.X = [("O2", 0.21), ("N2", 0.79)]
# setting pressure and temperature is not required in this case
air.pressure = 5.0 * ck.Patm
air.temperature = 1500.0

# products from the complete combustion of the fuel mixture and air
products = ["CO2", "H2O", "N2"]
# species mole fractions of added/inert mixture. can also create an additives mixture␣
↪here
add_frac = np.zeros(MyGasMech.KK, dtype=np.double)  # no additives: all zeros

# create the premixed mixture to be defined
premixed = ck.Mixture(MyGasMech)

iError = premixed.X_by_Equivalence_Ratio(
    MyGasMech, fuelmixture.X, air.X, add_frac, products, equivalenceratio=0.7
)
# check fuel-oxidizer mixture creation status
if iError != 0:
    print("Error: Failed to create the fuel-oxidizer mixture.")
    exit()

# list the composition of the premixed mixture for verification
premixed.list_composition(mode="mole")

# set mixture temperature and pressure (equivalent to setting the initial temperature␣
↪and pressure of the reactor)
premixed.temperature = 800.0
premixed.pressure = 3.0 * ck.Patm
```

Set up the rapid-compression machine

Create the rapid-compression machine as an instance of the `GivenVolumeBatchReactor_EnergyConservation` object because the reactor volume is assigned as a function of time. The batch reactors must be associated with a mixture that implicitly links the chemistry set (gas-phase mechanism and properties) to the batch reactor. Additionally, it also defines the initial reactor conditions (pressure, temperature, volume, and gas composition).

```
# create a constant volume batch reactor (with energy equation)
MyCONV = GivenVolumeBatchReactor_EnergyConservation(premixed, label="RCM")
# show initial gas composition inside the reactor
MyCONV.list_composition(mode="mole")
```

Set up additional reactor model parameters

You must provide reactor parameters, solver controls, and output instructions before running the simulations. For a batch reactor, the initial volume and the simulation end time are required inputs.

Note

You can reset the initial reactor temperature by using the `MyCONV.temperature = 800.0` method. In the run output, you see a warning message about the change.

```
# set other reactor properties
# set initial reactor volume [cm3]
MyCONV.volume = 10.0
# simulation end time [sec]
MyCONV.time = 0.1
```

Set the volume profile

Create a time-volume profile by using two arrays. Use the `set_volume_profile()` method to add the profile to the reactor model. The profile data overrides the initial volume value set earlier with the `volume()` method.

```
# number of profile data points
npoints = 3
# position array of the profile data
x = np.zeros(npoints, dtype=np.double)
# value array of the profile data
volprofile = np.zeros_like(x, dtype=np.double)
# set reactor volume data points
x = [0.0, 0.01, 2.0]  # [sec]
volprofile = [10.0, 4.0, 4.0]  # [cm3]
```

Set output options

You can turn on the adaptive solution saving to resolve the steep variations in the solution profile. Here additional solution data points are saved for every **100 [K]** change in gas temperature. The `set_ignition_delay()` method must be included for the reactor model to report the ignition delay times after the simulation is done. If `method="T_inflection"` is set, the reactor model treats the inflection points in the predicted gas temperature profile as the indication of an auto-ignition. You can choose a different auto-ignition definition.

Note

Type `ansys.chemkin.show_ignition_definitions()` to get the list of all available ignition delay time definitions in Chemkin.

Note

By default, time intervals for both print and save solution are **1/100** of the simulation end time. In this case $dt = time/100 = 0.001$. You can change them to different values.

```
# set the volume profile
MyCONV.set_volume_profile(x, volprofile)
# output controls
# set timestep between saving solution
MyCONV.timestep_for_saving_solution = 0.01
# turn ON adaptive solution saving
MyCONV.adaptive_solution_saving(mode=True, value_change=100, target="TEMPERATURE")
# specify the ignition definitions
MyCONV.set_ignition_delay(method="T_inflection")
```

Set solver controls

You can overwrite the default solver controls by using solver-related methods, such as those for tolerances.

```
# set tolerances in tuple: (absolute tolerance, relative tolerance)
MyCONV.tolerances = (1.0e-10, 1.0e-8)
# get solver parameters
ATOL, RTOL = MyCONV.tolerances
print(f"Dfault absolute tolerance = {ATOL}.")
print(f"Default relative tolerance = {RTOL}.")
# turn on the force non-negative solutions option in the solver
MyCONV.force_nonnegative = True
# show solver option
print(f"Timestep between solution printing: {MyCONV.timestep_for_printing_solution}.")
# show timestep between printing solution
print(f"Forced non-negative solution values: {MyCONV.force_nonnegative}.")
```

Display the added parameters (keywords)

You can use the `showkeywordinputlines()` method to verify the preceding parameters are correctly assigned to the reactor model.

```
# show the additional keywords given by user
MyCONV.showkeywordinputlines()
```

Run the simulation

Use the `run()` method to start the RCM simulation.

```
# run the CONV reactor model
runstatus = MyCONV.run()
```

(continues on next page)

(continued from previous page)

```
# check run status
if runstatus != 0:
    # Run failed.
    print(Color.RED + ">>> Run failed. <<<", end=Color.END)
    exit()

# Run succeeded.
print(Color.GREEN + ">>> Run completed. <<<", end=Color.END)
```

Get the ignition delay time from the solution

Use the `get_ignition_delay()` method to extract the ignition delay time after the run is completed.

Note

You need to deduct the initial compression time = 0.01 [sec] to get the *actual* ignition delay time.

```
# get ignition delay time (need to deduct the initial compression time = 0.01 [sec])
delaytime = MyCONV.get_ignition_delay() - 0.01 * 1.0e3
print(f"Ignition delay time = {delaytime} [msec].")
```

Postprocess the solution

The postprocessing step parses the solution and package the solution values at each time point into a mixture. There are two ways to access the solution profiles:

- The raw solution profiles (value as a function of distance) are available for distance, temperature, pressure, volume, and species mass fractions.

  -The mixtures permit the use of all property and rate utilities to extract
  information such as viscosity, density, and mole fractions.

You can use the `get_solution_variable_profile()` method to get the raw solution profiles. You can get solution mixtures using either the `get_solution_mixture_at_index()` method for the solution mixture at the given saved location or the `get_solution_mixture()` method for the solution mixture at the given distance. (In this case, the mixture is constructed by interpolation.)

Note

Use the `getnumbersolutionpoints()` method to get the size of the solution profiles before creating the arrays.

```
# postprocess the solutions
MyCONV.process_solution()
# get the number of solution time points
solutionpoints = MyCONV.getnumbersolutionpoints()
print(f"Number of solution points = {solutionpoints}.")

# easily access raw solution profiles
# get the time profile
timeprofile = MyCONV.get_solution_variable_profile("time")
# get the temperature profile
```

(continues on next page)

(continued from previous page)

```
tempprofile = MyCONV.get_solution_variable_profile("temperature")
# get the volume profile
volprofile = MyCONV.get_solution_variable_profile("volume")

# more involved postprocessing using mixtures
# reactor mass
massprofile = np.zeros_like(timeprofile, dtype=np.double)
# create arrays for CH4 mole fraction, CH4 ROP, and mixture viscosity
CH4profile = np.zeros_like(timeprofile, dtype=np.double)
CH4ROPprofile = np.zeros_like(timeprofile, dtype=np.double)
viscprofile = np.zeros_like(timeprofile, dtype=np.double)
CurrentROP = np.zeros(MyGasMech.KK, dtype=np.double)
# find CH4 species index
CH4_index = MyGasMech.get_specindex("CH4")

# loop over all solution time points
for i in range(solutionpoints):
    # get the mixture at the time point
    solutionmixture = MyCONV.get_solution_mixture_at_index(solution_index=i)
    # get gas density [g/cm3]
    den = solutionmixture.RHO
    # reactor mass [g]
    massprofile[i] = den * volprofile[i]
    # get CH4 mole fraction profile
    CH4profile[i] = solutionmixture.X[CH4_index]
    # get CH4 ROP profile
    currentROP = solutionmixture.ROP()
    CH4ROPprofile[i] = currentROP[CH4_index]
    # get mixture vicosity profile
    viscprofile[i] = solutionmixture.mixture_viscosity()
```

Validate the simulation result

Since the RCM is a closed reactor, the total gas mass inside RCM must be kept constant. You can verify it by computing the maximum mass deviation in the solution profile. If you find the mass variations are too large to be acceptable, you can set smaller tolerance values and/or adjust some solver parameters such as the maximum solver time step size and re-run the simulation.

```
del_mass = np.zeros_like(timeprofile, dtype=np.double)
mass0 = massprofile[0]
for i in range(solutionpoints):
    del_mass[i] = abs(massprofile[i] - mass0)
#
print(f">>> Maximum magnitude of reactor mass deviation = {np.max(del_mass)} [g].")
```

Plot the RCM solution profiles

```
# plot the profiles
plt.subplots(2, 2, sharex="col", figsize=(12, 6))
plt.subplot(221)
plt.plot(timeprofile, tempprofile, "r-")
plt.ylabel("Temperature [K]")
```

(continues on next page)

(continued from previous page)

```
plt.subplot(222)
plt.plot(timeprofile, CH4profile, "b-")
plt.ylabel("CH4 Mole Fraction")
plt.subplot(223)
plt.plot(timeprofile, CH4ROPprofile, "g-")
plt.xlabel("time [sec]")
plt.ylabel("CH4 Production Rate [mol/cm3-sec]")
plt.subplot(224)
plt.plot(timeprofile, viscprofile, "m-")
plt.xlabel("time [sec]")
plt.ylabel("Mixture Viscosity [g/cm-sec]")
# plot results
if interactive:
    plt.show()
else:
    plt.savefig("plot_RCM_solution.png", bbox_inches="tight")
```

12.3.4 Perform brute-force sensitivity analysis

One of the advantages of PyChemkin is customizability. You can easily build a specialized workflow to facilitate your simulation goals.

This tutorial demonstrates how to create a purpose-built workflow with PyChemkin by conducting a brute-force A-factor sensitivity analysis for ignition delay time of a premixed natural gas-air mixture at given initial temperature and pressure. Most Chemkin reactor models have the A-factor sensitivity analysis capability. The catch is that the subject variable of the analysis must be a member of the solution variables such as temperature, species mass fractions, and mass flow rate. However, for derived variables such as the ignition delay time, the built-in sensitivity analysis work as convenient. Thus, in this case, you may want to resort to the brute-force method to obtain those A-factor sensitivity coefficients with respect to the ignition delay time.

To conduct the brute-force A-factor sensitivity analysis, you will have to repeat the three steps for every reaction in the mechanism one by one

1. perturb the A-factor (the Arrhenius pre-exponent parameters) of a reaction
2. obtain the ignition delay time with this perturbed A-factor by running a constant pressure batch reactor simulation
3. restore the A-factor to its original value

The normalized ignition delay time sensitivity coefficient of reaction j is the difference between the original and the perturbed ignition delay time values divided by the size of the A-factor disturbance

$$S_j = \frac{(I_{j,ptb} - I_{j,org})}{(A_{j,ptb} - A_{j,org})/A_{j,org}}$$

Import PyChemkin package and start the logger

```
import os
import time

import ansys.chemkin as ck  # Chemkin
from ansys.chemkin import Color
```

(continues on next page)

(continued from previous page)

```
# chemkin batch reactor models (transient)
from ansys.chemkin.batchreactors.batchreactor import (
    GivenPressureBatchReactor_EnergyConservation,
)
from ansys.chemkin.logger import logger
import matplotlib.pyplot as plt  # plotting
import numpy as np  # number crunching

# check working directory
current_dir = os.getcwd()
logger.debug("working directory: " + current_dir)
# set verbose mode
ck.set_verbose(False)
# set interactive mode for plotting the results
# interactive = True: display plot
# interactive = False: save plot as a PNG file
global interactive
interactive = True
```

Create a chemistry set

The mechanism to load is the GRI 3.0 mechanism for methane combustion. This mechanism and its associated data files come with the standard Ansys Chemkin installation in the `/reaction/data` directory.

```
# set mechanism directory (the default Chemkin mechanism data directory)
data_dir = os.path.join(ck.ansys_dir, "reaction", "data")
mechanism_dir = data_dir
# create a chemistry set based on the diesel 14 components mechanism
MyGasMech = ck.Chemistry(label="GRI 3.0")
# set mechanism input files
# including the full file path is recommended
MyGasMech.chemfile = os.path.join(mechanism_dir, "grimech30_chem.inp")
MyGasMech.thermfile = os.path.join(mechanism_dir, "grimech30_thermo.dat")
```

Pre-process the `Chemistry Set`

```
iError = MyGasMech.preprocess()
# check preprocess status
if iError == 0:
    print("mechanism information:")
    print(f"number of gas species = {MyGasMech.KK:d}")
    print(f"number of gas reactions = {MyGasMech.IIGas:d}")
else:
    # When a non-zero value is returned from the process, check the text output files
    # chem.out, tran.out, or summary.out for potential error messages about the␣
↪mechanism data.
    print(f"Preprocessing error encountered. Code = {iError:d}.")
    print(f"see the summary file {MyGasMech.summaryfile} for details")
    exit()
```

Set up gas mixtures based on the species in this `Chemistry Set`

Use the *equivalence ratio method* so that you can easily set up the premixed fuel-oxidizer mixture composition by assigning an *equivalence ratio* value. In this case, the fuel mixture consists of methane, ethane, and propane as the simulated "natural gas". The premixed air-fuel mixture has an equivalence ratio of 1.1.

```
oxid = ck.Mixture(MyGasMech)
# set mole fraction
oxid.X = [("O2", 1.0), ("N2", 3.76)]
oxid.temperature = 900
oxid.pressure = ck.Patm  # 1 atm

fuel = ck.Mixture(MyGasMech)
# set mole fraction
fuel.X = [("C3H8", 0.1), ("CH4", 0.8), ("H2", 0.1)]
fuel.temperature = oxid.temperature
fuel.pressure = oxid.pressure

mixture = ck.Mixture(MyGasMech)
mixture.pressure = oxid.pressure
mixture.temperature = oxid.temperature
products = ["CO2", "H2O", "N2"]
add_frac = np.zeros(MyGasMech.KK, dtype=np.double)
# create the air-fuel mixture by using the equivalence ratio method
iError = mixture.X_by_Equivalence_Ratio(
    MyGasMech, fuel.X, oxid.X, add_frac, products, equivalenceratio=1.1
)
# check fuel-oxidizer mixture creation status
if iError != 0:
    print("Error: Failed to create the fuel-oxidizer mixture.")
    exit()
```

List the mixture composition

list the composition of the premixed mixture for verification.

```
if ck.verbose():
    mixture.list_composition(mode="mole")
```

Preparing for the sensitivity analysis

You need to perform some preparation work before running the brute-force sensitivity analysis. These tasks include: making backup copies of the original Arrhenius rate parameters, set up a *constant pressure* batch reactor object to compute the ignition delay times, and establish the baseline ignition delay time value with the original mechanism.

Get the original rate parameters

The first step is to save a copy of the Arrhenius rate parameters of all reactions. You can use the `get_reaction_parameters` method associated with the `MyGasMech` object. You can also verify the rate parameters by "screening" their values.

```
Afactor, Beta, ActiveEnergy = MyGasMech.get_reaction_parameters()
if ck.verbose():
    for i in range(MyGasMech.IIGas):
```

(continues on next page)

(continued from previous page)

```
        print(f"reaction: {i + 1}")
        print(f"A  = {Afactor[i]}")
        print(f"B  = {Beta[i]}")
        print(f"Ea = {ActiveEnergy[i]}\n")
        if np.isclose(0.0, Afactor[i], atol=1.0e-15):
            print("reaction pre-exponential factor = 0")
            exit()
```

Create the reactor object for ignition delay time calculations

Use the `GivenPressureBatchReactor_EnergyConservation` method to instantiate a *constant pressure batch reactor that also includes the energy equation*. You should use the `mixture` you just created.

```
MyCONP = GivenPressureBatchReactor_EnergyConservation(mixture, label="CONP")
# show initial gas composition inside the reactor for verification
MyCONP.list_composition(mode="mole")
```

Set up additional reactor model parameters

Reactor parameters, *solver controls*, and *output instructions* need to be provided before running the simulations. For a batch reactor, the *initial volume* and the *simulation end time* are required inputs. The `set_ignition_delay` method must be included for the reactor model to report the *ignition delay times* after the simulation is done. The *inflection points* definition is employed to detect the auto-ignition time because `method="T_inflection"` is specified. You can choose a different auto-ignition definition. Allow additional solution data point to be saved so that the predicted temperature profile can have enough resolution to provide more precise ignition delay time value. Here the adoptive solution saving is turned on by the `adaptive_solution_saving` method and the solution will be recorded for every **20** solver internal steps. Remember to set a simulation end time `time` that is long enough to catch the occurrence of auto-ignition.

Note

By default, time intervals for both print and save solution are **1/100** of the *simulation end time*. In this case $dt = time/100 = 0.001$. You can change them to different values.

```
# reactor volume [cm3]
MyCONP.volume = 10.0
# simulation end time [sec]
MyCONP.time = 2.0

# turn ON adaptive solution saving
MyCONP.adaptive_solution_saving(mode=True, steps=20)
# set ignition delay
MyCONP.set_ignition_delay(method="T_inflection")

# set tolerances in tuple: (absolute tolerance, relative tolerance)
MyCONP.tolerances = (1.0e-10, 1.0e-8)

# set the start wall time to get the total simulation run time
start_time = time.time()
```

Establish the baseline result

Run the nominal case and get the baseline ignition delay time value by using the `get_ignition_delay` method.

```
runstatus = MyCONP.run()
#
if runstatus == 0:
    # get ignition delay time
    delaytime_org = MyCONP.get_ignition_delay()
    print(f"ignition delay time = {delaytime_org} [msec]")
else:
    # if get this, most likely the END time is too short
    print(Color.RED + ">>> Run failed. <<<", end=Color.END)
    print("failed to find the ignition delay time of the nominal case")
    exit()
```

Run the sensitivity analysis cases

Now compute the "raw" A-factor sensitivity coefficients of ignition delay time. Firstly, you create an array `IGsen` to store the sensitivity coefficients, the size of `IGsen` must be no less than the number of reactions in the mechanism `MyGasMech`. Secondly, you introduce a small perturbation to the A-factor one reaction at a time by using the `set_reaction_AFactor` method. The advantage of this method is that you do not need to preprocess the `Chemistry Set` every time you make a change to the rate parameter. Then you run the same batch reactor `MyCONP` to get the ignition delay time.

Once the simulation is complete successfully, use the `get_ignition_delay` method to extract the ignition delay time. Compute the difference between this ignition delay time value (with altered A-factor) and the baseline value (from the original mechanism) and save the result to array `IGsen`. Remember to restore the A-factor to its original value before moving on to the next reaction.

```
# create sensitivity coefficient array
IGsen = np.zeros(MyGasMech.IIGas, dtype=np.double)
# set perturbation magnitude
perturb = 0.001  # increase by 0.1%
perturb_plus_1 = 1.0 + perturb
# loop over all reactions
for i in range(MyGasMech.IIGas):
    Anew = Afactor[i] * perturb_plus_1
    # actual reaction index
    ireac = i + 1
    # update the A factor
    MyGasMech.set_reaction_AFactor(ireac, Anew)
    # run the reactor model
    runstatus = MyCONP.run()
    #
    if runstatus == 0:
        # get ignition delay time
        delaytime = MyCONP.get_ignition_delay()
        print(f"ignition delay time = {delaytime} [msec]")
        # compute d(delaytime)
        IGsen[i] = delaytime - delaytime_org
        # restore the A factor
        MyGasMech.set_reaction_AFactor(ireac, Afactor[i])
    else:
```

(continues on next page)

(continued from previous page)

```
        # if get this, most likely the END time is too short
        print(f"trouble finding ignition delay time for raection {ireac}")
        print(Color.RED + ">>> Run failed. <<<", end=Color.END)
        exit()

# compute and report the total runtime (wall time)
runtime = time.time() - start_time
print(f"\ntotal simulation time: {runtime} [sec] over {MyGasMech.IIGas + 1} runs")
```

Compute the normalized sensitivity coefficients

Compute the normalized sensitivity coefficient = d(delaytime) * A[i] / d(A[i]).

```
IGsen /= perturb
```

Screen and rank the coefficients

Print top 5 positive and negative ignition delay time sensitivity coefficients to reveal the reactions of which the A-factor values have the strongest impact on the auto-ignition timing (positively or negatively). The ranking will change when the mixture composition or condition is changed.

```
top = 5
# rank the positive coefficients
posindex = np.argpartition(IGsen, -top)[-top:]
poscoeffs = IGsen[posindex]

# rank the negative coefficients
NegIGsen = np.negative(IGsen)
negindex = np.argpartition(NegIGsen, -top)[-top:]
negcoeffs = IGsen[negindex]

# print the top sensitivity coefficients
if ck.verbose():
    print("positive sensitivity coefficients")
    for i in range(top):
        print(f"reaction {posindex[i] + 1}: coefficient = {poscoeffs[i]}")
    print()
    print("negative sensitivity coefficients")
    for i in range(top):
        print(f"reaction {negindex[i] + 1}: coefficient = {negcoeffs[i]}")
```

Plot the ranked sensitivity coefficients

Create plots to show the reactions whose A-factors have most positive and negative influence on the ignition delay time.

```
plt.rcParams.update({"figure.autolayout": True, "ytick.color": "blue"})
plt.subplots(2, 1, sharex="col", figsize=(10, 5))
# convert reaction # from integers to strings
rxnstring = []
for i in range(len(posindex)):
    # the array index starting from 0 so the actual reaction # = index + 1
    rxnstring.append(MyGasMech.get_gas_reaction_string(posindex[i] + 1))
```

(continues on next page)

(continued from previous page)

```
# use horizontal bar chart
plt.subplot(211)
plt.barh(rxnstring, poscoeffs, color="orange", height=0.4)
plt.axvline(x=0, c="gray", lw=1)
# convert reaction # from integers to strings
rxnstring.clear()
fnegindex = np.flip(negindex)
fnegcoeffs = np.flip(negcoeffs)
for i in range(len(negindex)):
    # the array index starting from 0 so the actual reaction # = index + 1
    rxnstring.append(MyGasMech.get_gas_reaction_string(fnegindex[i] + 1))
plt.subplot(212)
plt.barh(rxnstring, fnegcoeffs, color="orange", height=0.4)
plt.axvline(x=0, c="gray", lw=1)
plt.xlabel("Sensitivity Coefficients")
plt.suptitle("Ignition Delay Time Sensitivity", fontsize=16)
# plot results
if interactive:
    plt.show()
else:
    plt.savefig("plot_sensitivity_analysis.png", bbox_inches="tight")
```

12.3.5 Explore cooling water vapor

How does the volume of water vapor evolves when it is cooled from a temperature above the boiling point to a temperature that is just above the freezing point at constant pressure? How fast does the vapor volume drop? This example uses the `GivenPressureBatchReactor_FixedTemperature` model to explore the different behaviors between an *ideal gas* water vapor and its *real gas* counterpart.

```

```

Import PyChemkin packages and start the logger

```
import os

import ansys.chemkin as ck  # Chemkin
from ansys.chemkin import Color

# chemkin batch reactor model (transient)
from ansys.chemkin.batchreactors.batchreactor import (
    GivenPressureBatchReactor_FixedTemperature,
)
from ansys.chemkin.logger import logger
import matplotlib.pyplot as plt  # plotting
import numpy as np  # number crunching

# check working directory
current_dir = os.getcwd()
logger.debug("working directory: " + current_dir)
# set interactive mode for plotting the results
# interactive = True: display plot
```

(continues on next page)

(continued from previous page)

```
# interactive = False: save plot as a PNG file
global interactive
interactive = True
```

Create a chemistry set

To compare the different behaviors of the water vapor under the ideal gas and real gas assumptions, you must use a *real gas EOS-enabled gas mechanism. The 'C2 NOx' mechanism that includes information about the *Soave* cubic Equation of State (EOS) is suitable for this endeavor. Therefore, the `MyMech` chemistry set is created from this gas phase mechanism.

```
# set mechanism directory (the default Chemkin mechanism data directory)
data_dir = os.path.join(ck.ansys_dir, "reaction", "data")
mechanism_dir = data_dir
# create a chemistry set based on C2_NOx using an alternative method
MyMech = ck.Chemistry(label="C2 NOx")
# set mechanism input files individually
# this mechanism file contains all the necessary thermodynamic and transport data
# thus, there is no need to specify thermodynamic and transport data files
MyMech.chemfile = os.path.join(mechanism_dir, "C2_NOx_SRK.inp")
```

Preprocess the C2 NOx chemistry set

```
# preprocess the mechanism file
iError = MyMech.preprocess()
if iError == 0:
    print(Color.GREEN + ">>> Preprocessing succeeded.", end=Color.END)
else:
    print(Color.RED + ">>> Preprocessing failed.", end=Color.END)
    exit()
```

Set up the air/water vapor mixture

Create the mixture of air and water vapor inside the imaginary strong but elastic container. The initial gas temperature is set to 500 [K] and at a constant pressure of 100 [atm]. At this temperature and pressure, the mixture should be entirely in gas form.

```
mist = ck.Mixture(MyMech)
# set mole fraction
mist.X = [("H2O", 2.0), ("O2", 1.0), ("N2", 3.76)]
mist.temperature = 500.0  # [K]
mist.pressure = 100.0 * ck.Patm
```

Create the reactor `tank` to perform the vapor cooling simulation

Use the `GivenPressureBatchReactor_FixedTemperature()` method to create a constant-pressure batch reactor (with a given temperature). Use the `mist` mixture that you just created to set the initial gas condition inside the `tank` reactor.

```
tank = GivenPressureBatchReactor_FixedTemperature(mist, label="tank")
```

Set up additional reactor model parameters

You must provide reactor parameters, solver controls, and output instructions before running the simulations. For a batch reactor, the initial volume and simulation end time are required inputs.

```
# verify initial gas composition inside the reactor
tank.list_composition(mode="mole")

# set other reactor properties
tank.volume = 10.0  # cm3
tank.time = 0.5  # sec
```

Set the gas temperature profile of the `tank` reactor

Create a time-temperature profile by using two arrays. Use the `set_temperature_profile()` method to add the profile to the reactor model.

```
# number of profile data points
npoints = 3
# position array of the profile data
x = np.zeros(npoints, dtype=np.double)
# value array of the profile data
TPROprofile = np.zeros_like(x, dtype=np.double)
# set tank temperature data points
x = [0.0, 0.2, 2.0]  # [sec]
TPROprofile = [500.0, 275.0, 275.0]  # [K]
# set the temperature profile
tank.set_temperature_profile(x, TPROprofile)
```

Switch on the real-gas EOS model

Use the `use_realgas_cubicEOS()` method to turn on the real-gas EOS model. For more information, type either `ansys.chemkin.help("real gas")` for information on real-gas model usage or `ansys.chemkin.help("manuals")` to access the online **Chemkin Theory** manual for descriptions of the real-gas EOS models.

Note

By default the *Van der Waals* mixing rule is applied to evaluate thermodynamic properties of a real-gas mixture. You can use `set_realgas_mixing_rule` to switch to a different mixing rule.

```
tank.userealgasEOS(mode=True)
```

Set output options

```
# set timestep between saving solution
tank.timestep_for_saving_solution = 0.01
```

Set solver controls

You can overwrite the default solver controls by using solver related methods, for example, `tolerances`.

```
# set tolerances in tuple: (absolute tolerance, relative tolerance)
tank.tolerances = (1.0e-10, 1.0e-8)
# get solver parameters
ATOL, RTOL = tank.tolerances
print(f"default absolute tolerance = {ATOL}")
print(f"default relative tolerance = {RTOL}")
# turn on the force non-negative solutions option in the solver
tank.force_nonnegative = True
```

Run the vapor cooling simulation with the *real gas EOS*

Run the CONP reactor model with given temperature profile with the *real gas EOS* model switched *ON*.

```
runstatus = tank.run()

# check run status
if runstatus != 0:
    # run failed!
    print(Color.RED + ">>> Run failed. <<<", end=Color.END)
    exit()
# run success!
print(Color.GREEN + ">>> Run completed. <<<", end=Color.END)
```

Postprocess the solution

The postprocessing step parses the solution and packages the solution values at each time point into a mixture. There are two ways to access the solution profiles:

- The raw solution profiles (value as a function of distance) are available for distance, temperature, pressure, volume, and species mass fractions.

  -The mixtures permit the use of all property and rate utilities to extract
  information such as viscosity, density, and mole fractions.

You can use the `get_solution_variable_profile()` method to get the raw solution profiles. You can get solution mixtures using either the `get_solution_mixture_at_index()` method for the solution mixture at the given saved location or the `get_solution_mixture()` method for the solution mixture at the given distance. (In this case, the mixture is constructed by interpolation.)

Note

Use the `getnumbersolutionpoints()` method to get the size of the solution profiles before creating the arrays.

```
tank.process_solution()
# get the number of solution time points
solutionpoints = tank.getnumbersolutionpoints()
print(f"number of solution points = {solutionpoints}")
# get the time profile
timeprofile = tank.get_solution_variable_profile("time")
# get the temperature profile
tempprofile = tank.get_solution_variable_profile("temperature")
# get the volume profile
volprofile = tank.get_solution_variable_profile("volume")
```

(continues on next page)

(continued from previous page)

```
# create array for mixture density
denprofile = np.zeros_like(timeprofile, dtype=np.double)
# create array for mixture enthalpy
Hprofile = np.zeros_like(timeprofile, dtype=np.double)
# loop over all solution time points
for i in range(solutionpoints):
    # get the mixture at the time point
    solutionmixture = tank.get_solution_mixture_at_index(solution_index=i)
    # get mixture density profile
    denprofile[i] = solutionmixture.RHO
    # get mixture enthalpy profile
    Hprofile[i] = solutionmixture.HML() / ck.ergs_per_joule * 1.0e-3
```

Turn off the real gas EOS model

Use the `use_realgas_cubicEOS()` method to turn off the real gas EOS model. Alternatively, use the `use_idealgas_law()` method to turn on the ideal gas law.

```
tank.userealgasEOS(mode=False)
```

Run the vapor cooling simulation with the ideal gas law

Run the CONP reactor model with given temperature profile with the *ideal gas law* turned back on.

```
runstatus = tank.run()

# check run status
if runstatus != 0:
    # run failed!
    print(Color.RED + ">>> Run failed. <<<", end=Color.END)
    exit()
# run success!
print(Color.GREEN + ">>> Run completed. <<<", end=Color.END)
```

Postprocess the ideal gas solution

postprocess the solutions

```
tank.process_solution()
# get the number of solution time points
solutionpoints = tank.getnumbersolutionpoints()
print(f"number of solution points = {solutionpoints}")
# get the time profile
timeprofile_IG = tank.get_solution_variable_profile("time")
# get the volume profile
volprofile_IG = tank.get_solution_variable_profile("volume")
# create array for mixture density
denprofile_IG = np.zeros_like(timeprofile, dtype=np.double)
# create array for mixture enthalpy
Hprofile_IG = np.zeros_like(timeprofile, dtype=np.double)
# loop over all solution time points
for i in range(solutionpoints):
```

(continues on next page)

(continued from previous page)

```
    # get the mixture at the time point
    solutionmixture = tank.get_solution_mixture_at_index(solution_index=i)
    # get mixture density profile
    denprofile_IG[i] = solutionmixture.RHO
    # get mixture enthalpy profile
    Hprofile_IG[i] = solutionmixture.HML() / ck.ergs_per_joule * 1.0e-3

ck.done()
```

Plot the RCM solution profiles

You should observe that the mixture volume obtained by the real gas model is noticeably lower than the ideal gas volume. When water vapor is cooled below the boiling point, formation of liquid water is expected due to condensation. The ideal gas law assumes that the mixture is always in gas phase and is unable to address the phase change phenomenon. The real gas EOS, on the other hand, can capture the formation of the liquid water as indicated by the sharper rise of the mixture density during the cooling process. The real gas mixture also has a lower enthalpy level than that of the ideal gas mixture. The enthalpy differences should largely represent the heat of vaporization of water at the temperature.

```
# plot the profiles
plt.subplots(2, 2, sharex="col", figsize=(12, 6))
thispres = str(mist.pressure / ck.Patm)
thistitle = "Cooling Vapor + Air at " + thispres + " atm"
plt.suptitle(thistitle, fontsize=16)
plt.subplot(221)
plt.plot(timeprofile, tempprofile, "r-")
plt.ylabel("Temperature [K]")
plt.subplot(222)
plt.plot(timeprofile, volprofile, "b-", label="real gas")
plt.plot(timeprofile_IG, volprofile_IG, "b--", label="ideal gas")
plt.legend(loc="upper right")
plt.ylabel("Volume [cm3]")
plt.subplot(223)
plt.plot(timeprofile, Hprofile, "g-", label="real gas")
plt.plot(timeprofile_IG, Hprofile_IG, "g--", label="ideal gas")
plt.legend(loc="upper right")
plt.xlabel("time [sec]")
plt.ylabel("Mixture Enthalpy [kJ/mole]")
plt.subplot(224)
plt.plot(timeprofile, denprofile, "m-", label="real gas")
plt.plot(timeprofile_IG, denprofile_IG, "m--", label="ideal gas")
plt.legend(loc="upper left")
plt.xlabel("time [sec]")
plt.ylabel("Mixture Density [g/cm3]")
# plot results
if interactive:
    plt.show()
else:
    plt.savefig("plot_vapor_condensation.png", bbox_inches="tight")
```

12.4 0-D engine models

PyChemkin offers two types of 0-D engine models: the *Homogeneous Charged Compression Ignition (HCCI)* engine model and the *Spark Ignition (SI)* engine model. These 0-D engine models simulate the combustion process (or the lack of, in case of a misfire) inside an engine cylinder between the *Intake Valve Close (IVC)* and the *Exhaust Valve Open (EVO)* when the cylinder is considered as a closed system. The Chemkin *Theory* manual has detailed descriptions of these 0-D engine models.

The examples show the steps of setting up simulations with different *Chemkin* engine models.

12.4.1 Simulate a single-zone HCCI engine

Ansys Chemkin offers some idealized internal combustion (IC) engine models commonly used for fuel combustion and engine performance research. The Chemkin IC engine model is a specialized transient 0-D *closed* gas-phase reactor that mainly performs combustion simulation between the intake valve closing (IVC) and the exhaust valve opening (EVO), that is, when the engine cylinder resembles a closed chamber. The cylinder volume is derived from the piston motion as a function of the engine crank angle (CA) and engine parameters such as engine speed (RPM) and stroke. The energy equation is always solved, and there are several wall heat transfer models specifically designed for engine simulations.

Note

For additional information on Chemkin IC engine models, use the `ansys.chemkin.manuals()` method to view the online **Theory** manual.

This example shows how to set up and run the simplest Chemkin IC engine model: the single-zone homogeneous charged compression ignition (HCCI) engine model. In addition to the basic engine parameters, many engine model-specific features such as the *exhaust gas recirculation* and *wall heat transfer* can be included in the engine simulation.

Import PyChemkin packages and start the logger

```
import os

import ansys.chemkin as ck  # Chemkin
from ansys.chemkin import Color

# chemkin homonegeous charge compression ignition (HCCI) engine model (transient)
from ansys.chemkin.engines.HCCI import HCCIengine
from ansys.chemkin.logger import logger
import matplotlib.pyplot as plt  # plotting
import numpy as np  # number crunching

# check working directory
current_dir = os.getcwd()
logger.debug("working directory: " + current_dir)
# set verbose mode
ck.set_verbose(True)
# set interactive mode for plotting the results
# interactive = True: display plot
# interactive = False: save plot as a PNG file
```

(continues on next page)

(continued from previous page)

```
global interactive
interactive = True
```

Create a chemistry set

The mechanism to load is the GRI 3.0 mechanism for methane combustion. This mechanism and its associated data files come with the standard Ansys Chemkin installation in the `/reaction/data` directory.

```
# set mechanism directory (the default Chemkin mechanism data directory)
data_dir = os.path.join(ck.ansys_dir, "reaction", "data")
mechanism_dir = data_dir
# create a chemistry set based on the GRI mechanism
MyGasMech = ck.Chemistry(label="GRI 3.0")
# set mechanism input files
# including the full file path is recommended
MyGasMech.chemfile = os.path.join(mechanism_dir, "grimech30_chem.inp")
MyGasMech.thermfile = os.path.join(mechanism_dir, "grimech30_thermo.dat")
MyGasMech.tranfile = os.path.join(mechanism_dir, "grimech30_transport.dat")
```

Preprocess the chemistry set

```
# preprocess the mechanism files
iError = MyGasMech.preprocess()
if iError != 0:
    msg = "preprocess failed"
    logger.critical("msg")
    Color.ckprint("critical", [msg, "!!!"])
    exit()
```

Set up the fuel-air mixture

You must set up the fuel-air mixture inside the engine cylinder right after the intake valve is closed. Here the `X_by_Equivalence_Ratio()` method is used. You create the `fuelmixture` and `air` mixtures first. You then define the *complete combustion product species* and provide the *additives* composition if there is any. And finally, you can simply set the value of `equivalenceratio` to create the fuel-air mixture. In this case, the fuel mixture consists of methane, ethane, and propane as the simulated natural gas. Because HCCI engines typically run on lean fuel-air mixtures, the equivalence ratio is set to 0.8.

```
# create a premixed fuel-oxidizer mixture by assigning the equivalence ratio
# create the fuel mixture
fuelmixture = ck.Mixture(MyGasMech)
# set fuel composition
fuelmixture.X = [("CH4", 0.9), ("C3H8", 0.05), ("C2H6", 0.05)]
# setting pressure and temperature is not required in this case
fuelmixture.pressure = 1.5 * ck.Patm
fuelmixture.temperature = 400.0

# create the oxidizer mixture: air
air = ck.Mixture(MyGasMech)
air.X = [("O2", 0.21), ("N2", 0.79)]
# setting pressure and temperature is not required in this case
air.pressure = 1.5 * ck.Patm
```

(continues on next page)

(continued from previous page)

```
air.temperature = 400.0

# products from the complete combustion of the fuel mixture and air
products = ["CO2", "H2O", "N2"]
# species mole fractions of added/inert mixture. You can also create an additives␣
↪mixture here.
add_frac = np.zeros(MyGasMech.KK, dtype=np.double)  # no additives: all zeros

# create the unburned fuel-air mixture
fresh = ck.Mixture(MyGasMech)
# mean equivalence ratio
equiv = 0.8
iError = fresh.X_by_Equivalence_Ratio(
    MyGasMech, fuelmixture.X, air.X, add_frac, products, equivalenceratio=equiv
)
# check fuel-oxidizer mixture creation status
if iError != 0:
    print("Error: Failed to create the fuel-oxidizer mixture.")
    exit()

# list the composition of the unburned fuel-air mixture
fresh.list_composition(mode="mole")
```

Specify pressure and temperature of the fuel-air mixture

Since you are going to use `fresh` to instantiate the engine object later, setting the mixture pressure and temperature is equivalent to setting the initial temperature and pressure of the engine cylinder.

```
fresh.temperature = 447.0
fresh.pressure = 1.065 * ck.Patm
```

Add EGR to the fresh fuel-air mixture

Many engines have the configuration for exhaust gas recirculation (EGR). Chemkin engine models allow you to add the EGR mixture to the fresh fuel-air mixture entered the cylinder. If the engine that you are modeling has EGR, you should have the EGR ratio, which is generally the volume ratio between the EGR mixture and the fresh fuel-air ratio. However, you know nothing about the composition of the exhaust gas so you cannot simply combine these two mixtures. In this case, you can use the `get_EGR_mole_fraction()` method to estimate the major components of the exhaust gas from the combustion of the fresh fuel-air mixture. The parameter `threshold=1.0e-8` tells the method to ignore any species with mole fractions below the threshold value. Once you have the EGR mixture composition, use the `X_by_Equivalence_Ratio()` method a second time to re-create the fuel-air mixture fresh with the original `fuelmixture` and `air` mixtures along with the EGR composition that you just got as the *additives*.

```
EGRratio = 0.3
# compute the EGR stream composition in mole fractions
add_frac = fresh.get_EGR_mole_fraction(EGRratio, threshold=1.0e-8)
# re-create the initial mixture with EGR
iError = fresh.X_by_Equivalence_Ratio(
    MyGasMech,
    fuelmixture.X,
    air.X,
    add_frac,
```

(continues on next page)

(continued from previous page)

```
    products,
    equivalenceratio=equiv,
    threshold=1.0e-8,
)

# list the composition of the fuel+air+EGR mixture for verification
fresh.list_composition(mode="mole", bound=1.0e-8)
```

Set up the HCCI engine reactor

Use the `HCCIengine()` method to create a single-zone HCCI engine named `MyEngine` and make the *new* `fresh` mixture the initial in-cylinder gas mixture at IVC. Set the `nzones` parameter to 1 for the *single-zone* HCCI simulation.

```
MyEngine = HCCIengine(reactor_condition=fresh, nzones=1)
# show initial gas composition inside the reactor
MyEngine.list_composition(mode="mole", bound=1.0e-8)
```

Set up basic engine parameters

Set the required engine parameters as shown in the following code. These engine parameters are used to describe the cylinder volume during the simulation. The `starting_CA` should be the crank angle corresponding to the cylinder IVC. The `ending_CA` is typically the EVC crank angle.

```
# cylinder bore diameter [cm]
MyEngine.bore = 12.065
# engine stroke [cm]
MyEngine.stroke = 14.005
# connecting rod length [cm]
MyEngine.connecting_rod_length = 26.0093
# compression ratio [-]
MyEngine.compression_ratio = 16.5
# engine speed [RPM]
MyEngine.RPM = 1000

# set piston pin offset distance [cm] (optional)
MyEngine.set_piston_pin_offset(offset=-0.5)

# set other required parameters
# simulation start CA [degree]
MyEngine.starting_CA = -142.0
# simulation end CA [degree]
MyEngine.ending_CA = 116.0

# list the engine parameters for verification
MyEngine.list_engine_parameters()
print(f"Engine displacement volume = {MyEngine.get_displacement_volume()} [cm3].")
print(f"Engine clearance volume = {MyEngine.get_clearance_volume()} [cm3].")
print(f"Number of zones = {MyEngine.get_number_of_zones()}.")
```

Set up engine wall heat transfer model

By default, the engine cylinder is adiabatic. You must set up a wall heat transfer model to include the heat loss effects in your engine simulation. Chemkin supports three widely used engine wall heat transfer models. These models and their parameters follow:

- `dimensionless`: [<a> <b> <c> <Twall>]
- `dimensional`: [<a> <b> <c> <Twall>]
- `hohenburg`: [<a> <b> <c> <d> <e> <Twall>]

There is also the incylinder gas velocity correlation (the Woschni correlation) that is associated with the engine wall heat transfer models. Here are the parameters of the Woschni correlation:

`[<C11> <C12> <C2> <swirl ratio>]`

You can also specify the surface areas of the piston head and the cylinder head for more precision heat transfer wall area. By default, both the piston head and the cylinder head surfaces are flat.

```
heattransferparameters = [0.035, 0.71, 0.0]
# set cylinder wall temperature [K]
Twall = 400.0
MyEngine.set_wall_heat_transfer("dimensionless", heattransferparameters, Twall)
# in-cylinder gas velocity correlation parameter (Woschni)
# [<C11> <C12> <C2> <swirl ratio>]
GVparameters = [2.28, 0.308, 3.24, 0.0]
MyEngine.set_gas_velocity_correlation(GVparameters)
# set piston head top surface area [cm2]
MyEngine.set_piston_head_area(area=124.75)
# set cylinder clearance surface area [cm2]
MyEngine.set_cylinder_head_area(area=123.5)
```

Set output options

You can turn on adaptive solution saving to resolve the steep variations in the solution profile. Here additional solution data points are saved for every 20 solver internal steps. You must include the `set_ignition_delay()` method for the engine model to report the ignition delay crank angle after the simulation is done. If `method="T_inflection"` is set, the reactor model treats the inflection points in the predicted gas temperature profile as the indication of an auto-ignition. You can choose a different auto-ignition definition.

Note

Type `ansys.chemkin.show_ignition_definitions()` to get the list of all available ignition delay time definitions in Chemkin.

Note

By default, time/crank angle intervals for both print and save solution are 1/100 of the simulation duration, which is in this case $dCA = (EVO - IVC)/100 = 2.58$. You can make the model report more frequently by using the `CAstep_for_saving_solution()` or `CAstep_for_printing_solution()` method to set different interval values in the crank angle.

```
# set the number of crank angles between saving solution
MyEngine.CAstep_for_saving_solution = 0.5
# set the number of crank angles between printing solution
MyEngine.CAstep_for_printing_solution = 10.0
# turn on adaptive solution saving
MyEngine.adaptive_solution_saving(mode=True, steps=20)
# specify the ignition definitions
MyEngine.set_ignition_delay(method="T_inflection")
```

Set solver controls

You can overwrite the default solver controls by using solver-related methods, such as those for tolerances.

```
# set tolerances in tuple: (absolute tolerance, relative tolerance)
MyEngine.tolerances = (1.0e-12, 1.0e-10)
# get solver parameters
ATOL, RTOL = MyEngine.tolerances
print(f"Default absolute tolerance = {ATOL}.")
print(f"Default relative tolerance = {RTOL}.")
# turn on the force non-negative solutions option in the solver
MyEngine.force_nonnegative = True
# show solver and output options
# show the number of crank angles between printing solution
print(
    f"Crank angles between solution printing: {MyEngine.CAstep_for_printing_solution}"
)
# show other transient solver setup
print(f"Forced non-negative solution values: {MyEngine.force_nonnegative}")
```

Display the added parameters (keywords)

Use the `showkeywordinputlines()` method to verify that the preceding parameters are correctly assigned to the engine model.

```
MyEngine.showkeywordinputlines()
```

Run the simulation

Use the `run()` method to start the single-zone HCCI engine simulation.

```
runstatus = MyEngine.run()
# check run status
if runstatus != 0:
    # Run failed.
    print(Color.RED + ">>> Run failed. <<<", end=Color.END)
    logger.error("Run failed.")
    exit()
# Run succeeded.
print(Color.GREEN + ">>> Run completed. <<<", end=Color.END)
logger.info("Run Completed.")
```

Get the ignition delay crank angle from the solution

Use the `get_ignition_delay()` method to extract the ignition delay crank angle (CA) after the run is completed.

```
# get ignition delay "time"
delayCA = MyEngine.get_ignition_delay()
print(f"Ignition delay CA = {delayCA} [degree].")
```

Get the heat release crank angles

The engine models also report the crank angles when the accumulated heat release reaches 10%, 50%, and 90% of the total heat release. Use the `get_engine_heat_release_CAs` method to extract these heat release crank angles (CA).

```
# get heat release information
HR10, HR50, HR90 = MyEngine.get_engine_heat_release_CAs()
print("Engine Heat Release Information")
print(f"10% heat release CA = {HR10} [degree].")
print(f"50% heat release CA = {HR50} [degree].")
print(f"90% heat release CA = {HR90} [degree].\n")
```

Postprocess the solution

The postprocessing step parses the solution and packages the solution values at each time point into a `Mixture` object. There are two ways to access the solution profiles:

- The raw solution profiles (value as a function of time) are available for time, temperature, pressure, volume, and species mass fractions.
- The mixtures permit the use of all property and rate utilities to extract information such as viscosity, density, and mole fractions.

You can use the `get_solution_variable_profile()` method to get the raw solution profiles. You can get solution mixtures using either the `get_solution_mixture_at_index()` method for the solution mixture at a given time point or the `get_solution_mixture()` method for the solution mixture at a given time. (In this case, the mixture is constructed by interpolation.)

Note

- For engine models, use the `process_engine_solution()` method to postprocess the solutions.
- Use the `getnumbersolutionpoints()` method to get the size of the solution profiles before creating the arrays.
- Use the `get_CA()` method to convert the time values reported in the solution to crank angles.

```
# postprocess the solutions
MyEngine.process_engine_solution()
# get the number of solution time points
solutionpoints = MyEngine.getnumbersolutionpoints()
print(f"Number of solution points = {solutionpoints}.")
# get the time profile
timeprofile = MyEngine.get_solution_variable_profile("time")
# convert time to crank angle
CAprofile = np.zeros_like(timeprofile, dtype=np.double)
count = 0
```

(continues on next page)

(continued from previous page)

```
for t in timeprofile:
    CAprofile[count] = MyEngine.get_CA(timeprofile[count])
    count += 1
# get the cylinder pressure profile
presprofile = MyEngine.get_solution_variable_profile("pressure")
presprofile *= 1.0e-6
# get the volume profile
volprofile = MyEngine.get_solution_variable_profile("volume")
# create arrays for mixture density, NO mole fraction, and mixture-specific heat capacity
denprofile = np.zeros_like(timeprofile, dtype=np.double)
Cpprofile = np.zeros_like(timeprofile, dtype=np.double)
# loop over all solution time points
for i in range(solutionpoints):
    # get the mixture at the time point
    solutionmixture = MyEngine.get_solution_mixture_at_index(solution_index=i)
    # get gas density [g/cm3]
    denprofile[i] = solutionmixture.RHO
    # get mixture-specific heat capacity profile [erg/mole-K]
    Cpprofile[i] = solutionmixture.CPBL() / ck.ergs_per_joule * 1.0e-3
```

Plot the engine solution profiles

Plot the profiles from the HCCI engine simulation.

Note

You can get profiles of the thermodynamic and the transport properties by applying `Mixture` utility methods to the solution mixtures.

```
plt.subplots(2, 2, sharex="col", figsize=(12, 6))
plt.subplot(221)
plt.plot(CAprofile, presprofile, "r-")
plt.ylabel("Pressure [bar]")
plt.subplot(222)
plt.plot(CAprofile, volprofile, "b-")
plt.ylabel("Volume [cm3]")
plt.subplot(223)
plt.plot(CAprofile, denprofile, "g-")
plt.xlabel("Crank Angle [degree]")
plt.ylabel("Mixture Density [g/cm3]")
plt.subplot(224)
plt.plot(CAprofile, Cpprofile, "m-")
plt.xlabel("Crank Angle [degree]")
plt.ylabel("Mixture Cp [kJ/mole]")
# plot results
if interactive:
    plt.show()
else:
    plt.savefig("plot_HCCI_engine.png", bbox_inches="tight")
```

12.4.2 Simulate a multi-zone HCCI engine

Ansys Chemkin offers some idealized internal combustion (IC) engine models commonly used for fuel combustion and engine performance research. The Chemkin IC engine model is a specialized transient 0-D *closed* gas-phase reactor that mainly performs combustion simulation between the intake valve closing (IVC) and the exhaust valve opening (EVO), that is, when the engine cylinder resembles a closed chamber. The cylinder volume is derived from the piston motion as a function of the engine crank angle (CA) and engine parameters such as engine speed (RPM) and stroke. The energy equation is always solved and there are several wall heat transfer models specifically designed for engine simulations.

Note

For additional information on the Chemkin IC engine models, use the `ansys.chemkin.manuals()` method to view the online **Theory** manual.

The multi-zone homogeneous charged compression ignition (HCCI) model is mainly intended to address the temperature variation inside the cylinder caused by the wall heat transfer and the imperfect mixing of the in-cylinder gas mixture. In addition, the multi-zone HCCI engine model allows the introduction of non-uniform temperature and/or the equivalence ratio distribution of the gas mixture at the IVC.

This example shows how to set up and run the Chemkin multi-zone HCCI engine model. Many engine model-specific features, such as basic engine parameters, exhaust gas recirculation, and wall heat transfer, are applied to the engine simulation. This example also shows how to create initial distributions of zone size, temperature, and composition.

Import PyChemkin packages and start the logger

```
import os

import ansys.chemkin as ck  # Chemkin
from ansys.chemkin import Color

# Chemkin homogeneous charge compression ignition (HCCI) engine model (transient)
from ansys.chemkin.engines.HCCI import HCCIengine
from ansys.chemkin.logger import logger
import matplotlib.pyplot as plt  # plotting
import numpy as np  # number crunching

# check working directory
current_dir = os.getcwd()
logger.debug("working directory: " + current_dir)
# set verbose mode
ck.set_verbose(True)
# set interactive mode for plotting the results
# interactive = True: display plot
# interactive = False: save plot as a PNG file
global interactive
interactive = True
```

Create a chemistry set

The mechanism to load is the GRI 3.0 mechanism for methane combustion. This mechanism and its associated data files come with the standard Ansys Chemkin installation in the `/reaction/data` directory.

```
# set mechanism directory (the default Chemkin mechanism data directory)
data_dir = os.path.join(ck.ansys_dir, "reaction", "data")
mechanism_dir = data_dir
# create a chemistry set based on the GRI mechanism
MyGasMech = ck.Chemistry(label="GRI 3.0")
# set mechanism input files
# including the full file path is recommended
MyGasMech.chemfile = os.path.join(mechanism_dir, "grimech30_chem.inp")
MyGasMech.thermfile = os.path.join(mechanism_dir, "grimech30_thermo.dat")
MyGasMech.tranfile = os.path.join(mechanism_dir, "grimech30_transport.dat")
```

Preprocess the chemistry set

```
# preprocess the mechanism files
iError = MyGasMech.preprocess()
```

Set up the fuel-air mixture

You must set up the fuel-air mixture inside the engine cylinder right after the intake valve is closed. Here the `X_by_Equivalence_Ratio()` method is used. You create the `fuelmixture` and the `air` mixtures first. You then define the *complete combustion product species* and provide the *additives* composition if there is any. Finally, you set the `equivalenceratio` value to create the fuel-air mixture. In this case, the fuel mixture consists of methane, ethane, and propane as the simulated natural gas. Because HCCI engines generally run on lean fuel-air mixtures, the equivalence ratio is set to 0.8.

```
# create the fuel mixture
fuelmixture = ck.Mixture(MyGasMech)
# set fuel composition
fuelmixture.X = [("CH4", 0.9), ("C3H8", 0.05), ("C2H6", 0.05)]
# setting pressure and temperature is not required in this case
fuelmixture.pressure = 1.5 * ck.Patm
fuelmixture.temperature = 400.0
# create the oxidizer mixture: air
air = ck.Mixture(MyGasMech)
air.X = [("O2", 0.21), ("N2", 0.79)]
# setting pressure and temperature is not required in this case
air.pressure = 1.5 * ck.Patm
air.temperature = 400.0
# create the unburned fuel-air mixture
fresh = ck.Mixture(MyGasMech)
# products from the complete combustion of the fuel mixture and air
products = ["CO2", "H2O", "N2"]
# species mole fractions of added/inert mixture. can also create an additives mixture␣
↪here
add_frac = np.zeros(MyGasMech.KK, dtype=np.double)  # no additives: all zeros
# mean equivalence ratio
equiv = 0.8
iError = fresh.X_by_Equivalence_Ratio(
```

(continues on next page)

(continued from previous page)

```
    MyGasMech, fuelmixture.X, air.X, add_frac, products, equivalenceratio=equiv
)
# check fuel-oxidizer mixture creation status
if iError != 0:
    print("Error: Failed to create the fuel-oxidizer mixture.")
    exit()

# list the composition of the unburned fuel-air mixture
fresh.list_composition(mode="mole")
```

Specify pressure and temperature of the fuel-air mixture

Since you are going to use the `fresh` fuel-air mixture to instantiate the engine object later, setting the mixture pressure and temperature is equivalent to setting the initial temperature and pressure of the engine cylinder.

```
fresh.temperature = 447.0
fresh.pressure = 1.065 * ck.Patm
```

Add EGR to the fresh fuel-air mixture

Many engines have the configuration for exhaust gas recirculation (EGR). Chemkin engine models allow you to add the EGR mixture to the fresh fuel-air mixture entered the cylinder. If the engine you are modeling has EGR, you should have the EGR ratio, which is generally the volume ratio between the EGR mixture and the fresh fuel-air ratio. However, because you know nothing about the composition of the exhaust gas, you cannot simply combine these two mixtures. In this case, you use the `get_EGR_mole_fraction()` method to estimate the major components of the exhaust gas from the combustion of the fresh fuel-air mixture. The `threshold=1.0e-8` parameter tells the method to ignore any species with a mole fraction below the threshold value. Once you have the EGR mixture composition, use the `X_by_Equivalence_Ratio()` method a second time to re-create the `fresh` fuel-air mixture with the original `fuelmixture` and `air` mixtures along with the EGR composition you just got as the *"additives"*.

```
EGRratio = 0.3
# compute the EGR stream composition in mole fractions
add_frac = fresh.get_EGR_mole_fraction(EGRratio, threshold=1.0e-8)
# recreate the initial mixture with EGR
iError = fresh.X_by_Equivalence_Ratio(
    MyGasMech,
    fuelmixture.X,
    air.X,
    add_frac,
    products,
    equivalenceratio=equiv,
    threshold=1.0e-8,
)

# list the composition of the fuel+air+EGR mixture for verification
fresh.list_composition(mode="mole", bound=1.0e-8)
```

Set up the HCCI engine reactor

Use the `HCCIengine()` method to create a multi-zone HCCI engine named `MyMZEngine` and make the new `fresh` mixture as the initial incylinder gas mixture at IVC. Set the `nzones` parameter to the number of zones in your multi-zone HCCI engine model.

```
# create a five-zone HCCI engine
numbzones = 5
MyMZEngine = HCCIengine(reactor_condition=fresh, nzones=numbzones)
# show initial gas composition inside the reactor
MyMZEngine.list_composition(mode="mole", bound=1.0e-8)
```

Set basic engine parameters

Set the required engine parameters as shown in the following code. These engine parameters are used to describe the cylinder volume during the simulation. The `starting_CA` argument should be the crank angle corresponding to the cylinder IVC. The `ending_CA` is typically the EVC crank angle.

```
# cylinder bore diameter [cm]
MyMZEngine.bore = 12.065
# engine stroke [cm]
MyMZEngine.stroke = 14.005
# connecting rod length [cm]
MyMZEngine.connecting_rod_length = 26.0093
# compression ratio [-]
MyMZEngine.compression_ratio = 16.5
# engine speed [RPM]
MyMZEngine.RPM = 1000

# set other parameters
# simulation start CA [degree]
MyMZEngine.starting_CA = -142.0
# simulation end CA [degree]
MyMZEngine.ending_CA = 116.0

# list the engine parameters
MyMZEngine.list_engine_parameters()
print(f"engine displacement volume {MyMZEngine.get_displacement_volume()} [cm3]")
print(f"engine clearance volume {MyMZEngine.get_clearance_volume()} [cm3]")
print(f"number of zone(s) = {MyMZEngine.get_number_of_zones()}")
```

Set up the engine wall heat transfer model

By default, the engine cylinder is adiabatic. You must set up a wall heat transfer model to include the heat loss effects in your engine simulation. Chemkin support three widely used engine wall heat transfer models. The models and their parameters follow.

- `dimensionless`: [<a> <b> <c> <Twall>]
- `dimensional`: [<a> <b> <c> <Twall>]
- `hohenburg`: [<a> <b> <c> <d> <e> <Twall>]

There is also the in-cylinder gas velocity correlation (the Woschni correlation) that is associated with the engine wall heat transfer models. Here are the parameters of the Woschni correlation:

`[<C11> <C12> <C2> <swirl ratio>]`

You can also specify the surface areas of the piston head and the cylinder head for more precision heat transfer wall area. By default, both the piston head and the cylinder head surfaces are flat.

```
heattransferparameters = [0.035, 0.71, 0.0]
# set cylinder wall temperature [K]
Twall = 400.0
MyMZEngine.set_wall_heat_transfer("dimensionless", heattransferparameters, Twall)
# in-cylinder gas velocity correlation parameter (Woschni)
# [<C11> <C12> <C2> <swirl ratio>]
GVparameters = [2.28, 0.308, 3.24, 0.0]
MyMZEngine.set_gas_velocity_correlation(GVparameters)
# set piston head top surface area [cm2]
MyMZEngine.set_piston_head_area(area=124.75)
# set cylinder clearance surface area [cm2]
MyMZEngine.set_cylinder_head_area(area=123.5)
```

Set zonal properties

By default, all zones in the multi-zone HCCI engine model have the same properties. You can artificially stratify the temperature and/or the equivalence ratio distribution in the cylinder at the IVC by utilizing the `set_zonal` methods of the `HCCI` object.

```
# zonal temperatures [K]
ztemperature = [447.5, 447.5, 447, 447, 447]
MyMZEngine.set_zonal_temperature(zonetemp=ztemperature)
# zonal volume fractions
zvolumefrac = [0.3, 0.25, 0.2, 0.2, 0.05]
MyMZEngine.set_zonal_volume_fraction(zonevol=zvolumefrac)
# wall heat transfer area fractions
zHTarea = [0.0, 0.15, 0.2, 0.25, 0.4]
MyMZEngine.set_zonal_heat_transfer_area_fraction(zonearea=zHTarea)
# zonal equivalence ratios
zphi = [equiv, equiv, equiv, equiv, equiv]
MyMZEngine.set_zonal_equivalence_ratio(zonephi=zphi)
# zonal EGR ratios
zEGRR = [0.3, 0.3, 0.3, 0.35, 0.35]
MyMZEngine.set_zonal_EGR_ratio(zoneegr=zEGRR)
# set fuel "molar" composition
MyMZEngine.define_fuel_composition([("CH4", 0.9), ("C3H8", 0.05), ("C2H6", 0.05)])
# set oxidizer "molar' composition
MyMZEngine.define_oxid_composition([("O2", 0.21), ("N2", 0.79)])
# set products
MyMZEngine.define_product_composition(["CO2", "H2O", "N2"])
# set EGR composition in mole fractions
zadd = [add_frac, add_frac, add_frac, add_frac, add_frac]
MyMZEngine.define_additive_fractions(addfrac=zadd)
```

Set output options

You can turn on the adaptive solution saving to resolve the steep variations in the solution profile. Here additional solution data points are saved for every 20*solver internal steps. You must include the `set_ignition_delay()` method for the engine model to report the ignition delay crank angle after the simulation is done. If `method="T_inflection"` is set, the reactor model treats the inflection points in the predicted gas temperature profile as the indication of an auto-ignition. You can choose a different auto-ignition definition.

Note

- Type `ansys.chemkin.show_ignition_definitions()` to get the list of all available ignition delay time definitions in Chemkin.
- The `set_ignition_delay()` method is required for the engine model to report the ignition delay time for each zone as well as the cylinder averaged ignition delay time derived from the cylinder averaged temperature profile.
- By default, time/crank angle intervals for both print and save solution are 1/100 of the simulation duration, which in this case is $dCA = (EVO - IVC)/100 = 2.58$. You can make the model report more frequently by using the `CAstep_for_saving_solution()` or the `CAstep_for_printing_solution()` method to set different interval values in the crank angle.

```
# set the number of crank angles between saving solution
MyMZEngine.CAstep_for_saving_solution = 0.5
# set the number of crank angles between printing solution
MyMZEngine.CAstep_for_printing_solution = 10.0
# turn on adaptive solution saving
MyMZEngine.adaptive_solution_saving(mode=True, steps=20)
# specify the ignition definitions
MyMZEngine.set_ignition_delay(method="T_inflection")
```

Set solver controls

You can overwrite the default solver controls by using solver-related methods, such as those for tolerances.

```
# set tolerances in tuple: (absolute tolerance, relative tolerance)
MyMZEngine.tolerances = (1.0e-12, 1.0e-10)
# get solver parameters
ATOL, RTOL = MyMZEngine.tolerances
print(f"Default absolute tolerance = {ATOL}.")
print(f"Default relative tolerance = {RTOL}")
# turn on the force non-negative solutions option in the solver
MyMZEngine.force_nonnegative = True
# show solver and output options
# show the number of crank angles between printing solution
print(
    f"Crank angles between solution printing: {MyMZEngine.CAstep_for_printing_solution}"
)
# show other transient solver setup
print(f"Forced non-negative solution values: {MyMZEngine.force_nonnegative}")
```

Display the added parameters (keywords)

Use the `showkeywordinputlines()` method to verify that the preceding parameters are correctly assigned to the engine model.

```
MyMZEngine.showkeywordinputlines()
```

Run the simulation

Use the `run()` method to start the multi-zone HCCI engine simulation.

```
runstatus = MyMZEngine.run()
# check run status
if runstatus != 0:
    # Run failed.
    print(Color.RED + ">>> Run failed. <<<", end=Color.END)
    exit()
# Run succeeded.
print(Color.GREEN + ">>> Run completed. <<<", end=Color.END)
```

Get the ignition delay crank angle from the solution

Use the `get_ignition_delay()` method to extract the cylinder averaged ignition delay crank angle (CA) after the run is completed.

```
# get ignition delay "time"
delayCA = MyMZEngine.get_ignition_delay()
print(f"Ignition delay CA = {delayCA} [degree].")
```

Get the heat release crank angles

The engine models also report the crank angles when the accumulated heat release reaches 10%, 50%, and 90% of the total heat release. Use the `get_engine_heat_release_CAs()` method to extract these heat release crank angles (CA).

```
HR10, HR50, HR90 = MyMZEngine.get_engine_heat_release_CAs()
print("Engine Heat Release Information:")
print(f"10% heat release CA = {HR10} [degree].")
print(f"50% heat release CA = {HR50} [degree].")
print(f"90% heat release CA = {HR90} [degree].\n")
```

Postprocess the solution ==========_== The postprocessing step parses the solution and packages the solution values at each time point into a mixture. There are two ways to access the solution profiles:

- The raw solution profiles (value as a function of time) are available for time, temperature, pressure, volume, and species mass fractions.
- The mixtures permit the use of all property and rate utilities to extract information such as viscosity, density, and mole fractions.

You can use the `get_solution_variable_profile()` method to get the raw solution profiles. You can get solution mixtures using either the `get_solution_mixture_at_index()` method for the solution mixture at a given time point or the `get_solution_mixture()` method for the solution mixture at a given time. (In this case, the mixture is constructed by interpolation.)

Note

- For engine models, use the `process_engine_solution()` method to postprocess the solutions.
- Use the `getnumbersolutionpoints()` method to get the size of the solution profiles before creating the arrays.
- Use the `get_CA()` method to convert the time values reported in the solution to crank angles.

Postprocess the solution profiles in selected zone

The solution of the multi-zone HCCI engine model contains the results of the individual zones plus the cylinder averaged results. This means that if there are n zones in the multi-zone engine model, there are (n+1) solution records: n zonal results and the cylinder averaged results.

To process the result of the zone number j, $(1 \leq j \leq n)$, set the parameter value of `zoneID` to j when you call the engine postprocessor with the `process_engine_solution()` method. Otherwise, the cylinder averaged results are postprocessed by default, that is, when the `zoneID` parameter is omitted.

Note

Because The `process_engine_solution()` method can process only one set of results at a time (one zonal result or the cylinder averaged result), you must postprocess the zones one by one to obtain all solution data of the multi-zone simulation.

```
thiszone = 1
MyMZEngine.process_engine_solution(zoneID=thiszone)
plottitle = "Zone " + str(thiszone) + " Solution"
# get the number of solution time points
solutionpoints = MyMZEngine.getnumbersolutionpoints()
print(f"Number of solution points = {solutionpoints}.")
# get the time profile
timeprofile = MyMZEngine.get_solution_variable_profile("time")
# convert time to crank angle
CAprofile = np.zeros_like(timeprofile, dtype=np.double)
count = 0
for t in timeprofile:
    CAprofile[count] = MyMZEngine.get_CA(timeprofile[count])
    count += 1
# get the cylinder pressure profile
presprofile = MyMZEngine.get_solution_variable_profile("pressure")
presprofile *= 1.0e-6
# get the zonal volume profile
volprofile = MyMZEngine.get_solution_variable_profile("volume")
# create arrays for zonal mixture density and mixture specific heat capacity
denprofile = np.zeros_like(timeprofile, dtype=np.double)
viscprofile = np.zeros_like(timeprofile, dtype=np.double)
# loop over all solution time points
for i in range(solutionpoints):
    # get the zonal mixture at the time point
    solutionmixture = MyMZEngine.get_solution_mixture_at_index(solution_index=i)
    # get zonal gas density [g/cm3]
    denprofile[i] = solutionmixture.RHO
    # get zonal mixture viscosity profile [g/cm-sec] or [Poise]
    viscprofile[i] = solutionmixture.mixture_viscosity() * 1.0e2

# post-process cylinder-averged solution
# do NOT set the zoneID parameter
MyMZEngine.process_average_engine_solution()
# get the cylinder volume profile
cylindervolprofile = MyMZEngine.get_solution_variable_profile("volume")
# create arrays for cylinder-averaged mixture density
```

(continues on next page)

(continued from previous page)

```
cylinderdenprofile = np.zeros_like(timeprofile, dtype=np.double)
# loop over all solution time points
for i in range(solutionpoints):
    # get the zonal mixture at the time point
    solutionmixture = MyMZEngine.get_solution_mixture_at_index(solution_index=i)
    # get zonal gas density [g/cm3]
    cylinderdenprofile[i] = solutionmixture.RHO
```

Plot the engine solution profiles

Plot the zonal and the cylinder averaged profiles from the multi-zone HCCI engine simulation.

Note

You can get profiles of the thermodynamic and the transport properties by applying `Mixture` utility methods to the solution mixtures.

```
plt.subplots(2, 2, sharex="col", figsize=(12, 6))
plt.suptitle(plottitle, fontsize=16)
plt.subplot(221)
plt.plot(CAprofile, presprofile, "r-")
plt.ylabel("Pressure [bar]")
plt.subplot(222)
plt.plot(CAprofile, volprofile, "b-")
plt.plot(CAprofile, cylindervolprofile, "b--")
plt.ylabel("Volume [cm3]")
plt.legend(["Zone", "Cylinder"], loc="upper right")
plt.subplot(223)
plt.plot(CAprofile, denprofile, "g-")
plt.plot(CAprofile, cylinderdenprofile, "g--")
plt.xlabel("Crank Angle [degree]")
plt.ylabel("Mixture Density [g/cm3]")
plt.legend(["Zone", "Averaged"], loc="upper left")
plt.subplot(224)
plt.plot(CAprofile, viscprofile, "m-")
plt.xlabel("Crank Angle [degree]")
plt.ylabel("Mixture Viscosity [cP]")
# plot results
if interactive:
    plt.show()
else:
    plt.savefig("plot_multizone_HCCI_engine.png", bbox_inches="tight")
```

12.4.3 Simulate a spark ignition engine

Ansys chemkin offers some idealized internal combustion (IC) engine models commonly used for fuel combustion and engine performance research. The Chemkin IC engine model is a specialized transient 0-D closed gas-phase reactor that mainly performs combustion simulation between the intake valve closing (IVC) and the exhaust valve opening (EVO), that is, when the engine cylinder resembles a closed chamber. The cylinder volume is derived from the piston motion as a function of the engine crank angle (CA) and engine parameters such as engine speed (RPM) and stroke. The energy equation is always solved. There are several wall heat transfer models specifically designed for engine

simulations.

Note

For additional information on Chemkin IC engine models, use the `ansys.chemkin.manuals()` method to view the online **Theory** manual.

The Chemkin spark ignition (SI) engine model offers a simple way to simulate the chemical kinetics taking place in the spark ignition engine. The Chemkin SI engine model does not predict the fuel mass burning rate profile. On the contrary, it requires the burning rate profile as input in the form of the Wiebe function parameters, the burn profile anchor points, or normalized profile data. The main uses of the Chemkin SI engine model are conducting parameter studies of the burning rate profile on engine performance, emissions, and the onset of engine knock due to end gas autoignition.

This example shows how to set up and run the simplest Chemkin IC engine model, the SI engine model. In addition to the basic engine parameters, many engine model-specific features such as the exhaust gas recirculation and the wall heat transfer can be included in the engine simulation.

Import PyChemkin packages and start the logger

```
import os
import time

import ansys.chemkin as ck  # Chemkin
from ansys.chemkin import Color

# chemkin spark ignition (SI) engine model (transient)
from ansys.chemkin.engines.SI import SIengine
from ansys.chemkin.logger import logger
import matplotlib.pyplot as plt  # plotting
import numpy as np  # number crunching

# check working directory
current_dir = os.getcwd()
logger.debug("working directory:" + current_dir)
# set verbose mode
ck.set_verbose(True)
# set interactive mode for plotting the results
# interactive = True: display plot
# interactive = False: save plot as a PNG file
global interactive
interactive = True
```

Create a chemistry set

For PRF, the encrypted 14-component gasoline mechanism, `gasoline_14comp_WBencrypted.inp`, is used. The chemistry set is named `gasoline`.

Note

Because this gasoline mechanism does not come with any transport data, you do not need to provide a transport data file.

```
# set mechanism directory (the default Chemkin mechanism data directory)
data_dir = os.path.join(ck.ansys_dir, "reaction", "data")
mechanism_dir = data_dir
# create a chemistry set based on the gasoline 14 components mechanism
MyGasMech = ck.Chemistry(label="Gasoline")
# set mechanism input files
# including the full file path is recommended
MyGasMech.chemfile = os.path.join(mechanism_dir, "gasoline_14comp_WBencrypt.inp")
```

Preprocess the gasoline chemistry set

```
# preprocess the mechanism files
iError = MyGasMech.preprocess()
print("Mechanism information:")
print(f"Number of gas species = {MyGasMech.KK:d}.")
print(f"Number of gas reactions = {MyGasMech.IIGas:d}.")
```

Set up the stoichiometric gasoline-air mixture

You must set up the stoichiometric gasoline-air mixture for the subsequent SI engine calculations. Here the `X_by_Equivalence_Ratio()``method is used. You create the ``fuel` and the `air` mixtures first. You then define the *complete combustion product species* and provide the *additives* composition if applicable. Finally, you simply set `equivalenceratio=1` to create the stoichiometric gasoline-air mixture.

For PRF 90 gasoline, the recipe is `[("ic8h18", 0.9), ("nc7h16", 0.1)]`.

```
# create the fuel mixture
fuelmixture = ck.Mixture(MyGasMech)
# set fuel composition
fuelmixture.X = [("ic8h18", 0.9), ("nc7h16", 0.1)]
# setting pressure and temperature is not required in this case
fuelmixture.pressure = ck.Patm
fuelmixture.temperature = 353.0

# create the oxidizer mixture: air
air = ck.Mixture(MyGasMech)
air.X = [("o2", 0.21), ("n2", 0.79)]
# setting pressure and temperature is not required in this case
air.pressure = ck.Patm
air.temperature = 353.0

# products from the complete combustion of the fuel mixture and air
products = ["co2", "h2o", "n2"]
# species mole fractions of added/inert mixture. You can also create an additives␣
↪mixture here.
add_frac = np.zeros(MyGasMech.KK, dtype=np.double)  # no additives: all zeros

# create the unburned fuel-air mixture
fresh = ck.Mixture(MyGasMech)
```

(continues on next page)

(continued from previous page)

```
# mean equivalence ratio
equiv = 1.0
iError = fresh.X_by_Equivalence_Ratio(
    MyGasMech, fuelmixture.X, air.X, add_frac, products, equivalenceratio=equiv
)
# check fuel-oxidizer mixture creation status
if iError != 0:
    print("Error: Failed to create the fuel-oxidizer mixture.")
    exit()
```

List the mixture composition

List the composition of the premixed mixture for verification.

```
fresh.list_composition(mode="mole")
```

Specify pressure and temperature of the fuel-air mixture

Since you are going to use `fresh` fuel-air mixture to instantiate the engine object later, setting the mixture pressure and temperature is equivalent to setting the initial temperature and pressure of the engine cylinder.

```
fresh.temperature = fuelmixture.temperature
fresh.pressure = fuelmixture.pressure
```

Add EGR to the fresh fuel-air mixture

Many engines have the configuration for exhaust gas recirculation (EGR). Chemkin engine models let you add the EGR mixture to the fresh fuel-air mixture entering the cylinder. If the engine you are modeling has EGR, you should have the EGR ratio, which is generally the volume ratio between the EGR mixture and the fresh fuel-air ratio. However, because you know nothing about the composition of the exhaust gas, you cannot simply combine these two mixtures. In this case, you use the `get_EGR_mole_fraction()` method to estimate the major components of the exhaust gas from the combustion of the fresh fuel-air mixture. The `threshold=1.0e-8` parameter tells the method to ignore any species with a mole fraction below the threshold value. Once you have the EGR mixture composition, use the `X_by_Equivalence_Ratio()` method a second time to re-create the fuel-air mixture `fresh` with the original `fuelmixture` and `air` mixtures, along with the EGR composition that you just got as the *additives*.

```
EGRratio = 0.3
# compute the EGR stream composition in mole fractions
add_frac = fresh.get_EGR_mole_fraction(EGRratio, threshold=1.0e-8)
# recreate the initial mixture with EGR
iError = fresh.X_by_Equivalence_Ratio(
    MyGasMech,
    fuelmixture.X,
    air.X,
    add_frac,
    products,
    equivalenceratio=equiv,
    threshold=1.0e-8,
)
# list the composition of the fuel+air+EGR mixture for verification
fresh.list_composition(mode="mole", bound=1.0e-8)
```

Set up the SI engine reactor

Use the `SIengine()` method to create an SI engine named `MyEngine` and make the new `fresh` mixture as the initial in-cylinder gas mixture at the IVC. The Chemkin SI engine model consists of two*zones: the unburned zone and the burned zone. At the IVC, the unburned zone is filled with the `fresh` mixture and occupies the entire engine cylinder. On the other hand, the burned zone is empty and has zero volume/mass.

```
MyEngine = SIengine(reactor_condition=fresh)
# show initial gas composition inside the reactor
MyEngine.list_composition(mode="mole", bound=1.0e-8)
```

Set up basic engine parameters

Set the required engine parameters as shown in the following code. These engine parameters are used to describe the cylinder volume during the simulation. The `starting_CA` argument should be the crank angle corresponding to the cylinder IVC. The `ending_CA` argument is typically the EVC crank angle.

```
# cylinder bore diameter [cm]
MyEngine.bore = 8.5
# engine stroke [cm]
MyEngine.stroke = 10.82
# connecting rod length [cm]
MyEngine.connecting_rod_length = 17.853
# compression ratio [-]
MyEngine.compression_ratio = 12
# engine speed [RPM]
MyEngine.RPM = 600

# set other parameters
# simulation start CA [degree]
MyEngine.starting_CA = -120.2
# simulation end CA [degree]
MyEngine.ending_CA = 139.8
# list the engine parameters
MyEngine.list_engine_parameters()
print(f"Engine displacement volume = {MyEngine.get_displacement_volume()} [cm3].")
print(f"Engine clearance volume = = {MyEngine.get_clearance_volume()} [cm3].")
```

Set up the SI engine specific parameters

Set up the mass burned fraction profile

The burning rate profile is required because the SI engine model uses it to determine the mass transfer rate from the unburned zone to the burned zone during the simulation. You can use one of three methods to specify the mass burning fraction profile for the SI engine simulation:

- Wiebe function

  You must provide two sets of parameters:

  - `set_burn_timing`: Start of combustion crank angle (CA) and burn duration.
  - `wiebe_parameters`: Wiebe function parameters.

- Burn profile anchor points

  You must provide one set of parameters: `set_burn_anchor_points`.

- Burned mass fraction profile data

  You must provide two sets of parameters:

  - `set_burn_timing`: Start of combustion CA and burn duration.
  - `set_mass_burned_profile`: Normalized burn rate profile.

```
# start of combustion CA
MyEngine.set_burn_timing(SOC=-14.5, duration=45.6)
MyEngine.wiebe_parameters(n=4.0, b=7.0)
# set minimum zonal mass [g]
MyEngine.set_minimum_zone_mass(minmass=1.0e-5)
```

Set up engine wall heat transfer model

By default, the engine cylinder is adiabatic. You must set up a wall heat transfer model to include the heat loss effects in your engine simulation. Chemkin supports three widely used engine wall heat transfer models. The models and their parameters follow:

- `dimensionless`: [<a> <b> <c> <Twall>]
- `dimensional`: [<a> <b> <c> <Twall>]
- `hohenburg`: [<a> <b> <c> <d> <e> <Twall>]

There is also the in-cylinder gas velocity correlation (the Woschni correlation) that is associated with the engine wall heat transfer models. Here are the parameters of the Woschni correlation:

`[<C11> <C12> <C2> <swirl ratio>]`

You can also specify the surface areas of the piston head and the cylinder head for more precision heat transfer wall area. By default, both the piston head and the cylinder head surfaces are flat.

```
heattransferparameters = [0.1, 0.8, 0.0]
# set cylinder wall temperature [K]
Twall = 434.0
MyEngine.set_wall_heat_transfer("dimensionless", heattransferparameters, Twall)
# in-cylinder gas velocity correlation parameter (Woschni)
# [<C11> <C12> <C2> <swirl ratio>]
GVparameters = [2.28, 0.318, 0.324, 0.0]
MyEngine.set_gas_velocity_correlation(GVparameters)
# set piston head top surface area [cm2]
MyEngine.set_piston_head_area(area=56.75)
# set cylinder clearance surface area [cm2]
MyEngine.set_cylinder_head_area(area=56.75)
```

Set output options

You can turn on the adaptive solution saving to resolve the steep variations in the solution profile. Here additional solution data point are saved for every 20 solver internal steps. Since ignition in SI engine is controlled by the spark timing, that is, the start of combustion CA of the chemkin SI engine model, the ignition delay time is not reported.

Note

By default, time/crank angle intervals for both print and save solution are 1/100 of the simulation duration, which in this case is $dCA = (EVO - IVC)/100 = 2.58$. You can make the model report more frequently by using

the `CAstep_for_saving_solution()` or `CAstep_for_printing_solution()` method to set different interval values in the CA.

```
# set the number of crank angles between saving solution
MyEngine.CAstep_for_saving_solution = 0.5
# set the number of crank angles between printing solution
MyEngine.CAstep_for_printing_solution = 10.0
# turn ON adaptive solution saving
MyEngine.adaptive_solution_saving(mode=True, steps=20)
```

Set solver controls

You can overwrite the default solver controls by using solver-related methods, such as those for tolerances.

```
# set tolerances in tuple: (absolute tolerance, relative tolerance)
MyEngine.tolerances = (1.0e-15, 1.0e-6)
# get solver parameters
ATOL, RTOL = MyEngine.tolerances
print(f"Default absolute tolerance = {ATOL}.")
print(f"Default relative tolerance = {RTOL}.")
# turn on the force non-negative solutions option in the solver
MyEngine.force_nonnegative = True
# show solver option
# show the number of crank angles between printing solution
print(
    f"Crank angles between solution printing: {MyEngine.CAstep_for_printing_solution}"
)
# show other transient solver setup
print(f"Forced non-negative solution values: {MyEngine.force_nonnegative}")
```

Display the added parameters (keywords)

Use the `showkeywordinputlines()` method to verify the preceding parameters are correctly assigned to the engine model.

```
MyEngine.showkeywordinputlines()
# set the start wall time
start_time = time.time()
```

Run the simulation

Use the `run()` method to start the SI engine simulation. The simulation might take more than 30 seconds because the gasoline mechanism contains a lot of species and reactions.

```
runstatus = MyEngine.run()
# compute the total runtime
runtime = time.time() - start_time
# check run status
if runstatus != 0:
    # Run failed.
    print(Color.RED + ">>> Run failed. <<<", end=Color.END)
    exit()
```

(continues on next page)

(continued from previous page)

```
# run success!
print(Color.GREEN + ">>> Run completed. <<<", end=Color.END)
print(f"Total simulation duration: {runtime} [sec]")
```

Postprocess the solution

The postprocessing step parses the solution and packages the solution values at each time point into a mixture. There are two ways to access the solution profiles:

- The raw solution profiles (value as a function of time) are available for time, temperature, pressure, volume, and species mass fractions.
- The mixtures permit the use of all property and rate utilities to extract information such as viscosity, density, and mole fractions.

You can use the `get_solution_variable_profile()` method to get the raw solution profiles. You can get solution mixtures using either the `get_solution_mixture_at_index()` method for the solution mixture at a given time point or the `get_solution_mixture()` method for the solution mixture at a given time. (In this case, the mixture is constructed by interpolation.)

Note

- For engine models, use the `process_engine_solution()` method to postprocess the solutions.
- Use the `getnumbersolutionpoints()` method to get the size of the solution profiles before creating the arrays.
- Use the `get_CA()` method to convert the time values reported in the solution to crank angles.

Postprocess the solution profiles in selected zone

The solution of the SI engine model contains the results of the *unburned zones* and *burned zone*, along with the cylinder averaged results. That is, there are three solution records. The unburned zone is designated as `zone 1`, and the burned zone is designated as `zone 2`. To process the result of the burned zone, you use `zoneID=2` when you call the engine postprocessor with the `process_engine_solution()` method. If you omit the `zoneID` parameter, the cylinder averaged results are be postprocessed by default.

Note

Because the `process_engine_solution` method can process only one set of result at a time (one zonal result or the cylinder averaged result), you must postprocess the zones one by one to obtain all solution data of the multi-zone simulation.

```
unburnedzone = 1
burnedzone = 2
zonestrings = ["Unburned Zone", "Burned Zone"]
thiszone = burnedzone
MyEngine.process_engine_solution(zoneID=thiszone)
plottitle = zonestrings[thiszone - 1] + " Solution"
# get the number of solution time points
solutionpoints = MyEngine.getnumbersolutionpoints()
print(f"Number of solution points = {solutionpoints}.")
```

(continues on next page)

(continued from previous page)

```
# get the time profile
timeprofile = MyEngine.get_solution_variable_profile("time")
# convert time to crank angle
CAprofile = np.zeros_like(timeprofile, dtype=np.double)
count = 0
for t in timeprofile:
    CAprofile[count] = MyEngine.get_CA(timeprofile[count])
    count += 1
# get the cylinder pressure profile
presprofile = MyEngine.get_solution_variable_profile("pressure")
presprofile *= 1.0e-6
# get zonal temperature profile [K]
tempprofile = MyEngine.get_solution_variable_profile("temperature")
# get the zonal volume profile
volprofile = MyEngine.get_solution_variable_profile("volume")
# create arrays for zonal CO mole fraction
# find CO species index
COindex = MyGasMech.get_specindex("co")
COprofile = np.zeros_like(timeprofile, dtype=np.double)
# loop over all solution time points
for i in range(solutionpoints):
    # get the zonal mixture at the time point
    solutionmixture = MyEngine.get_solution_mixture_at_index(solution_index=i)
    # get zonal CO mole fraction
    COprofile[i] = solutionmixture.X[COindex]

# postprocess cylinder-averged solution
MyEngine.process_average_engine_solution()
# get the cylinder volume profile
cylindervolprofile = MyEngine.get_solution_variable_profile("volume")
# create arrays for cylinder-averaged mixture temperature
cylindertempprofile = MyEngine.get_solution_variable_profile("temperature")
```

Plot the engine solution profiles

Plot the zonal and the cylinder averaged profiles from the SI engine simulation.

Note

You can get profiles of the thermodynamic and the transport properties by applying `Mixture` utility methods to the solution mixtures.

```
plt.subplots(2, 2, sharex="col", figsize=(13, 6.5))
plt.suptitle(plottitle, fontsize=16)
plt.subplot(221)
plt.plot(CAprofile, presprofile, "r-")
plt.ylabel("Pressure [bar]")
plt.subplot(222)
plt.plot(CAprofile, volprofile, "b-")
plt.plot(CAprofile, cylindervolprofile, "b--")
plt.ylabel("Volume [cm3]")
```

(continues on next page)

(continued from previous page)

```
plt.legend(["Zone", "Cylinder"], loc="lower right")
plt.subplot(223)
plt.plot(CAprofile, tempprofile, "g-")
plt.plot(CAprofile, cylindertempprofile, "g--")
plt.xlabel("Crank Angle [degree]")
plt.ylabel("Mixture Temperature [K]")
plt.legend(["Zone", "Averaged"], loc="upper left")
plt.subplot(224)
plt.plot(CAprofile, COprofile, "m-")
plt.xlabel("Crank Angle [degree]")
plt.ylabel("CO Mole Fraction")
# plot results
if interactive:
    plt.show()
else:
    plt.savefig("plot_spark_ignition_engine.png", bbox_inches="tight")
```

12.5 Perfectly stirred reactor

The *perfectly Stirred reactor (PSR)* model is a 0-D steady-state reactor. This ideal reactor model assumes the gas mixture inside is uniform and the properties of the outlet stream are exactly the same as the gas properties inside the reactor. A PSR can have either its pressure or its volume "specified", and the reactor temperature can either be solved from the energy equation or be "given" as an input parameter.

The examples will show the procedures of setting up single PSR simulations with different constraints and with multiple inlet streams.

12.5.1 Set up a PSR parameter study for the inlet stream equivalence ratio

Ansys Chemkin offers some idealized reactor models commonly used for studying chemical processes and for developing reaction mechanisms. The PSR (perfectly stirred reactor) model is a steady-state 0-D model of the open perfectly mixed gas-phase reactor. There is no limit on the number of inlets to the PSR. As soon as the inlet gases enter the reactor, they are thoroughly mixed with the gas mixture inside. The PSR has only one outlet, and the outlet gas is assumed to be exactly the same as the gas mixture in the PSR.

There are two basic types of PSR models:

- **constrained-pressure** (or set residence time)
- **constrained-volume**

By default, the PSR model is running under constant pressure. PyChemkin PSR models always require the connected inlets to be defined, that is, the total inlet flow rate to the PSR is always known. Therefore, either the residence time or the reactor volume is needed to satisfy the basic setup of the PSR model. In this case, you specify the residence time of the PSR, and the PSR model automatically calculates the reactor volume from the given residence time and the total inlet volumetric flow rate.

For each type of the PSR, you can choose either to specify the reactor temperature (as a fixed value or by a piecewise-linear profile) or to solve the energy conservation equation. In total, you get four variations of the PSR model.

The PSR model is mostly employed in chemical kinetics studies. By controlling the reactor temperature, pressure, and/or residence time, you can gain knowledge about the major intermediates of a complex chemical process and postulate possible reaction pathways. The parameter study in this example shows how the inlet equivalence ratio impacts the hydrogen combustion process in a fixed residence time PSR.

Import PyChemkin packages and start the logger

```
import os
import time

import ansys.chemkin as ck  # Chemkin
from ansys.chemkin import Color
from ansys.chemkin.inlet import Stream  # external gaseous inlet
from ansys.chemkin.logger import logger

# Chemkin PSR model (steady-state)
from ansys.chemkin.stirreactors.PSR import PSR_SetResTime_EnergyConservation as PSR
from ansys.chemkin.utilities import find_file
import matplotlib.pyplot as plt  # plotting
import numpy as np  # number crunching

# check working directory
current_dir = os.getcwd()
logger.debug("working directory: " + current_dir)
# set verbose mode
ck.set_verbose(True)
# set interactive mode for plotting the results
# interactive = True: display plot
# interactive = False: save plot as a PNG file
global interactive
interactive = True
```

Create a chemistry set

This example uses the encrypted hydrogen-ammonia mechanism, `Hydrogen-Ammonia-NOx_chem_MFL2021.inp`. This mechanism is developed under Chemkin's *Model Fuel Library* (MFL) project. Like the rest of the MFL mechanisms, it is in the *ModelFuelLibrary* in the `/reaction/data` directory of the standard Ansys Chemkin installation.

```
# set mechanism directory (the default Chemkin mechanism data directory)
data_dir = os.path.join(
    ck.ansys_dir, "reaction", "data", "ModelFuelLibrary", "Skeletal"
)
mechanism_dir = data_dir
# create a chemistry set based on the gasoline 14-components mechanism
MyGasMech = ck.Chemistry(label="hydrogen")
# set mechanism input files
# including the full file path is recommended
MyGasMech.chemfile = find_file(
    mechanism_dir,
    "Hydrogen-Ammonia-NOx_chem_MFL",
    "inp",
)
```

Preprocess the gasoline chemistry set

```
# preprocess the mechanism files
iError = MyGasMech.preprocess()
```

Set up the H_2-air stream

Instantiate a stream feed for the inlet gas mixture. The stream is a mixture with the addition of the inlet flow rate. You specify inlet gas properties in the same way as you set up a mixture. Here, the `X_by_Equivalence_Ratio()` method is used.

First create the fuel and air mixtures. Then, define the complete combustion product species and provide the additives composition if applicable. Finally, set the equivalence ratio to 1 to create the stoichiometric hydrogen-air mixture. Use the `mass_flowrate()` method to assign the inlet mass flow rate. Use a fixed inlet mass flow rate of 432 [g/sec] since you are to change the PSR residence time in the parameter study.

```
# create the fuel mixture
fuel = ck.Mixture(MyGasMech)
# set fuel composition: hydrogen diluted by nitrogen
fuel.X = [("h2", 0.8), ("n2", 0.2)]
# setting the pressure and temperature is not required in this case
fuel.pressure = ck.Patm
fuel.temperature = 298.0  # inlet temperature

# create the oxidizer mixture: air
air = ck.Mixture(MyGasMech)
air.X = [("o2", 0.21), ("n2", 0.79)]
# setting the pressure and temperature is not required in this case
air.pressure = fuel.pressure
air.temperature = fuel.temperature

# create the fuel-oxidizer inlet to the PSR
feed = Stream(MyGasMech, label="feed_1")
# products from the complete combustion of the fuel mixture and air
products = ["h2o", "n2"]
# species mole fractions of added/inert mixture. You can also create an additives␣
↪mixture here.
add_frac = np.zeros(MyGasMech.KK, dtype=np.double)  # no additives: all zeros
# mean equivalence ratio
equiv = 1.0
iError = feed.X_by_Equivalence_Ratio(
    MyGasMech, fuel.X, air.X, add_frac, products, equivalenceratio=equiv
)
# check fuel-oxidizer mixture creation status
if iError != 0:
    print("Error: Failed to create the fuel-oxidizer mixture.")
    exit()

# set reactor pressure [dynes/cm2]
feed.pressure = fuel.pressure
# set inlet gas temperature [K]
feed.temperature = fuel.temperature
# set inlet mass flow rate [g/sec]
feed.mass_flowrate = 432.0
```

Create the PSR to predict the gas composition of the outlet stream

Use the `PSR_SetResTime_EnergyConservation()` method to instantiate the PSR `sphere` object because the goal is to see how the residence time affects the hydrogen combustion process. The gas property of the inlet feed is applied as the estimated reactor condition of the `sphere` object by default. You can overwrite any estimated reactor conditions by using appropriate methods. For example, `sphere.temperature = 1700.0` changes the estimated reactor temperature from 298 to 1700 [K]. The residence time of the nominal case is set by the `residence_time()` method.

```
sphere = PSR(feed, label="PSR_1")
```

Set up additional reactor model parameters

Before you can run the simulation, you must provide reactor parameters, solver controls, and output instructions. For a steady-state PSR, you must provide either the residence time or reactor volume. You can also make changes to any estimated reactor conditions if desired.

```
# reset the estimated reactor temperature [K]
sphere.temperature = 1700.0
# set PSR residence time (sec): required for PSR_SetResTime_EnergyConservation model
sphere.residence_time = 3.0 * 1.0e-5
```

Connect the inlet to the reactor

You must connect at least one inlet to the open reactor. Use the `set_inlet()` method to add a stream to the PSR. Inversely, use the `remove_inlet()` to disconnect an inlet from the PSR.

Note

There is no limit on the number of inlets that can be connected to a PSR.

```
# connect the inlet to the reactor
sphere.set_inlet(feed)
```

Set solver controls

You can overwrite the default solver controls by using solver-related methods, such as those for tolerances. Here the tolerances that the steady-state solver is to use for the *steady-state search* and the *pseudo time stepping* stages are changed. Sometimes, during the iterations, some species mass fractions might become negative, causing the solver to report an error and stop. To overcome this issue, you can use the `set_species_floor()` method to provide a small cushion, allowing species mass fractions to go slightly negative by resetting the mass fraction floor value.

```
# reset the tolerances in the steady-state solver
sphere.steady_state_tolerances = (1.0e-9, 1.0e-6)
sphere.timestepping_tolerances = (1.0e-9, 1.0e-6)
# reset the gas species floor value in the steady-state solver
sphere.set_species_floor(-1.0e-10)
```

Run the inlet equivalence ratio parameter study

In the parameter study, the equivalence ratio ϕ of the inlet gas mixture is increased from 1.0 to 1.4. This is done by applying the `X_by_Equivalence_Ratio()` method on the feed inlet. Changing the equivalence ratio of the inlet gas mixture inevitably has impact on the inlet stream density and hence the inlet volumetric flow rate. By using the

PSR_SetResTime_EnergyConservation model, the PSR residence time remains constant for all runs. The effects from any variations of the inlet are reflected on the PSR volume.

Use the process_solution() method to convert the result from each PSR to a mixture. You can either overwrite the solution mixture or use a new one for each simulation result.

```
# inlet gas equivalence ratio increment
deltaequiv = 0.05
numbruns = 9
# solution arrays
inletequiv = np.zeros(numbruns, dtype=np.double)
tempSSsolution = np.zeros_like(inletequiv, dtype=np.double)
# set the start wall time
start_time = time.time()
# loop over all inlet temperature values
for i in range(numbruns):
    # run the PSR model
    runstatus = sphere.run()
    # check run status
    if runstatus != 0:
        # Run failed.
        print(Color.RED + ">>> Run failed. <<<", end=Color.END)
        exit()
    # Run succeeded.
    print(Color.GREEN + ">>> Run completed. <<<", end=Color.END)
    # postprocess the solution profiles
    solnmixture = sphere.process_solution()
    # print the steady-state solution values
    # print(f"Steady-state temperature = {solnmixture.temperature} [K].")
    # solnmixture.list_composition(mode="mole")
    # store solution values
    inletequiv[i] = equiv
    tempSSsolution[i] = solnmixture.temperature
    # update inlet gas equivalence ratio (composition)
    equiv += deltaequiv
    iError = feed.X_by_Equivalence_Ratio(
        MyGasMech, fuel.X, air.X, add_frac, products, equivalenceratio=equiv
    )
    # check fuel-oxidizer mixture creation status
    if iError != 0:
        print(f"Error encountered with inlet equivalence ratio = {equiv}.")
        exit()

# compute the total runtime
runtime = time.time() - start_time
print(f"Total simulation duration: {runtime} [sec] over {numbruns} runs.")
```

Plot the parameter study results

Plot the steady-state PSR temperature against the equivalence ratio of the inlet H_2-air mixture. You should see that the maximum combustion temperature does not correspond to the $\phi = 1$ mixture. Instead, the temperature peak occurs when the mixture is slightly fuel rich. You can run the same parameter study on a different fuel species such as CH_4 to see if you observe the same behavior.

```
plt.plot(inletequiv, tempSSsolution, "b-")
plt.xlabel("Inlet Gas Equivalence Ratio")
plt.ylabel("Reactor Temperature [K]")
plt.title("PSR Solution")
# plot results
if interactive:
    plt.show()
else:
    plt.savefig("plot_PSR_gas.png", bbox_inches="tight")
```

12.5.2 Run PSR calculations for mechanism validation

Ansys Chemkin offers some idealized reactor models commonly used for studying chemical processes and for developing reaction mechanisms. The PSR (perfectly stirred reactor) model is a steady-state 0-D model of the open perfectly mixed gas-phase reactor. There is no limit on the number of inlets to the PSR. As soon as the inlet gases enter the reactor, they are thoroughly mixed with the gas mixture inside. The PSR has only one outlet, and the outlet gas is assumed to be exactly the same as the gas mixture in the PSR.

There are two basic types of PSR models:

- **constrained-pressure** (or set residence time)
- **constrained-volume**

By default, the PSR is running under constant pressure. In this case, you specify the residence time of the PSR. For the constrained-volume type of application, you must provide the PSR volume. You can calculate the residence time from the reactor volume and the total inlet volumetric flow rate.

For each type of PSR, you either specify the reactor temperature (as a fixed value or by a piecewise-linear profile) or solve the energy conservation equation. In total, you get four variations of the PSR model.

The JSR (jet-stirred reactor is mostly employed in chemical kinetics studies. By controlling the reactor temperature, pressure, and/or residence time, you can gain knowledge about the major intermediates of a complex chemical process and postulate possible reaction pathways.

This example shows how to use the PSR model to validate the reaction mechanism against the measured data from hydrogen oxidation experiments.

```

```

Import PyChemkin packages and start the logger

```
import os
import time

import ansys.chemkin as ck  # Chemkin
from ansys.chemkin import Color
from ansys.chemkin.inlet import Stream  # external gaseous inlet
from ansys.chemkin.logger import logger

# chemkin perfectly stirred reactor (PSR) model (steady-state)
from ansys.chemkin.stirreactors.PSR import PSR_SetResTime_FixedTemperature as PSR
from ansys.chemkin.utilities import find_file
import matplotlib.pyplot as plt  # plotting
import numpy as np  # number crunching
```

(continues on next page)

(continued from previous page)

```
# check working directory
current_dir = os.getcwd()
logger.debug("working directory: " + current_dir)
# set verbose mode
ck.set_verbose(True)
# set interactive mode for plotting the results
# interactive = True: display plot
# interactive = False: save plot as a PNG file
global interactive
interactive = True
```

Create a chemistry set

This code uses the encrypted hydrogen-ammonia mechanism, `Hydrogen-Ammonia-NOx_chem_MFL2021.inp`. This mechanism is developed under Chemkin's **Model Fuel Library (MFL)** project. Like the rest of the MFL mechanisms, it is in the `ModelFuelLibrary` in the `/reaction/data` directory of the standard Ansys Chemkin installation.

```
# set mechanism directory (the default Chemkin mechanism data directory)
data_dir = os.path.join(
    ck.ansys_dir, "reaction", "data", "ModelFuelLibrary", "Skeletal"
)
mechanism_dir = data_dir
# create a chemistry set based on the hydrogen-ammonia mechanism
MyGasMech = ck.Chemistry(label="hydrogen")
# set mechanism input files
# including the full file path is recommended
MyGasMech.chemfile = find_file(
    mechanism_dir,
    "Hydrogen-Ammonia-NOx_chem_MFL",
    "inp",
)
```

Preprocess the gasoline chemistry set

```
# preprocess the mechanism files
iError = MyGasMech.preprocess()
```

Set up the H_2-O_2-N_2 stream

Instantiate a stream feed for the inlet gas mixture. A stream is a mixture with the addition of the inlet flow rate. You set up the inlet gas properties in the same way you set up a mixture. Set up the gas properties of the feed as described by the experiment.

Use the `mass_flowrate()` method to assign the inlet mass flow rate. The experiments use a fixed inlet mass flow rate of 0.11 [g/sec].

```
# create the fuel-oxidizer inlet to the JSR
feed = Stream(MyGasMech)
# set H2-O2-N2 composition
feed.X = [("h2", 1.1e-2), ("n2", 9.62e-1), ("o2", 2.75e-2)]
# set reactor pressure [dynes/cm2]
```

(continues on next page)

(continued from previous page)

```
feed.pressure = ck.Patm
# set inlet gas temperature [K]
temp = 800.0
feed.temperature = temp
# set inlet mass flow rate [g/sec]
feed.mass_flowrate = 0.11
```

Create the PSR to predict the gas composition of the outlet stream

Use the `PSR_SetResTime_FixedTemperature()` method to instantiate the JSR because both the reactor temperature and residence time are fixed during the experiments. The gas property of the inlet feed is applied as the estimated reactor condition of the JSR.

```
JSR = PSR(feed, label="JSR")
```

Connect the inlets to the reactor

You must connect at least one inlet to the open reactor. Use the `set_inlet()` method to add a stream to the PSR. Inversely, use the `remove_inlet()` method to disconnect an inlet from the PSR.

```
# connect the inlet to the reactor
JSR.set_inlet(feed)
```

Set up additional reactor model parameters

You must provide reactor parameters, solver controls, and output instructions before running the simulations. For the steady-state PSR, you must provide either the residence time or the reactor volume.

```
# set PSR residence time (sec): required for PSR_SetResTime_FixedTemperature model
JSR.residence_time = 120.0 * 1.0e-3
```

Set solver controls

You can overwrite the default solver controls by using solver-related methods, such as those for tolerances. Here, particular to this mechanism, a number of *pseudo timesteps* is required before attempting to actually search for the steady-state solution. This is done by using the steady-state solver control method, `set_initial_timesteps()`.

```
# set the number of initial pseudo timesteps in the steady-state solver
JSR.set_initial_timesteps(1000)
```

Run the parameter study to replicate the experiments

Different inlet and reactor temperatures are used in the experiments. The temperature value varies from 800 to 1050 [K].

Use the `process_solution()` method to convert the result from each PSR run to a mixture. You can either overwrite the solution mixture or use a new one for each simulation result.

```
# inlet gas temperature increment
deltatemp = 25.0
numbruns = 19
# find "h2o" species index
```

(continues on next page)

(continued from previous page)

```
H2Oindex = MyGasMech.get_specindex("h2o")
# solution arrays
inletTemp = np.zeros(numbruns, dtype=np.double)
h2oSSsolution = np.zeros_like(inletTemp, dtype=np.double)
# set the start wall time
start_time = time.time()
# loop over all inlet temperature values
for i in range(numbruns):
    # run the PSR model
    runstatus = JSR.run()
    # check run status
    if runstatus != 0:
        # Run failed.
        print(Color.RED + ">>> Run failed. <<<", end=Color.END)
        exit()
    # Run succeeded.
    print(Color.GREEN + ">>> Run completed. <<<", end=Color.END)
    # postprocess the solution profiles
    solnmixture = JSR.process_solution()
    # print the steady-state solution values
    # print(f"Steady-state temperature = {solnmixture.temperature} [K].")
    # solnmixture.list_composition(mode="mole")
    # store solution values
    inletTemp[i] = solnmixture.temperature
    h2oSSsolution[i] = solnmixture.X[H2Oindex]
    # update reactor temperature
    temp += deltatemp
    JSR.temperature = temp

# compute the total runtime
runtime = time.time() - start_time
print(f"Total simulation duration: {runtime} [sec] over {numbruns} runs")
#
# experimental data
# JSR temperature [K]
TEMP_data = [
    803.0,
    823.0,
    850.0,
    875.0,
    902.0,
    925.0,
    951.0,
    973.0,
    1002.0,
    1023.0,
    1048.0,
]
# measured H2O mole fractions in the JSR exit flow
H2O_data = [
    0.000312,
    0.000313,
```

(continues on next page)

(continued from previous page)

```
    0.000318,
    0.000303,
    0.003292,
    0.006226,
    0.008414,
    0.009745,
    0.010103,
    0.011036,
    0.010503,
]
```

Plot the ignition delay curve

Compare the predicted H_2O mole fraction to the measurement.

```
# plot results
plt.plot(inletTemp, h2oSSsolution, "b-", label="prediction")
plt.plot(TEMP_data, H2O_data, "ro", label="data", markersize=4, fillstyle="none")
plt.xlabel("Reactor Temperature [K]")
plt.ylabel("H2O Mole Fraction")
plt.legend(loc="lower right")
plt.title("JSR Solution")
# plot results
if interactive:
    plt.show()
else:
    plt.savefig("plot_jet_stirred_reactor.png", bbox_inches="tight")
```

12.5.3 Determine the impact of residence time on combustion in a PSR

Ansys Chemkin offers some idealized reactor models commonly used for studying chemical processes and for developing reaction mechanisms. The PSR (perfectly stirred reactor) model is a steady-state 0-D model of the open perfectly mixed gas-phase reactor. There is no limit on the number of inlets to the PSR. As soon as the inlet gases enter the reactor, they are thoroughly mixed with the gas mixture inside. The PSR has only one outlet, and the outlet gas is assumed to be exactly the same as the gas mixture in the PSR. There are two basic types of PSR models:

- **constrained-pressure** (or set residence time)
- **constrained-volume**

By default, the PSR model is running under constant pressure. The PyChemkin PSR models always require the connected inlets to be defined, that is, the total inlet flow rate to the PSR is always known. Therefore, either the residence time or the reactor volume is needed to satisfy the basic setup of the PSR model.

This example specifies the reactor volume of the PSR. The residence time is calculated from the reactor volume and the total inlet volumetric flow rate.

For each type of PSR model, you can either specify the reactor temperature (as a fixed value or by a piecewise-linear profile) or solve the energy conservation equation. In total, you get four variations of the PSR model.

PSR models are mostly employed in chemical kinetics studies. By controlling the reactor temperature, pressure, and/or residence time, you can gain knowledge about the major intermediates of a complex chemical process and postulate possible reaction pathways.

This example describes a parameter study of the influence of the PSR residence time on the hydrogen combustion process. It uses two inlet streams, one for the fuel mixture and the other for the air mixture. The fuel-to-air ratio inside

the PSR is determined by the mass or the volumetric flow rate ratio of the two inlet streams.

Import PyChemkin packages and start the logger

```
import os
import time

import ansys.chemkin as ck  # Chemkin
from ansys.chemkin import Color
from ansys.chemkin.inlet import Stream  # external gaseous inlet
from ansys.chemkin.logger import logger

# Chemkin PSR model (steady-state)
from ansys.chemkin.stirreactors.PSR import PSR_SetVolume_EnergyConservation as PSR
from ansys.chemkin.utilities import find_file
import matplotlib.pyplot as plt  # plotting
import numpy as np  # number crunching

# check working directory
current_dir = os.getcwd()
logger.debug("working directory: " + current_dir)
# set verbose mode
ck.set_verbose(True)
# set interactive mode for plotting the results
# interactive = True: display plot
# interactive = False: save plot as a PNG file
global interactive
interactive = True
```

Create a chemistry set

This example uses the encrypted hydrogen-ammonia mechanism, `Hydrogen-Ammonia-NOx_chem_MFL2021.inp`. This mechanism is developed under Chemkin's **Model Fuel Library (MFL)** project. Like the rest of the MFL mechanisms, it is located in `ModelFuelLibrary` in the `/reaction/data` directory of the standard Ansys Chemkin installation.

```
# set mechanism directory (the default Chemkin mechanism data directory)
data_dir = os.path.join(
    ck.ansys_dir, "reaction", "data", "ModelFuelLibrary", "Skeletal"
)
mechanism_dir = data_dir
# create a chemistry set based on the gasoline 14 components mechanism
MyGasMech = ck.Chemistry(label="hydrogen")
# set mechanism input files
# including the full file path is recommended
MyGasMech.chemfile = find_file(
    mechanism_dir,
    "Hydrogen-Ammonia-NOx_chem_MFL",
    "inp",
)
```

Preprocess the gasoline chemistry set

```
# preprocess the mechanism files
iError = MyGasMech.preprocess()
```

Set up the fuel and air streams

Instantiate a stream named `fuel` for the inlet stream containing the fuel mixture and a stream named `air` for the inlet stream containing the air mixture. The `Inlet` object is a mixture with the addition of the inlet flow rate.

You specify inlet gas properties in the same way as you set up a mixture. Here, the `fuel` and `air` inlets are created separately. You can adjust their inlet volumetric flow rates to create the desired hydrogen-air mixture. You use the `vol_flowrate()` method to assign the inlet volumetric flow rate. In this project, the `fuel` and `air` inlets have fixed volumetric flow rates of 25 and 50 [cm3/sec], respectively.

Note

This equation is used to determine the hydrogen-to-oxygen molar ratio:

$$H_2 : O_2 = 0.21[cm^3/cm^3] * 25[cm^3/s] : 0.21[cm^3/cm^3] * 50[cm^3/s] = 1 : 2$$

Note

The PSR residence time is specified indirectly through the reactor volume in the parameter study.

```
# create the fuel inlet
fuel = Stream(MyGasMech, label="Fuel")
# set fuel composition
fuel.X = [("h2", 0.21), ("n2", 0.79)]
# setting pressure and temperature is not required in this case
fuel.pressure = ck.Patm
fuel.temperature = 450.0  # inlet temperature
# set inlet volumetric flow rate [cm3/sec]
fuel.vol_flowrate = 25.0

# create the oxidizer inlet: air
air = Stream(MyGasMech, label="Oxid")
air.X = [("o2", 0.21), ("n2", 0.79)]
# setting pressure and temperature is not required in this case
air.pressure = fuel.pressure
air.temperature = fuel.temperature
# set inlet volumetric flow rate [cm3/sec]
air.vol_flowrate = 50.0
```

Create the PSR to predict the gas composition of the outlet stream

Use the `PSR_SetVolume_EnergyConservation()` method to instantiate a PSR named `combustor`. You must include the energy equation because the goal is to see how the residence time would affect the hydrogen combustion process. The `combustor` PSR is initiated with the parameter set to the `fuel` inlet, Hence, the gas property of the `fuel` inlet is applied as the estimated reactor condition of the `combustor` PSR. You can overwrite any estimated reactor conditions by using appropriate methods. For example, `combustor.temperature = 2000.0` changes the estimated

reactor temperature from 450 (the temperature of the `fuel` inlet) to 2000 [K]. The `volume()` method sets the reactor volume of the nominal case.

```
# create a PSR with fixed reactor volume and
# with the fuel inlet composition as the estimated reactor condition
combustor = PSR(fuel, label="tincan")
```

Set up additional reactor model parameters

You must provide reactor parameters, solver controls, and output instructions before running the simulations. For the steady-state PSR, you must provide either the residence time or the reactor volume. You can also make changes to any estimated reactor conditions if desired.

```
# reset the estimated reactor temperature [K]
combustor.temperature = 2000.0
# set the reactor volume (cm3): required for PSR_SetVolume_EnergyConservation model
combustor.volume = 200.0
```

Connect the inlets to the reactor

You must connect at least one inlet to the open reactor. Use the `set_inlet()` method to add a stream object to the PSR. Inversely, use the `remove_inlet()` to disconnect an inlet from the PSR. Here two inlets, `fuel` and `air`, are connected to the `combustor` PSR. The fuel-to-air ratio is controlled by the mass or the volumetric flow rate ratio of the `fuel` and the `air` inlets.

```
# add external inlets to the PSR
combustor.set_inlet(fuel)
combustor.set_inlet(air)
```

Set solver controls

You can overwrite the default solver controls by using solver-related methods, such as those for tolerances. The following code changes the tolerances that the steady-state solver is to use for the steady-state search and changes the pseudo time stepping stages. Sometimes, during the iterations, some species mass fractions might become negative, causing the solver to report an error and stop. To overcome this issue, you can provide a small cushion to allow species mass fractions to go slightly negative by using the # `set_species_floor()` method to reset the mass fraction floor value.

```
# reset the tolerances in the steady-state solver
combustor.steady_state_tolerances = (1.0e-9, 1.0e-6)
combustor.timestepping_tolerances = (1.0e-9, 1.0e-6)
# reset the gas species floor value in the steady-state solver
combustor.set_species_floor(-1.0e-10)
```

Run the PSR residence time parameter study

The PSR residence time τ is calculated as follows:

$$\tau = \frac{reactor\ volume}{total\ volumetric\ flow\ rate}$$

In this parameter study, the reactor volume is decreased from 200 to 160 [cm3]. Accordingly, τ is decreased from 2.67 to 2.13 [sec] because the total inlet volumetric flow rate is kept constant at 75 [cm3/sec]. Usually you want to run the burning cases (large residence times) first in the PSR parameter study.

The `process_solution` method converts the result from each PSR run to a mixture. You can either overwrite the solution mixture or use a new one for each simulation result.

```
# reactor volume increment
deltaVol = -5
numbruns = 9
# solution arrays
residencetime = np.zeros(numbruns, dtype=np.double)
tempSSsolution = np.zeros_like(residencetime, dtype=np.double)
# set the start wall time
start_time = time.time()
# loop over all inlet temperature values
for i in range(numbruns):
    # run the PSR model
    runstatus = combustor.run()
    # check run status
    if runstatus != 0:
        # Run failed.
        print(Color.RED + ">>> Run failed. <<<", end=Color.END)
        exit()
    # Run succeeded.
    print(Color.GREEN + ">>> Run completed. <<<", end=Color.END)
    # postprocess the solution profiles
    solnmixture = combustor.process_solution()
    # print the steady-state solution values
    print(f"steady-state temperature = {solnmixture.temperature} [K]")
    # solnmixture.list_composition(mode="mole")
    # store solution values
    # final reactor gas density [g/cm3]
    # density = solnmixture.RHO
    # final reactor mass [g]
    # mass = density * combustor.volume
    # PSR apparent residence time [sec]
    residencetime[i] = combustor.volume / combustor.net_vol_flowrate
    tempSSsolution[i] = solnmixture.temperature
    # update reactor volume
    combustor.volume += deltaVol

# compute the total runtime
runtime = time.time() - start_time
print(f"Total simulation duration: {runtime} [sec] over {numbruns} runs")
```

Plot the parameter study results

Plot the predicted PSR temperature against the residence time. You can observe that the hydrogen-air mixture ceases to burn when the residence time becomes too small. Gas turbine terminology refers to this as *blown out* because the fuel-air mixture gets blown out of the combustor before any significant chemical reaction can take place.

```
plt.plot(residencetime, tempSSsolution, "bo-")
plt.xlabel("Apparent Residence Time [sec]")
plt.ylabel("Exit Gas Temperature [K]")
plt.title("PSR Solution")
# plot results
if interactive:
    plt.show()
else:
```

(continues on next page)

(continued from previous page)

```
    plt.savefig("plot_multi_inlet_PSR.png", bbox_inches="tight")
```

12.6 Plug-flow reactor

Plug-Flow Reactor (PFR) model is an idealized 1-D steady-state flow reactor model with single inlet stream. The reactor pressure is assumed to be given, and the mass (flow rate) conservation is maintained by the velocity change. The gas temperature along the reactor can be either specified or solved by the energy equation.

12.6.1 Simulate NO reduction in combustion exhaust

The PFR (plug flow reactor) model is widely adopted in the chemical kinetic studies of emission abatement processes because of its simplicity in flow treatment as well as its resemblance of an exhaust duct or a tailpipe.

The PFR model is a steady-state open reactor because materials are allowed to move in and out of the reactor. The solutions are obtained along the length of the reactor as the inlet materials march toward the reactor exit. Typically, the pressure of the PFR is fixed. However, Chemkin does permit the use of a pressure profile to enforce a certain pressure gradient in the PFR.

Use the `PlugFlowReactor_FixedTemperature()` or `PlugFlowReactor_EnergyConservation()` method to create a PFR. Unlike the closed batch reactor model, which is instantiated by a mixture, the open reactor model, such as the PFR model, is initiated by a stream, which is simply a mixture with the addition of the inlet mass/volumetric flow rate or velocity. You already know how to create a stream if you know how to create a mixture. You can specify the inlet flow rate using one of these methods: `velocity()`, `mass_flowrate()`, `vol_flowrate()`, or `sccm()` (standard cubic centimeters per minute).

This example shows how to use the Chemkin PFR model to study the reduction of nitric oxide (NO) in the combustion exhaust by using the CH_4 reburning process. This is achieved by injecting CH_4 into the hot exhaust gas mixture. As the exhaust gas travels along the tubular reactor, the injected CH_4 is oxidized by the NO to form N_2, CO, and H_2. This is why this NO reduction process is called *methane reburning*.

Import PyChemkin packages and start the logger

```
import os
import time

import ansys.chemkin as ck  # Chemkin
from ansys.chemkin import Color

# chemkin plug flow reactor model
from ansys.chemkin.flowreactors.PFR import PlugFlowReactor_FixedTemperature
from ansys.chemkin.inlet import Stream
from ansys.chemkin.logger import logger
import matplotlib.pyplot as plt  # plotting
import numpy as np  # number crunching

# check working directory
current_dir = os.getcwd()
logger.debug("working directory: " + current_dir)
# set interactive mode for plotting the results
# interactive = True: display plot
```

(continues on next page)

(continued from previous page)

```
# interactive = False: save plot as a PNG file
global interactive
interactive = True
```

Create a chemistry set

The mechanism to load is the C2 NOx mechanism for the combustion of C1-C2 hydrocarbons. The mechanism comes with the standard Ansys Chemkin installation in the `/reaction/data` directory.

```
# set mechanism directory (the default Chemkin mechanism data directory)
data_dir = os.path.join(ck.ansys_dir, "reaction", "data")
mechanism_dir = data_dir
# create a chemistry set based on the GRI mechanism
MyGasMech = ck.Chemistry(label="C2 NOx")
# set mechanism input files
# including the full file path is recommended
MyGasMech.chemfile = os.path.join(mechanism_dir, "C2_NOx_SRK.inp")
```

Preprocess the GRI 3.0 chemistry set

```
# preprocess the mechanism files
iError = MyGasMech.preprocess()
```

Instantiate and set up the stream

Create a hot exhaust gas mixture by assigning the mole fractions of the species. A stream is simply a mixture with the addition of mass/volumetric flow rate or velocity. Here, the gas composition `exhaust.X` is given by a recipe consisting of the common species in the combustion exhaust, such as CO_2 and $H_{2`O and the CH:sub:`4}$ injected. The inlet velocity is given by `exhaust.velocity = 26.815`.

```
# create the inlet (mixture + flow rate)
exhaust = Stream(MyGasMech)
# set inlet temperature [K]
exhaust.temperature = 1750.0
# set inlet/PFR pressure [atm]
exhaust.pressure = 0.83 * ck.Patm
# set inlet molar composition directly
exhaust.X = [
    ("CO", 0.0145),
    ("CO2", 0.0735),
    ("H2O", 0.1107),
    ("NO", 0.0012),
    ("N2", 0.7981),
    ("CH4", 0.002),
]
# set inlet velocity [cm/sec]
exhaust.velocity = 26.815
#
```

Create the plug flow reactor object

Use the `PlugFlowReactor_FixedTemperature()` method to create a plug flow reactor. The required input parameter of the open reactor models in PyChemkin is the stream, while PyChemkin batch reactor models take a mixture as the input parameter. In this case, the exhaust stream is used to create a PFR named `tubereactor`.

```
tubereactor = PlugFlowReactor_FixedTemperature(exhaust)
```

Set up additional reactor model parameters

For the PFR, the required reactor parameters are the reactor diameter [cm] or the cross-sectional flow area [cm2] and the reactor length [cm]. The mixture condition and inlet mass flow rate of the inlet are already defined by the stream when the `tubereactor` PFR is instantiated.

```
# set PFR diameter [cm]
tubereactor.diameter = 5.8431
# set PFR length [cm]
tubereactor.length = 5.0
```

Verify the inlet condition of the PFR

```
print(f"PFR inlet mass flow rate {tubereactor.mass_flowrate} [g/sec]")
print(f"PFR inlet velocity {tubereactor.velocity} [cm/sec]")
# show inlet gas composition of the PFR
print("PFR inlet gas compsition")
tubereactor.list_composition(mode="mass", bound=1.0e-8)
```

Set output options

You can turn on adaptive solution saving to resolve the steep variations in the solution profile. Here, additional solution data points are saved for every **100** internal solver steps.

Note

By default, distance intervals for both print and save solution are 1/100 of the reactor length. In this case, $dt = time/100 = 0.001$. You can change them to different values.

```
# set distance between saving solution
tubereactor.timestep_for_saving_solution = 0.0005
# turn on adaptive solution saving
tubereactor.adaptive_solution_saving(mode=True, steps=100)
```

Display the added parameters (keywords)

Use the `showkeywordinputlines()` method to verify that the preceding parameters are correctly assigned to the reactor model.

```
tubereactor.showkeywordinputlines()
```

Run the parameter study to replicate the experiments

Use the `run()` method to start the plug flow simulation.

Note

You can use two `time` calls (one before the run and one after the run) to get the simulation run time (wall time).

```
# set the start wall time
start_time = time.time()
# run the PFR model
runstatus = tubereactor.run()
# compute the total runtime
runtime = time.time() - start_time
# check run status
if runstatus != 0:
    # Run failed.
    print(Color.RED + ">>> Run failed. <<<", end=Color.END)
    exit()
# run succeeded.
print(Color.GREEN + ">>> Run completed. <<<", end=Color.END)
print(f"Total simulation duration: {runtime * 1.0e3} [msec]")
```

Postprocess the solution

The postprocessing step parses the solution and packages the solution values at each time point into a mixture. There are two ways to access the solution profiles:

- The raw solution profiles (value as a function of distance) are available for distance, temperature, pressure, volume, and species mass fractions.
- The mixtures permit the use of all property and rate utilities to extract information such as viscosity, density, and mole fractions.

You can use the `get_solution_variable_profile()` method to get the raw solution profiles. You can get solution mixtures using either the `get_solution_mixture_at_index()` method for the solution mixture at the given saved location or the `get_solution_mixture()` method for the solution mixture at the given distance. (In this case, the mixture is constructed by interpolation.)

You can get the outlet solution by simply grabbing the solution values at the very last point by using $(outletsolutionindex) = (numberofsolutions) - 1$.

```
# postprocess the solution profiles
tubereactor.process_solution()

# get the number of solution time points
solutionpoints = tubereactor.getnumbersolutionpoints()
print(f"number of solution points = {solutionpoints}")

# get the grid profile [cm]
xprofile = tubereactor.get_solution_variable_profile("time")
# get the temperature profile [K]
tempprofile = tubereactor.get_solution_variable_profile("temperature")
# get the CH4 mass fraction profile
```

(continues on next page)

(continued from previous page)

```
YCH4profile = tubereactor.get_solution_variable_profile("CH4")
# get the CO mass fraction profile
YCOprofile = tubereactor.get_solution_variable_profile("CO")
# get the H2O mass fraction profile
YH2Oprofile = tubereactor.get_solution_variable_profile("H2O")
# get the H2 mass fraction profile
YH2profile = tubereactor.get_solution_variable_profile("H2")
# get the NO mass fraction profile
YNOprofile = tubereactor.get_solution_variable_profile("NO")
# get the N2 mass fraction profile
YN2profile = tubereactor.get_solution_variable_profile("N2")

# inlet NO mass fraction
YNO_inlet = exhaust.Y[MyGasMech.get_specindex("NO")]
# outlet grid index
xout_index = solutionpoints - 1
print("At the reactor inlet: x = 0 [xm]")
print(f"The NO mass fraction = {YNO_inlet}.")
print(f"At the reactor outlet: x = {xprofile[xout_index]} [cm]")
print(f"The NO mass fraction = {YNOprofile[xout_index]}.")
print(
    "The NO conversion rate = "
    + f"{(YNO_inlet - YNOprofile[xout_index]) / YNO_inlet * 100.0} %\n."
)

# more involved postprocessing using mixtures
#
# create arrays for the gas velocity profile
velocityprofile = np.zeros_like(xprofile, dtype=np.double)
# reactor mass flow rate (constant) [g/sec]
massflowrate = tubereactor.mass_flowrate
# reactor cross-section area [cm2]
areaflow = tubereactor.flowarea
print(f"Mass flow rate: {massflowrate} [g/sec]\nflow area: {areaflow} [cm2]")
# ratio
ratio = massflowrate / areaflow
# loop over all solution time points
for i in range(solutionpoints):
    # get the mixture at the time point
    solutionmixture = tubereactor.get_solution_mixture_at_index(solution_index=i)
    # get gas density [g/cm3]
    den = solutionmixture.RHO
    # gas velocity [g]
    velocityprofile[i] = ratio / den
```

Plot the solution profiles

Plot the species and the gas velocity profiles along the tubular reactor.

You can see from the plots that CH_4 is mainly oxidized by NO to form N_2 and H_2in the hot exhaust. The gas velocity accelerates because of the net increase in total number of moles of gas species due to the chemical reactions. You can conduct more in depth analyses of the reaction pathways of the CH_4 reburning mechanism by applying other PyChemkin utilities.

```
plt.subplots(2, 2, sharex="col", figsize=(12, 6))
plt.suptitle("Constant Temperature Plug-Flow Reactor", fontsize=16)
plt.subplot(221)
plt.plot(xprofile, YN2profile, "r-")
plt.ylabel("N2 Mass Fraction")
plt.subplot(222)
plt.plot(xprofile, YH2Oprofile, "b-", label="H2O")
plt.ylabel("H2O Mass Fraction")
plt.subplot(223)
plt.plot(xprofile, YNOprofile, "g-", label="NO")
plt.plot(xprofile, YCH4profile, "g:", label="CH4")
plt.plot(xprofile, YH2profile, "g--", label="H2")
plt.legend()
plt.xlabel("distance [cm]")
plt.ylabel("Mass Fraction")
plt.subplot(224)
plt.plot(xprofile, velocityprofile, "m-")
plt.xlabel("distance [cm]")
plt.ylabel("Gas Velocity [cm/sec]")
# plot results
if interactive:
    plt.show()
else:
    plt.savefig("plot_plug_flow_reactor.png", bbox_inches="tight")
```

12.7 PSR network

PyChemkin can be used to create an *Equivalence Reactor Network (ERN)* as a reduced-order (in geometry) model for complex combustion applications such as gas turbine combustor. Each reactor in the network represents a sub-region (zone) in the actual equipment. Normally these zones are created according to specific criteria such as temperature, equivalence ratio, or location; and the gas flow (mean, turbulent, and diffusive) between the zones becomes the internal streams between the reactors.

There are several ways to build an *ERN* in **PyChemkin**. The first method is to create and solve the reactors one-by-one starting from the most upstream (first) reactor. Adjust/manipulate the external and outlet streams from connected reactors (that is, solutions from the reactors) using the *Mixture*/*Stream* utilities and apply the desired stream as the inlet for the downstream reactors. The second method takes advantage of the *hybridreactornetwork* "model" of **PyChemkin**. Simply defined the reactors with the associated external inlets and add the reactors to the *hybridreactornetwork*. If there are *recycling* streams from the downstream reactor to the upstream ones, the *hybridreactornetwork* will solve the entire *ERN* iteratively. In this case, a *tear point* must be explicitly specified. The last method, the coupled method, is to create the network and its connectivity first, and the entire *ERN* will be solved in a coupled manner. This method is the default method in Chemkin GUI and is available as the *PSRCluster* "model" in **PyChemkin**.

Chemkin *ERN* has a few limitations:

- The first reactor of the reactor network must have at least one external inlet.
- when using the *coupled* model, the entire reactor network can have only one outlet to the surroundings, and the outlet must be attached to the last reactor in the network.
- PSR is the only reactor model allowed in the *PSRCluster* and the *hybridreactornetwork* reactor networks.

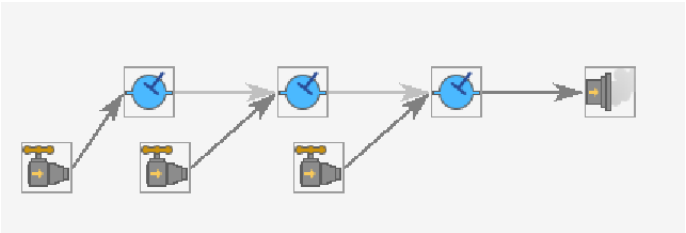
The *PSRChain_xxx* examples show the two methods to build and run a series of linked PSRs (no stream recycling) can be modeled in **PyChemkin**. The *PSRnetwork* example goes over the steps of creating and running an *ERN* with recycling streams by using the *hybridreactornetwork* method. The *PSRnetwork_coupled* example solves the same *ERN*

as the *PSRnetwork* example but utilizes the *coupled* method. Because the sole outlet from the *coupled ERN* must be attached to the last reactor, the order of the downstream zones is different from the *PSRnetwork* example.

12.7.1 Use a chain of individual reactors to model a gas combustor

This example shows how to set up and solve a series of linked PSRs (perfectly-stirred reactors). This is the simplest reactor network as it does not contain any recycling streams or outflow splittings.

Here is the PSR chain model of a fictional gas combustor.



The primary inlet stream to the first reactor, the *combustor*, is the fuel-lean methane-air mixture that is formed by mixing the fuel (methane) and the heated air. The exhaust from the combustor enters the second reactor, the *dilution zone*, where the hot combustion products are cooled by the introduction of additional cool air. The cooled and diluted gas mixture in the *dilution zone* then travels to the third reactor, the *reburning zone*. A mixture of fuel (methane) and carbon dioxide is injected to the gas in the reburning zone, attempting to convert any remaining carbon monoxide or nitric oxide in the exhaust gas to carbon dioxide or nitrogen, respectively.

This example solves the reactors one by one, from upstream to downstream. Once the solution of the upstream reactor is obtained, it is used to set up the external inlet of the immediate downstream reactor. This process continues until all reactors in the chain network are solved. Since there is no recycling stream in this configuration, the entire reactor network can be solved in one sweep.

Import PyChemkin packages and start the logger

```
import os
import time

import ansys.chemkin as ck  # Chemkin
from ansys.chemkin import Color
from ansys.chemkin.inlet import Stream  # external gaseous inlet
from ansys.chemkin.inlet import adiabatic_mixing_streams
from ansys.chemkin.logger import logger

# Chemkin PSR model (steady-state)
from ansys.chemkin.stirreactors.PSR import PSR_SetResTime_EnergyConservation as PSR

# check working directory
current_dir = os.getcwd()
logger.debug("working directory: " + current_dir)
# set verbose mode
ck.set_verbose(True)
# set interactive mode for plotting the results
```

(continues on next page)

(continued from previous page)

```
# interactive = True: display plot
# interactive = False: save plot as a PNG file
global interactive
interactive = True
```

Create a chemistry set

The mechanism to load is the GRI 3.0 mechanism for methane combustion. This mechanism and its associated data files come with the standard Ansys Chemkin installation in the `/reaction/data` directory.

```
# set mechanism directory (the default Chemkin mechanism data directory)
data_dir = os.path.join(ck.ansys_dir, "reaction", "data")
mechanism_dir = data_dir
# create a chemistry set based on the GRI mechanism
MyGasMech = ck.Chemistry(label="GRI 3.0")
# set mechanism input files
# including the full file path is recommended
MyGasMech.chemfile = os.path.join(mechanism_dir, "grimech30_chem.inp")
MyGasMech.thermfile = os.path.join(mechanism_dir, "grimech30_thermo.dat")
```

Preprocess the gasoline chemistry set

```
# preprocess the mechanism files
iError = MyGasMech.preprocess()
```

Set up gas mixtures based on the species in this chemistry set

Create the `fuel` and `air` mixtures to initialize the external inlet streams. The fuel for this case is pure methane.

```
# fuel is pure methane
fuel = Stream(MyGasMech)
fuel.temperature = 300.0  # [K]
fuel.pressure = 2.1 * ck.Patm  # [atm] => [dyne/cm2]
fuel.X = [("CH4", 1.0)]
fuel.mass_flowrate = 3.275  # [g/sec]

# air is modeled as a mixture of oxygen and nitrogen
air = Stream(MyGasMech)
air.temperature = 550.0  # [K]
air.pressure = 2.1 * ck.Patm
air.X = ck.Air.X()  # use predefined "air" recipe in mole fractions
air.mass_flowrate = 45.0  # [g/sec]
```

Create external inlet streams from the mixtures

Use the stream `adiabatic_mixing_streams()` method to combine the `fuel` and `air` streams. The final gas temperature should land between the temperatures of the two source streams. The mass flow rate of the `premixed` stream should be the sum of the sources. A simple PyChemkin composition recipe is used to create the `reburn_fuel` stream.

```
# premixed stream for the combustor
premixed = adiabatic_mixing_streams(fuel, air)
print(str(premixed.temperature))
```

(continues on next page)

(continued from previous page)

```
print(str(premixed.mass_flowrate))

# additional fuel injection for the reburning zone
reburn_fuel = Stream(MyGasMech)
reburn_fuel.temperature = 300.0  # [K]
reburn_fuel.pressure = 2.1 * ck.Patm  # [atm] => [dyne/cm2]
reburn_fuel.X = [("CH4", 0.6), ("CO2", 0.4)]
reburn_fuel.mass_flowrate = 0.12  # [g/sec]

# find the species index
CH4_index = MyGasMech.get_specindex("CH4")
O2_index = MyGasMech.get_specindex("O2")
NO_index = MyGasMech.get_specindex("NO")
CO_index = MyGasMech.get_specindex("CO")
```

Create PSRs for each zone

Set up the PSR for each zone one by one with external inlets only. PyChemkin requires that the first reactor/zone must have at least one external inlet. There are three reactors in the network. From upstream to downstream, they are `combustor`, `dilution zone`, and `reburning zone`. All of them have one external inlet. For the two downstream reactors, you must create the through-flow stream from their respective upstream reactor by using the solution of the upstream reactor. Use the `process_solution()` method to get the solution stream from the upstream reactor. Then, use the `set_inlet()` method to connect the resulting solution stream to the downstream reactor.

Note

- PyChemkin requires that the first reactor/zone must have at least one external inlet. The rest of the reactors have at least the through-flow from the immediate upstream reactor so they do not require an external inlet.
- The `Stream` parameter used to instantiate a PSR object is used to establish the *guessed reactor solution* and is modified when the network is solved by the `ERN`.
- The reactors in the network must be postprocessed individually.

```
# PSR #1: combustor
combustor = PSR(premixed, label="combustor")
# use the equilibrium state of the inlet gas mixture as the guessed solution
combustor.set_estimate_conditions(option="HP")
# set PSR residence time (sec): required for PSR_SetResTime_EnergyConservation model
combustor.residence_time = 2.0 * 1.0e-3
# add external inlet
combustor.set_inlet(premixed)
# set the start wall time
start_time = time.time()
# run PSR #1
status = combustor.run()
if status != 0:
    print(Color.RED + combustor.label + " Run failed.")
    exit()
# postprocess the solution profiles
solnstream1 = combustor.process_solution()
solnstream1.label = "PSR1"
```

(continues on next page)

(continued from previous page)

```
print("=" * 40)
print("Combustor exited.")
print("=" * 40)
print(f"Temperature = {solnstream1.temperature} [K].")
print(f"Mass flow rate = {solnstream1.mass_flowrate} [g/sec].")
print(f"CH4 = {solnstream1.X[CH4_index]}.")
print(f"O2 = {solnstream1.X[O2_index]}.")
print(f"CO = {solnstream1.X[CO_index]}.")
print(f"NO = {solnstream1.X[NO_index]}.")

# PSR #2: cooling
cooling = PSR(solnstream1, label="cooling zone")
# set PSR residence time (sec): required for PSR_SetResTime_EnergyConservation model
cooling.residence_time = 1.5 * 1.0e-3
# add external inlet
air.mass_flowrate = 62.0  # [g/sec]
cooling.set_inlet(air)
# add the through-flow from PSR #1
cooling.set_inlet(solnstream1)
# run PSR #2
status = cooling.run()
if status != 0:
    print(Color.RED + cooling.label + " Run failed.")
    exit()
# post-process the solution profiles
solnstream2 = cooling.process_solution()
solnstream2.label = "PSR2"
print()
print("=" * 40)
print("Dilution zone exited.")
print("=" * 40)
print(f"Temperature = {solnstream2.temperature} [K].")
print(f"Mass flow rate = {solnstream2.mass_flowrate} [g/sec].")
print(f"CH4 = {solnstream2.X[CH4_index]}.")
print(f"O2 = {solnstream2.X[O2_index]}.")
print(f"CO = {solnstream2.X[CO_index]}.")
print(f"NO = {solnstream2.X[NO_index]}.")

# PSR #3: reburn
reburn = PSR(solnstream2, label="reburn zone")
# set PSR residence time (sec): required for PSR_SetResTime_EnergyConservation model
reburn.residence_time = 3.5 * 1.0e-3
# add external inlet
reburn.set_inlet(reburn_fuel)
# add the through flow from PSR #2
reburn.set_inlet(solnstream2)
# run PSR #3
status = reburn.run()
if status != 0:
    print(Color.RED + reburn.label + " Run failed.")
    exit()
# post-process the solution profiles
```

(continues on next page)

(continued from previous page)

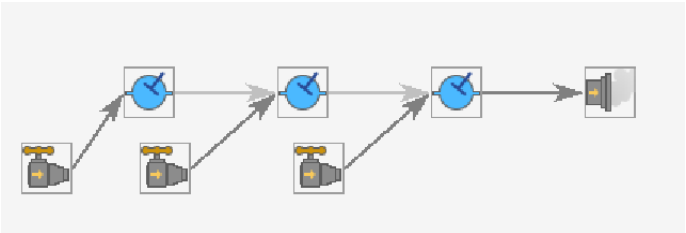
```
outflow = reburn.process_solution()
outflow.label = "outflow"
print()
print("=" * 40)
print("Outflow exited.")
print("=" * 40)
print(f"Temperature = {outflow.temperature} [K].")
print(f"Mass flow rate = {outflow.mass_flowrate} [g/sec].")
print(f"CH4 = {outflow.X[CH4_index]}.")
print(f"O2 = {outflow.X[O2_index]}.")
print(f"CO = {outflow.X[CO_index]}.")
print(f"NO = {outflow.X[NO_index]}.")

# compute the total runtime
runtime = time.time() - start_time
print()
print(f"Total simulation duration: {runtime} [sec].")
```

12.7.2 Use a chain reactor network to model a gas combustor

This example shows how to set up and solve a series of linked PSRs (perfectly-stirred reactors). This is the simplest reactor network as it does not contain any recycling streams or outflow splittings.

Here is a PSR chain model of a fictional gas combustor:



The primary inlet stream to the first reactor, the *combustor*, is the fuel-lean methane-air mixture that is formed by mixing the fuel (methane) and the heated air. The exhaust from the combustor enters the second reactor, the *dilution zone*, where the hot combustion products are cooled by the introduction of additional cool air. The cooled and diluted gas mixture in the dilution zone then travel to the third reactor, the *reburning zone*. A mixture of fuel (methane) and carbon dioxide is injected to the gas in the reburning zone, attempting to convert any remaining carbon monoxide or nitric oxide in the exhaust gas to carbon dioxide or nitrogen, respectively.

This example uses the `ReactorNetwork` module to configure and solve this chain reactor network. This module automatically handles the tasks of running the individual reactors and setting up the inlet to the downstream reactor.

Import PyChemkin packages and start the logger

```
import os
import time

import ansys.chemkin as ck  # Chemkin
```

(continues on next page)

(continued from previous page)

```
from ansys.chemkin import Color
from ansys.chemkin.hybridreactornetwork import ReactorNetwork as ERN
from ansys.chemkin.inlet import Stream  # external gaseous inlet
from ansys.chemkin.inlet import adiabatic_mixing_streams
from ansys.chemkin.logger import logger

# Chemkin PSR model (steady-state)
from ansys.chemkin.stirreactors.PSR import PSR_SetResTime_EnergyConservation as PSR

# check working directory
current_dir = os.getcwd()
logger.debug("working directory: " + current_dir)
# set verbose mode
ck.set_verbose(True)
```

Create a chemistry set

The mechanism to load is the GRI 3.0 mechanism for methane combustion. This mechanism and its associated data files come with the standard Ansys Chemkin installation in the `/reaction/data` directory.

```
# set mechanism directory (the default Chemkin mechanism data directory)
data_dir = os.path.join(ck.ansys_dir, "reaction", "data")
mechanism_dir = data_dir
# create a chemistry set based on the GRI mechanism
MyGasMech = ck.Chemistry(label="GRI 3.0")
# set mechanism input files
# including the full file path is recommended
MyGasMech.chemfile = os.path.join(mechanism_dir, "grimech30_chem.inp")
MyGasMech.thermfile = os.path.join(mechanism_dir, "grimech30_thermo.dat")
```

Preprocess the gasoline chemistry set

```
# preprocess the mechanism files
iError = MyGasMech.preprocess()
```

Set up gas mixtures based on the species in this chemistry set

Create the `fuel` and `air` streams before setting up the external inlet streams. The fuel in this case is pure methane. The main `premixed` inlet stream to the combustor is formed by mixing the `fuel` and `air` streams adiabatically. The fuel-to-air mass ratio is provided implicitly by the mass flow rates of the two streams. The external inlet to the second reactor, the `dilution zone`, is simply the `air` stream with a different mass flow rate (and different temperature if desirable). The `reburn_fuel` stream to inject to the downstream reburning zone is a mixture of methane and carbon dioxide.

Note

PyChemkin has `air` predefined as a convenient way to set up the air stream/mixture in simulations. Use the `ansys.chemkin.Air.X()` or `ansys.chemkin.Air.Y()` method when the mechanism uses "O2" and "N2" for oxygen and nitrogen. Use the `ansys.chemkin.air.X()` or `ansys.chemkin.air.Y()` method when the mechanism uses "o2" and "n2" for oxygen and nitrogen.

```
# fuel is pure methane
fuel = Stream(MyGasMech)
fuel.temperature = 300.0  # [K]
fuel.pressure = 2.1 * ck.Patm  # [atm] => [dyne/cm2]
fuel.X = [("CH4", 1.0)]
fuel.mass_flowrate = 3.275  # [g/sec]

# air is modeled as a mixture of oxygen and nitrogen
air = Stream(MyGasMech)
air.temperature = 550.0  # [K]
air.pressure = 2.1 * ck.Patm
# use predefined "air" recipe in mole fractions (with upper cased symbols)
air.X = ck.Air.X()
air.mass_flowrate = 45.0  # [g/sec]
```

Create external inlet streams from the mixtures

Use the `adiabatic_mixing_streams()` method to combine the `fuel` and the `air` streams. The final gas temperature should land between the temperatures of the two source streams. The mass flow rate of the `premixed` stream should be the sum of the sources. Use a simple PyChemkin composition recipe to create the `reburn_fuel` stream.

```
# premixed stream for the combustor
premixed = adiabatic_mixing_streams(fuel, air)

# verify the premixed stream properties
print(f"Premixed stream temperature = {premixed.temperature} [K].")
print(f"Premixed stream mass flow rate = {premixed.mass_flowrate} [g/sec].")

# additional fuel injection for the reburning zone
reburn_fuel = Stream(MyGasMech)
reburn_fuel.temperature = 300.0  # [K]
reburn_fuel.pressure = 2.1 * ck.Patm  # [atm] => [dyne/cm2]
reburn_fuel.X = [("CH4", 0.6), ("CO2", 0.4)]
reburn_fuel.mass_flowrate = 0.12  # [g/sec]

# find the species index
CH4_index = MyGasMech.get_specindex("CH4")
O2_index = MyGasMech.get_specindex("O2")
NO_index = MyGasMech.get_specindex("NO")
CO_index = MyGasMech.get_specindex("CO")
```

Create PSRs for each zone

Set up the PSR for each zone one by one with *external inlets only*. For PSR creation, use the `set_inlet()` method to add the external inlets to the reactor. A PFR always requires one external inlet when it is instantiated.

There are three reactors in the network. From upstream to downstream, they are `combustor`, `dilution zone`, and `reburning zone`. All of them have one external inlet.

Note

PyChemkin requires that the first reactor/zone must have at least one external inlet. Because the rest of the reactors have at least the through flow from the immediate upstream reactor, they do not require an external inlet.

Note

The `Stream` parameter used to instantiate a PSR is used to establish the *guessed reactor solution* and is modified when the network is solved by the `ERN`.

```
# PSR #1: combustor
combustor = PSR(premixed, label="combustor")
# use the equilibrium state of the inlet gas mixture as the guessed solution
combustor.set_estimate_conditions(option="HP")
# set PSR residence time (sec): required for PSR_SetResTime_EnergyConservation model
combustor.residence_time = 2.0 * 1.0e-3
# add external inlet
combustor.set_inlet(premixed)

# PSR #2: dilution zone
dilution = PSR(premixed, label="dilution zone")
# set PSR residence time (sec): required for PSR_SetResTime_EnergyConservation model
dilution.residence_time = 1.5 * 1.0e-3
# add external inlet
# assign the correct mass flow rate to the "air" stream
air.mass_flowrate = 62.0  # [g/sec]
dilution.set_inlet(air)

# PSR #3: reburning zone
reburn = PSR(premixed, label="reburning zone")
# set PSR residence time (sec): required for PSR_SetResTime_EnergyConservation model
reburn.residence_time = 3.5 * 1.0e-3
# add external inlet
reburn.set_inlet(reburn_fuel)
```

Create the reactor network

Create a hybrid reactor network named `PSRChain` and use the `add_reactor()` method to add the reactors one by one from upstream to downstream. For a simple chain network such as the one used in this example, you do not need to define the connectivity among the reactors. The reactor network model automatically figures out the through-flow connections.

Note

- Use the `show_reactors()` method to get the list of reactors in the network in the order they are added.
- Use the `remove_reactor()` method to remove an existing reactor from the network by the reactor `name/label`. Similarly, use the `clear_connections()` method to undo the network connectivity.
- The order of the reactor addition is important as it dictates the solution

sequence and thus the convergence rate.

```
# instantiate the chain PSR network as a hybrid reactor network
PSRChain = ERN(MyGasMech)

# add the reactors from upstream to downstream
```

(continues on next page)

(continued from previous page)

```
PSRChain.add_reactor(combustor)
PSRChain.add_reactor(dilution)
PSRChain.add_reactor(reburn)

# list the reactors in the network
PSRChain.show_reactors()
```

Solve the reactor network

Use the `run()` method to solve the entire reactor network. The hybrid reactor network solves the reactors one by one in the order that they are added to the network.

```
# set the start wall time
start_time = time.time()

# solve the reactor network
status = PSRChain.run()
if status != 0:
    print(Color.RED + "Failed to solve the reactor network." + Color.END)
    exit()

# compute the total runtime
runtime = time.time() - start_time
print()
print(f"Total simulation duration: {runtime} [sec]")
```

Postprocess reactor network results

There are two ways to process results from a reactor network. You can extract the solution of an individual reactor member as a stream by using the `get_reactor_stream()` method to get the reactor by its name. Or, you can get the stream properties of a specific network outlet by using the `get_external_stream()` method to get the stream by its outlet index. Once you get the solution as a stream, you can use any stream or mixture method to further manipulate the solutions.

Note

Use the `number_external_outlets()` method to find out the number of external outlets of the reactor network.

```
# get the outlet stream from the reactor network solutions
# find the number of external outlet streams from the reactor network
print(f"Number of outlet streams = {PSRChain.number_external_outlets}.")

# get the first (and the only) external outlet stream properties
network_outflow = PSRChain.get_external_stream(1)
# set the stream label
network_outflow.label = "outflow"

# print the desired outlet stream properties
print()
print("=" * 10)
print("outflow")
```

(continues on next page)

(continued from previous page)

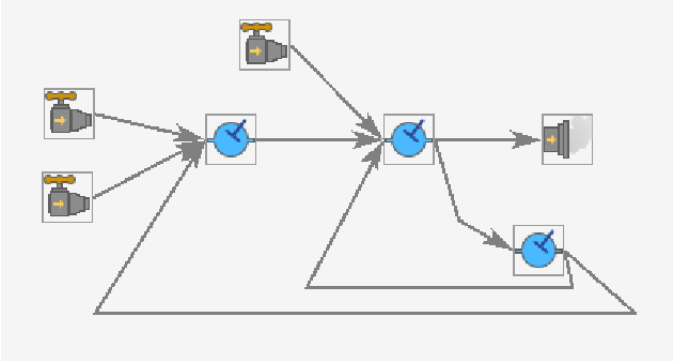
```
print("=" * 10)
print(f"temperature = {network_outflow.temperature} [K]")
print(f"mass flow rate = {network_outflow.mass_flowrate} [g/sec]")
print(f"CH4 = {network_outflow.X[CH4_index]}")
print(f"O2 = {network_outflow.X[O2_index]}")
print(f"CO = {network_outflow.X[CO_index]}")
print(f"NO = {network_outflow.X[NO_index]}")
```

12.7.3 Simulate a combustor using an equivalent reactor network with stream recycling

This example shows how to set up and solve an ERN (equivalent reactor network) in PyChemkin.

An ERN is employed as a reduced-order model to simulate the steady-state combustion process inside a gas turbine combustor chamber. This reduced-order reactor network model retains the complexity of the combustion chemistry by sacrificing details of the combustor geometries, the spatial resolution, and the mass and energy transfer processes. An ERN usually comprises PSRs (perfectly-stirred reactors) and PFRs (plug-flow reactors). The network configuration and connectivity, the reactor parameters, and the mass flow rates can be determined from "hot" steady-state CFD simulation results and/or from observations and measured data of the actual/similar devices. Once a reactor network is calibrated against the experimental data of a gas combustor, it becomes a handy tool for quickly estimating the emissions from the combustor when it is subjected to certain variations in the fuel compositions.

This figure shows a proposed ERN model of a fictional gas turbine combustor.



The *primary fuel* is mixed with the incoming air to form a *fuel-lean* mixture before entering the chamber through the primary inlet. Additional air, the *primary air*, is introduced to the combustion chamber separately through openings surrounding the primary inlet. The *secondary air* is entrained into the combustion chamber through well-placed holes on the liners at a location slightly downstream from the primary inlet.

The first PSR (reactor #1) represents the *mixing zone* around the main injector where the cool *premixed fuel-air* stream and the *primary air* stream are preheated by mixing with the hot combustion products from the *recirculation zone*.

Downstream from PSR #1, PSR #2, the *flame zone* is where the combustion of the heated fuel-air mixture takes place. The secondary air is injected here to cool down the combustion exhaust before it exits the combustion chamber. A portion of the exhaust gas coming out of PSR #2 does not leave the combustion chamber directly and is diverted to PSR #3, the *recirculation zone*.

The majority of the outlet flow from PSR #3 is recirculated back to the flame zone to sustain the fuel-lean premixed flame there. The rest of the hot gas from PSR #3 travels further back to PSR #1, the mixing zone, to preheat the fuel-lean mixture just entering the combustion chamber. Finally, the cooled flue gas leaves the chamber in a stream-like manner. Typically, a PFR is applied to simulation of the outflow.

The reactors in the network are solved individually one by one. When there is a *tear stream* in the network, the ERN should be solved iteratively. A tear stream is usually a *recycle" stream* that can serve as a pivot point for solving the recycle network iteratively. The convergence of the ERN is then determined by the absence of the per-iteration variation of the computed tear stream properties, such as species composition. In the current project, the recycling streams from PSR #3 to PSR #1 and PSR #2 are tear streams.

Import PyChemkin packages and start the logger ==========+======

```
import os
import time

import ansys.chemkin as ck  # Chemkin
from ansys.chemkin import Color
from ansys.chemkin.hybridreactornetwork import ReactorNetwork as ERN
from ansys.chemkin.inlet import Mixture
from ansys.chemkin.inlet import Stream  # external gaseous inlet
from ansys.chemkin.logger import logger

# Chemkin PSR model (steady-state)
from ansys.chemkin.stirreactors.PSR import PSR_SetResTime_EnergyConservation as PSR
import numpy as np  # number crunching

# check working directory
current_dir = os.getcwd()
logger.debug("working directory: " + current_dir)
# set verbose mode
ck.set_verbose(True)
# set interactive mode for plotting the results
# interactive = True: display plot
# interactive = False: save plot as a PNG file
global interactive
interactive = True
```

Create a chemistry set

The mechanism to load is the GRI 3.0 mechanism for methane combustion. This mechanism and its associated data files come with the standard Ansys Chemkin installation in the `/reaction/data` directory.

```
# set mechanism directory (the default Chemkin mechanism data directory)
data_dir = os.path.join(ck.ansys_dir, "reaction", "data")
mechanism_dir = data_dir
# create a chemistry set based on the GRI mechanism
MyGasMech = ck.Chemistry(label="GRI 3.0")
# set mechanism input files
# including the full file path is recommended
MyGasMech.chemfile = os.path.join(mechanism_dir, "grimech30_chem.inp")
MyGasMech.thermfile = os.path.join(mechanism_dir, "grimech30_thermo.dat")
```

Preprocess the gasoline chemistry set

```
# preprocess the mechanism files
iError = MyGasMech.preprocess()
```

Set up gas mixtures based on the species in this chemistry set

Create the "fuel" and the "air" mixtures to initialize the external inlet streams. The fuel for this case is pure methane.

```
# fuel is pure methane
fuel = Mixture(MyGasMech)
fuel.temperature = 650.0  # [K]
fuel.pressure = 10.0 * ck.Patm  # [atm] => [dyne/cm2]
fuel.X = [("CH4", 1.0)]

# air is modeled as a mixture of oxygen and nitrogen
air = Mixture(MyGasMech)
air.temperature = 650.0  # [K]
air.pressure = 10.0 * ck.Patm
air.X = ck.Air.X()  # mole fractions
```

Create external inlet streams from the mixtures

Create the `fuel` and the `air` streams/mixtures before setting up the external inlet streams. The fuel in this case is pure methane. The `X_by_Equivalence_Ratio()` method is used to form the main `premixed` inlet stream to the *mixing zone* reactor. The external inlets to the first reactor, the *mixing zone*, and the second reactor, the *flame zone*, are simply `air` mixtures with different mass flow rates and temperatures.

Note

PyChemkin has *air* redefined as a convenient way to set up the air stream/mixture in the simulations. Use the `ansys.chemkin.Air.X()` or `ansys.chemkin.Air.Y()` method when the mechanism uses `O2` and `N2` for oxygen and nitrogen. Use the `ansys.chemkin.air.X()` or `ansys.chemkin.air.Y()` method when oxygen and nitrogen are represented by `o2` and `n2`.

```
# primary fuel-air mixture
# products from the complete combustion of the fuel mixture and air
products = ["CO2", "H2O", "N2"]
# species mole fractions of added/inert mixture. (You can also create an additives␣
↪mixture here.)
add_frac = np.zeros(MyGasMech.KK, dtype=np.double)  # no additives: all zeros

# create the unburned fuel-air mixture
premixed = Stream(MyGasMech)
# mean equivalence ratio
equiv = 0.6
iError = premixed.X_by_Equivalence_Ratio(
    MyGasMech, fuel.X, air.X, add_frac, products, equivalenceratio=equiv
)
# check fuel-oxidizer mixture creation status
if iError != 0:
    print("Error: Failed to create the fuel-oxidizer mixture.")
```

(continues on next page)

(continued from previous page)

```
    exit()

# list the composition of the unburned fuel-air mixture
premixed.list_composition(mode="mole")

# set premixed fuel-air inlet temperature and mass flow rate
premixed.temperature = fuel.temperature
premixed.pressure = fuel.pressure
premixed.mass_flowrate = 500.0  # [g/sec]

# primary air stream to mix with the primary fuel-air stream
primary_air = Stream(MyGasMech, label="Primary_Air")
primary_air.X = air.X
primary_air.pressure = air.pressure
primary_air.temperature = air.temperature
primary_air.mass_flowrate = 50.0  # [g/sec]

# secondary bypass air stream
secondary_air = Stream(MyGasMech, label="Secondary_Air")
secondary_air.X = air.X
secondary_air.pressure = air.pressure
secondary_air.temperature = 670.0  # [K]
secondary_air.mass_flowrate = 100.0  # [g/sec]

# find the species index
CH4_index = MyGasMech.get_specindex("CH4")
O2_index = MyGasMech.get_specindex("O2")
NO_index = MyGasMech.get_specindex("NO")
CO_index = MyGasMech.get_specindex("CO")
```

Define reactors in the reactor network

Set up the PSR for each zone one by one with *external inlets only*. For PSRs, use the `set_inlet()` method to add the external inlets to the reactor. A PFR always requires one external inlet when it is instantiated. From upstream to downstream, they are `premix zone`, `flame zone`, and `recirculation zone`. The `recirculation zone` does not have an external inlet.

Note

PyChemkin requires that the first reactor/zone must have at least one external inlet. Because the rest of the reactors have at least the through-flow from the immediate upstream reactor, they do not require an external inlet.

Note

The stream parameter used to instantiate a PSR is used to establish the *guessed reactor solution* and is modified when the network is solved by the ERN.

```
# PSR #1: mixing zone
mix = PSR(premixed, label="mixing zone")
```

(continues on next page)

(continued from previous page)

```
# use different guess temperature
mix.set_estimate_conditions(option="TP", guess_temp=800.0)
# set PSR residence time (sec): required for PSR_SetResTime_EnergyConservation model
mix.residence_time = 0.5 * 1.0e-3
# add external inlets
mix.set_inlet(premixed)
mix.set_inlet(primary_air)


# PSR #2: flame zone
flame = PSR(premixed, label="flame zone")
# use the equilibrium state of the inlet gas mixture as the guessed solution
flame.set_estimate_conditions(option="TP", guess_temp=1600.0)
# set PSR residence time (sec): required for PSR_SetResTime_EnergyConservation model
flame.residence_time = 1.5 * 1.0e-3
# add external inlet
flame.set_inlet(secondary_air)

# PSR #3: recirculation zone
recirculation = PSR(premixed, label="recirculation zone")
# use the equilibrium state of the inlet gas mixture as the guessed solution
recirculation.set_estimate_conditions(option="TP", guess_temp=1600.0)
# set PSR residence time (sec): required for PSR_SetResTime_EnergyConservation model
recirculation.residence_time = 1.5 * 1.0e-3
```

Create the reactor network

Create a hybrid reactor network named `PSRnetwork` and add the reactors one by one from upstream to downstream using the `add_reactor()` method. For a simple chain network you do not need to define the connectivity among the reactors. The reactor network model automatically figures out the through-flow connections.

Note

- Use the `show_reactors()` method to get the list of reactors in the network in the order that they are added.
- Use the `remove_reactor()` method to remove an existing reactor from the network by the reactor `name/label`. Similarly, use the `clear_connections()` method to undo the network connectivity.
- The order of the reactor addition is important as it dictates the solution sequence and thus the convergence rate.

```
# instantiate the PSR network as a hybrid reactor network
PSRnetwork = ERN(MyGasMech)

# add the reactors from upstream to downstream
PSRnetwork.add_reactor(mix)
PSRnetwork.add_reactor(flame)
PSRnetwork.add_reactor(recirculation)

# list the reactors in the network
PSRnetwork.show_reactors()
```

Define the PSR connectivity

Because the current reactor network contains recycling streams, for example, from PSR #3 to PSR #1 and PSR #2, you must explicitly specify the network connectivity. To do this, you use a *split* dictionary to define the outflow connections among the PSRs in the network. The *key* of the split dictionary is the label of the *originate PSR*. The *value* is a list of tuples consisting of the label of the *target PSR* and its mass flow rate fraction.

```
{ "originate PSR" : [("target PSR1", fraction), ("target PSR2", fraction)],
  "another originate PSR" : [("target PSR1", fraction), ... ], ... }
```

For example, the outflow from reactor PSR1``is split into two streams. 90% of the mass flow rate goes to ``PSR2 and the rest is diverted to PSR5. The split dictionary entry for the outflow split of PSR1 is as follows:

```
"PSR1" : [("PSR2", 0.9), ("PSR5", 0.1)]
```

Note

The outlet flow from a reactor that is leaving the reactor network must be labeled as EXIT>> when you define the outflow splitting.

```
# PSR #1 outlet flow splitting
split_table = [(flame.label, 1.0)]
PSRnetwork.add_outflow_connections(mix.label, split_table)
# PSR #2 outlet flow splitting
# part of the outlet flow from PSR #2 exits the reactor network
split_table = [(recirculation.label, 0.2), ("EXIT>>", 0.8)]
PSRnetwork.add_outflow_connections(flame.label, split_table)
# PSR #3 outlet flow splitting
# PSR #3 is the last reactor of the network. However, it does not have an external␣
↪outlet.
split_table = [(mix.label, 0.15), (flame.label, 0.85)]
PSRnetwork.add_outflow_connections(recirculation.label, split_table)
```

Define the tear point for the iteration

Because the reactor network contains recycling streams, it must be solved iteratively by applying the *tear stream* method. You use the add_tearingpoint() `` method to explicitly define the *tear points* of the network. In this example, the stream recycling occurs at reactor #3, where the outlet flow is sent back to the upstream reactors, reactor #1 and reactor #2. Therefore, reactor #3, the *recirculation zone*, should be set as the only tear point of this reactor network. You use the ``set_tear_tolerance() method to set the relative tolerance for the overall reactor network convergence. The default tolerance is 1.0e-6.

```
# define the tear point reactor as reactor #3
PSRnetwork.add_tearingpoint(recirculation.label)

# reset the network relative tolerance
PSRnetwork.set_tear_tolerance(1.0e-5)
```

Solve the reactor network

Use the `run()` method to solve the entire reactor network. The `ReactorNetwork` hybrid solves the reactors one by one in the order that they are added to the network.

Note

Use the `set_tear_iteration_limit(count)` method to change the limit on the number of tear loop iterations that can be taken before declaring failure. The default limit is 200.

Note

The `set_relaxation_factor(factor)` method can be employed to make solution updating at the end of each iteration to be more aggressive (factor > 1.0) or more conservative (factor < 1.0). By default, the relaxation factor is set to 1.

```
# set the start wall time
start_time = time.time()
# solve the reactor network iteratively
status = PSRnetwork.run()
if status != 0:
    print(Color.RED + "Failed to solve the reactor network." + Color.END)
    exit()

# compute the total runtime
runtime = time.time() - start_time
print()
print(f"Total simulation duration: {runtime} [sec].")
print()
```

Postprocess the reactor network solutions

There are two ways to process the results from a reactor network. You can extract the solution of an individual reactor member as a stream by using the `get_reactor_stream()` method with the reactor name. Or, you can get the stream properties of a specific network outlet by using the `get_external_stream()` method with the outlet index. Once you get the solution as a stream, you can use any stream or mixture method to further manipulate the solutions.

Note

Use the `number_external_outlets()` method to find out the number of external exists of the reactor network.

```
# get the outlet stream from the reactor network solutions
# find the number of external outlet streams from the reactor network
print(f"number of outlet streams = {PSRnetwork.number_external_outlets}.")

# display the external outlet stream properties
for m in range(PSRnetwork.number_external_outlets):
    n = m + 1
    network_outflow = PSRnetwork.get_external_stream(n)
    # set the stream label
```

(continues on next page)

(continued from previous page)

```
    network_outflow.label = "outflow"

    # print the desired outlet stream properties
    print("=" * 10)
    print(f"outflow # {n}")
    print("=" * 10)
    print(f"Temperature = {network_outflow.temperature} [K].")
    print(f"Mass flow rate = {network_outflow.mass_flowrate} [g/sec].")
    print(f"CH4 = {network_outflow.X[CH4_index]}.")
    print(f"O2 = {network_outflow.X[O2_index]}.")
    print(f"CO = {network_outflow.X[CO_index]}.")
    print(f"NO = {network_outflow.X[NO_index]}.")
    print("-" * 10)

# display the reactor solutions
print()
print("=" * 10)
print("reactor/zone")
print("=" * 10)
for index, stream in PSRnetwork.reactor_solutions.items():
    name = PSRnetwork.get_reactor_label(index)
    print(f"Reactor: {name}.")
    print(f"Temperature = {stream.temperature} [K].")
    print(f"Mass flow rate = {stream.mass_flowrate} [g/sec].")
    print(f"CH4 = {stream.X[CH4_index]}.")
    print(f"O2 = {stream.X[O2_index]}.")
    print(f"CO = {stream.X[CO_index]}.")
    print(f"NO = {stream.X[NO_index]}.")
    print("-" * 10)
```

12.8 Premixed flame models

The *Premixed Flame* model is a 1-D steady-state application, and it contains two sub-models: *"flame speed calculator"* (or the freely propagating flame model) and *"flat flame"* model. The *flame speed calculator* is commonly used to calculate the laminar flame speed of a premixed fuel-oxidizer mixture. The predicted/computed flame speeds can be used to validate a combustion mechanism or can serve as a prre-processor to construct a "flame speed table" or a "combustion progress table" for other applications. The *flat flame* burner-stablized model is primarily used to study the flame chemistry in conjunction with the flat flame experiments.

The examples show different ways to perform the "premixed flame" calculations.

12.8.1 Compute the laminar flame speed of diluted hydrogen at low pressure

The *freely propagating premixed flame* model can be utilized to obtain the laminar *flame speed* of a gas mixture at given initial temperature and pressure. The premixed flame model will calculate the temperature and species composition profiles across the flame, and the laminar flame speed will be derived from the mass flow rate of the gas mixture, a constant across the entire calculation domain because of the mass conservation.

This tutorial demonstrates the application of the "freely propagating" premixed flame model to calculate the laminar flame speed of an N_2 diluted hydrogen-air mixture at low pressure. Since the transport processes are critical for flame calculations, the *transport data* must be included in the mechanism data and pre-processed.

Import PyChemkin packages and start the logger

```
import os
import time

import ansys.chemkin as ck  # Chemkin
from ansys.chemkin import Color
from ansys.chemkin.inlet import Stream  # external gaseous inlet
from ansys.chemkin.logger import logger

# Chemkin 1-D premixed freely propagating flame model (steady-state)
from ansys.chemkin.premixedflames.premixedflame import FreelyPropagating as FlameSpeed
import matplotlib.pyplot as plt  # plotting
import numpy as np  # number crunching

# check working directory
current_dir = os.getcwd()
logger.debug("working directory: " + current_dir)
# set verbose mode
ck.set_verbose(True)
# set interactive mode for plotting the results
# interactive = True: display plot
# interactive = False: save plot as a PNG file
global interactive
interactive = True
```

Create a chemistry set

The 'C2 NOx' mechanism is from the default *"/reaction/data"* directory. This mechanism also includes information about the *Soave* cubic Equation of State (EOS) for the real-gas applications. PyChemkin preprocessor will indicate the availability of the real-gas model in the `Chemistry Set` processed.

Note

The transport data *must* be included and pre-processed because the transport processes, *convection and diffusion*, are important to sustain the flame structure.

Note

Use the `preprocess_transportdata` method if the transport data is embedded in the gas-phase mechanism file.

```
# set mechanism directory (the default Chemkin mechanism data directory)
data_dir = os.path.join(ck.ansys_dir, "reaction", "data")
mechanism_dir = data_dir
# create a chemistry set based on the C2 NOx mechanism
MyGasMech = ck.Chemistry(label="C2 NOx")
# set mechanism input files
```

(continues on next page)

(continued from previous page)

```
# including the full file path is recommended
MyGasMech.chemfile = os.path.join(mechanism_dir, "C2_NOx_SRK.inp")

# direct the preprocessor to include the transport properties
# only when the mechanism file contains all the transport data
MyGasMech.preprocess_transportdata()
```

Pre-process the `Chemistry Set`

```
# preprocess the mechanism files
iError = MyGasMech.preprocess()
if iError != 0:
    print("Error: Failed to preprocess the mechanism!")
    print(f"       Error code = {iError}")
    exit()
```

Set up the (H_2 + N_2)-air mixture for the flame speed calculation

Instantiate an `Stream` object `premixed` for the inlet gas mixture. The `Stream` object is a `Mixture` object with the addition of the *inlet flow rate*. You can specify the inlet gas properties the same way you set up a `Mixture`. Here the `X_by_Equivalence_Ratio` method is used. You create the `fuel` and the `air` mixtures first. Then define the *complete combustion product species* and provide the *additives* composition if applicable. And finally you can simply set `equivalenceratio=1` to create the stoichiometric hydrogen-air mixture. The estimated inlet mass flow rate can be assigned by the `mass_flowrate()` method.

```
# create the fuel mixture
fuel = ck.Mixture(MyGasMech)
# set fuel composition: hydrogen diluted by nitrogen
fuel.X = [("H2", 0.7), ("N2", 0.3)]
# setting pressure and temperature is not required in this case
fuel.pressure = 0.0125 * ck.Patm
fuel.temperature = 300.0  # inlet temperature

# create the oxidizer mixture: air
air = ck.Mixture(MyGasMech)
air.X = ck.Air.X()
# setting pressure and temperature is not required in this case
air.pressure = fuel.pressure
air.temperature = fuel.temperature

# create the fuel-air Stream for the premixed flame speed calculation
premixed = Stream(MyGasMech, label="premixed")
# products from the complete combustion of the fuel mixture and air
products = ["H2O", "N2"]
# species mole fractions of added/inert mixture. can also create an additives mixture␣
↪here
add_frac = np.zeros(MyGasMech.KK, dtype=np.double)  # no additives: all zeros
# mean equivalence ratio
equiv = 1.0
iError = premixed.X_by_Equivalence_Ratio(
    MyGasMech, fuel.X, air.X, add_frac, products, equivalenceratio=equiv
)
```

(continues on next page)

(continued from previous page)

```
# check fuel-oxidizer mixture creation status
if iError != 0:
    print("Error: Failed to create the hydrogen-air mixture!")
    exit()

# setting inlet pressure [dynes/cm2]
premixed.pressure = fuel.pressure
# set inlet/unburnt gas temperature [K]
premixed.temperature = fuel.temperature
# set estimated value of the flame speed [cm/sec]
premixed.velocity = 65.0
```

Instantiate the laminar speed calculator

Set up the *freely propagating premixed flame* model by using the stream representing the premixed fuel-oxidizer mixture (with the estimated mass flow rate value). When the flame speed is expected to be very small (< 10 [cm/sec]) or very large (> 300 [cm/sec]), it might be beneficial to set the `mass_flowrate` of this inlet to the mass flux [g/cm^2-sec] based on the estimated flame speed value. There are many options and parameters related to the treatment of the species boundary condition, the transport properties. All the available options and parameters are described in the *Chemkin Input* manual.

Note

The stream parameter used to instantiate a `FlameSpeed` object is the properties of the unburned fuel-oxidizer mixture of which the *laminar flame speed* will be determined.

```
flamespeedcalculator = FlameSpeed(premixed, label="premixed_hydrogen")
```

Set up initial mesh and grid adaption options

The premixed flame models provides several methods to set up the initial mesh. Here a uniform mesh of 35 grid points is used at the start of the simulation. The flame models would add more grid points to where they are needed as determined by the solution quality parameters specified by the `set_solution_quality` method.

The `end_poistion` is a required input as it defines the length of the calculation domain. Typically, the length of the calculation domain is between 1 to 10 [cm]. For low pressure conditions, the flame thickness becomes wider and a larger calculation domain is required.

Note

There are three methods to set up the initial mesh for the premixed flame calculations:

1. `use_TPRO_grids` method (default) to use the grid points in the estimate temperature profile.
2. `set_numb_grid_points` method to create a uniform mesh of the given number of grid points.
3. `set_grid_profile` method to specify the initial grid point profile.

```
# set the initial mesh to 35 uniformly distributed grid points
flamespeedcalculator.set_numb_grid_points(35)
# set the maximum total number of grid points allowed in the calculation (optional)
```

(continues on next page)

(continued from previous page)

```
flamespeedcalculator.set_max_grid_points(150)
# define the calculation domain [cm]
flamespeedcalculator.end_position = 40.0
# maximum number of grid points can be added during each grid adaption event (optional)
flamespeedcalculator.set_max_adaptive_points(20)
# set the maximum values of the grdient and the curvature of the solution profiles␣
↪(optional)
flamespeedcalculator.set_solution_quality(gradient=0.1, curvature=0.2)
```

Set transport property options

Ansys Chemkin offers three methods for computing mixture properties:

- **Mixture averaged**
- **Multi-component**
- **Constant Lewis number

When the system pressure is not too low, the mixture averaged method should be adequate. The multi-component method, although it is slightly more accurate, makes the simulation time longer and is harder to converge. Using the constant Lewis number method implies that all the species would have the same transport properties. Include the thermal diffusion effect, when there are large amount of light species (molecular weight < 5.0).

```
# use the mixture averaged formulism to evaluate the mixture transport properties
flamespeedcalculator.use_mixture_averaged_transport()
# include the thermal diffusion effect (because the unburned mixture has hydrogen␣
↪(molecular weight < 5.0))
flamespeedcalculator.use_thermal_diffusion(mode=True)
```

Set species composition boundary option

There two types of boundary condition treatments for the species composition available from the premixed flame models: `comp` and `flux`. You can find the descriptions of these two treatments in the *Chemkin Input* manual.

```
# specific the species composition boundary treatment ('comp' or 'flux')
# use 'comp' to keep the inlet species mass fraction values the same as the "given inlet␣
↪stream".
flamespeedcalculator.set_species_boundary_types(mode="comp")
```

Set solver parameters

The steady state solver parameters for the premixed flame model are optional because all the solver parameters have their own default values. Change the solver parameters when the premixed flame simulation does not converge with the default settings.

Note

The `FreelyPropagating` flame speed calculator has an option to automatically generate an estimate temperature profile that might improve the convergence performance. Use the `automatic_temperature_profile_estimate` method to turn this option on.

```
# reset the tolerances in the steady-state solver (optional)
flamespeedcalculator.steady_state_tolerances = (1.0e-9, 1.0e-6)
flamespeedcalculator.time_stepping_tolerances = (1.0e-6, 1.0e-4)
# reset the gas species floor value in the steady-state solver (optional)
flamespeedcalculator.set_species_floor(-1.0e-4)
# skip the fixed-temperature step (optional)
flamespeedcalculator.skip_fix_T_solution(mode=True)
# reduce the Jacobian age during the pseudo time stepping phase
flamespeedcalculator.set_pseudo_Jacobian_age(10)
```

Run the premixed flame calculation

Use the `run()` method to run the freely propagating premixed flame (flame speed) model. will solve the reactors one by one in the order they are added to the network. After the premixed flame calculation concludes successfully, use the `process_solution()` method to postprocess the solutions. The predicted laminar flame speed can be obtained by using the `get_flame_speed()` method. You can create other property profiles by looping through the solution streams with proper `Mixture` methods.

Note

When the inlet stream condition is close to the flammability limit, the flame speed calculation might fail. Remember that the reaction mechanism (reaction rates, thermodynamic properties, and transport properties) and the reactor model are *models* that contain assumptions and uncertainties.

```
# set the start wall time
start_time = time.time()

status = flamespeedcalculator.run()
if status != 0:
    print(Color.RED + "Failed to calculate the laminar flame speed!" + Color.END)
    exit()

# compute the total runtime
runtime = time.time() - start_time
print()
print(f"Total simulation duration: {runtime} [sec].")
print()
```

Postprocess the premixed flame results

The postprocessing step will parse the solution and package the solution values at each time point into a stream. There are two ways to access the solution profiles:

1. the raw solution profiles (value as a function of time) are available for “distance”, “temperature”, and species “mass fractions”;

2. the streams that permit the use of all property and rate utilities to extract information such as viscosity, density, and mole fractions.

You can use the `get_solution_variable_profile()` method to get the raw solution profiles. You solution streams are accessed using either the `get_solution_stream_at_grid()` method for the solution stream at the given grid point or the `get_solution_stream()` method for the solution stream at the given location. (In this case, the stream is constructed by interpolation.)

Note

- Use the `get_solution_size()` to get the number of grid pints in the solution profiles before creating the arrays.
- The `mass_flowrate` from the solution streams is actually the *mass flux* [g/cm^2-sec]. It can used to derive the velocity at the corresponding location by dividing it by the local gas mixture density [g/cm^3]. The `get_flame_speed()` can also be used to retrieve the predict laminar flame speed value.

```
# postprocess the solutions
flamespeedcalculator.process_solution()

# print the predicted laminar flame speed
print(
    f"The predicted laminar flame speed = {flamespeedcalculator.get_flame_speed()} [cm/
↪sec]."
)
# get the number of solution grid points
solutionpoints = flamespeedcalculator.get_solution_size()
print(f"Number of solution points = {solutionpoints}.")
# get the grid profile
mesh = flamespeedcalculator.get_solution_variable_profile("distance")
# get the temperature profile
tempprofile = flamespeedcalculator.get_solution_variable_profile("temperature")
# get HO2 mass fraction profile
HO2profile = flamespeedcalculator.get_solution_variable_profile("HO2")

# create arrays for mixture density, mixture viscosity, and mixture specific heat␣
↪capacity
denprofile = np.zeros_like(mesh, dtype=np.double)
viscprofile = np.zeros_like(mesh, dtype=np.double)
# loop over all solution grid points
for i in range(solutionpoints):
    # get the stream at the grid point
    solutionstream = flamespeedcalculator.get_solution_stream_at_grid(grid_index=i)
    # get gas density [g/cm3]
    denprofile[i] = solutionstream.RHO
    # get mixture viscosity profile [g/cm-sec] or [Poise]
    viscprofile[i] = solutionstream.mixture_viscosity() * 1.0e2
```

Plot the premixed flame solution profiles

Plot the solution profiles of the premixed flame.

Note

You can get profiles of the thermodynamic and the transport properties by applying `Mixture` utility methods to the solution `Stream`.

```
plt.subplots(2, 2, sharex="col", figsize=(12, 6))
plt.subplot(221)
```

(continues on next page)

(continued from previous page)

```
plt.plot(mesh, tempprofile, "r-")
plt.ylabel("Temperature [K]")
plt.subplot(222)
plt.plot(mesh, denprofile, "b-")
plt.ylabel("Mixture Density [g/cm3]")
plt.subplot(223)
plt.plot(mesh, HO2profile, "g-")
plt.xlabel("Distance [cm]")
plt.ylabel("HO2 Mass Fraction")
plt.subplot(224)
plt.plot(mesh, viscprofile, "m-")
plt.xlabel("Distance [cm]")
plt.ylabel("Mixture Viscosity [cP]")
# plot results
if interactive:
    plt.show()
else:
    plt.savefig("plot_hydrogen_premixed_flame.png", bbox_inches="tight")
```

12.8.2 Construct atmospheric methane-air flame speed versus equivalence ratio table

One of the prevailing use case of the *freely propagating premixed flame* model is to build a *flame speed* table to be imported by another combustion simulation tools. PyChemkin provides the flexibility to customize the data structure of the flame speed table depending on the simulation goals and the tool. Furthermore, over the years, the chemkin flame speed calculator has derived a set of default solver settings that would greatly improve the convergence performance, especially for those widely adopted hydrocarbon fuel combustion mechanisms. The required input parameters the flame speed calculator are reduced to the composition of the fuel-oxidizer mixture, the initial/inlet pressure and temperature, and the calculation domain.

This tutorial shows the "minimal" effort to create a flame speed table of CH_4-air mixtures at the atmospheric pressure. The predicted flame speed values are compared against the experimental data as a function of the mixture equivalence ratio. Since the transport processes are critical for flame calculations, the transport data must be included in the mechanism data and preprocessed.

```

```

Import PyChemkin packages and start the logger

```
import os
import time

import ansys.chemkin as ck  # Chemkin
from ansys.chemkin import Color
from ansys.chemkin.inlet import Stream  # external gaseous inlet
from ansys.chemkin.logger import logger

# Chemkin 1-D premixed freely propagating flame model (steady-state)
from ansys.chemkin.premixedflames.premixedflame import FreelyPropagating as FlameSpeed
import matplotlib.pyplot as plt  # plotting
import numpy as np  # number crunching

```

(continues on next page)

(continued from previous page)

```
# check working directory
current_dir = os.getcwd()
logger.debug("working directory: " + current_dir)
# set verbose mode
ck.set_verbose(True)
# set interactive mode for plotting the results
# interactive = True: display plot
# interactive = False: save plot as a PNG file
global interactive
interactive = True
```

Create an instance of the Chemistry Set

The mechanism loaded is the GRI 3.0 mechanism for methane combustion. The mechanism and its associated data files come with the standard Ansys Chemkin installation under the subdirectory *"/reaction/data"*.

Note

The transport data *must* be included and preprocessed because the transport processes, *convection and diffusion*, are important to sustain the flame structure.

```
# set mechanism directory (the default Chemkin mechanism data directory)
data_dir = os.path.join(ck.ansys_dir, "reaction", "data")
mechanism_dir = data_dir
# including the full file path is recommended
chemfile = os.path.join(mechanism_dir, "grimech30_chem.inp")
thermfile = os.path.join(mechanism_dir, "grimech30_thermo.dat")
tranfile = os.path.join(mechanism_dir, "grimech30_transport.dat")
# create a chemistry set based on GRI 3.0
MyGasMech = ck.Chemistry(chem=chemfile, therm=thermfile, tran=tranfile, label="GRI 3.0")
```

Preprocess the Chemistry Set

```
# preprocess the mechanism files
iError = MyGasMech.preprocess()
if iError != 0:
    print("Error: failed to preprocess the mechanism!")
    print(f"       error code = {iError}")
    exit()
```

Set up the CH_4-air mixture for the flame speed calculation

Instantiate a stream named `premixed` for the inlet gas mixture. This stream is a mixture with the addition of the inlet flow rate. You can specify the inlet gas properties the same way you set up a `Mixture`. Here the `X_by_Equivalence_Ratio` method is used. You create the `fuel` and the `air` mixtures first. Then define the *complete combustion product species* and provide the *additives* composition if applicable. And finally, during the parameter iteration runs, you can simply set different values to `equivalenceratio` to create different methane-air mixtures.

```
# create the fuel mixture
fuel = ck.Mixture(MyGasMech)
```

(continues on next page)

(continued from previous page)

```
# set fuel composition: methane
fuel.X = [("CH4", 1.0)]
# setting pressure and temperature condition for the flame speed calculations
fuel.pressure = 1.0 * ck.Patm
fuel.temperature = 300.0  # inlet temperature

# create the oxidizer mixture: air
air = ck.Mixture(MyGasMech)
air.X = ck.Air.X()
# setting pressure and temperature is not required in this case
air.pressure = fuel.pressure
air.temperature = fuel.temperature

# create the fuel-air Stream for the premixed flame speed calculation
premixed = Stream(MyGasMech, label="premixed")
# products from the complete combustion of the fuel mixture and air
products = ["CO2", "H2O", "N2"]
# species mole fractions of added/inert mixture. can also create an additives mixture␣
↪here
add_frac = np.zeros(MyGasMech.KK, dtype=np.double)  # no additives: all zeros

# setting pressure and temperature is not required in this case
premixed.pressure = fuel.pressure
premixed.temperature = fuel.temperature

# set estimated value of the flame mass flux [g/cm2-sec]
premixed.mass_flowrate = 0.4

# equivalence ratio for the first case
phi = 0.6
# create mixture by using the equivalence ratio
iError = premixed.X_by_Equivalence_Ratio(
    MyGasMech, fuel.X, air.X, add_frac, products, equivalenceratio=phi
)
# check fuel-oxidizer mixture creation status
if iError != 0:
    print(
        "Error: failed to create the methane-air mixture "
        + "for equivalence ratio = "
        + str(phi)
    )
    exit()
```

Instantiate the laminar speed calculator

Set up the *freely propagating premixed flame* model by using the stream representing the premixed fuel-oxidizer mixture (with the estimated mass flow rate value). When the flame speed is expected to be very small (< 10 [cm/sec]) or very large (> 300 [cm/sec]), it might be beneficial to set the `mass_flowrate` of this inlet to the mass flux [g/cm^2-sec] based on the estimated flame speed value. There are many options and parameters related to the treatment of the species boundary condition, the transport properties. All the available options and parameters are described in the *Chemkin Input* manual.

Note

The stream parameter used to instantiate a `FlameSpeed` object is the properties of the unburned fuel-oxidizer mixture of which the *laminar flame speed* will be determined.

```
flamespeedcalculator = FlameSpeed(premixed, label="premixed_methane")
```

Set up initial mesh and grid adaption options

The `end_poistion` is a required input as it defines the length of the calculation domain. Typically, the length of the calculation domain is between 1 to 10 [cm]. For low pressure conditions, the flame thickness becomes wider and a larger calculation domain is required.

```
# set the maximum total number of grid points allowed in the calculation (optional)
# flamespeedcalculator.set_max_grid_points(150)
# define the calculation domain [cm]
flamespeedcalculator.end_position = 1.0
```

Run the flame speed parameter study

Use the `run()` method to run the freely propagating premixed flame (flame speed) model. After the premixed flame calculation concludes successfully, use the `process_solution()` method to postprocess the solutions. The predicted laminar flame speed can be obtained by using the `get_flame_speed()` method. You can create other property profiles by looping through the solution streams with proper `Mixture` methods. The parameter in this project is the equivalence ratio of the methane-air mixture. You can start the parameter run from the most fuel-lean or from the most fuel-rich case. Normally, the most "extreme" cases are difficult to converge. When running into these situations, start the parameter runs from the stoichiometric condition and go down the lean and/or the rich branch. Here the runs start from the most fuel-lean case ($\phi = 0.6$) and progress all the way to the most fuel-rich case ($\phi = 1.6$) in steps of 0.05.

Note

- When the inlet stream condition is close to the flammability limit, the flame speed calculation might fail. Remember that the reaction mechanism (reaction rates, thermodynamic properties, and transport properties) and the reactor model are *models* that contain assumptions and uncertainties.
- After complete the first run, you can use the `continuation()` method to start the new runs from the solution of the previous run. However, by doing this, the later runs will contain a lot of grid points accumulated from all previous runs.
- Use the `set_molefractions` method to update the inlet gas composition before each run. Similarly, use the `pressure` and the `temperature` methods to change the inlet condition.

```
# total number of parameter cases
points = 21
# equivalence ratio increment
delta_phi = 0.05
# create solution arrays
equival = np.zeros(points, dtype=np.double)
flamespeed = np.zeros_like(equival, dtype=np.double)
# set the start wall time
start_time = time.time()
```

(continues on next page)

(continued from previous page)

```
# start the parameter study runs
for i in range(points):
    # run the flame speed calculation for this equivalence ratio
    status = flamespeedcalculator.run()
    if status != 0:
        print(
            Color.RED
            + "failed to calculate the laminar flame speed"
            + "for equivalence ratio = "
            + str(phi)
            + Color.END
        )
        exit()
    # get flame speed
    # postprocess the solutions
    flamespeedcalculator.process_solution()
    # save data
    equival[i] = phi
    # get flame speed
    flamespeed[i] = flamespeedcalculator.get_flame_speed()
    # print the predicted laminar flame speed
    print(
        f"methane-air equivalence ratio = {phi} :\n"
        + f"the predicted laminar flame speed = {flamespeed[i]} [cm/sec]"
    )
    #
    # update parameter
    phi += delta_phi
    # create mixture by using the equivalence ratio
    iError = premixed.X_by_Equivalence_Ratio(
        MyGasMech, fuel.X, air.X, add_frac, products, equivalenceratio=phi
    )
    # check fuel-oxidizer mixture creation status
    if iError != 0:
        print(
            "Error: failed to create the methane-air mixture ",
            "for equivalence ratio = ",
            str(phi),
        )
        exit()
    # update initial gas composition
    flamespeedcalculator.set_molefractions(premixed.X)

# compute the total runtime
runtime = time.time() - start_time
print()
print(f"total simulation duration: {runtime} [sec]")
print()

# experimental data by Vagelopoulos
# equivalence ratios
data_equiv = [
```

(continues on next page)

(continued from previous page)

```
    0.6126,
    0.6619,
    0.7109,
    0.7533,
    0.8268,
    0.9109,
    0.9826,
    1.0387,
    1.0901,
    1.1321,
    1.1858,
    1.2347,
    1.2695,
    1.325,
    1.3563,
    1.4279,
    1.4977,
]
# methane flame speeds at 1 atm
data_speed = [
    9.4434,
    12.7281,
    17.4088,
    21.0219,
    26.5237,
    33.2573,
    37.0347,
    38.677,
    38.8412,
    37.8558,
    34.9818,
    31.1223,
    25.9489,
    21.3504,
    17.2445,
    12.7281,
    9.7719,
]
```

Plot the premixed flame solution profiles

Plot the predicted flame speeds against the experimental data

```
plt.plot(data_equiv, data_speed, label="data", linestyle="", marker="^", color="blue")
plt.plot(equival, flamespeed, label=MyGasMech.label, linestyle="-", color="blue")
plt.legend()
plt.ylabel("Flame Speed [cm/sec]")
plt.xlabel("Equivalence Ratio")
# plot results
if interactive:
    plt.show()
else:
```

(continues on next page)

(continued from previous page)

```
    plt.savefig("plot_flame_speed_table.png", bbox_inches="tight")
```

12.8.3 Simulate a burner stabilized premixed flame

The *burner stabilized premixed flame* model is frequently used as a numerical tool to develop and to validate combustion mechanisms in the context of a burner stabilized flat flame. The premixed burner stabilized flat flame experiments are important because, in addition to the reaction pathways and rate parameters, they provide opportunities to learn about the impact of the transport processes on the combustion chemistry and flame structure. The premixed burner stabilized flame model calculates the temperature and the species concentrations along the burner centerline, and the results will be compared against the measurements. The burner stabilized flame model assumes that the flow is laminar and the temperature at the burner rim is constant (in order to anchor the flame).

This example shows how to use the burner stabilized premixed flame model to calculate the species profiles of a C_2H_4-O_2-AR mixture along the burner centerline with the measured temperature profile data. Since the transport processes are critical for flame calculations, transport data must be included in the mechanism data and preprocessed.

```

```

Import PyChemkin packages and start the logger

```
import os
import time

import ansys.chemkin as ck  # Chemkin
from ansys.chemkin import Color
from ansys.chemkin.inlet import Stream  # external gaseous inlet
from ansys.chemkin.logger import logger

# Chemkin 1-D premixed burner-stabilized flame model (steady-state)
from ansys.chemkin.premixedflames.premixedflame import (
    BurnedStabilized_GivenTemperature as Burner,
)
import matplotlib.pyplot as plt  # plotting
import numpy as np  # number crunching

# check working directory
current_dir = os.getcwd()
logger.debug("working directory: " + current_dir)
# set verbose mode
ck.set_verbose(True)
# set interactive mode for plotting the results
# interactive = True: display plot
# interactive = False: save plot as a PNG file
global interactive
interactive = True
```

Create the first chemistry set

The mechanism to load is the GRI 3.0 mechanism for methane combustion. This mechanism and its associated data files come with the standard Ansys Chemkin installation in the `/reaction/data` directory.

Note

The transport data *must* be included and preprocessed because the transport processes, *convection and diffusion*, are important to sustain the flame structure.

```
# set mechanism directory (the default Chemkin mechanism data directory)
data_dir = os.path.join(ck.ansys_dir, "reaction", "data")
mechanism_dir = data_dir
# including the full file path is recommended
chemfile = os.path.join(mechanism_dir, "grimech30_chem.inp")
thermfile = os.path.join(mechanism_dir, "grimech30_thermo.dat")
tranfile = os.path.join(mechanism_dir, "grimech30_transport.dat")
# create a chemistry set based on GRI 3.0
MyGasMech = ck.Chemistry(chem=chemfile, therm=thermfile, tran=tranfile, label="GRI 3.0")
```

Preprocess the chemistry set

```
# preprocess the mechanism files
iError = MyGasMech.preprocess()
if iError != 0:
    print("Error: Failed to preprocess the mechanism.")
    print(f"       Error code = {iError}.")
    exit()
```

Set up the (C_2H_4 + O_2+ AR) mixture to the flat burner

Instantiate a stream named `premixed` for the inlet gas mixture. This stream is a mixture with the addition of the inlet flow rate. You specify the inlet gas properties in the same way you set up a mixture. You can simply use a composition *"recipe"* to create the C_2H_4 + O_2+ AR mixture. You use the `mass_flowrate()` method to assign the burner inlet mass flow rate.

Note

By default, the burner flow area is set to unity (1.0 [cm^2]) because the flame models use the mass flux [g/cm^2-sec] in the calculations instead of the flow rate [g/sec]. If the mass flow rate is used, the actual burner cross-sectional flow area must be provided by using the `flowarea()` stream method.

```
# create the fuel-air stream for the premixed flame speed calculation
premixed = Stream(MyGasMech, label="premixed")
# set inlet premixed stream molar composition
premixed.X = [("C2H4", 0.163), ("O2", 0.237), ("AR", 0.6)]
# setting inlet pressure [dynes/cm2]
premixed.pressure = 1.0 * ck.Patm
# set inlet/unburnt gas temperature [K]
premixed.temperature = 300.0  # inlet temperature

# set the burner outlet cross sectional flow area to unity (1 [cm2])
# because the flame models use the mass flux in the calculation instead of the mass flow␣
↪rate
premixed.flowarea = 1.0
```

(continues on next page)

(continued from previous page)

```
# set value for the inlet mass flow rate [g/sec]
premixed.velocity = 0.0147
```

Instantiate the laminar flat flame burner

Set up the premixed burner stabilized flame model by using the stream representing the premixed fuel-oxidizer mixture.There are many options and parameters related to the treatment of the species boundary condition and the transport properties. All the available options and parameters are described in the *Chemkin Input* manual.

Note

The stream parameter used to instantiate a `Burner` object consists of the properties of the unburned fuel-oxidizer mixture leaving the burner outlet.

```
flatflame = Burner(premixed, label="ethylene flame")
```

Set up the temperature profile along the burner centerline

Use the `set_temperature_profile()` method to set up the temperature profile data. The temperature profile along the burner centerline is typically obtained from the corresponding experiments.

```
TPRO_data_points = 21
gridpoints = np.zeros(TPRO_data_points, dtype=np.double)
temp_data = np.zeros_like(gridpoints, dtype=np.double)
gridpoints = [
    0.0,
    0.01,
    0.02,
    0.0345643,
    0.0602659,
    0.100148,
    0.118759,
    0.135598,
    0.15421,
    0.179911,
    0.211817,
    0.233087,
    0.260561,
    0.293353,
    0.348301,
    0.436928,
    0.517578,
    0.7161,
    0.935894,
    1.16632,
    1.2,
]
temp_data = [
    300.0,
    450.0,
    600.0,
```

(continues on next page)

(continued from previous page)

```
    828.498,
    1104.4,
    1496.7,
    1634.65,
    1714.85,
    1768.33,
    1801.12,
    1807.2,
    1803.78,
    1799.51,
    1788.35,
    1778.09,
    1767.01,
    1760.23,
    1744.13,
    1737.55,
    1730.99,
    1729.31,
]
flatflame.set_temperature_profile(x=gridpoints, temp=temp_data)
```

Set up initial mesh and grid adaption options

The premixed flame models provides several methods to set up the initial mesh. In this case, the temperature profile is given by the experimental data. The `use_TPRO_grid()` method lets the flame model recycle the grid points of the temperature profile as the initial mesh at the start of the calculation. The flame models would add more grid points to where they are needed as determined by the solution quality parameters specified by the `set_solution_quality()` method.

The `end_position` argument is a required input as it defines the length of the calculation domain. Typically, the length of the calculation domain is between 1 to 10 [cm].

```
# set the maximum total number of grid points allowed in the calculation (optional)
flatflame.set_max_grid_points(250)
# define the calculation domain [cm]
flatflame.end_position = 1.2
# maximum number of grid points can be added during each grid adaption event (optional)
flatflame.set_max_adaptive_points(10)
# set the maximum values of the gradient and the curvature of the solution profiles␣
↪(optional)
flatflame.set_solution_quality(gradient=0.1, curvature=0.1)
```

Set transport property options

Ansys Chemkin offers three methods for computing mixture properties:

- **Mixture averaged**
- **Multi-component**
- **Constant Lewis number

When the system pressure is not too low, the mixture averaged method should be adequate. The multi-component method, although it is slightly more accurate, makes the simulation time longer and is harder to converge. Using the constant Lewis number method implies that all the species would have the same transport properties.

```
# use the mixture averaged method to evaluate the mixture transport properties
flatflame.use_mixture_averaged_transport()
```

Set species composition boundary option

There two types of boundary condition treatments for the species composition available from the premixed flame models: `comp` and `flux`. You can find the descriptions of these two treatments in the *Chemkin Input* manual.

```
# specify the species composition boundary treatment ('comp' or 'flux')
# use 'flux' to ensure that the "net" species mass fluxes are zero at the burner outlet.
flatflame.set_species_boundary_types(mode="flux")
```

Set solver parameters

The steady-state solver parameters for the premixed flame model are optional because all the solver parameters have their own default values. Change the solver parameters when the premixed flame simulation does not converge with the default settings.

```
# reset the tolerances in the steady-state solver (optional)
flatflame.steady_state_tolerances = (1.0e-9, 1.0e-4)
# reset the gas species floor value in the steady-state solver (optional)
flatflame.set_species_floor(-1.0e-3)
```

Run the premixed flame calculation

Use the `run()` method to run the freely propagating premixed flame (flame speed) model. This method solves the reactors one by one in the order that they are added to the network. After the premixed flame calculation concludes successfully, use the `process_solution()` method to postprocess the solutions. You can create other property profiles by looping through the solution streams with proper mixture methods.

```
# set the start wall time
start_time = time.time()

status = flatflame.run()
if status != 0:
    print(Color.RED + "Failed to solve the reactor network." + Color.END)
    exit()

# get the number of solution grid points
solutionpoints = flatflame.get_solution_size()
print(f"Number of solution points = {solutionpoints}.")
```

Refine the solution profiles

When the simulation is hard to converge in one go, you can use a number of continuations to gradually get to the solution of the intended condition. For example, you can get to the solution of a very fuel-lean mixture by starting with a slightly fuel-rich mixture flame and use the `continuation()` method to run the more difficult fuel-lean case from the converged solution.

```
# tightening the convergence criteria
flatflame.set_solution_quality(gradient=0.05, curvature=0.1)

# restart the flame simulation by continuation
```

(continues on next page)

(continued from previous page)

```
flatflame.continuation()

# compute the total runtime
runtime = time.time() - start_time
print()
print(f"Total simulation duration: {runtime} [sec].")
print()
```

Postprocess the premixed flame results

The postprocessing step parses the solution and packages the solution values at each time point into a mixture. There are two ways to access the solution profiles:

- The raw solution profiles (value as a function of distance) are available for distance, temperature, pressure, volume, and species mass fractions.

  -The streams permit the use of all property and rate utilities to extract
  information such as viscosity, density, and mole fractions.

You can use the `get_solution_variable_profile()` method to get the raw solution profiles. You solution streams are accessed using either the `get_solution_stream_at_grid()` method for the solution stream at the given grid point or the `get_solution_stream()` method for the solution stream at the given location. (In this case, the stream is constructed by interpolation.)

Note

- Use the `get_solution_size()` method to get the number of grid points in the solution profiles before creating the arrays.
- The `mass_flowrate` from the solution streams is actually the *mass flux* [g/cm^2-sec]. It can used to derive the velocity at the corresponding location by dividing it by the local gas mixture density [g/cm^3].

```
# postprocess the solutions
flatflame.process_solution()

# get the number of solution grid points
solutionpoints = flatflame.get_solution_size()
print(f"Number of final solution points = {solutionpoints}.")
# get the grid profile
mesh = flatflame.get_solution_variable_profile("distance")
# get the temperature profile
tempprofile = flatflame.get_solution_variable_profile("temperature")
# get OH mass fraction profile
OHprofile = flatflame.get_solution_variable_profile("OH")

# create arrays for mixture conductivity and mixture-pecific heat capacity
Cpprofile = np.zeros_like(mesh, dtype=np.double)
condprofile = np.zeros_like(mesh, dtype=np.double)
# loop over all solution grid points
for i in range(solutionpoints):
    # get the stream at the grid point
    solutionstream = flatflame.get_solution_stream_at_grid(grid_index=i)
```

(continues on next page)

(continued from previous page)

```
    # get mixture-specific heat capacity profile [erg/mole-K]
    Cpprofile[i] = solutionstream.CPBL() / ck.ergs_per_joule * 1.0e-3
    # get thermal conductivity profile [ergs/cm-K-sec]
    condprofile[i] = solutionstream.mixture_conductivity() * 1.0e-5
```

Plot the premixed flame solution profiles

Plot the solution profiles of the premixed flame.

Note

You can get profiles of the thermodynamic and the transport properties by applying `Mixture` utility methods to the solution stream.

```
plt.subplots(2, 2, sharex="col", figsize=(12, 6))
plt.subplot(221)
plt.plot(mesh, tempprofile, "r-")
plt.ylabel("Temperature [K]")
plt.subplot(222)
plt.plot(mesh, Cpprofile, "b-")
plt.ylabel("Mixture Cp [kJ/mole]")
plt.subplot(223)
plt.plot(mesh, OHprofile, "g-")
plt.xlabel("Distance [cm]")
plt.ylabel("OH Mass Fraction")
plt.subplot(224)
plt.plot(mesh, condprofile, "m-")
plt.xlabel("Distance [cm]")
plt.ylabel("Mixture conductivity [W/m-K]")
# plot results
if interactive:
    plt.show()
else:
    plt.savefig("plot_premixed_burner_stabilised_flame.png", bbox_inches="tight")
```

CHAPTER

THIRTEEN

CONTRIBUTE

Thank you for your interest in contributing to PyChemkin. Contributions for making the project better can include fixing bugs, adding new features, and improving the documentation.

Overall guidance on contributing to a PyAnsys library appears in the Contributing topic in the *PyAnsys developer's guide*. Ensure that you are thoroughly familiar with this guide before attempting to contribute to PyChemkin.

The following contribution information is specific to PyChemkin.

- **Clone the repository.**
  To clone and install the latest PyChemkin release in development mode, run these commands:

  ```
  git clone https://github.com/ansys/pychemkin/
  cd pychemkin
  python -m pip install --upgrade pip
  pip install -e .
  ```

- **Follow the code style.**
  PyChemkin follows the PEP 8 standard as described in PEP 8 in the *PyAnsys developer's guide* and uses `pre-commit` for style checking.

  To ensure your code meets minimum code styling standards, run these commands:

  ```
  pip install pre-commit
  pre-commit run --all-files
  ```

  You can also install this as a pre-commit hook by running this command:

  ```
  pre-commit install
  ```

- **Run the tests.**
  Prior to running the tests, you must run this command to install the test dependencies:

  ```
  pip install - .tests
  ```

  To run the tests, navigate to the root directory of the repository and run this command:

  ```
  pytest
  ```

- **Build the documentation.**
  Prior to building the documentation, you must run this command to install the documentation dependencies:

  ```
  pip install -e .doc
  ```

  To build the documentation, run the following commands:

```
cd doc
```

- On Linux:

```
make html
```

- On Windows:

```
./make.bat html
```

The documentation is built in the `docs/_build/html` directory.

CHAPTER

FOURTEEN

API REFERENCE

This section describes PyChemkin endpoints, their capabilities, and how to interact with them programmatically.

14.1 The `ansys.chemkin.grid` library

14.1.1 Summary

Classes

Grid	Grid quality control parameters for Chemkin 1-D steady-state reactor models.

Grid

class ansys.chemkin.grid.**Grid**

Grid quality control parameters for Chemkin 1-D steady-state reactor models.

Overview

Methods

set_numb_grid_points	Set the number of uniform grid points at the start of simulation.
set_max_grid_points	Set the maximum number of grid points allowed during the solution refinement.
set_reaction_zone_center	Set the coordinate value of the reaction/mixing zone center.
set_reaction_zone_width	Set the width of the reaction/mixing.
set_max_adaptive_points	Set the maximum number of adaptive grid points allowed per solution refinement.
set_solution_quality	Set the maximum gradient and curvature ratios in the final solution profile.
set_grid_profile	Specify the grid point coordinates of the initial grid points.

Properties

start_position	Get the coordinate value of the first grid point, that is, the inlet/entrance.
end_position	Get the coordinate value of the last grid point, that is, the outlet/exit/gap.

Attributes

max_numb_grid_points
max_numb_adapt_points
gradient
curvature
numb_grid_points
starting_x
ending_x
reaction_zone_center_x
reaction_zone_width
grid_profile
numb_grid_profile

Import detail

```
from ansys.chemkin.grid import Grid
```

Property detail

property Grid.**start_position: float**

Get the coordinate value of the first grid point, that is, the inlet/entrance.

Returns

position: double
coordinate of the first grid point [cm]

property Grid.**end_position: float**

Get the coordinate value of the last grid point, that is, the outlet/exit/gap.

Returns

position: double
coordinate of the last grid point [cm]

Attribute detail

Grid.**max_numb_grid_points = 250**

Grid.**max_numb_adapt_points = 10**

Grid.**gradient = 0.1**

Grid.**curvature = 0.5**

Grid.**numb_grid_points = 6**

Grid.**starting_x = 0.0**

Grid.**ending_x = 0.0**

Grid.**reaction_zone_center_x = 0.0**

Grid.**reaction_zone_width = 0.0**

Grid.**grid_profile = []**

Grid.**numb_grid_profile = 0**

Method detail

Grid.**set_numb_grid_points**(*numb_points: int*)

Set the number of uniform grid points at the start of simulation.

Parameters

numb_points: integer, default = 6
number of initial grid points

Grid.**set_max_grid_points**(*numb_points: int*)

Set the maximum number of grid points allowed during the solution refinement.

Parameters

numb_points: integer, default = 250
maximum number of grid points

Grid.**set_reaction_zone_center**(*position: float*)

Set the coordinate value of the reaction/mixing zone center.

Parameters

position: double
coordinate of center of the reaction/mixing zone [cm]

Grid.**set_reaction_zone_width**(*size: float*)

Set the width of the reaction/mixing.

Parameters

size: double
width of the reaction/mixing zone [cm]

Grid.**set_max_adaptive_points**(*numb_points: int*)

Set the maximum number of adaptive grid points allowed per solution refinement.

Parameters

numb_points: integer, default = 10
maximum number of adapted grid points can be added

Grid.**set_solution_quality**(*gradient: float = 0.1*, *curvature: float = 0.5*)

Set the maximum gradient and curvature ratios in the final solution profile. The solver will attempt to add more grid points to improve the resolution of the solution profiles till both gradient and curvature ratios are below the specified values or till the number of grid points exceeds the maximum quantity allowed.

Parameters

gradient: double, default = 0.1
the maximum gradient ratio of in the final solution profiles.

curvature: double, default = 0.5
the maximum curvature ratio of in the final solution profiles.

`Grid.set_grid_profile`(*mesh: numpy.typing.NDArray[numpy.double]*) → int

Specify the grid point coordinates of the initial grid points.

Parameters

mesh: 1-D double array
initial grid point positions [cm]

Returns

error code

14.1.2 Description

Chemkin grid quality control parameters.

14.2 The `ansys.chemkin.info` library

14.2.1 Summary

Functions

setup_hints	Set up Chemkin keyword hints.
clear_hints	Clear the Chemkin keyword data.
keyword_hints	Get hints about the Chemkin keyword.
phrase_hints	Get keyword hints by using key phrase in the description.
help	Provide assistance on finding information about Chemkin keywords.
show_realgas_usage	Show Chemkin real-gas model usage and options.
show_equilibrium_options	Show the equilibrium calculation usage and options.
show_ignition_definitions	Show the ignition definitions available in Chemkin.
manuals	Access the Chemkin manuals page on the Ansys Help portal.

Attributes

CKdict

14.2.2 Description

Chemkin help menu for keywords and key phrases.

14.2.3 Module detail

info.**setup_hints**()

Set up Chemkin keyword hints.

info.**clear_hints**()

Clear the Chemkin keyword data.

info.**keyword_hints**(*mykey: str*)

Get hints about the Chemkin keyword.

Parameters

mykey: string
keyword phrase

info.**phrase_hints**(*phrase: str*)

Get keyword hints by using key phrase in the description.

Parameters

phrase: string
search phrase

info.**help**(*topic: str | None = None*)

Provide assistance on finding information about Chemkin keywords.

Parameters

topic: string
the keyword topic of which additional hints are requested

info.**show_realgas_usage**()

Show Chemkin real-gas model usage and options.

info.**show_equilibrium_options**()

Show the equilibrium calculation usage and options.

info.**show_ignition_definitions**()

Show the ignition definitions available in Chemkin.

info.**manuals**()

Access the Chemkin manuals page on the Ansys Help portal.

info.**CKdict**

14.3 The ansys.chemkin.flame library

14.3.1 Summary

Classes

Flame	Generic steady state, one dimensional flame model

Flame

class ansys.chemkin.flame.**Flame**(*fuelstream:* ansys.chemkin.inlet.Stream, *label: str*)

Bases: *ansys.chemkin.reactormodel.ReactorModel*, *ansys.chemkin.steadystatesolver.SteadyStateSolver*, *ansys.chemkin.grid.Grid*

Generic steady state, one dimensional flame model

Overview

Methods

set_temperature_profile	Specify temperature profile
use_temp_profiel_initial_me	Use the grid points in the user defined initial/estimated temperature profile
set_convection_differencing	Set the finite differencing scheme for the convective terms in the transport equations.
set_mesh_keywords	Set mesh related keywords
set_SSsolver_keywords	add steady-state solver parameter keywoprds to the keyword list
use_mixture_averaged_transp	Use the mixture-averaged transport properties.
use_multicomponent_transpor	Use the multi-component transport properties.
use_fixed_Lewis_number_tran	Use a fixed Lewis number to compute the species diffusion coefficient from mixture conductivity.
use_thermal_diffusion	Include the thermal diffusion (Doret) effect.
set_species_boundary_types	Set the species boundary condition type (at inlet and outlet).

Attributes

mass_flow_rate
temp_profile_set
grid_T_profile
EnergyTypes
transport_mode

Import detail

```
from ansys.chemkin.flame import Flame
```

Attribute detail

Flame.**mass_flow_rate**

Flame.**temp_profile_set = False**

Flame.**grid_T_profile = False**

Flame.**EnergyTypes**

Flame.**transport_mode = 0**

Method detail

Flame.**set_temperature_profile**(*x: numpy.typing.NDArray[numpy.double]*, *temp: numpy.typing.NDArray[numpy.double]*) → int

Specify temperature profile

Parameters

x: 1D double array
position value of the profile data [cm]

temp: 1D double array
temperature value of the profile data [K]

Returns

error code: integer

Flame.**use_temp_profiel_initial_mesh**(*on: bool = False*)

Use the grid points in the user defined initial/estimated temperature profile as the initial/starting grid points.

Parameters

on: boolean, default = False
use the grid points of the temperature profile as the initial grid points

Flame.**set_convection_differencing_type**(*mode: str*)

Set the finite differencing scheme for the convective terms in the transport equations.

Parameters

mode: string, {"central", "upwind"}
finite difference discretizing scheme

Flame.**set_mesh_keywords**() → int

Set mesh related keywords

Returns

error code: integer

Flame.**set_SSsolver_keywords**()

add steady-state solver parameter keywoprds to the keyword list

Flame.**use_mixture_averaged_transport**()

Use the mixture-averaged transport properties.

Flame.**use_multicomponent_transport**()

Use the multi-component transport properties. Use of the multi-component transport properties is recommended when the pressure is low.

Flame.**use_fixed_Lewis_number_transport**(*Lewis: float = 1.0*)

Use a fixed Lewis number to compute the species diffusion coefficient from mixture conductivity.

$$Le = Sc/Pr = \frac{(\kappa/\rho C_p)}{D}$$

Parameters

Lewis: double
Lewis number

`Flame.use_thermal_diffusion`(*mode: bool = True*)

Include the thermal diffusion (Doret) effect. The inclusion of thermal diffucivity is recommended when there are significant amount of "light" species in the system. Species with molecular weight less than 5 g/mol is considered a light species, for example, hydrogen.

Parameters

mode: boolean {True, False}
ON/OFF

`Flame.set_species_boundary_types`(*mode: str = 'comp'*)

Set the species boundary condition type (at inlet and outlet). When the species value is fixed at the boundary, the "back" diffusion of the species into the inlet stream is not considered. That is, when the species profile is positive at the inlet, a fixed species value implies the species concentration in the inlet stream is actually lower than its boundary concentration. When the "flux" option is employed, species mass flux will be balanced at the inlet. That is, the species concentration at the inlet will be slightly varied from its specified boundary value so that the "net" species mass flux at the inlet is zero.

Parameters

mode: string, {"flux", "comp"}
keep the species mole/mass fraction at a fixed value or keep the species mass flux conserved

14.3.2 Description

Chemkin utilities for steady state, 1-D flame models.

14.4 The `ansys.chemkin.color` library

14.4.1 Summary

Classes

Color	Define colors used by PyChemkin for printing text messages

Color

class ansys.chemkin.color.**Color**

Define colors used by PyChemkin for printing text messages

Overview

Attributes

RED
MAGENTA
PURPLE
CYAN
GREEN
BLUE
YELLOW
WHITE
BOLD
ULINE
END
SPACE
SPACEx6
msg_modes
log_modes

Static methods

ckprint	Customized text messages

Import detail

```
from ansys.chemkin.color import Color
```

Attribute detail

Color.**RED = '\x1b[91m'**

Color.**MAGENTA = '\x1b[035m'**

Color.**PURPLE = '\x1b[95m'**

Color.**CYAN = '\x1b[96m'**

Color.**GREEN = '\x1b[92m'**

Color.**BLUE = '\x1b[94m'**

Color.**YELLOW = '\x1b[93m'**

Color.**WHITE = '\x1b[037m'**

Color.**BOLD = '\x1b[1m'**

Color.**ULINE = '\x1b[4m'**

Color.**END = Multiline-String**

```
"""
 [0m"""
```

Color.**SPACE = ' '**

Color.**SPACEx6 = ' '**

Color.**msg_modes**

Color.**log_modes**

Method detail

static Color.**ckprint**(*mode: str*, *msg: list = []*)

Customized text messages

Parameters

mode: string, {"normal, "info", "warning, "error", "fatal", "ok"}, default = ""
message mode/type

msg: list of strings
the message to be printed

14.5 The ansys.chemkin.inlet library

14.5.1 Summary

Classes

Stream	Generic inlet stream consists of the gas species defined in the given chemistry set

Functions

clone_stream	Copy the properties of the source Stream to the target Stream.
compare_streams	Compare properties of stream B against those of stream A. The stream properties
adiabatic_mixing_streams	Create a new Stream object by mixing two streams adiabatically. The
create_stream_from_mixture	Create a new Stream object from the given Mixture object.

Stream

class ansys.chemkin.inlet.**Stream**(*chem*, *label: str | None = None*)

Bases: *ansys.chemkin.mixture.Mixture*

Generic inlet stream consists of the gas species defined in the given chemistry set for Chemkin open reactor models

Overview

Methods

convert_to_mass_flowrate	convert different types of flow rate value to mass flow rate
convert_to_vol_flowrate	convert different types of flow rate value to volumetric flow rate
convert_to_SCCM	convert different types of flow rate value to SCCM

Properties

flowarea	Get inlet flow area
mass_flowrate	Get inlet mass flow rate
vol_flowrate	Get inlet volumetric flow rate
sccm	Get inlet SCCM volumetric flow rate
velocity	Get inlet gas velocity
velocity_gradient	Get inlet gas axial velocity gradient (for premixed, oppdif, and spin)
label	Get the label of the Stream.

Import detail

```
from ansys.chemkin.inlet import Stream
```

Property detail

property Stream.**flowarea: float**

Get inlet flow area

Returns

flowarea: double
cross-sectional flow area [cm2]

property Stream.**mass_flowrate: float**

Get inlet mass flow rate

Returns

mflowrate: double
mass flow rate [g/sec]

property Stream.**vol_flowrate: float**

Get inlet volumetric flow rate

Returns

vflowrate: double
volumetric flow rate [cm3/sec]

property Stream.**sccm: float**

Get inlet SCCM volumetric flow rate

Returns

vflowrate: double
SCCM volumetric flow rate [standard cm3/min]

property Stream.**velocity: float**

Get inlet gas velocity

Returns

vel: double
velocity [cm/sec]

property Stream.**velocity_gradient: float**

Get inlet gas axial velocity gradient (for premixed, oppdif, and spin) or radial velocity spreading rate (v_r/r) at the inlet.

Returns

velgrad: double
velocity gradient [1/sec]

property Stream.**label: str**

Get the label of the Stream.

Returns

label: string
label of the Stream

Method detail

Stream.**convert_to_mass_flowrate**() → float

convert different types of flow rate value to mass flow rate

Returns

mrate: double
mass flow rate [g/sec]

Stream.**convert_to_vol_flowrate**() → float

convert different types of flow rate value to volumetric flow rate

Returns

vrate: double
volmetric flow rate [cm3/sec]

Stream.**convert_to_SCCM**() → float

convert different types of flow rate value to SCCM

Returns

sccm: double
volumetric flow rate in SCCM [standard cm3/min]

14.5.2 Description

Chemkin reactor inlet/stream utilities.

14.5.3 Module detail

inlet.**clone_stream**(*source:* Stream, *target:* Stream)

Copy the properties of the source Stream to the target Stream.

Parameters

source: Stream object
the "source" Stream to be cloned

target: Stream object
the "target" Stream to get new properties

inlet.**compare_streams**(*streamA:* Stream, *streamB:* Stream, *atol: float = 1e-10*, *rtol: float = 0.001*, *mode: str = 'mass'*) → tuple[bool, float, float]

Compare properties of stream B against those of stream A. The stream properties include mixture properties such as pressure [atm], temperature [K], and species mass/mole fractions and the stream mass flow rate. When the differences in the property values satisfy both the absolute and the relative tolerances, this method will return "True", that is, stream B is essentially identical to stream A; otherwise, "False" will be returned.

Parameters

streamA: Stream object
mixture A, the target stream

streamB: Stream object
stream B, the sample stream

atol: double, default = 1.0e-10
the absolute tolerance for the max property differences

rtol: double, default = 1.0e-3
the relative tolerance for the max property differences

mode: string {"mass", "mole"}, default = "mass"
compare species "mass" or "mole" fractions

Returns

issame: boolean
the equivalence of the two mixtures

atol_max: double
the max absolute difference value

rtol_max: double
the max relative difference value

inlet.**adiabatic_mixing_streams**(*streamA:* Stream, *streamB:* Stream) → *Stream*

Create a new Stream object by mixing two streams adiabatically. The enthalpy of the final stream is the sum of the enthalpies of the two streams, and so is the final mass flow rate.

Parameters

streamA: Stream object
mixture A, the target stream

streamB: Stream object
stream B, the sample stream

Returns

final_stream: Stream object
the final stream from combining steamA and streamB

inlet.**create_stream_from_mixture**(*chem:* ansys.chemkin.chemistry.Chemistry, *mixture:* ansys.chemkin.mixture.Mixture, *flow_rate: float = 0.0*, *mode: str = 'mass'*) → *Stream*

Create a new Stream object from the given Mixture object.

Parameters

chem: Chemistry object
the Chemistry Set used to instantiate the mixture

mixture: Mixture object
the Mixture whose properties will be used to set up the new Stream

flow_rate: double, >= 0.0, default = 0.0
the flow rate/velocity of the new Stream, [g/sec], [cm3/sec], [cm/sec], [standard cm3/minute]

mode: string, {"mass", "vol", "vel", "sccm"}
the type of flow rate data is given by the flow_rate parameter

Returns

new_stream: Stream object
the new Stream based on the given Mixture

14.6 The ansys.chemkin.logger library

14.6.1 Summary

Classes

SingletonType	**no-index** Provides the singleton helper class for the logger.
ChemkinLogger	Provides the singleton logger for the PyChemkin.

Attributes

logger

SingletonType

class ansys.chemkin.logger.**SingletonType**

Bases: type

No-index
Provides the singleton helper class for the logger.

Overview

Special methods

__call__	Call to redirect new instances to the singleton instance.

Import detail

```
from ansys.chemkin.logger import SingletonType
```

Method detail

SingletonType.**__call__**(**args*, ***kwargs*)

Call to redirect new instances to the singleton instance.

ChemkinLogger

class ansys.chemkin.logger.**ChemkinLogger**(*level: int = logging.ERROR*, *logger_name: str = 'PyChemkin'*)

Bases: object

Provides the singleton logger for the PyChemkin.

Parameters

to_file
[bool, default: False] Whether to include the logs in a file.

Overview

Methods

get_logger	Get the logger.
set_level	Set the logger output level.
enable_output	Enable logger output to a given stream.
add_file_handler	Save logs to a file in addition to printing them to the standard output.

Import detail

```
from ansys.chemkin.logger import ChemkinLogger
```

Method detail

ChemkinLogger.**get_logger**() → logging.Logger

Get the logger.

Returns

Logger
Logger.

ChemkinLogger.**set_level**(*level: int*) → None

Set the logger output level.

Parameters

level
[int, {0, 10, 20, 30, 40, 50}] Output Level of the logger. 0 = NOTSET 10 = DEBUG 20 = INFO 30 = WARNING 40 = ERROR 50 = CRITICAL

ChemkinLogger.**enable_output**(*stream=None*)

Enable logger output to a given stream.

If a stream is not specified, `sys.stderr` is used.

Parameters

stream: TextIO, default: `sys.stderr`
Stream to output the log output to.

ChemkinLogger.**add_file_handler**(*logs_dir: str = './.log'*)

Save logs to a file in addition to printing them to the standard output.

Parameters

logs_dir
[str, default: `"./.log"`] Directory of the logs.

14.6.2 Description

Provides the singleton helper class for the logger.

14.6.3 Module detail

logger.**logger**

14.7 The `ansys.chemkin.mixture` library

14.7.1 Summary

Classes

Mixture	define a mixture based on the gas species in the given chemistry set

Functions

isothermal_mixing	Find the resulting gas mixture properties from mixing a number of gas mixtures at the given mixture temperature
adiabatic_mixing	Find the resulting gas mixture properties from mixing a number of gas mixtures with constant total enthalpy
calculate_mixture_temperatu	Compute the mixture temperature from the given mixture enthalpy,
interpolate_mixtures	Create a new mixture object by interpolating the two mixture objects with a specific weight ratio
compare_mixtures	Compare properties of mixture B against those of mixture A. The mixture properties
calculate_equilibrium	Get the equilibrium mixture composition corresponding to the given initial mixture composition at the given pressure and temperature
equilibrium	Find the equilibrium state mixture corresponding to the given mixture
detonation	Find the Chapman-Jouguet state mixture and detonation wave speed corresponding to the given mixture

Mixture

class ansys.chemkin.mixture.**Mixture**(*chem:* ansys.chemkin.chemistry.Chemistry)

define a mixture based on the gas species in the given chemistry set

Overview

Methods

list_composition	list the mixture composition
Find_Equilibrium	Create the equilibrium state mixture corresponding to mixture itself
HML	Get enthalpy of the mixture
CPBL	Get specific heat capacity of the mixture
ROP	Get species molar rate of production from the given mixture condition: pressure, temperature, and species compositions
RxnRates	Get molar rates of the gas reactions from the given mixture condition: pressure, temperature, and species composition
species_Cp	Get species specific heat capacity at constant pressure
species_H	Get species enthalpy
species_Visc	Get species viscosity
species_Cond	Get species conductivity
species_Diffusion_Coeff	Get species diffusion coefficients
mixture_viscosity	Get viscosity of the gas mixture
mixture_conductivity	Get conductivity of the gas mixture
mixture_diffusion_coeff	Get mixture-averaged species diffusion coefficients of the gas mixture
mixture_binary_diffusio	Get multi-component species binary diffusion coefficients of the gas mixture
mixture_thermal_diffusi	Get thermal diffusivity of the gas mixture
volHRR	Get volumetric heat release rate
massROP	Get species mass rates of production
list_ROP	list information about species molar production rate in descending order
list_massROP	list information about species mass rate of production in descending order
list_reaction_rates	list information about reaction rate in descending order
X_by_Equivalence_Ratio	Specify the mixture molar composition by providing the equivalence ratio, the mole fractions of the fuel mixture,
Y_by_Equivalence_Ratio	Specify the mixture molar composition by providing the equivalence ratio, the mole fractions of the fuel mixture,
get_EGR_mole_fraction	Compute the EGR composition in mole fraction corresponding to this mixture
validate	Check whether the mixture is fully defined before being used by other methods
use_realgas_cubicEOS	Turn ON the real-gas cubic EOS to compute mixture properties if the mechanism contains necessary data
use_idealgas_law	Turn on the ideal gas law to compute mixture properties
set_realgas_mixing_rule	Set the mixing rule to be used for calculating the real-gas mixture properties

Properties

chemID	Get chemistry set index
KK	Get the number of gas species
pressure	Get gas mixture pressure [dynes/cm2]
temperature	Get gas mixture temperature
volume	Get mixture volume
X	Get mixture mole fraction
Y	Get mixture mass fraction
concentration	Get mixture molar concentrations
EOS	Get the available real-gas EOS model that is provided in the mechanism
WT	Get species molecular masses
WTM	Get mean molar mass of the gas mixture
RHO	Get mixture mass density

Attributes

transport_data
userealgas
mole_fractions
mass_fractions
species_molar_weight
mean_molar_weight

Static methods

normalize	Normalize the mixture composition
mean_molar_mass	Get mean molar mass of the gas mixture
*mole_fraction_to_mass_*	Convert mole fraction to mass fraction
*mass_fraction_to_mole_*	Convert mass fraction to mole fraction
mass_fraction_to_conce	Convert mass fractions to molar concentrations
mole_fraction_to_conce	Convert mole fractions to molar concentrations
density	Get mass density from the given mixture condition: pressure, temperature, and species composition
mixture_specific_heat	Get mixture specific heat capacity from the given mixture condition: pressure, temperature, and species composition
mixture_enthalpy	Get mixture enthalpy from the given mixture condition: pressure, temperature, and species composition
rate_of_production	Get species molar rate of production from the given mixture condition: pressure, temperature, and species composition
reaction_rates	Get molar rates of the gas reactions from the given mixture condition: pressure, temperature, and species composition

Import detail

```
from ansys.chemkin.mixture import Mixture
```

Property detail

property Mixture.**chemID: int**

Get chemistry set index

Returns

chemID: integer
chemistry set index associated with this Mixture

property Mixture.**KK: int**

Get the number of gas species

Returns

num_spec: integer
number of gas species in the mixture

property Mixture.pressure: float

Get gas mixture pressure [dynes/cm2]

Returns

pressure: double
mixture pressure [dynes/cm2]

property Mixture.temperature: float

Get gas mixture temperature

Returns

temperature: double
temperature [K]

property Mixture.volume: float

Get mixture volume

Returns

volume: double
mixture volume [cm3]

property Mixture.X: numpy.typing.NDArray[numpy.double]

Get mixture mole fraction

Returns

X: 1-D double array, dimensdion = number_species
mixture composition in mole fractions

property Mixture.Y: numpy.typing.NDArray[numpy.double]

Get mixture mass fraction

Returns

Y: 1-D double array, dimensdion = number_species
mixture composition in mass fractions

property Mixture.concentration: numpy.typing.NDArray[numpy.double]

Get mixture molar concentrations

Returns

c: 1-D double array, dimensdion = number_species
mixture compisition in molar concentrations [mole/cm3]

property Mixture.EOS: int

Get the available real-gas EOS model that is provided in the mechanism

Returns

EOS: integer
index of the realgas EOS model defined in the gas-phase mechanism input file

property Mixture.**WT: numpy.typing.NDArray[numpy.double]**

Get species molecular masses

Returns

WT: 1-D double array, dimension = number_species
species molecular masses [gm/mole]

property Mixture.**WTM: float**

Get mean molar mass of the gas mixture

Returns

WTM: double
mean molecular mass of the mixture [gm/mol]

property Mixture.**RHO: float**

Get mixture mass density

Returns

RHO: double
mixture density [gm/cm3]

Attribute detail

Mixture.**transport_data**

Mixture.**userealgas**

Mixture.**mole_fractions**

Mixture.**mass_fractions**

Mixture.**species_molar_weight**

Mixture.**mean_molar_weight**

Method detail

static Mixture.**normalize**(*frac: numpy.typing.ArrayLike*) → tuple[int, numpy.typing.NDArray[numpy.double]]

Normalize the mixture composition

Parameters

frac: 1-D double array
mixture composition to be normalized

Returns

error code: integer
error code

localfrac: 1-D double array
normalized fraction array

static Mixture.**mean_molar_mass**(*frac: numpy.typing.NDArray[numpy.double]*, *wt: numpy.typing.NDArray[numpy.double]*, *mode: str*) → float

Get mean molar mass of the gas mixture

Parameters

frac: 1-D double array, dimensdion = number_species
mixture composition in 'mass' or mole fraction as indicated by mode

wt: 1-D double array, dimensdion = number_species
species molar mass [gm/mol]

mode: string, {'mole', 'mass'}
flag indicates the frac array is 'mass' or 'mole' fractions

Returns

mwt: double
mean molar mass [gm/mol]

static Mixture.**mole_fraction_to_mass_fraction**(*molefrac: numpy.typing.NDArray[numpy.double]*, *wt: numpy.typing.NDArray[numpy.double]*) → numpy.typing.NDArray[numpy.double]

Convert mole fraction to mass fraction

Parameters

molefrac: 1-D double array, dimension = number_species
mixture composition in mole fractions

wt: 1-D double array, dimension = number_species
species molar mass [gm/mol]

Returns

massfrac: 1-D double array, dimension = number_species
mass fractions

static Mixture.**mass_fraction_to_mole_fraction**(*massfrac: numpy.typing.NDArray[numpy.double]*, *wt: numpy.typing.NDArray[numpy.double]*) → numpy.typing.NDArray[numpy.double]

Convert mass fraction to mole fraction

Parameters

massfrac: 1-D double array, dimension = number_species
mixture composition in mass fractions

wt: 1-D double array, dimension = number_species
species molar mass [gm/mol]

Returns

molefrac: 1-D double array, dimension = number_species
mole fractions

static Mixture.**mass_fraction_to_concentration**(*chemID: int*, *p: float*, *t: float*, *massfrac: numpy.typing.NDArray[numpy.double]*, *wt: numpy.typing.NDArray[numpy.double]*) → numpy.typing.NDArray[numpy.double]

Convert mass fractions to molar concentrations

Parameters

chemID: integer
chemistry set index associated with the mixture

p: double
pressure [dynes/cm2]

t: double
temperature [K]

massfrac: 1-D double array, dimension = number_species
mixture compisition in mass fractions

wt: 1-D double array, dimension = number_species
species molecular masses [gm/mole]

Returns

c: 1-D double array, dimension = number_species
molar concentrations [mole/cm3]

static Mixture.**mole_fraction_to_concentration**(*chemID: int*, *p: float*, *t: float*, *molefrac: numpy.typing.NDArray[numpy.double]*, *wt: numpy.typing.NDArray[numpy.double]*) → numpy.typing.NDArray[numpy.double]

Convert mole fractions to molar concentrations

Parameters

chemID: integer
chemistry set index associated with the mixture

p: double
pressure [dynes/cm2]

t: double
temperature [K]

molefrac: 1-D double array, dimension = number_species
mixture compisition in mole fractions

wt: 1-D double array, dimension = number_species
species molecular masses [gm/mole]

Returns

c: 1-D double array, dimension = number_species
molar concentrations [mole/cm3]

`Mixture.list_composition`(*mode: str*, *option: str = ' '*, *bound: float = 0.0*)

list the mixture composition

Parameters

mode: string, {'mole', 'mass'}
flag indicates the fractions returned are 'mass' or 'mole' fractions

option: string, {'all, ' '}, default = 'all'
flag indicates to list 'all' species or just the species with non-zero fraction

bound: double
minimum fraction value for the species to be printed

`static Mixture.density`(*chemID: int*, *p: float*, *t: float*, *frac: numpy.typing.NDArray[numpy.double]*, *wt: numpy.typing.NDArray[numpy.double]*, *mode: str*) → float

Get mass density from the given mixture condition: pressure, temperature, and species composition

Parameters

chemID: integer
chemistry set index associated with the mixture

p: double
mixture pressure in [dynes/cm2]

t: double
mixture temperature in [K]

frac: 1-D double array, dimension = number_species
mixture composition given by either mass or mole fractions as specified by mode

wt: 1-D double array, dimension = number_species
molar masses of the species in the mixture in [gm/mol]

mode: string, {'mole', 'mass'}
flag indicates the frac array is 'mass' or 'mole' fractions

Returns

den: double
mass density in [gm/cm3]

`static Mixture.mixture_specific_heat`(*chemID: int*, *p: float*, *t: float*, *frac: numpy.typing.NDArray[numpy.double]*, *wt: numpy.typing.NDArray[numpy.double]*, *mode: str*) → float

Get mixture specific heat capacity from the given mixture condition: pressure, temperature, and species composition

Parameters

chemID: integer
chemistry set index associated with the mixture

p: double
mixture pressure in [dynes/cm2]

t: double
mixture temperature in [K]

frac: 1-D double array, dimension = number_species
mixture composition given by either mass or mole fractions as specified by mode

wt: 1-D double array, dimension = number_species
molar masses of the species in the mixture in [gm/mol]

mode: string, {'mole', 'mass'}
flag indicates the frac array is 'mass' or 'mole' fractions

Returns

CpB: double
mixture specific heat capacity [erg/mol-K]

`static` Mixture.**mixture_enthalpy**(*chemID: int*, *p: float*, *t: float*, *frac: numpy.typing.NDArray[numpy.double]*, *wt: numpy.typing.NDArray[numpy.double]*, *mode: str*) → float

Get mixture enthalpy from the given mixture condition: pressure, temperature, and species composition

Parameters

chemID: integer
chemistry set index associated with the mixture

p: double
mixture pressure in [dynes/cm2]

t: double
mixture temperature in [K]

frac: 1-D double array, dimension = number_species
mixture composition given by either mass or mole fractions as specified by mode

wt: 1-D double array, dimension = number_species
molar masses of the species in the mixture in [gm/mol]

mode: string, {'mole', 'mass'}
flag indicates the frac array is 'mass' or 'mole' fractions

Returns

H: double
mixture enthalpy [erg/mol]

static Mixture.**rate_of_production**(*chemID: int*, *p: float*, *t: float*, *frac: numpy.typing.NDArray[numpy.double]*, *wt: numpy.typing.NDArray[numpy.double]*, *mode: str*) → numpy.typing.NDArray[numpy.double]

Get species molar rate of production from the given mixture condition: pressure, temperature, and species composition

Parameters

chemID: integer
chemistry set index associated with the mixture

p: double
mixture pressure in [dynes/cm2]

t: double
mixture temperature in [K]

frac: 1-D double array, dimension = number_species
mixture composition given by either mass or mole fractions as specified by mode

wt: 1-D double array, dimension = number_species
molar masses of the species in the mixture in [gm/mol]

mode: string, {'mole', 'mass'}
flag indicates the frac array is 'mass' or 'mole' fractions

Returns

ROP: 1-D double array, dimension = number_species
species molar rate of production in [mol/cm3-sec]

static Mixture.**reaction_rates**(*chemID: int*, *numbreaction: int*, *p: float*, *t: float*, *frac: numpy.typing.NDArray[numpy.double]*, *wt: numpy.typing.NDArray[numpy.double]*, *mode: str*) → tuple[numpy.typing.NDArray[numpy.double], numpy.typing.NDArray[numpy.double]]

Get molar rates of the gas reactions from the given mixture condition: pressure, temperature, and species composition

Parameters

chemID: integer
chemistry set index associated with the mixture

numbreaction: integer
number of gas reactions associated with the chemistry set

p: double
mixture pressure in [dynes/cm2]

t: double
mixture temperature in [K]

frac: 1-D double array, dimension = number_species
mixture composition given by either mass or mole fractions as specified by mode

wt: 1-D double array, dimension = number_species
molar masses of the species in the mixture in [gm/mol]

mode: string, {'mole', 'mass'}
flag indicates the frac array is 'mass' or 'mole' fractions

Returns

Kforward: 1-D double array, dimension = numbreaction
forward molar rates of the reactions in [mol/cm3-sec]

Kreverse: 1-D double array, dimension = numbreaction
reverse molar rates of the reactions in [mol/cm3-sec]

`Mixture.Find_Equilibrium()`

Create the equilibrium state mixture corresponding to mixture itself

Returns

EQState: Mixture object
gas mixture at the equilibrium state

`Mixture.HML()` → float

Get enthalpy of the mixture

Returns

hml: double
mixture enthalpy [erg/mol]

`Mixture.CPBL()` → float

Get specific heat capacity of the mixture

Returns

cpbl: double
mixture specific heat capacity [erg/mol-K]

`Mixture.ROP()` → numpy.typing.NDArray[numpy.double]

Get species molar rate of production from the given mixture condition: pressure, temperature, and species compositions

Returns

ROP: 1-D double array, dimension = number_species
species molar rate of production in [mol/cm3-sec]

`Mixture.RxnRates()` → tuple[numpy.typing.NDArray[numpy.double], numpy.typing.NDArray[numpy.double]]

Get molar rates of the gas reactions from the given mixture condition: pressure, temperature, and species composition

Returns

Kforward: 1-D double array, dimension = numbreaction
forward molar rates of the reactions in [mol/cm3-sec]

Kreverse: 1-D double array, dimension = numbreaction
reverse molar rates of the reactions in [mol/cm3-sec]

`Mixture.species_Cp()` → numpy.typing.NDArray[numpy.double]

Get species specific heat capacity at constant pressure

Returns

Cp: 1-D double array, dimension = number_species
species specific heat capacities at constant pressure [ergs/mol-K]

`Mixture.species_H()` → numpy.typing.NDArray[numpy.double]

Get species enthalpy

Returns

H: 1-D double array, dimension = number_species
species enthalpy [ergs/mol]

`Mixture.species_Visc()` → numpy.typing.NDArray[numpy.double]

Get species viscosity

Returns

visc:
[1-D double array, dimension = number_species] species viscosity [gm/cm-sec]

`Mixture.species_Cond()` → numpy.typing.NDArray[numpy.double]

Get species conductivity

Returns

cond: 1-D double array, dimension = number_species
species conductivity [ergs/cm-K-sec]

`Mixture.species_Diffusion_Coeffs()` → numpy.typing.NDArray[numpy.double]

Get species diffusion coefficients

Returns

diffusioncoeffs: 2-D double array, dimension = [number_species, number_species]
species diffusion coefficients [cm2/sec]

`Mixture.mixture_viscosity()` → float

Get viscosity of the gas mixture

Returns

visc: double
mixture viscosity [gm/cm-sec]

Mixture.**mixture_conductivity**() → float
Get conductivity of the gas mixture

Returns

cond: double
mixture conductivity [erg/cm-K-sec]

Mixture.**mixture_diffusion_coeffs**() → numpy.typing.NDArray[numpy.double]
Get mixture-averaged species diffusion coefficients of the gas mixture

Returns

diffusioncoeffs: 1-D double array, dimension = number_species
mixture-averaged diffusion coefficients [cm2/sec]

Mixture.**mixture_binary_diffusion_coeffs**() → numpy.typing.NDArray[numpy.double]
Get multi-component species binary diffusion coefficients of the gas mixture

Returns

binarydiffusioncoeffs: 2-D double array, dimension = [number_species, number_species]
binary diffusion coefficients [cm2/sec]

Mixture.**mixture_thermal_diffusion_coeffs**() → numpy.typing.NDArray[numpy.double]
Get thermal diffusivity of the gas mixture

Returns

thermaldiffusioncoeffs: 1-D double array, dimension = number_species
thermal diffusivity [gm/cm-sec]

Mixture.**volHRR**() → float
Get volumetric heat release rate

Returns

vol_HRR: double
volumetric heat release rate [ergs/cm3-sec]

Mixture.**massROP**() → numpy.typing.NDArray[numpy.double]
Get species mass rates of production

Returns

massROP: 1-D double array, dimension = number_species
mass rates of production [gm/cm3-sec]

Mixture.**list_ROP**(*threshold: float = 0.0*) → tuple[numpy.typing.NDArray[numpy.int32], numpy.typing.NDArray[numpy.double]]

list information about species molar production rate in descending order

Parameters

threshold: double, optional, default = 0.0
minimum absolute ROP value to be printed

Returns

order: 1-D integer array, dimension = number_species
sorted species index

sorted_ROP: 1-D double array, dimension = number_species
sorted ROP values [gm/cm3-sec]

Mixture.**list_massROP**(*threshold: float = 0.0*) → tuple[numpy.typing.NDArray[numpy.int32], numpy.typing.NDArray[numpy.double]]

list information about species mass rate of production in descending order

Parameters

threshold: double, optional, default = 0.0
minimum absolute mass ROP value to be printed

Returns

order: 1-D integer array, dimension = number_species
sorted species index

sorted_massROP: 1-D double array, dimension = number_species
sorted mass ROP values [gm/cm3-sec]

Mixture.**list_reaction_rates**(*threshold: float = 0.0*) → tuple[numpy.typing.NDArray[numpy.int32], numpy.typing.NDArray[numpy.double]]

list information about reaction rate in descending order

Parameters

threshold: double, optional, default = 0.0
minimum absolute reaction rate value to be printed

Returns

order: 1-D integer array, dimension = numb_reactions
sorted reaction index

sorted_RxnRates: 1-D double array, dimension = numb_reactions
sorted reaction rate values [mol/cm3-sec]

Mixture.**X_by_Equivalence_Ratio**(*chemistryset:* ansys.chemkin.chemistry.Chemistry, *fuel_molefrac: numpy.typing.NDArray[numpy.double]*, *oxid_molefrac: numpy.typing.NDArray[numpy.double]*, *add_molefrac: numpy.typing.NDArray[numpy.double]*, *products: list[str]*, *equivalenceratio: float*, *threshold: float = 1e-10*) → int

Specify the mixture molar composition by providing the equivalence ratio, the mole fractions of the fuel mixture, the oxidizer mixture, and the additives mixture, and the list of the complete combustion product species.

Parameters

chemistryset: Chemistry object
the chemistry set used to create the mixtures

fuel_molefrac: 1-D double array, dimension = number_species
mole fractions of the fuel mixture

oxid_molefrac: 1-D double array, dimension = number_species
mole fractions of the oxidizer mixture

add_molefrac: 1-D double array, dimension = number_species
mole fractions of the additives mixture

products: list of string
list of the complete combustion species symbols

equivalenceratio: double
equivalence ratio of the final mixture (double scalar)

threshold: double, optional, default = 1.0e-10
minimum species fraction value to be included in the stoichiometric coefficient calculation

Returns

Error status: integer

`Mixture.Y_by_Equivalence_Ratio`(*chemistryset:* ansys.chemkin.chemistry.Chemistry, *fuel_massfrac: numpy.typing.NDArray[numpy.double]*, *oxid_massfrac: numpy.typing.NDArray[numpy.double]*, *add_massfrac: numpy.typing.NDArray[numpy.double]*, *products: list[str]*, *equivalenceratio: float*, *threshold: float = 1e-10*) → int

Specify the mixture molar composition by providing the equivalence ratio, the mole fractions of the fuel mixture, the oxidizer mixture, and the additives mixture, and the list of the complete combustion product species.

Parameters

chemistryset: Chemistry object
the chemistry set used to create the mixtures

fuel_massfrac: 1-D double array, dimension = number_species
mass fractions of the fuel mixture

oxid_massfrac: 1-D double array, dimension = number_species
mass fractions of the oxidizer mixture

add_massfrac: 1-D double array, dimension = number_species
mass fractions of the additives mixture

products: list of string
list of the complete combustion species symbols

equivalenceratio: double
equivalence ratio of the final mixture (double scalar)

threshold: double, optional, default = 1.0e-10
minimum species fraction value to be included in the stoichiometric coefficient calculation

Returns

Error status: integer

`Mixture.get_EGR_mole_fraction`(*EGRratio: float*, *threshold: float = 1e-08*) → numpy.typing.NDArray[numpy.double]

Compute the EGR composition in mole fraction corresponding to this mixture

Parameters

EGRratio: double
exhaust gas recirculation (EGR) molar ratio

threshold: double, optional, default = 1.0e-8
minimum species fraaction value to be included in the EGR stream

Returns

EGR_molefrac: 1-D double array, dimension = number_species
EGR stream compostion in mole fractions

`Mixture.validate`() → int

Check whether the mixture is fully defined before being used by other methods

Returns

Error status: integer

`Mixture.use_realgas_cubicEOS`()

Turn ON the real-gas cubic EOS to compute mixture properties if the mechanism contains necessary data

`Mixture.use_idealgas_law`()

Turn on the ideal gas law to compute mixture properties

`Mixture.set_realgas_mixing_rule`(*rule: int = 0*)

Set the mixing rule to be used for calculating the real-gas mixture properties

Parameters

rule: integer, optional, default = 0

mixing rule:
0 for the Van der Waals mixing rule; 1 for the critical properties mixing rule (integer scalar)

14.7.2 Description

Chemkin Mixture utilities.

14.7.3 Module detail

mixture.**isothermal_mixing**(*recipe: list[tuple[*Mixture*, float]]*, *mode: str*, *finaltemperature: float*) → *Mixture*

Find the resulting gas mixture properties from mixing a number of gas mixtures at the given mixture temperature

Parameters

recipe: list of tuples, [(Mixture object, fraction), ...]
non-zero mixture composition corresponding to the given mole/mass fraction array

mode: string, {'mass', 'mole'}, default = 'mole'
indicting the fractions given in the recipe are in 'mole' or 'mass' ratios

finaltemperature: double
temperature of the resulting gas mixture after mixing

Returns

finalmixture: Mixture object
the resulting gas mixture after mixing

mixture.**adiabatic_mixing**(*recipe: list[tuple[*Mixture*, float]]*, *mode: str*) → *Mixture*

Find the resulting gas mixture properties from mixing a number of gas mixtures with constant total enthalpy

Parameters

recipe: list of tuples, [(Mixture object, fraction), ...]
non-zero mixture composition corresponding to the given mole/mass fraction array

mode: string, {'mass', 'mole'}, default = 'mole'
indicting the fractions given in the recipe are in 'mole' or 'mass' ratios

Returns

finalmixture: Mixture object
the resulting gas mixture after mixing

mixture.**calculate_mixture_temperature_from_enthalpy**(*mixture:* Mixture, *mixtureH: float*, *guesstemperature: float = 0.0*) → int

Compute the mixture temperature from the given mixture enthalpy, the solved mixture temperature is stored as the temperature attribute of the given gas mixture (i.e., as mixture.temperature)

Parameters

mixture: Mixture object
gas mixture of interest

mixtureH: double
mixture enthalpy of the given gas mixture [erg/mol]

guesstemperature: double, optional
a guessed value for the mixture temperature at the start of the iteration process

Returns

error code: integer

mixture.**interpolate_mixtures**(*mixtureleft:* Mixture, *mixtureright:* Mixture, *ratio: float*) → *Mixture*

Create a new mixture object by interpolating the two mixture objects with a specific weight ratio

::

mixture_new = (1 - ratio) * mixtureleft + ratio * mixtureright

Parameters

mixtureleft: Mixture object
mixture A to be mixed

mixtureright: Mixture object
mixture B to be mixed

ratio: double
the weight parameters for interpolation, 0 <= ratio <= 1

Returns

mixturenew: Mixture object
the resulting gas mixture

mixture.**compare_mixtures**(*mixtureA:* Mixture, *mixtureB:* Mixture, *atol: float = 1e-10*, *rtol: float = 0.001*, *mode: str = 'mass'*) → tuple[bool, float, float]

Compare properties of mixture B against those of mixture A. The mixture properties include pressure [atm], temperature [K], and species mass/mole fractions. When the differences in the property values satisfy both the absolute and the relative tolerances, this method will return "True", that is, mixture B is essentially identical to mixture A; otherwise, "False" will be returned.

Parameters

mixtureA: Mixture object
mixture A, the target mixture

mixtureB: Mixture object
mixture B, the sample mixture

atol: double, default = 1.0e-10
the absolute tolerance for the max property differences

rtol: double, default = 1.0e-3
the relative tolerance for the max property differences

mode: string {"mass", "mole"}, default = "mass"
compare species "mass" or "mole" fractions

Returns

issame: boolean
the equivalence of the two mixtures

atol_max: double
the max absolute difference value

rtol_max: double
the max relative difference value

mixture.**calculate_equilibrium**(*chemID: int*, *p: float*, *t: float*, *frac: numpy.typing.NDArray[numpy.double]*, *wt: numpy.typing.NDArray[numpy.double]*, *mode_in: str*, *mode_out: str*, *EQOption: int = 1*, *useRealGas: int = 0*) → tuple[list[float], numpy.typing.NDArray[numpy.double]]

Get the equilibrium mixture composition corresponding to the given initial mixture composition at the given pressure and temperature

Parameters

chemID: integer
chemistry set index associated with the mixture

p: double
initial mixture pressure in [dynes/cm2]

t: double
initial mixture temperature in [K]

frac: 1-D double array
initial mixture composition given by either mass or mole fractions as specified by mode_in

wt: 1-D double arrays
molar masses of the species in the mixture in [gm/mol]

mode_in: string, {'mass', 'mole'}, default = 'mole'
flag indicates the frac array is 'mass' or 'mole' fractions

mode_out: string, {'mass', 'mole'}, default = 'mole'
flag to indicate the returning composition is in 'mole' or 'mass' fraction

EQOption: integer, {1, 2, 3, 4, 5, 6, 7, 8, 9, 10}
equilibrium type (see below)

::

1. SPECIFIED T AND P
2. SPECIFIED T AND V
3. SPECIFIED T AND S
4. SPECIFIED P AND V
5. SPECIFIED P AND H
6. SPECIFIED P AND S
7. SPECIFIED V AND U
8. SPECIFIED V AND H
9. SPECIFIED V AND S
10. CHAPMAN-JOUGUET DETONATION

useRealGas: integer, {0, 1}
option to turned ON/OFF (1/0) the real-gas cubic EOS if available

Returns

state_variables_equilibrium: list of doubles,
equilibrium pressure [dynes/cm2], equilibrium temperature [K], speed of sound at equilibrium [cm/sec], detonation wave speed [cm/sec]. Note: if Chapmen-Jouguet option is not used, both speed of sound and detonation wave speed are set to 0.0

equilibrium composition: 1-D double array
given in fractions indicated by the parameter mode_out

`mixture.`**`equilibrium`**(*mixture:* Mixture, *opt: int = 1*) → *Mixture*

Find the equilibrium state mixture corresponding to the given mixture

Parameters

mixture: Mixture object
initial gas mixture

opt: integer, {1, 2, 4, 5, 7, 8}
equilibrium type

::

1. SPECIFIED T AND P
2. SPECIFIED T AND V
3. SPECIFIED T AND S (*)
4. SPECIFIED P AND V
5. SPECIFIED P AND H
6. SPECIFIED P AND S (*)
7. SPECIFIED V AND U
8. SPECIFIED V AND H
9. SPECIFIED V AND S (*)
10. CHAPMAN-JOUGUET DETONATION (*)

(*) indicates the options are not available

Returns

finalmixture: Mixture object
gas mixture at the equilibrium state

`mixture.`**`detonation`**(*mixture:* Mixture) → tuple[list[float], *Mixture*]

Find the Chapman-Jouguet state mixture and detonation wave speed corresponding to the given mixture

Parameters

mixture: Mixture object
initial gas mixture

Returns

speed_values: list of doubles
speed of sound [cm/sec], detonation wave speed [cm/sec].

finalmixture: Mixture object
gas mixture at the equilibrium state

14.8 The ansys.chemkin.utilities library

14.8.1 Summary

Functions

where_element_in_array_1D	Find the number of occurrence and the element index in the 1D arr array that matches the target value.
bisect	Use bisectional method to find the largest index in the xarray of which its value is small or equal to the target x value
find_interpolate_parameter	Find the index ileft that
interpolate_array	Find the value in the y_array from the interpolation parameters ileft and ratio
create_mixture_recipe_from	Build a PyChemkin mixture recipe/formula from a species fraction array (i.e., mixture mole/mass composition).
calculate_stoichiometrics	calculate the stoichiometric coefficients of the complete combustion reaction of the given fuel and oxidizer mixtures.
random	Generate a (reproducible) random floating number value >= 0.0 and < 1.0
find_file	Find the correct version of the given partial file name.

Attributes

ck_rng

14.8.2 Description

pychemkin utilities

14.8.3 Module detail

utilities.**where_element_in_array_1D**(*arr: numpy.typing.NDArray[numpy.double] | numpy.typing.NDArray[numpy.int32]*, *target: int | float*) → tuple[int, numpy.typing.NDArray[numpy.int32]]

Find the number of occurrence and the element index in the 1D arr array that matches the target value. Using numpy.argwhere might be more efficient. However, the numpy method returns a list of lists of occurrence indices while this might be necessary for general applications, it is an overkill for simple 1D array cases.

Parameters

arr: integer or double array
the reference 1D integer or double array

target: integer or double scalar
the target value to be matched

Returns

number_of_occurrences: integer occurrence_index: integer array

utilities.**bisect**(*ileft: int*, *iright: int*, *x: float*, *xarray*) → int

Use bisectional method to find the largest index in the xarray of which its value is small or equal to the target x value

Parameters

ileft: integer
index of xarray that represents the current lower bound

iright: integer
index of xarray that represents the current upper bound

x: double
target value

xarray: double array
a sorted array containing all x values in strictly ascending order x[i] < x[i+1]

Returns

itarget: integer
the largest index in the xarray of which its value is small or equal to the target x value

utilities.**find_interpolate_parameters**(*x: float*, *xarray: numpy.typing.NDArray[numpy.double]*) → tuple[int, float]

Find the index ileft that
xarray[ileft] <= x <= xarray[iright] where iright = ileft + 1

Parameters

x: double
target value

xarray: double array
a sorted array containing all x values in strictly ascending order x[i] < x[i+1]

Returns

itarget: integer
the largest index in the xarray of which its value is small or equal to the target x value

ratio: double
the distance ratio = (x - xarray[ileft])/(xarray[ileft+1] - xarray[ileft])

utilities.**interpolate_array**(*x: float*, *x_array: numpy.typing.NDArray[numpy.double]*, *y_array: numpy.typing.NDArray[numpy.double]*) → float

Find the value in the y_array from the interpolation parameters ileft and ratio
y = (1-ratio)* y_array[ileft] + ratio * y_array[ileft+1] where ileft and ratio are determined from the target x value and the xarray

Parameters

x: double
target value

x_array: double array
a sorted array containing all x values in strictly ascending order x[i] < x[i+1]

y_array: double array
dependent variable array

Returns

y: double
the interpolated dependent variable value corresponding the given x value

utilities.**create_mixture_recipe_from_fractions**(*chemistry_set:* ansys.chemkin.chemistry.Chemistry, *frac: numpy.typing.NDArray[numpy.double]*) → tuple[int, list[tuple[str, float]]]

Build a PyChemkin mixture recipe/formula from a species fraction array (i.e., mixture mole/mass composition). This mixture recipe can then be used to create the corresponding Mixture object.

Parameters

chemistry_set: Chemistry object
the Chemistry object will be used to create the mixture

frac: double array
mole or mass fractions of the mixture

Returns

count: integer
the size of the recipe list containing [gas species, mole/mass fraction] tuples

recipe: list of tuples, [(species_symbol, fraction), ...]
non-zero mixture composition corresponding to the given mole/mass fraction array

utilities.**calculate_stoichiometrics**(*chemistryset:* ansys.chemkin.chemistry.Chemistry, *fuel_molefrac: numpy.typing.NDArray[numpy.double]*, *oxid_molefrac: numpy.typing.NDArray[numpy.double]*, *prod_index: numpy.typing.NDArray[numpy.int32]*) → tuple[float, numpy.typing.NDArray[numpy.double]]

calculate the stoichiometric coefficients of the complete combustion reaction of the given fuel and oxidizer mixtures. Consider the complete combustion of the fuel + oxidizer mixture

```
(fuel species) + alpha*(oxidizer species) <=> nu(1)*prod(1) + ... + nu(numb_
↪prod)*prod(numb_prod)
```

The number of unknowns is equal to the number of elements that make of all the fuel and oxidizer species. And the number of product species must be one less than the number of unknowns. The unknowns

```
alpha is the stoichiometric coefficient multiplier of the oxidizer species
nu(1), ... nu(numb_prod) are the stoichiometric coefficients of the product species
```

The conservation of elements yields a set of linear algebraic equations

```
A x = b
```

in which x = [-alpha | nu(1), ..., nu(numb_prod)] (a vector of size numb_elem) can be obtained.

Parameters

chemistryset: Chemistry object
the Chemistry object used to create the fuel and the oxidizer mixtures

fuel_molefrac: 1-D double array
mole fractions of the fuel mixture

oxid_molefrac: 1-D double array
mole fractions of the oxidizer mixture

prod_index: 1-D integer array
the species indices of the complete combustion products

Returns

alpha: double
oxidizer_coefficient_multiplier

nu: 1-D double array
stoichiometric_coefficients_of_products

utilities.**random**(*range: None | tuple[float, float] = None*) → float

Generate a (reproducible) random floating number value >= 0.0 and < 1.0 by using the Numpy pseudo-random number generator. If the range tuple (a, b) is given, the random number will have a value >= a and < b.

Parameters

range: tuple of floats (a, b) and b > a, default = (0.0, 1.0)
the range of the random number values

Returns

random: float
random number

utilities.**find_file**(*filepath: str*, *partialfilename: str*, *fileext: str*) → str

Find the correct version of the given partial file name. This is mostly to handle the different years/versions of the MFL mechanisms that come with the Ansys Chemkin installation.

Parameters

filepath: string
the directory where the file is located

partialfilename: string
the leading portion of the file name

fileext: string
file extension

Returns

thefile: string
full path name of the file, = "" if no file matches the 'partialname' in the 'filepath'

utilities.**ck_rng = None**

14.9 The ansys.chemkin.chemistry library

14.9.1 Summary

Classes

Chemistry	define and preprocess Chemkin chemistry set

Functions

verbose	return the global verbose mode indicating the status (ON/OFF) of printing statements that do not have the leading '**' characters
set_verbose	set the global verbose mode to turn ON(True) or OFF(False) of printing statements that do not have the leading '**' characters
chemkin_version	Return the Chemkin-CFD-API version number currently in use
verify_version	Check if the version of Chemkin-CFD-API currently in use meets
done	Release Chemkin license and reset the Chemistry sets
check_chemistryset	check whether the Chemistry Set is initialized in Chemkin-CFD-API
activate_chemistryse	Switch to (re-activate) the work spaces of the current Chemistry Set
force_activate_chem	activate the Chemistry Set automatically and silently.
chemistryset_new	Create a new Chemistry Set initialization flag and set the value to False
chemistryset_initia	Set the Chemistry Set Initialization flag to True
check_active_chemis	Verify if the chemistry set is currently activated.

Attributes

LP_c_char
chemkin_verbose

Constants

MAX_SPECIES_LENGTH
COMPLETE

Chemistry

class ansys.chemkin.chemistry.**Chemistry**(*chem: str = ''*, *surf: str = ''*, *therm: str = ''*, *tran: str = ''*, *label: str = ''*)

define and preprocess Chemkin chemistry set

Overview

Methods

preprocess_transportd	Instruct the preprocessor to include the transport data
set_file_names	Assign all input files of the chemistry set
preprocess	Run Chemkin preprocessor
verify_realgas_model	Verify the availability of real-gas data in the mechanism
verify_transport_data	Verify the availability of transport property data in the mechanism
verify_surface_mechan	Verify the availability of surface chemistry data in the mechanism
get_specindex	Get index of the gas species
SpeciesCp	Get species specific heat capacity at constant pressure
SpeciesCv	Get species specific heat capacity at constant volume (ideal gas only)
SpeciesH	Get species enthalpy
SpeciesU	Get species internal energy
SpeciesVisc	Get species viscosity
SpeciesCond	Get species conductivity
SpeciesDiffusionCoefi	Get species diffusion coefficients
SpeciesComposition	Get elemental composition of a species
use_realgas_cubicEOS	Turn ON the real-gas cubic EOS to compute mixture properties if the mechanism contains necessary data
use_idealgas_law	Turn on the ideal gas law to compute mixture properties
get_reaction_paramete	Get the Arrhenius reaction rate parameters of all gas-phase reactions
set_reaction_AFactor	(Re)set the Arrhenius A-Factor of the given reaction
get_reaction_AFactor	get the Arrhenius A-Factor of the given reaction
get_gas_reaction_stri	Get the reaction string of the gas-phase reaction specified by the reaction index.
save	Store the work spaces of the current Chemistry Set
activate	Switch to (re-activate) the work spaces of the current Chemistry Set

Properties

chemfile	Get gas-phase mechanism file name of this chemistry set
thermfile	Get thermodynamic data filename of this chemistry set
tranfile	Get transport data filename of this chemistry set
summaryfile	Get the name of the summary file from the preprocessor
surffile	Get surface mechanism filename of this chemistry set
species_symbols	Get list of gas species symbols
element_symbols	Get the list of element symbols
chemID	Get chemistry set index
surfchem	Get surface chemistry status
KK	Get number of gas species
MM	Get number of elements in the chemistry set
IIGas	Get number of gas-phase reactions
AWT	compute atomic masses
WT	compute gas species molecular masses
EOS	Get the available real-gas EOS model

Attributes

realgas_CuEOS
realgas_mixing_rules
userealgas
KSymbol
ESymbol
label
number_species
number_elements
number_gas_reactions
atomic_weight
species_molar_weight

Import detail

```
from ansys.chemkin.chemistry import Chemistry
```

Property detail

property Chemistry.**chemfile: str**

Get gas-phase mechanism file name of this chemistry set

Returns

chemfile: string
Full path and name of the Chemkin gas-phase mechanism input file

property Chemistry.**thermfile: str**

Get thermodynamic data filename of this chemistry set

Returns

thermfile: string
Full path and name of the Chemkin thermodynamic data file

property Chemistry.**tranfile: str**

Get transport data filename of this chemistry set

Returns

tranfile: string
Full path and name of the Chemkin thransport data file

property Chemistry.**summaryfile: str**

Get the name of the summary file from the preprocessor

Returns

tranfile: string
Full path and name of the preprocessing summary file

property Chemistry.surffile: str
Get surface mechanism filename of this chemistry set

Returns

tranfile: string
Full path and name of the Chemkin surface mechanism input file

property Chemistry.species_symbols
Get list of gas species symbols

Returns

Ksymbol: list of strings
list of species symbols in the gas-phase mechanism

property Chemistry.element_symbols
Get the list of element symbols

Returns

Esymbol: list of strings
list of element symbols in the mechanism

property Chemistry.chemID: int
Get chemistry set index

Returns

chemIDx: integer
index of the Chemistry set

property Chemistry.surfchem: int
Get surface chemistry status

Returns

status: integer, {0, 1}
indicating whether the Chemistry set includes a surface mechanism 0 = this chemistry set does NOT include a surface chemistry 1 = this chemistry set includes a surface chemistry

property Chemistry.KK: int
Get number of gas species

Returns

KK: integer
total number of gas-phase species in the Chemistry set

property Chemistry.**MM: int**

Get number of elements in the chemistry set

Returns

MM: integer
total number of elements in the Chemistry set

property Chemistry.**IIGas: int**

Get number of gas-phase reactions

Returns

IIGas: integer
total number of gas-phase reactions in the Chemistry set

property Chemistry.**AWT: numpy.typing.NDArray[numpy.double]**

compute atomic masses

Returns

AWT: 1-D double array
masses of the elements in the Chemistry set [g/mole]

property Chemistry.**WT: numpy.typing.NDArray[numpy.double]**

compute gas species molecular masses

Returns

WT: 1-D double array
molecular masses of the gas-phase species in the Chemistry set [g/mole]

property Chemistry.**EOS: int**

Get the available real-gas EOS model

Returns

count: integer
number of available cubic EOS models in Chemkin

Attribute detail

Chemistry.**realgas_CuEOS = ['ideal gas', 'Van der Waals', 'Redlich-Kwong', 'Soave', 'Aungier', 'Peng-Robinson']**

Chemistry.**realgas_mixing_rules = ['Van der Waals', 'pseudocritical']**

Chemistry.**userealgas = False**

Chemistry.**KSymbol: list[str] = []**

Chemistry.**ESymbol: list[str] = []**

Chemistry.**label = ' '**

Chemistry.**number_species**

Chemistry.**number_elements**

Chemistry.**number_gas_reactions**

Chemistry.**atomic_weight**

Chemistry.**species_molar_weight**

Method detail

Chemistry.**preprocess_transportdata**()

Instruct the preprocessor to include the transport data

Chemistry.**set_file_names**(*chem: str = ''*, *surf: str = ''*, *therm: str = ''*, *tran: str = ''*)

Assign all input files of the chemistry set

Parameters

chem: string, optional
name of the gas mechanism file with the full path

surf: string, optional
name of the surface mechanism file with the full path

therm: string, optional
name of the thermodynamic data file with the full path

tran: string, optional
name of the transport data file with the full path

Chemistry.**preprocess**() → int

Run Chemkin preprocessor

Returns

Error code: integer

Chemistry.**verify_realgas_model**()

Verify the availability of real-gas data in the mechanism

Chemistry.**verify_transport_data**() → bool

Verify the availability of transport property data in the mechanism

Returns

availability: boolean
True = the transport property is available

Chemistry.**verify_surface_mechanism**() → bool

Verify the availability of surface chemistry data in the mechanism

Returns

availability: boolean
True = the surface chemistry data is available

Chemistry.**get_specindex**(*specname: str*) → int
Get index of the gas species

Returns

specindex: integer
index of the given species symbols in the gas-phase mechanism

Chemistry.**SpeciesCp**(*temp: float = 0.0*, *pres: float | None = None*) → numpy.typing.NDArray[numpy.double]
Get species specific heat capacity at constant pressure

Parameters

temp: double
Temperature [K]

pres: double, optional
Pressure [dynes/cm2] required when real gas model is activated

Returns

Cp: 1-D double array
species specific heat capacities at constant pressure [ergs/mol-K]

Chemistry.**SpeciesCv**(*temp: float = 0.0*, *pres: float | None = None*) → numpy.typing.NDArray[numpy.double]
Get species specific heat capacity at constant volume (ideal gas only)

Parameters

temp: double
Temperature [K]

pres: double, optional
Pressure [dynes/cm2] required when real gas model is activated

Returns

Cv: 1-D double array
species specific heat capacities at constant volume [ergs/mol-K]

Chemistry.**SpeciesH**(*temp: float = 0.0*, *pres: float | None = None*) → numpy.typing.NDArray[numpy.double]
Get species enthalpy

Parameters

temp: double
Temperature [K]

pres: double, optional
Pressure [dynes/cm2] required when real gas model is activated

Returns

H: 1-D double array
species enthalpy [ergs/mol]

Chemistry.**SpeciesU**(*temp: float = 0.0*, *pres: float | None = None*) → numpy.typing.NDArray[numpy.double]

Get species internal energy

Parameters

temp: double
Temperature [K]

pres: double, optional
Pressure [dynes/cm2] required when real gas model is activated

Returns

U: 1-D double array
species internal energy [ergs/mol]

Chemistry.**SpeciesVisc**(*temp: float = 0.0*) → numpy.typing.NDArray[numpy.double]

Get species viscosity

Parameters

temp: double
Temperature [K]

Returns

visc: 1-D double array
species viscosity [gm/cm-sec]

Chemistry.**SpeciesCond**(*temp: float = 0.0*) → numpy.typing.NDArray[numpy.double]

Get species conductivity

Parameters

temp: double
Temperature [K]

Returns

cond: 1-D double array
species conductivity [ergs/cm-K-sec]

Chemistry.**SpeciesDiffusionCoeffs**(*press: float = 0.0*, *temp: float = 0.0*) → numpy.typing.NDArray[numpy.double]

Get species diffusion coefficients

Parameters

press: double
Pressure [dynes/cm2]

temp: double
Temperature [K]

Returns

diffusioncoeffs: 2-D double array, dimension = [number_species, number_species]
species diffusion coefficients [cm2/sec]

Chemistry.**SpeciesComposition**(*elemindex: int = -1*, *specindex: int = -1*) → int

Get elemental composition of a species

Parameters

elemindex: integer
index of the element

specindex: integer
index of the gas species

Returns

count: integer
number of the element in the given gas species

Chemistry.**use_realgas_cubicEOS**()

Turn ON the real-gas cubic EOS to compute mixture properties if the mechanism contains necessary data

Chemistry.**use_idealgas_law**()

Turn on the ideal gas law to compute mixture properties

Chemistry.**get_reaction_parameters**() → tuple[numpy.typing.NDArray[numpy.double], numpy.typing.NDArray[numpy.double], numpy.typing.NDArray[numpy.double]]

Get the Arrhenius reaction rate parameters of all gas-phase reactions

Returns

AFactor: 1-D double array
Arrhenius pre-exponent A-Factor of reaction in [mole-cm3-sec-K]

TBeta: 1-D double array
Arrhenius temperature exponent [-]

AEnergy: 1-D double array
activation temperature [K]

Chemistry.**set_reaction_AFactor**(*reaction_index: int*, *AFactor: float*)

(Re)set the Arrhenius A-Factor of the given reaction

Parameters

reaction_index: integer
index of the gas-phase reaction of which the A-Factor to be reset

AFctor: double
new A-Factor value in [mole-cm3-sec-K]

Chemistry.**get_reaction_AFactor**(*reaction_index: int*) → float

get the Arrhenius A-Factor of the given reaction

Parameters

reaction_index: integer
index of the reaction

Returns

AFactor: double
Arrhenius A-Factor of the given reaction in [mole-cm3-sec-K]

Chemistry.**get_gas_reaction_string**(*reaction_index: int*) → str

Get the reaction string of the gas-phase reaction specified by the reaction index.

Parameters

reaction_index: integer
(base-1) gas-phase reaction index

Returns

reactionstring: string
reaction string of the given reaction

Chemistry.**save**()

Store the work spaces of the current Chemistry Set if new Chemistry Set will be created later in the same project

Chemistry.**activate**()

Switch to (re-activate) the work spaces of the current Chemistry Set when there are multiple Chemistry Sets in the same project

14.9.2 Description

Chemkin Chemistry utilities.

14.9.3 Module detail

chemistry.**verbose**() → bool

return the global verbose mode indicating the status (ON/OFF) of printing statements that do not have the leading '**' characters

Returns

mode: boolean, {True, False}, default = True
the verbose mode

chemistry.**set_verbose**(*OnOff: bool*)

set the global verbose mode to turn ON(True) or OFF(False) of printing statements that do not have the leading '**' characters

Parameters

OnOff: boolean, {True, False}
the verbose mode

chemistry.**chemkin_version**() → int

Return the Chemkin-CFD-API version number currently in use

Returns

version: integer
Chemkin-CFD-API version number

chemistry.**verify_version**(*min_version: int*) → bool

Check if the version of Chemkin-CFD-API currently in use meets the minimum version required by certain operations

Parameters

min_version: integer
minimum chemkin-CFD-API version required to perform the operation

Returns

status: boolean

chemistry.**done**()

Release Chemkin license and reset the Chemistry sets

chemistry.**check_chemistryset**(*chem_index: int*) → bool

check whether the Chemistry Set is initialized in Chemkin-CFD-API

Parameters

chem_index: integer
chemistry set index associated with the Chemistry Set

Returns

status: boolean
the initialization status of the Chemistry set associated with the given Chemistry set index

chemistry.**activate_chemistryset**(*chem_index: int*) → int

Switch to (re-activate) the work spaces of the current Chemistry Set when there are multiple Chemistry Sets in the same project

Parameters

chem_index: integer
chemistry set index associated with the Chemistry Set

Returns

error flag: integer

chemistry.**force_activate_chemistryset**(*chem_index: int*)

activate the Chemistry Set automatically and silently.

Parameters

chem_index: integer
chemistry set index associated with the Chemistry Set

chemistry.**chemistryset_new**(*chem_index: int*)

Create a new Chemistry Set initialization flag and set the value to False

Parameters

chem_index: integer
chemistry set index associated with the Chemistry Set

chemistry.**chemistryset_initialized**(*chem_index: int*)

Set the Chemistry Set Initialization flag to True

Parameters

chem_index: integer
chemistry set index associated with the Chemistry Set

chemistry.**check_active_chemistryset**(*chem_index: int*) → bool

Verify if the chemistry set is currently activated.

Parameters

chem_index: integer
chemistry set index associated with a Chemistry Set

Returns

status: boolean
active status of the Chemistry Set

chemistry.**MAX_SPECIES_LENGTH = 17**

chemistry.**COMPLETE = 0**

chemistry.**LP_c_char**

chemistry.**chemkin_verbose = True**

14.10 The ansys.chemkin.constants library

14.10.1 Summary

Classes

Air	define the “air” composition in PyChemkin with a fixed mixture “recipe”.
air	define the “air” composition in PyChemkin with a fixed mixture “recipe”.

Functions

water_heat_vaporization	Get the heat if vporization of water at the given temperature [K]

Attributes

boltzmann
avogadro
Patm
Ptorrs
ergs_per_joule
joules_per_calorie
ergs_per_calorie
ergs_per_eV
eV_per_K
RGas
RGas_Cal

Air

class ansys.chemkin.constants.**Air**

define the “air” composition in PyChemkin with a fixed mixture “recipe”. A “recipe” is a list of tuples of (“species symbol”, fraction) to define a gas mixture in PyChemkin. This class uses the upper case symbols for oxygen and nitrogen.

Overview

Static methods

X
Y

Import detail

```
from ansys.chemkin.constants import Air
```

Method detail

static Air.**X**() → list[tuple[str, float]]

static Air.**Y**() → list[tuple[str, float]]

air

class ansys.chemkin.constants.**air**

define the "air" composition in PyChemkin with a fixed mixture "recipe". A "recipe" is a list of tuples of ("species symbol", fraction) to define a gas mixture in PyChemkin. This class uses the lower case symbols for oxygen and nitrogen.

Overview

Static methods

X
Y

Import detail

```
from ansys.chemkin.constants import air
```

Method detail

static air.**X**() → list[tuple[str, float]]

static air.**Y**() → list[tuple[str, float]]

14.10.2 Description

Constants used by Chemkin utilities and models.

14.10.3 Module detail

constants.**water_heat_vaporization**(*temperature: float*) → float

Get the heat if vporization of water at the given temperature [K]

Parameters

temperature: double
water temperature [K]

Returns

enthalpy: double
enthalpy of vaporization of water at the given temperature [erg/g-water]

constants.**boltzmann = 1.3806504e-16**

constants.**avogadro = 6.02214179e+23**

constants.**Patm = 1013250.0**

constants.**Ptorrs = 1333.2236842105262**

constants.**ergs_per_joule = 10000000.0**

constants.**joules_per_calorie = 4.184**

constants.**ergs_per_calorie = 41840000.0**

constants.**ergs_per_eV = 1.602176487e-12**

constants.**eV_per_K = 11604.505289680863**

constants.**RGas = 83144724.71220216**

constants.**RGas_Cal = 1.9872066135803572**

14.11 The ansys.chemkin.realgaseos library

14.11.1 Summary

Functions

check_realgas_status	Check whether the real-gas cubic EOS is active
set_current_pressure	Set gas mixture pressure for real-gas EOS calculations

14.11.2 Description

Real-gas cubic EOS model.

14.11.3 Module detail

realgaseos.**check_realgas_status**(*chem_index: int*) → bool

Check whether the real-gas cubic EOS is active

Parameters

chem_index: integer
chemistry set index associated with the Chemistry Set

Returns

status: boolean
the activation status of the Chemkin real-gas model

realgaseos.**set_current_pressure**(*chem_index: int*, *pressure: float*) → int

Set gas mixture pressure for real-gas EOS calculations

Parameters

chem_index: integer
chemistry set index associated with the Chemistry Set

pressure: double
gas pressure [dynes/cm2]

Returns

iErr: inetger
error code

14.12 The ansys.chemkin.reactormodel library

14.12.1 Summary

Classes

Keyword	A Chemkin keyword
BooleanKeyword	Chemkin boolean keyword
IntegerKeyword	A Chemkin integer keyword
RealKeyword	A Chemkin real keyword
StringKeyword	A Chemkin string keyword
Profile	Generic Chemkin profile keyword class
ReactorModel	A generic Chemkin reactor model framework

Keyword

class ansys.chemkin.reactormodel.**Keyword**(*phrase: str*, *value: int | float | bool | str*, *data_type: str*)

A Chemkin keyword

Overview

Methods

show	display the Chemkin keyword and its parameter value
resetvalue	Reset the parameter value of an existing keyword
parametertype	get parameter type of the keyword
getvalue_as_string	Create the keyword input line for Chemkin applications

Properties

value	Get parameter value of the keyword
keyphrase	Get the phrase of the keyword
keyprefix	Get the status of the keyword

Attributes

gasspecieskeywords
flowratekeywords
profilekeywords
fourspaces
noFullKeyword

Static methods

setfullkeywords	All keywords and their parameters must be specified by using the setkeyword method

Import detail

```
from ansys.chemkin.reactormodel import Keyword
```

Property detail

property Keyword.**value: int | float | bool | str**

Get parameter value of the keyword

Returns

parameter value: {integer, floating, string, or boolean}

property Keyword.**keyphrase: str**

Get the phrase of the keyword

Returns

keyword phrase: string

property Keyword.**keyprefix: bool**

Get the status of the keyword

Returns

status: boolean
keyword is ON/OFF

Attribute detail

Keyword.**gasspecieskeywords = ['REAC', 'XEST', 'FUEL', 'OXID']**

Keyword.**flowratekeywords = ['FLRT', 'VDOT', 'VEL', 'SCCM']**

Keyword.**profilekeywords = ['TPRO', 'PPRO', 'VPRO', 'QPRO', 'AINT', 'AEXT', 'DPRO', 'FPRO', 'SCCMPRO', 'VDOTPRO', 'VELPRO',...**

Keyword.**fourspaces = ' '**

Keyword.**noFullKeyword = True**

Method detail

static Keyword.**setfullkeywords**(*mode: bool*)

All keywords and their parameters must be specified by using the setkeyword method and will be passed to the reactor model for further processing

Parameters

mode: boolean
turn the full keyword mode ON/OFF

Keyword.**show**()
display the Chemkin keyword and its parameter value

Keyword.**resetvalue**(*value: int | float | bool | str*)
Reset the parameter value of an existing keyword

Parameters

value: indicated by the data_type, {int, float, string, bool}
keyword parameter

Keyword.**parametertype**() → type
get parameter type of the keyword

Returns

parameter_data_type: string, {'int', 'float', 'string', 'bool'}
parameter data type

Keyword.**getvalue_as_string**() → tuple[int, str]
Create the keyword input line for Chemkin applications

Returns

linelength: integer
line length

line: string
keyword value

BooleanKeyword

class ansys.chemkin.reactormodel.**BooleanKeyword**(*phrase: str*)
Bases: Keyword

Chemkin boolean keyword

Import detail

```
from ansys.chemkin.reactormodel import BooleanKeyword
```

IntegerKeyword

class ansys.chemkin.reactormodel.**IntegerKeyword**(*phrase: str*, *value: int = 0*)
Bases: Keyword

A Chemkin integer keyword

Import detail

```
from ansys.chemkin.reactormodel import IntegerKeyword
```

RealKeyword

class ansys.chemkin.reactormodel.**RealKeyword**(*phrase: str*, *value: float = 0.0*)

Bases: Keyword

A Chemkin real keyword

Import detail

```
from ansys.chemkin.reactormodel import RealKeyword
```

StringKeyword

class ansys.chemkin.reactormodel.**StringKeyword**(*phrase: str*, *value: str = ''*)

Bases: Keyword

A Chemkin string keyword

Import detail

```
from ansys.chemkin.reactormodel import StringKeyword
```

Profile

class ansys.chemkin.reactormodel.**Profile**(*key: str*, *x: numpy.typing.NDArray[numpy.double]*, *y: numpy.typing.NDArray[numpy.double]*)

Generic Chemkin profile keyword class

Overview

Methods

show	Show the profile data
resetprofile	Reset the profile data
getprofile_as_string_list	Create the keyword input lines as a list for Chemkin applications

Properties

size	Get number of data points in the profile
status	Get the validity of the profile object
pos	Get position values of profiles data
value	Get variable values of profile data
profilekey	Get profile keyword

Import detail

```
from ansys.chemkin.reactormodel import Profile
```

Property detail

property Profile.**size: int**

Get number of data points in the profile

Returns

size: integer
number of profile data points

property Profile.**status: int**

Get the validity of the profile object

Returns

status: integer
profile status

property Profile.**pos: numpy.typing.NDArray[numpy.double]**

Get position values of profiles data

Returns

pos: 1-D double array
position [sec, cm]

property Profile.**value: numpy.typing.NDArray[numpy.double]**

Get variable values of profile data

Returns

val: 1-D double array
variable value

property Profile.**profilekey: str**

Get profile keyword

Returns

profilekeyword: string
keyword associated with the profile data

Method detail

Profile.**show**()

Show the profile data

Profile.**resetprofile**(*size: int*, *x: numpy.typing.NDArray[numpy.double]*, *y: numpy.typing.NDArray[numpy.double]*)

Reset the profile data

Parameters

size: integer
number of points of the new profile data

x: 1-D double array
position of the new profile data points

y:1-D double array
variable value of the new profile data

Profile.**getprofile_as_string_list**() → tuple[int, list[str]]

Create the keyword input lines as a list for Chemkin applications

Returns

size: integer
number of profile lines

line: list of strings
list of profile related keywords

ReactorModel

class ansys.chemkin.reactormodel.**ReactorModel**(*reactor_condition:* ansys.chemkin.inlet.Stream, *label: str*)

A generic Chemkin reactor model framework

Overview

Methods

usefullkeywords	Specify all necessary keywords explicitly
setkeyword	Set a Chemkin keyword and its parameter
removekeyword	Remove an existing Chemkin keyword and its parameter
showkeywordinputlines	list all currently-defined keywords and their parameters line by line
createkeywordinputlines	Create keyword input lines for Chemkin applications
showkeywordinputlines_with	list all currently-defined keywords, their parameters, and an
createkeywordinputlines_wi	Create keyword input lines for Chemkin applications
setprofile	Set a Chemkin profile and its parameter
createprofileinputlines	Create profile keyword input lines for Chemkin applications
createspeciesinputlines	Create keyword input lines for initial/estimated species mole fraction inside the batch reactor
createspeciesinputlineswit	Create keyword input lines for initial/estimated species mole fraction inside the batch reactor
chemID	Get chemistry set index
list_composition	List the gas mixture composition inside the reactor
setsensitivityanalysis	Switch ON/OFF A-factor sensitivity analysis
setROPanalysis	Switch ON/OFF the ROP (Rate Of Production) analysis
userealgasEOS	Set the option to turn ON/OFF the real gas model
setrealgasmixingmodel	Set the real gas mixing rule/model
setrunstatus	Set the simulation run status
getrunstatus	Get the reactor model simuation status
run	Generic Chemkin run reactor model method
setsolutionspeciesfracmode	Set the type of species fractions in the solution
getrawsolutionstatus	Get the status of the post-process
getmixturesolutionstatus	Get the status of the post-process
get_solution_size	Get the number of reactors and the number of solution points
getnumbersolutionpoints	Get the number of solution points per reactor
parsespeciessolutiondata	Parse the species fraction solution data that are stored in a 2D array (number_species x numb_solution)
process_solution	Post-process solution to extract the raw solution variable data

Properties

temperature	Get reactor initial temperature
pressure	Get reactor pressure
massfraction	Get the initial/guessed/estimate gas species mass fractions inside the reactor
molefraction	Get the initial/guessed/estimate gas species mole fractions inside the reactor
concentration	Get the initial/guessed/estimate gas species molar concentrations inside the reactor
gasratemultiplier	Get the value of the gas-phase reaction rate multiplier
STD_Output	Get text output status
XML_Output	Get XML solution output status
realgas	Get the real gas EOS status

Attributes

label
runstatus

Import detail

```
from ansys.chemkin.reactormodel import ReactorModel
```

Property detail

property ReactorModel.**temperature: float**

Get reactor initial temperature

Returns

temperature: double
reactor temperature [K]

property ReactorModel.**pressure: float**

Get reactor pressure

Returns

pressure: double
reactor pressure [dynes/cm2]

property ReactorModel.**massfraction: float**

Get the initial/guessed/estimate gas species mass fractions inside the reactor

Returns

reactormixture: 1-D double array
mixture mass fraction [-]

property ReactorModel.**molefraction: numpy.typing.NDArray[numpy.double]**

Get the initial/guessed/estimate gas species mole fractions inside the reactor

Returns

X: 1-D double array
mixture mole fraction

property ReactorModel.**concentration: numpy.typing.NDArray[numpy.double]**

Get the initial/guessed/estimate gas species molar concentrations inside the reactor

Returns

concentration: 1-D double array
mixture molar concentration [mole/cm3]

property ReactorModel.**gasratemultiplier: float**

Get the value of the gas-phase reaction rate multiplier

Returns

rate_factor: double
gas-phase reaction rate multiplier

property ReactorModel.**STD_Output: bool**

Get text output status

Returns

status: boolean
text output ON=True/OFF=False

property ReactorModel.**XML_Output: bool**

Get XML solution output status

Returns

status: boolean
XML solution output ON=True/OFF=False

property ReactorModel.**realgas: bool**

Get the real gas EOS status

Returns

status: boolean
status of the real-gas EOS model True: real gas EOS is turned ON

Attribute detail

ReactorModel.**label**

ReactorModel.**runstatus = -100**

Method detail

ReactorModel.**usefullkeywords**(*mode: bool*)

Specify all necessary keywords explicitly

Parameters

mode: boolean, default = False
turn full keyword mode ON/OFF

ReactorModel.**setkeyword**(*key: str*, *value: bool | int | float | str*)

Set a Chemkin keyword and its parameter

Parameters

key: string
Chemkin keyword phrase

value: integer, double, string, or boolean depending on the keyword
value associated with the keyword phrase

ReactorModel.**removekeyword**(*key: str*)

Remove an existing Chemkin keyword and its parameter

Parameters

key: string
Chemkin keyword phrase

ReactorModel.**showkeywordinputlines**()

list all currently-defined keywords and their parameters line by line

ReactorModel.**createkeywordinputlines**() → tuple[int, int]

Create keyword input lines for Chemkin applications one keyword per line: <keyword> <parameter>

Returns

Error code: integer number of lines: integer

number of keyword lines to be added to the inputs

ReactorModel.**showkeywordinputlines_with_tag**(*tag: str = ''*)

list all currently-defined keywords, their parameters, and an extra tag string line by line

Parameters

tag: string
additional tag for the keywords, for example, the reactor index

ReactorModel.**createkeywordinputlines_with_tag**(*tag: str = ''*) → tuple[int, int]

Create keyword input lines for Chemkin applications one keyword per line: <keyword> <parameter> <tag>

Parameters

tag: string
additional tag for the keywords, for example, the reactor index

Returns

Error code: integer number of lines: integer

number of keyword lines to be added to the inputs

ReactorModel.**setprofile**(*key: str*, *x: numpy.typing.NDArray[numpy.double]*, *y: numpy.typing.NDArray[numpy.double]*) → int

Set a Chemkin profile and its parameter

Parameters

key: string
Chemkin profile keyword phrase

x: 1-D double array
position values of the profile data

y: 1-D double array
variable values of the profile data

Returns

Error code: integer

ReactorModel.**createprofileinputlines**() → tuple[int, int, list[str]]

Create profile keyword input lines for Chemkin applications

one keyword per line: <profile keyword> <position> <value>

Returns

Error code: integer numblines: integer

total number of profile keyword lines

keyword_lines: list of string lists, [[string, . . .], [string, . . .], . . .]
string list containing lists of profile keywords

ReactorModel.**createspeciesinputlines**(*solvertype: int*, *threshold: float = 1e-12*, *molefrac: numpy.typing.NDArray[numpy.double] = None*) → tuple[int, list[str]]

Create keyword input lines for initial/estimated species mole fraction inside the batch reactor

Parameters

solvertype: integer
solver type of the reactor model

threshold: double
minimum species mole fraction value to be included in the species keyword

molefrac: 1-D double array
species composition in mole fractions

Returns

numb_lines: integer
Number of keyword lines

lines: list of strings
list of keyword line strings

ReactorModel.**createspeciesinputlineswithaddon**(*key: str = 'XEST'*, *threshold: float = 1e-12*, *molefrac: numpy.typing.NDArray[numpy.double] = None*, *addon: str = ''*) → tuple[int, list[str]]

Create keyword input lines for initial/estimated species mole fraction inside the batch reactor

Parameters

key: string
Chemkin reactor keyword for species value

threshold: douoble
minimum species mole fraction value to be included in the species keyword

molefrac: 1-D double array
species composition in mole fractions

addon: string
add-on string to the species input, usually the reactor/zone number

Returns

numb_lines: integer
Number of keyword lines

lines: list of keyword line strings

ReactorModel.**chemID**() → int
Get chemistry set index

Returns

chemID: integer
chemistry set index

ReactorModel.**list_composition**(*mode: str*, *option: str = ' '*, *bound: float = 0.0*)
List the gas mixture composition inside the reactor

Parameters

mode: string, {'mass', 'mole'}, default = 'mole'
flag specifies the fractions returned are 'mass' or 'mole' fractions

option: string, {'all', ' '}, default = ' '
flag specifies to list 'all' species or just the species with non-zero fraction

bound: double, default = 0.0
minimum fraction value for the species to be printed

ReactorModel.**setsensitivityanalysis**(*mode: bool = True*, *absolute_tolerance: float | None = None*, *relative_tolerance: float | None = None*, *temperature_threshold: float | None = None*, *species_threshold: float | None = None*)
Switch ON/OFF A-factor sensitivity analysis

Parameters

mode: boolean
turn A-factor sensitivity ON/OFF

absolute_tolerance: double
absolute tolerance of the sensitivity parameters

relative_tolerance: double
relative tolerance of the sensitivity parameters

temperature_threshold: double
threshold normalized temperature sensitivity parameter value to print out to the text output file

species_threshold: double
threshold normalized species sensitivity parameter value to print out to the text output file

`ReactorModel.setROPanalysis(mode=True, threshold=None)`
Switch ON/OFF the ROP (Rate Of Production) analysis

Parameters

mode: boolean, default = False
turn ROP ON/OFF

threshold: double
threshold ROP value to print out to the text output file

`ReactorModel.userealgasEOS(mode: bool)`
Set the option to turn ON/OFF the real gas model for cubic EOS enabled gas-phase mechanism

Parameters

mode: boolean
turn the Chemkin real-gas cubic EOS model ON/OFF

`ReactorModel.setrealgasmixingmodel(model: int)`
Set the real gas mixing rule/model for cubic EOS enabled gas-phase mechanism

Parameters

model: integer, {0, 1}
Chemkin real-gas mixing rule method 0 = Van der Waals 1 = pseudocritical

`ReactorModel.setrunstatus(code: int)`
Set the simulation run status

Parameters

run_status: integer
error code

`ReactorModel.getrunstatus(mode: str = 'silent')` → int
Get the reactor model simuation status

Parameters

mode: string {'verbose', 'silent'}, default = 'silent'
option for additional print information

Returns

run_status: integer
error code: 0=success; -100=not run; other=failed

ReactorModel.**run**() → int

Generic Chemkin run reactor model method to be overridden by child classes

Returns

error code: integer

ReactorModel.**setsolutionspeciesfracmode**(*mode: str = 'mass'*)

Set the type of species fractions in the solution

Parameters

mode: string {'mass', 'mole'}
species fraction type to be returned by the post-processor

ReactorModel.**getrawsolutionstatus**() → bool

Get the status of the post-process

Returns

status: boolean
True = raw solution is ready, False = raw solution is yet to be processed

ReactorModel.**getmixturesolutionstatus**() → bool

Get the status of the post-process

Returns

status: boolean
True = solution mixtures is ready, False = solution mixtures are yet to be processed

ReactorModel.**get_solution_size**() → tuple[int, int]

Get the number of reactors and the number of solution points

Returns

nreactor: integer
number of reactors

npoints: integer
number of solution points

ReactorModel.**getnumbersolutionpoints**() → int

Get the number of solution points per reactor

Returns

npoints: integer
number of solution points

ReactorModel.**parsespeciessolutiondata**(*frac: numpy.typing.NDArray[numpy.double]*)

Parse the species fraction solution data that are stored in a 2D array (number_species x numb_solution)

Parameters

frac: 2-D double array, dimension = [number_species, numb_solution]
species fractions of each solution point

ReactorModel.**process_solution**()
Post-process solution to extract the raw solution variable data to be overridden by child classes

14.12.2 Description

Chemkin general reactor model utilities.

14.13 The ansys.chemkin.chemkin_wrapper library

14.13.1 Summary

Attributes

status
this_date
msg
this_msg
status
chemkin

14.13.2 Description

Chemkin-CFD-API python interfaces.

14.13.3 Module detail

chemkin_wrapper.**status = 0**

chemkin_wrapper.**this_date**

chemkin_wrapper.**msg**

chemkin_wrapper.**this_msg = ''**

chemkin_wrapper.**status = 1**

chemkin_wrapper.**chemkin**

14.14 The ansys.chemkin.steadystatesolver library

14.14.1 Summary

Classes

SteadyStateSolver	Common steady-state solver controlling parameters

SteadyStateSolver

class ansys.chemkin.steadystatesolver.**SteadyStateSolver**

Common steady-state solver controlling parameters

Overview

Methods

set_max_pseudo_transient_ca	set the maximum number of call to the pseudo transient algorithm
set_max_timestep_iteration	set the maximum number of iterations per time step when performing the pseudo transient algorithm
set_max_search_iteration	set the maximum number of iterations per search when performing the steady-state search algorithm
set_initial_timesteps	set the number of pseudo time steps to be performed to establish a "better"
set_species_floor	set the minimum species fraction value allowed during steady-state solution search
set_temperature_ceiling	set the maximum temperature value allowed during steady-state solution search
set_species_reset_value	set the positive value to reset any negative species fraction in
set_max_pseudo_timestep_siz	set the maximum time step sizes allowed by the pseudo time stepping solution
set_min_pseudo_timestep_siz	set the minimum time step size allowed by the pseudo time stepping solution
set_pseudo_timestep_age	set the minimum number of time steps taken before allowing time step size increase
set_Jacobian_age	set the number of steady-state searches before re-evaluate the Jacobian matrix
set_pseudo_Jacobian_age	set the number of time steps taken before re-evaluate the Jacobian matrix
set_damping_option	turn ON (True) or OFF (False) the damping option of the steady-state solver
set_legacy_option	turn ON (True) or OFF (False) the legacy steady-state solver
set_print_level	set the level of information to be provided by the steady-state solver
set_pseudo_timestepping_par	Set the parameters for the pseudo time stepping process of the steady state solver.

Properties

steady_state_tolerances	Get tolerance for the steady-state search algorithm
time_stepping_tolerances	Get tolerance for the pseudo time stepping solution algorithm

Attributes

SSabsolute_tolerance
SSrelative_tolerance
SSmaxiteration
SSJacobianage
maxpseudotransient
numbinitialpseudosteps
maxTbound
speciesfloor
species_positive
use_legacy_technique
SSdamping
absolute_perturbation
relative_perturbation
TRabsolute_tolerance
TRrelative_tolerance
TRmaxiteration
timestepsizeage
TRminstepsize
TRmaxstepsize
TRupfactor
TRdownfactor
TRJacobianage
TRstride_fixT
TRnumbsteps_fixT
TRstride_ENRG
TRnumbsteps_ENRG
print_level
SSsolverkeywords

Import detail

```
from ansys.chemkin.steadystatesolver import SteadyStateSolver
```

Property detail

property SteadyStateSolver.**steady_state_tolerances: tuple[float, float]**

Get tolerance for the steady-state search algorithm

Returns

tuple, [absolute_tolerance, relative_tolerance]

absolute_tolerance: double
absolute tolerance

relative_tolerance: double
relative tolerance

property SteadyStateSolver.**time_stepping_tolerances: tuple[float, float]**

Get tolerance for the pseudo time stepping solution algorithm

Returns

tuple, [absolute_tolerance, relative_tolerance]

absolute_tolerance: double
absolute tolerance for time stepping algorithm

relative_tolerance: double
relative tolerance for time stepping algorithm

Attribute detail

SteadyStateSolver.**SSabsolute_tolerance = 1e-09**

SteadyStateSolver.**SSrelative_tolerance = 0.0001**

SteadyStateSolver.**SSmaxiteration = 100**

SteadyStateSolver.**SSJacobianage = 20**

SteadyStateSolver.**maxpseudotransient = 100**

SteadyStateSolver.**numbinitialpseudosteps = 0**

SteadyStateSolver.**maxTbound = 5000.0**

SteadyStateSolver.**speciesfloor = -1e-14**

SteadyStateSolver.**species_positive = 0.0**

SteadyStateSolver.**use_legacy_technique = False**

SteadyStateSolver.**SSdamping = 1**

SteadyStateSolver.**absolute_perturbation = 0.0**

SteadyStateSolver.**relative_perturbation = 0.0**

SteadyStateSolver.**TRabsolute_tolerance = 1e-09**

SteadyStateSolver.**TRrelative_tolerance = 0.0001**

SteadyStateSolver.**TRmaxiteration = 25**

SteadyStateSolver.**timestepsizeage = 25**

SteadyStateSolver.**TRminstepsize = 1e-10**

SteadyStateSolver.**TRmaxstepsize = 0.01**

SteadyStateSolver.**TRupfactor = 2.0**

SteadyStateSolver.**TRdownfactor = 2.2**

SteadyStateSolver.**TRJacobianage = 20**

SteadyStateSolver.**TRstride_fixT = 1e-06**

SteadyStateSolver.**TRnumbsteps_fixT = 100**

SteadyStateSolver.**TRstride_ENRG = 1e-06**

SteadyStateSolver.**TRnumbsteps_ENRG = 100**

SteadyStateSolver.**print_level = 1**

SteadyStateSolver.**SSsolverkeywords: dict[str, int | float | str | bool]**

Method detail

SteadyStateSolver.**set_max_pseudo_transient_call**(*maxtime: int*)

set the maximum number of call to the pseudo transient algorithm in an attempt to find the steady-state solution

Parameters

maxtime: integer
max number of pseudo transient calls/attempts

SteadyStateSolver.**set_max_timestep_iteration**(*maxiteration: int*)

set the maximum number of iterations per time step when performing the pseudo transient algorithm

Parameters

maxtime: integer
max number of iterations per pseudo time step

SteadyStateSolver.**set_max_search_iteration**(*maxiteration: int*)

set the maximum number of iterations per search when performing the steady-state search algorithm

Parameters

maxtime: integer
max number of iterations per steady-state search

SteadyStateSolver.**set_initial_timesteps**(*initsteps: int*)

set the number of pseudo time steps to be performed to establish a "better" set of guessed solution before start the actual steady-state solution search

Parameters

initsteps: integer
number of initial pseudo time steps

SteadyStateSolver.**set_species_floor**(*floor_value: float*)

set the minimum species fraction value allowed during steady-state solution search

Parameters

floor_value: double
minimum species fraction value

SteadyStateSolver.**set_temperature_ceiling**(*ceilingvalue: float*)

set the maximum temperature value allowed during steady-state solution search

Parameters

ceilingvalue: double
maximum temperature value

SteadyStateSolver.**set_species_reset_value**(*resetvalue: float*)

set the positive value to reset any negative species fraction in intermediate solutions during iterations

Parameters

resetvalue: double
positive value to reset negative species fraction

SteadyStateSolver.**set_max_pseudo_timestep_size**(*dtmax: float*)

set the maximum time step sizes allowed by the pseudo time stepping solution

Parameters

dtmax: double
maximum time step size allowed

SteadyStateSolver.**set_min_pseudo_timestep_size**(*dtmin: float*)

set the minimum time step size allowed by the pseudo time stepping solution

Parameters

dtmin: double
minimum time step size allowed

SteadyStateSolver.**set_pseudo_timestep_age**(*age: int*)

set the minimum number of time steps taken before allowing time step size increase

Parameters

age: integer
min age of the pseudo time step size

SteadyStateSolver.**set_Jacobian_age**(*age: int*)

set the number of steady-state searches before re-evaluate the Jacobian matrix

Parameters

age: integer
age of the steady-state Jacobian matrix

SteadyStateSolver.**set_pseudo_Jacobian_age**(*age: int*)

set the number of time steps taken before re-evaluate the Jacobian matrix

Parameters

age: integer
age of the pseudo time step Jacobian matrix

SteadyStateSolver.**set_damping_option**(*ON: bool*)

turn ON (True) or OFF (False) the damping option of the steady-state solver

Parameters

ON: boolean
turn On the damping option

SteadyStateSolver.**set_legacy_option**(*ON: bool*)

turn ON (True) or OFF (False) the legacy steady-state solver

Parameters

ON: boolean
turn On the legacy solver

SteadyStateSolver.**set_print_level**(*level: int*)

set the level of information to be provided by the steady-state solver to the text output

Parameters

level: integer, {0, 1, 2}
solver message details level (0 ~ 2)

SteadyStateSolver.**set_pseudo_timestepping_parameters**(*numb_steps: int = 100*, *step_size: float = 1e-06*, *stage: int = 1*)

Set the parameters for the pseudo time stepping process of the steady state solver.

Parameters

numb_step: integer, default = 100
the number of pseudo time steps to be taken during each time stepping process

step_size: double, default = 1.0e-6 [sec]
the initial time step size for each time stepping process

stage: integer, {1, 2}
the stage the time stepping process is in. 1 = fixed temperature stage 2 = solving energy equation

14.14.2 Description

Chemkin steady-state solver controlling parameters.

14.15 The ansys.chemkin.hybridreactornetwork library

14.15.1 Summary

Classes

ReactorNetwork	The hybrid reactor network allows internal recycling stream and reactor outflow splitting.

ReactorNetwork

class ansys.chemkin.hybridreactornetwork.**ReactorNetwork**(*chem:* ansys.chemkin.chemistry.Chemistry)

The hybrid reactor network allows internal recycling stream and reactor outflow splitting. The reactors are solved individually in terms. For network with complex internal connections, "Tearing points" can be manually defined and "tear stream" method is applied to solve the entire network iteratively.

Overview

Methods

get_reactor_label	Get the name/label of the reactor corresponding to the reactor index in the
add_reactor	Add a reactor to the network in order.
add_reactor_list	Add a list of reactors to the network in order.
show_reactors	Show the reactor labels in the network.
show_internal_outflow_connec	Show the ouflow connections to other reactors in the network.
show_internal_inflow_connect	Show the incoming flow connections from other reactors in the network.
add_outflow_connections	Add outflow connections to other reactors in the network. The connection is given
clear_connections	Clear the internal connection configurations.
remove_reactor	Remove the named reactor from the network.
set_reactor_outflow	Set up and verify the the outlet flow connections from this reactor
set_inflow_connections	Set up the sources of the internal network inlet stream to the reactor.
set_external_outlet	Add a new network external outlet to the reactor.
calculate_incoming_streams	Calculate the combined internal incoming streams from other reactors in the network.
set_internal_inlet	Create or update the merged inlet stream to the reactor from the rest of
create_internal_inlet	Create a new inlet stream that combines all incoming streams from the other
get_network_run_status	Get network run status
run	Solve the hybrid reactor network by solving the individual reactors in
get_reactor_stream	Get the solution Stream object of the given reactor name/label.
set_external_streams	Set up external outlet streams.
get_external_stream	Get the list of external outlet Stream objects.
run_without_tearstream	Run the individual reactors in the network one by one without using
run_with_tearstream	Run the individual reactors in the network one by one with
remove_tearpoint	Remove the tear point from the list.
add_tearingpoint	Add a new tear point to the list.
set_tear_tolerance	Set the relative tolerance to test the tear stream convergencce.
set_tear_iteration_limit	Set the maximum number of tear loop iterations.
check_iteration_count	Check the iteration count for over the set limit.
set_relaxation_factor	Set the relaxation factor when updating the tear stream
check_tearstream_convergence	Compare the last and the current reactor solution at the tear point.
update_tear_solution	Update the old/temporary tear point stream properties (with the used of a relaxation factor).

Properties

number_reactors	Get the number of reactors in the network
number_external_outlets	Get the number of external outlets from the network

Attributes

numb_reactors
last_reactor
numb_external_outlet
external_outlets
external_outlet_streams
reactor_map
reactor_objects
reactor_solutions
outflow_targets
outflow_altered
external_connections
inflow_sources
internal_inflow
internal_inflow_ready
numb_tearpoints
tearpoint
max_tearloop_count
tolerance
relaxation_factor
tear_converged
network_run_status

Import detail

```
from ansys.chemkin.hybridreactornetwork import ReactorNetwork
```

Property detail

property ReactorNetwork.**number_reactors: int**

Get the number of reactors in the network

Returns

numb_reactors: integer
number of reactors in the network

property ReactorNetwork.**number_external_outlets: int**

Get the number of external outlets from the network

Returns

numb_outlets: integer
number of external outlets from the network

Attribute detail

ReactorNetwork.**numb_reactors = 0**

ReactorNetwork.**last_reactor = 0**

ReactorNetwork.**numb_external_outlet = 0**

ReactorNetwork.**external_outlets: dict[int, int]**

ReactorNetwork.**external_outlet_streams: dict[int, *ansys.chemkin.inlet.Stream*]**

ReactorNetwork.**reactor_map: dict[str, int]**

ReactorNetwork.**reactor_objects: dict[int,**
ansys.chemkin.stirreactors.PSR.perfectlystirredreactor |
ansys.chemkin.flowreactors.PFR.PlugFlowReactor]

ReactorNetwork.**reactor_solutions: dict[int, *ansys.chemkin.inlet.Stream*]**

ReactorNetwork.**outflow_targets: dict[int, list[tuple[int, float]]]**

ReactorNetwork.**outflow_altered = True**

ReactorNetwork.**external_connections: dict[int, int]**

ReactorNetwork.**inflow_sources: dict[int, list[tuple[int, float]]]**

ReactorNetwork.**internal_inflow: dict[int, *ansys.chemkin.inlet.Stream*]**

ReactorNetwork.**internal_inflow_ready: dict[int, bool]**

ReactorNetwork.**numb_tearpoints = 0**

ReactorNetwork.**tearpoint: list[int] = []**

ReactorNetwork.**max_tearloop_count = 200**

ReactorNetwork.**tolerance = 1e-06**

ReactorNetwork.**relaxation_factor = 1.0**

ReactorNetwork.**tear_converged = False**

ReactorNetwork.**network_run_status = -100**

Method detail

ReactorNetwork.**get_reactor_label**(*reactor_index: int*) → str

Get the name/label of the reactor corresponding to the reactor index in the reactor network.

Parameters

reactor_index: integer
reactor index

Returns

name: string
reactor name/label

ReactorNetwork.**add_reactor**(*reactor: ansys.chemkin.stirreactors.PSR.perfectlystirredreactor | ansys.chemkin.flowreactors.PFR.PlugFlowReactor*)

Add a reactor to the network in order. Plan the order (sequence) of reactor addition carefully. The order of the reactors is somewhat important as it might affect the convergence rate of the network, especially with the presence of any "tearing stream".

Parameters

reactor: open reactor (PSR or PFR) object
the reactor object to be added to the network

ReactorNetwork.**add_reactor_list**(*reactor_list: list[ansys.chemkin.stirreactors.PSR.perfectlystirredreactor | ansys.chemkin.flowreactors.PFR.PlugFlowReactor]*)

Add a list of reactors to the network in order. Plan the order (sequence) of reactor addition carefully. The order of the reactors is somewhat important as it might affect the convergence rate of the network, especially with the presence of any "tearing stream".

Parameters

reactor: list of open reactor (PSR or PFR) object
the reactor objects to be added to the network

ReactorNetwork.**show_reactors**()

Show the reactor labels in the network.

ReactorNetwork.**show_internal_outflow_connections**()

Show the ouflow connections to other reactors in the network.

ReactorNetwork.**show_internal_inflow_connections**()

Show the incoming flow connections from other reactors in the network.

ReactorNetwork.**add_outflow_connections**(*source_label: str*, *outflow_split: list[tuple[str, float]]*)

Add outflow connections to other reactors in the network. The connection is given by a tuple consistinig of the target reactor name and the mass flow rate split fraction. The connection to the immediate downstream reactor (through flow) is optional.

Parameters

source_label: string
name/label of the source reactor

outflow_split: list of tuples of (target reactor name, split fraction)
outflow connections from the source reactor. target reactor name: string split fraction: double, <= 1.0

ReactorNetwork.**clear_connections**()

Clear the internal connection configurations.

ReactorNetwork.**remove_reactor**(*name: str*)

Remove the named reactor from the network.

Parameters

name: string
reactor name/label

ReactorNetwork.**set_reactor_outflow**()

Set up and verify the the outlet flow connections from this reactor to the target reactors in the network.

ReactorNetwork.**set_inflow_connections**()

Set up the sources of the internal network inlet stream to the reactor.

ReactorNetwork.**set_external_outlet**(*reactor_index: int*)

Add a new network external outlet to the reactor.

Parameters

reactor_index: integer
reactor index

ReactorNetwork.**calculate_incoming_streams**(*reactor_index: int*) → *ansys.chemkin.inlet.Stream* | None

Calculate the combined internal incoming streams from other reactors in the network.

Parameters

reactor_index: integer
index of the current (target) reactor

Returns

incoming_stream: Stream object
the total internal stream going into the current PSR

ReactorNetwork.**set_internal_inlet**(*reactor_index: int*) → int

Create or update the merged inlet stream to the reactor from the rest of the reactors in the network.

Parameters

reactor_index: integer
reactor index

Returns

status: integer
error code

ReactorNetwork.**create_internal_inlet**(*reactor_index: int*)

Create a new inlet stream that combines all incoming streams from the other reactor network to the current reactor.

Parameters

reactor_index: integer
index of the current reactor

ReactorNetwork.**get_network_run_status**() → int

Get network run status

Returns

status: integer
run status, 0=all reactor success; <-100=not run; other=failed

ReactorNetwork.**run**() → int

Solve the hybrid reactor network by solving the individual reactors in the sequence as they are added to the network. If there is any "tear stream" in the network, the solution process will be repeated till the properties of the "tear stream" are converged.

Returns

run status: integer

ReactorNetwork.**get_reactor_stream**(*reactor_name: str*) → *ansys.chemkin.inlet.Stream*

Get the solution Stream object of the given reactor name/label.

Parameters

reactor_name: string
reactor name

Returns

solution_stream: Stream object
solution of the reactor specified

ReactorNetwork.**set_external_streams**()

Set up external outlet streams.

ReactorNetwork.**get_external_stream**(*stream_index: int*) → list[*ansys.chemkin.inlet.Stream*]

Get the list of external outlet Stream objects.

Parameters

stream_index: integer
the index of the external outlet

Returns

external_stream: Stream object
external outlet stream properties

ReactorNetwork.**run_without_tearstream**() → int

Run the individual reactors in the network one by one without using tear stream iteration.

Returns

run_status: integer
error code

ReactorNetwork.**run_with_tearstream**() → int
Run the individual reactors in the network one by one with tear stream iteration.

Returns

run_status: integer
error code

ReactorNetwork.**remove_tearpoint**(*reactor_name: str*)
Remove the tear point from the list.

Parameters

reactor_name: string
reactor name/label

ReactorNetwork.**add_tearingpoint**(*reactor_name: str*)
Add a new tear point to the list.

Parameters

reactor_name: string
reactor name/label

ReactorNetwork.**set_tear_tolerance**(*tol: float = 1e-06*)
Set the relative tolerance to test the tear stream convergencce.

Parameters

tol: double, default = 1.0e-6
relative tolerance

ReactorNetwork.**set_tear_iteration_limit**(*max_count: int*)
Set the maximum number of tear loop iterations.

Parameters

max_count: integer
tear loop iteration limit

ReactorNetwork.**check_iteration_count**(*count: int*) → bool
Check the iteration count for over the set limit.

Parameters

count: integer
current tear loop iteration count

Returns

status: boolean
True=the current count is under the limit False=the current count is over the limit

`ReactorNetwork.`**`set_relaxation_factor`**`(`*relax: float*`)`

Set the relaxation factor when updating the tear stream properties from their values of the previous iteration step.

Parameters

relax: double
iteration relaxation factor

`ReactorNetwork.`**`check_tearstream_convergence`**`(`*streamA*, *streamB*`)` → tuple[bool, float]

Compare the last and the current reactor solution at the tear point.

Parameters

streamA: Stream object
reactor solution to be compared against

streamB: Stream object
reactor solution used for the comparison

Returns

converged: boolean
True=the two streams are close within the relative tolerance False=the differences of the two streams are greater than the relative tolerance

residual: double
the maximum relative difference between the properties of the streams

`ReactorNetwork.`**`update_tear_solution`**`(`*new_stream:* ansys.chemkin.inlet.Stream, *old_stream:* ansys.chemkin.inlet.Stream`)` → *ansys.chemkin.inlet.Stream*

Update the old/temporary tear point stream properties (with the used of a relaxation factor).

Parameters

new_stream: Stream object
tear point stream properties from the current iteration step

old_stream: Stream object
tear point stream properties from the last iteration step

Returns

updated_stream: Stream object
updated tear point stream properties

14.15.2 Description

Hybrid reactor network comprised of a mix of open reactors such as PSR and PFR.

PYTHON MODULE INDEX

a