The PYCHEMKIN User Manual

Jane Huang, Kimia Mavon, Weidong Xu, Zeyu Zhao

1 Introduction

`pychemkin` is a Python 3 chemical kinetics library. The core functionality is to compute species' reaction rates and concentrations in a system of elementary reactions. New extended functionality now allows reaction rates and concentrations to be computed as a function of time and to be visualized, in order to facilitate understanding of the timescales upon which the system operates and the production yields of the system.

1.1 Key chemical concepts and terminology

A system consisting of M elementary reactions involving N species has the general form

$$\sum_{i=1}^{N} \nu'_{ij} \mathcal{S}_i \longrightarrow \sum_{i=1}^{N} \nu''_{ij} \mathcal{S}_i, \qquad j = 1, \ldots, M. \tag{1}$$

for **irreversible reactions** (i.e., the reaction only proceeds in the forward direction) and

$$\sum_{i=1}^{N} \nu'_{ij} \mathcal{S}_i \rightleftharpoons \sum_{i=1}^{N} \nu''_{ij} \mathcal{S}_i, \qquad j = 1, \ldots, M. \tag{2}$$

for **reversible reactions** (i.e., the reaction can proceed in either the forward or backward directions).

S_i is the ith specie in the system, ν'_{ij} is its stoichiometric coefficient (dimensionless) on the reactants side of the jth reaction, and ν''_{ij} is its stoichiometric coefficient (dimensionless) on the product side for the jth reaction.

Each specie is characterized by a concentration x_i, in units of [mol/vol]. The **reaction rate** of each specie is the time rate of change of its concentration, $\frac{dx_i}{dt}$. The reaction rate is usually represented by the symbol f_i, such that

$$f_i = \sum_{j=1}^{M} (\nu''_{ij} - \nu'_{ij}) \omega_j = \sum_{j=1}^{M} \nu_{ij} \omega_j, \qquad i = 1, \ldots, N. \tag{3}$$

The **progress rate** of the jth reaction is given by

$$\omega_j = k_j^{(f)} \prod_{i=1}^{N} x_i^{\nu'_{ij}} - k_j^{(b)} \prod_{i=1}^{N} x_i^{\nu''_{ij}}, \qquad j = 1, \ldots, M. \tag{4}$$

The **forward reaction rate coefficient** $k_j^{(f)}$ is assumed to take one of three possible forms:

1. $k =$ constant
2. Arrhenius: $k = A \exp(-\frac{E}{RT})$, where A is the pre-factor, E is the activation energy, R is the universal gas constant, and T is the temperature.
3. Modified Arrhenius: $k = AT^b \exp(-\frac{E}{RT})$, where A is the pre-factor, E is the activation energy, R is the universal gas constant, T is the temperature, and b is the temperature scaling parameter.

The forward and backward reaction rate coefficients are related by

$$k_j^{(b)} = \frac{k_j^{(f)}}{k_j^e}, \qquad j = 1, \ldots, M, \tag{5}$$

where the **equilibrium coefficient** k_j^e is given by

$$k_j^e = \left(\frac{p_0}{RT}\right)^{\gamma_j} \exp\left(\frac{\Delta S_j}{R} - \frac{\Delta H_j}{RT}\right), \qquad j = 1, \ldots, M. \tag{6}$$

The pressure p_0 is fixed at 10^5 Pa in this package. $\gamma_j = \sum_{i=1}^{N} \nu_{ij}$. The **entropy change** of reaction j is

$$\Delta S_j = \sum_{i=1}^{N} \nu_{ij} S_i, \qquad j = 1, \ldots, M, \tag{7}$$

where S_i is the entropy of specie i. Likewise, the **enthalpy change** of reaction j is

$$\Delta H_j = \sum_{i=1}^{N} \nu_{ij} H_i, \qquad j = 1, \ldots, M. \tag{8}$$

An irreversible reaction can be thought of as the limiting case where k_j^e approaches ∞, in which case the backwards reaction rate coefficient $k_j^{(b)}$ approaches 0. The progress rate expression then simplifies to

$$\omega_j = k_j^{(f)} \prod_{i=1}^{N} x_i^{\nu'_{ij}}, \qquad j = 1, \ldots, M. \tag{9}$$

Traditionally for combustion chemistry, the entropy and enthalpy of each species are approximated by polynomial fits to numerical calculations from Gordon and McBride's 1963 report, *The Thermodynamic Properties of Chemical Substances to 6000 K, NASA Report SP-3001*:

$$\frac{H_i}{RT} = a_{i1} + \frac{1}{2}a_{i2}T + \frac{1}{3}a_{i3}T^2 + \frac{1}{4}a_{i4}T^3 + \frac{1}{5}a_{i5}T^4 + \frac{a_{i6}}{T} \tag{10}$$

and

$$\frac{S_i}{R} = a_{i1}\ln(T) + a_{i2}T + \frac{1}{2}a_{i3}T^2 + \frac{1}{3}a_{i4}T^3 + \frac{1}{4}a_{i5}T^4 + a_{i7}. \tag{11}$$

These are known as the **NASA polynomials**. For each specie, there are two sets of coefficients a_i, the first of which is applicable at low temperatures and a second that is applicable at high temperatures.

1.2 Features

1.2.1 Basic

The package can solve for the reaction rates of a system of elementary reactions. The number of reactions and species is arbitrary. For each system of reactions, the user supplies the species participating in the reactions, the chemical equations, the stoichiometric coefficients for the reactants and products, and the rate coefficient parameters (e.g., E and A for Arrhenius rates). For a given system, the user can then specify a temperature and a vector of species concentrations in order to return the reaction rates in the form of a NumPy array. `pychemkin` stores the NASA polynomials for computing thermodynamic quantities (taken from http://burcat.technion.ac.il/dir/) in an sqlite database and retrieves values for species requested by the user.

1.2.2 Advanced

The updated version of `pychemkin` can now compute the time-evolution of reaction rates and concentration rates and visualize the results. See Section 4 for more details.

2 Installation

2.1 Where to find and download the code

2.1.1 Using pip

`pychemkin` v. 1.1.0 is hosted on PyPi at https://pypi.python.org/pypi/pychemkin. To download and install the package, simply type `pip install pychemkin` into your terminal.

2.1.2 Installing from source and contributing to the source code

While `pip` is the preferred method of installation, you may prefer to obtain the most up-to-date version from GitHub, especially if you are interested in developing the code further. The most up-to-date version is hosted at https://github.com/cs207group4/cs207-FinalProject. If you have a GitHub account, you can simply open your terminal and type `git clone git@github.com:cs207group4/cs207-FinalProject.git`. Otherwise, you can download the package by clicking on the green button in the upper right corner of the page that says "Clone or download," then click "Download ZIP" to download the entire repository as a ZIP file. Once you download the contents of the repository from GitHub, enter the directory and type `python setup.py install`.

In order to run the test suite, you need to have pytest v. 3.00+ and pytest-cov v. 2.5+ installed. When you're in the top level of the package directory, type `pytest` into the terminal. The results of the test code will be printed out to the terminal.

2.2 Third-party dependencies

This package has dependencies that usually come standard with the Anaconda distribution, but will otherwise automatically be installed for you if you use pip, along with their individual dependencies (e.g., pandas requires PyTables to process HDF5 output). Other than the packages lised above for running the test-suite, the explicitly required packages outside the Python Standard Library are as follows:

- NumPy v. 1.13+
- SciPy v. 1.0.0+
- Matplotlib v. 2.0.2+
- pandas v. 0.21.0 +

3 Basic Usage and Examples

Note: All output files described in the following section can be found in the 'examples' folder in the GitHub repository. As an example of basic code usage, we first consider the following system of elementary, irreversible reactions:

1. $H + O_2 \xrightarrow{k_1} OH + O$
2. $H_2 + O \xrightarrow{k_2} OH + H$
3. $H_2 + OH \xrightarrow{k_3} H_2O + H$

Reaction 1 has an Arrhenius rate coefficient with $A = 3.52 \times 10^{10}$ and $E = 7.14 \times 10^4$. Reaction 2 has a modified Arrhenius rate coefficient with $A = 5.06 \times 10^{-2}$, $b = 2.7$, and $E = 2.63 \times 10^4$. Finally, reaction 3 has a constant rate coefficient of $k_3 = 10^3$.

3.1 User-required input

In an xml input file, the user provides the species participating in the reactions, the chemical equations, the stoichiometric coefficients, and the rate coefficient parameters. See `rxns.xml` in the `tests/test_xml` folder for an example of how to format the input file.

The xml file will be processed and stored in a `chemkin` object as follows:

```
>>>from pychemkin import chemkin
>>>rxn_system = chemkin('rxns.xml')
Finished␣reading␣xml␣input␣file
```

We can print out information about the reaction system as follows:

```
>>>print(rxn_system)
chemical equations:
[
H + O2 =] OH + O
H2 + O =] OH + H
H2 + OH =] H2O + H
]
species: ['H', 'O', 'OH', 'H2', 'H2O', 'O2']
nu_react:
[[1 0 0]
 [0 1 0]
 [0 0 1]
 [0 1 1]
 [0 0 0]
 [1 0 0]]
nu_prod:
[[0 1 1]
 [1 0 0]
 [1 1 0]
 [0 0 0]
 [0 0 1]
 [0 0 0]]
reaction coefficients:
[
Arrhenius Reaction Coeffs: {'A': 35200000000.0, 'E': 71400.0, 'R': 8.314}
modifiedArrhenius Reaction Coeffs: {'A': 0.0506, 'b': 2.7, 'E': 26300.0, 'R'
    ↪ : 8.314}
```

```
Constant Reaction Coeffs: {'k': 1000.0, 'R': 8.314}
]
reaction types: ['Elementary', 'Elementary', 'Elementary']
reversible: [False False False]
```

3.2 Computing reaction rates

Given the reaction data from a user-provided input file, the reaction rates can be computed for an arbitrary temperature and set of species concentrations.

```
>>> import numpy as np
>>> T = 1000 #K
>>> x = np.array([1,1,1,1,1,1])
>>> rxn_system.reaction_rate_T(x,T)
array([ -6.28889929e+06, 6.28989929e+06, 6.82761528e+06,
          -2.70357993e+05, 1.00000000e+03, -6.55925729e+06])
```

3.2.1 Obtaining intermediate calculations

Rate coefficients and progress rates are calculated in the course of computing the reaction rates. While these methods do not have to be called by the user to obtain the reaction rates, they are accessible if the user wishes to obtain these values.

To obtain the reaction rate coefficients, the user can call

```
>>> x, kf, kb = rxn_system._init_progress_rate(x,T)
>>> print(kf) #forward reaction rate coefficients
[ 6.55925729e+06 2.69357993e+05 1.00000000e+03]
>>> print(kb) #backward reaction rate coefficients (should be empty array
    ↪ since all reactions are irreversible)
[]
```

The progress rate values w_i can then be computed in the following manner:

```
>>> T = 1000 #K
>>> x = np.array([1,1,1,1,1,1])
>>> rxn_system.progress_rate(x,T)
array([ 6.55925729e+06, 2.69357993e+05, 1.00000000e+03])
```

3.3 Computing the time evolution of reaction rates and species concentrations

The reaction rate function outputs the right-hand side of the equation

$$\frac{dx_i}{dt} = f(x\_i, t) \tag{12}$$

You can solve for $x_i(t)$ with the following function calls, for which we use the file from test_xml/rxns_reversible.xml for illustrative purposes.

```
>>> import numpy as np
>>> from pychemkin import chemkin, ChemSolver
>>> T = 1000 #temperature in K
>>> x_init = np.ones(8) #initial concentration
>>> t_max = 5.e-13 # integration end time in seconds
>>> dt = 1.e-16 # step size in seconds
>>> cs = ChemSolver(chemkin('../tests/test_xml/rxns_reversible.xml'))
>>> cs.solve(x_init, T, t_max, dt)
```

The `solve` function is wrapped around `scipy.integrate.ode` and can therefore take as an argument any of the optional parameters in `scipy.integrate.ode.setintegrator`. For example, while the default integrator is set to `lsoda`, one can instead specify the `vode` integrator using backward differentiation formulas and a maximum number of steps of 500 per call to the integrator with:

```
>>> cs.solve(x_init, T, t_max, dt, algorithm = 'vode', method = 'bdf',
    ↪ nsteps = 500).
```

See Section 4 for a more detailed discussion of the implementation.

The results of the ODE solver can be accessed in several ways.
`time_array, conc_array, rxnrate_array = cs.get_results()` allows the arrays for the time steps and corresponding concentrations and reaction rates to be accessed most directly. The user also has the option of viewing and manipulating the data in the form of a `pandas` dataframe: `df = cs.to_df()`.

If the calculations are lengthy, we recommend saving them in either csv or HDF5 format with the following commands:
`df.save_results('simulationdata.csv')` or `df.save_results('simulationdata.h5')`.
To quickly explore parameter space with different temperatures and starting concentration values, the user can do the following:

```
>>> T = [500,1000,1500] #temperature in K
>>> #initial concentration (each row corresponds to a different
    ↪ concentration vector)
>>> x_init = np.array([[1 , 1, 1, 1, 1, 1, 1, 1],
                       [1, 1, 1, 1, 0, 0, 0, 0]])
>>> t_max = 5.e-13 # integration end time in seconds
>>> dt = 1.e-16 # step size in seconds
>>> cs = ChemSolver(chemkin('../tests/test_xml/rxns_reversible.xml'))
>>> cs.grid_solve(x_init, T, t_max, dt)
```

To get the parameters explored for the grid and a dictionary of the grid ouput, you can call `initial_conditions, results = cs.get_grid_result()` The results can be saved to csv or hdf5 with the following command:

```
>>> cs.save_grid_results('gridoutput', filetype = 'csv')
```

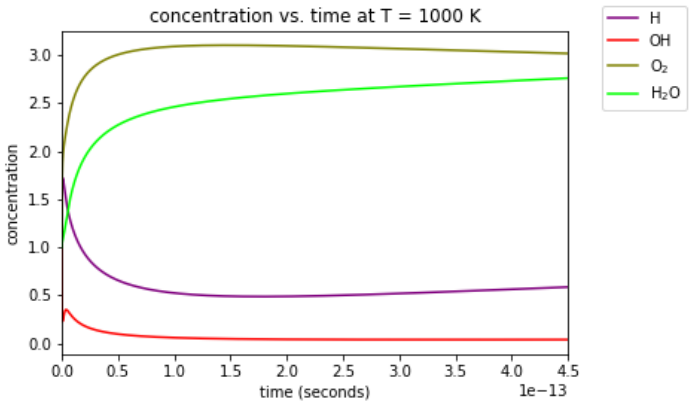
For the example given above, this will result in six csv files being saved, with 'gridoutput_T0_x0.csv' corresponding to the results for $T = 500$ and an initial concentration vector of [1,1,1,1,1,1,1,1], 'gridoutput_T1_x1.csv' corresponding to $T = 1000$ and an initial concentration vector of [1,1,1,1,0,0,0,0], etc.

3.4 Visualizing the time evolution of the reaction system

The evolution of concentration and reaction rates with time as computed by the ChemSolver module can be plotted with the ChemViz module. Going back to the first example given for the solver, we can use the following commands to plot the concentration rates of H, OH, O_2, and H_2O: from 0 to 4.5×10^{13} s. (By default, all species and the entire simulation timerange are plotted). The output image is shown in Figure 1.

```
>>> import numpy as np
>>> from pychemkin import chemkin, ChemSolver
>>> T = 1000 #temperature in K
>>> x_init = np.ones(8) #initial concentration
>>> t_max = 5.e-13 # integration end time in seconds
>>> dt = 1.e-16 # step size in seconds
>>> cs = ChemSolver(chemkin('../tests/test_xml/rxns_reversible.xml'))
>>> cs.solve(x_init, T, t_max, dt)
>>> cv.plot_time_series('concentration',tmin=0, tmax = 4.5e-13, species = ['
    ↪ H', 'OH', 'O2', 'H2O'],outputfile = 'modeldocfig1.png')
```

Figure 1: Example output for concentration as a function of time for user-selected species from a given simulation.

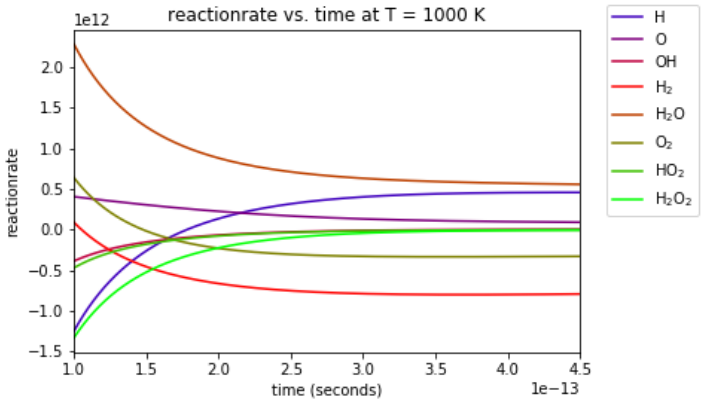


Similarly, the reaction rates can be plotted with the command

```
>>> cv.plot_time_series('reactionrate',tmin=1.e-13, tmax = 4.5e-13,
    ↪ outputfile = 'modeldocfig2.png')
```

The output is shown in Figure 2.

Figure 2: Example output for reaction rates as a function of time for user-selected species from a given simulation.



As a complementary approach for visualizing how a reaction system evolves, the species network can be diagrammed with

```
>>> cv.plot_network([0, 1.5e-13, 3e-13], figsize = (8, 15),outputfile = '
    ↪ modeldocfig3.png')
```

The output is shown in Figure 3

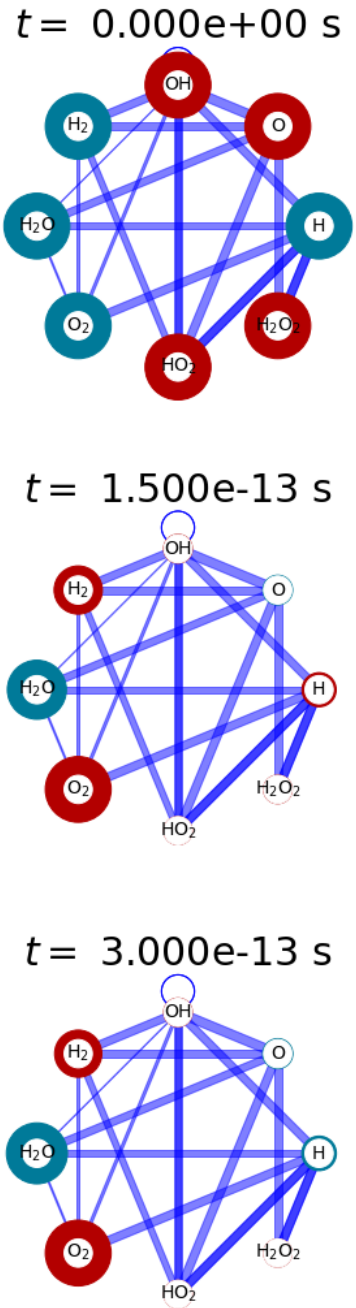
Each specie is plotted with a bubble, the size of which depends linearly with its concentration. Red bubbles indicate that the species concentration is decreasing at that time and teal bubbles indicate that the concentration is increasing. Two species are connected by a blue line if they react with one another (including reverse reactions). If the specie reacts with itself, a loop is drawn. The width of the connecting line varies logarithmically with the reaction rate coefficients (k_f for the forward reactions and k_b for the backward reactions). If two reactants are associated with multiple reaction coefficients (due to formation of different products), the width of the connecting line is based on the largest coefficient, and the line is shaded darker to indicate the existence of multiple pathways involving the same reactants.

The chemical equations, reaction rate coefficients, time series plots, and network plots can be combined in a summary HTML report with the following command: `cv.html_report(file_name)`. For the HTML report that would be generated with the reaction system demonstrated in this section, see 'examples/modeldocexample.html.'

Time series of combinations of initial concentrations and temperatures can be plotted easily as well via the combination of ChemSolver and ChemVis:

```
>>> T = [1000,1500] #temperature in K
```

Figure 3: Example output for species network at three different timepoints

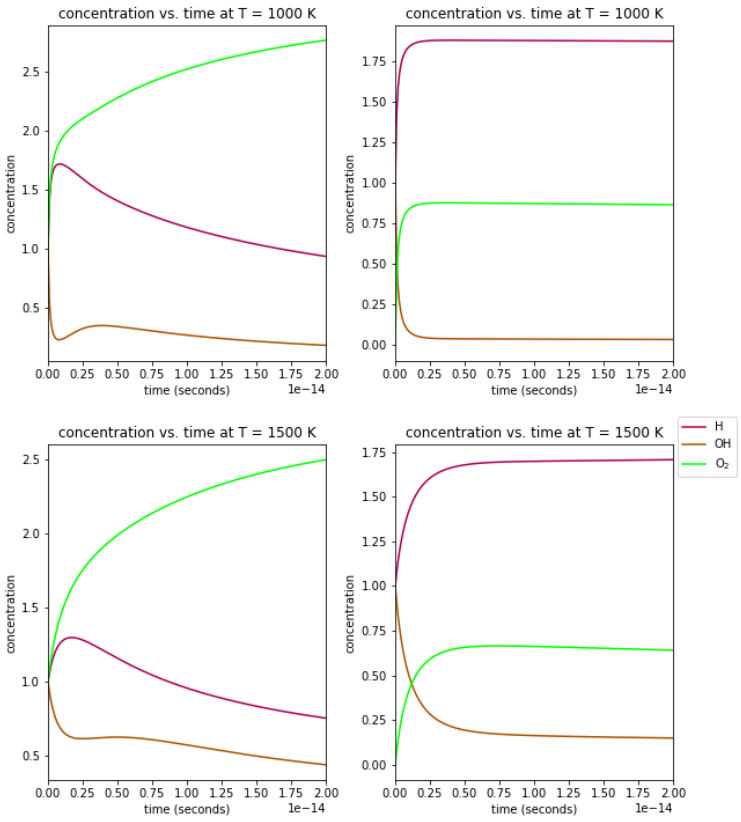



```
>>> #initial concentration (each row corresponds to a different
    ↪ concentration vector)
>>> x_init = np.array([[1 , 1, 1, 1, 1, 1, 1, 1],
```

```
                    [1, 1, 1, 1, 0, 0, 0, 0]])
>>> t_max = 5.e-13 # integration end time in seconds
>>> dt = 1.e-16 # step size in seconds
>>> cs = ChemSolver(chemkin('../tests/test_xml/rxns_reversible.xml'))
>>> cs.grid_solve(x_init, T, t_max, dt)
>>> cv = ChemViz(cs)
>>> cv.plot_gridtime_series('concentration', tmax = 2e-14, species = ['H', '
    ↪ OH', 'O2'], outputfile = 'modeldocfig4.png')
```

The example output is shown in Figure 4, demonstrating how highly sensitive species concentrations are to temperature and the initial concentrations of other species in the system.

Figure 4: Example output of plot_gridtime_series



3.5 Simple IO

Users can save and load pychemkin objects by using the `simpleIO` module. Users are allowed to load the saved pychemkin objects (such as `ChemSolver` or `ChemViz` without explicitly preloading dependent objects (such as `chemkin`), and restart calculations or visualization from any previous steps they want.

```
>>> import numpy as np
>>> from pychemkin import chemkin, ChemSolver, simpleIO
>>> T = 1000 #temperature in K
>>> x_init = np.ones(8) #initial concentration
>>> t_max = 5.e-13 # integration end time in seconds
>>> dt = 1.e-16 # step size in seconds
>>> cs = ChemSolver(chemkin('../tests/test_xml/rxns_reversible.xml'))
>>> cs.solve(x_init, T, t_max, dt)
>>> simpleIO('cs.pkl').to_pickle(cs)
>>> cs2 = simpleIO('cs.pkl').read_pickle()
```

4 Implementation details of new code feature

4.1 Motivation

Understanding how reaction systems evolve with time (i.e. the study of chemical kinetics) is important for employing chemical processes for human benefit. For example, we need to know whether the timescales over which the processes act are suitable for the desired application, we need to estimate production yields in order to determine whether a process is sufficient and to identify any problematic byproducts, and we need to be able to explore how the behavior of the system depends on initial conditions in order to identify the optimal starting conditions. Our package specifically addresses the issue of calculating and visualizing how reaction rates and species concentrations change over time. Visualization of these quantities is an essential tool for understanding the behavior of the system, since it 1) serves as a quick way to diagnose whether the inputs and calculations appear to be correct and 2) presents information in an easily digestible form and makes it easier to identify trends or relationships between different variables.

4.2 New modules

The extension of pychemkin involved the creation of three new classes: `ChemSolver`, `ChemViz`, and `simpleIO`. The design and implementation of each new class is described below.

4.2.1 ChemSolver

The `ChemSolver` module solves reaction rate equations (a first-order ordinary differential equation (ODE) system) to obtain species reaction rates and concentrations as a function of time. The module wrap arounds `SciPy`'s ODE library, which is a well-tested and widely used Python implementation of popular ODE solving algorithms. ChemSolver is initialized with a chemkin object.

The user-facing methods are as follows:

- `solve(self, y0, T, t1, dt, algorithm='lsoda', **options)`: This method wraps around `scipy.integrate.ode` to solve for reaction rates and species concentrations as a function of time. $y0$ is the initial concentration, T is the temperature, $t1$ is the endpoint for the integration, dt is the interval for which solutions should be returned, 'algorithm' is the choice of SciPy algorithm for solving the ODE system, and the optional parameters are the options listed in the documentation for scipy.integrate.ode. The five integrator options available through SciPy are `lsoda`, `vode`, `zvode`, `dop853`, and `dopri5`. Because reaction rate equation systems are often stiff, meaning that algorithms have to take very small timesteps to achieve numerical stability. For this reason, we recommend the usage of either `lsoda` or `vode` with 'method = "bdf"' (backwards differentiation formula), which are specifically designed for stiff systems. By default, we use `lsoda`, which can automatically transition between stiff and non-stiff methods, which allows for more efficient calculations in cases where the system is not stiff. Note that dt corresponds to the interval for which the solutions are returned to the user, NOT to the internal timestep that the integrator takes. This timestep is chosen automatically through SciPy, but if the integrator appears to be terminating prematurely or the solver appears to be unstable, the internal timesteps can be adjusted using the optional parameters `nsteps` (number of internal steps taken per integration interval), and `min_step` and `max_step` for the smallest and largest possible step sizes taken by the integrator.

- `get_results(return_reaction_rate=True)`, `to_df()`, `save_results(file_name)`, and `load_results(file_name)` are the various methods to save and retrieve the results, depending on the user's needs. `get_results` simply returns a tuple of (time, concentrations, reaction rates) if the user plans to manipulate these variables further in the program. `to_df` creates a pandas dataframe for the output data, for ease of viewing and to take advantage of convenient indexing and slicing. The user has the option of saving the output data in the form of a table of the concentration and reaction rates at every time step with `save_results`. The two output formats are csv and HDF5. The csv format is convenient because it can easily be processed by many external programs, and the user can simply open up the file to retrieve values as necessary. The HDF5 format is convenient for large files, if the user has a large number of species or timesteps. `load_results` will load the output csv or HDF5 file back into memory, to make it convenient for the user to manipulate that data in other forms (e.g., in the forms returned by `to_df` and `get_results`.

- `is_equilibrium(tol = 1.)` is a function that assesses whether a chemical kinetics simulation has reached steady-state by checking whether the reaction rates are sufficiently close to zero (given a tolerance level by the user). As a complementary check, the user may also wish to examine plots of concentration vs. time to see whether the curves level off (see next section).

- `grid_solve(y0s, Ts, t1, dt, algorithm='lsoda', return_reaction_rate=True, **options)` is a function intended to solve for the concentrations and reaction rates of a given system for different combinations of system temperature and initial concentrations. The inputs are similar to `solve`, except *Ts* is now a vector of temperature values and *y0s* is a two-dimensional array for which each row represents a different initial concentration vector. The user may be interested in this option if they wish to systematically explore parameter space by examining how varying initial concentrations or temperatures affect the time evolution of a system.

- `get_grid_result()` and `save_grid_results(file_prefix, filetype = 'csv')` are functions to retrieve and store the calculations from `grid_solve`. `get_grid_result` returns a tuple of the starting conditions and a dictionary of the derived concentrations and reaction rates, while `save_grid_results` will save the results of the simulations to csv or HDF5 files (one for each unique combination of temperature and initial concentration, each of which has a filename starting with the same 'file_prefix'.

4.2.2 ChemViz

The `ChemViz` module is a companion to the `ChemSolver` module, plotting the solutions found for the system of reaction rate ODEs. This module is built on top of Matplotlib. A ChemViz object is initialized with a ChemSolver object.

The user-facing methods are as follows:

- `plot_time_series(yaxis, species = None, tmin = 0, tmax = None, outputfile=None)` allows for concentration or reaction rates to be plotted as a function of time after ChemSolver solves for the time evolution for user's chosen reaction system. By default, all species are plotted over the entire time range for which calculations have been performed. However, since the user may only be interested in a subset of species or part of the time range, the user is allowed to enter a list of species to be plotted via the 'species' keyword and the time limits for the plot via 'tmin' and 'tmax.' The resulting image can be saved as a png file by setting the 'outputfile' keyword.

- `plot_gridtime_series(yaxis, species = None, tmin = 0, tmax = None, outputfile=None)` is similar to the previous function, except it plots a grid of time series for temperature/initial concentration combos as calculated by `ChemSolver.grid_solve`

- `plot_network(yaxis, species = None, tmin = 0, tmax = None, outputfile=None)` presents a complementary way of visually summarizing the evolution of the reaction

system. Whereas the time series plots are focused on the individual species, this function plots a diagram of the species in relation to one another. It is intended to help the user quickly identify the major species in the system, assess how favorable certain reactions are, and visualize how species are consumed. For example, the time series plot may show that H is decreasing rapidly with time, but it does not identify why H is decreasing with time. The diagram provides that complementary information by showing which species H reacts with, and which pairs of reactants have more favorable reaction rate coefficients.

- `html_report(file_name)` generates an HTML file containing a summary of the chemical equations, the reaction rate coefficients, the initial and end concentrations, and time series and network plots. The html template is based on GitHub's cayman theme.

4.2.3 simpleIO

`simpleIO` allows users to save and load pychemkin objects. Users can load the saved pychemkin objects (such as `ChemSolver` or `ChemViz` without explicitly preloading dependent objects (such as `chemkin`), and restart calculations or visualization from any previous steps they want.

The user-facing methods are as follows:

- `to_pickle(obj)` allows users to save a Python object to the Python-specific pickle format. Users can save `pychemkin` objects such as `ChemSolver` or `ChemViz` to pickle-format files.

- `read_pickle()` allows users to load a Python object from a pickle file. Specifically, instance variables and methods will also be implicitly loaded.