`chemical-kinetics` User Manual

Introduction

The subject of chemical kinetics studies the rates of chemical processes. Analyzing how reaction rates change with respect to reactant abundances, temperature, pressure, and other conditions gives insight into reaction mechanisms.

The present library, `chemical-kinetics`, is a Python 3 library for mathematical modeling of chemical reaction systems. Features include handling systems of elementary reactions, prediction of initial reaction rates, and simulation of reaction progress using deterministic and stochastic methods.

Reaction systems are specified in a standard XML format. The user may choose to use the provided interactive interface or program using the library directly.

Scientific background

A chemical equation describes a chemical reaction, relating the stoichiochetric ratios of reactants and products. For example, this is the equation for the combustion of hydrogen gas:

$$2\mathrm{H}_{2(g)} + \mathrm{O}_{2(g)} \rightarrow 2\mathrm{H}_2\mathrm{O}_{(l)} \tag{1}$$

A chemical equation may describe an overall net reaction, indicating the ratio of reactants consumed to products produced. On the other hand, such a net reaction may occur by way of several *elementary* reaction steps, each of which can be written as a separate chemical equation. A set of elementary reactions that together add up to yield the net reaction represents a *reaction mechanism*. Such a mechanism describes the exact reactions that lead to formation of the reaction products. An elementary reaction step describes a specific chemical reaction that occurs; i.e. which chemical species collide to produce which reaction products.

[still work in progress] Laboratory and computational experiments studying chemical kinetics have several reasons. They include.

- Reaction mechanisms: proposed mechanisms can be validated and tested.
- Equilibrium:
- Rate-determining steps can be determined

Predicting reaction rates

In a system of N chemical species undergoing M *elementary* reactions, the reaction rate of species i, f_i, is computed by

$$f_i = \sum_{j=1}^{M} \nu_{ij}\omega_j, \qquad i = 1, \ldots, N \tag{2}$$

where ω_j is the *progress rate* of the jth reaction and ν_{ij} is the net stoichiometric coefficient for species i in the jth reaction. The progress rate (ω) for each reaction is computed by

$$\omega_j = k_j^{(f)} \prod_{i=1}^{N} x_i^{\nu'_{ij}} - k_j^{(b)} \prod_{i=1}^{N} x_i^{\nu''_{ij}}, \qquad j = 1, \ldots, M \tag{3}$$

where, for the jth reaction, $k_j^{(f)}$ is the forward reaction rate coefficient and $k_j^{(b)}$ is the corresponding backward reaction rate cofficient. The x_i's represent abundances (usually concentrations) of the chemical species.

Given the particular nature of a chemical reaction, the reaction rate coefficient, k, can be a constant value or exhibit dependence on an activation energy, $E^{\ddagger}$, and reaction temperature, T. In the latter case, the reaction rate coefficient is often modeled using the Arrhenius equation:

$$k = AT^b \exp[-E^{\ddagger}/RT] \tag{4}$$

where A is the pre-exponential factor, b gives the temperature dependent of the pre-exponential factor ($b = 0$ indicating no dependence), and R is the universal gas constant.

The forward and reverse reaction rate coefficients for a chemical reaction are related to each other through the reaction equilibrium constant, K:

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

The equilibrium constant depends on temperature and the thermodynamic properties of the reactants and products in the following way:

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

$$\gamma_j = \sum_{i=1}^{N} \nu_{ij} \tag{7}$$

In particular, the equilibrium constant depends on the entropy and enthalpy change of the reaction, respectively ΔS and ΔH, which are calculated from entropy and enthalpy values of the chemical species. The change in entropy (enthalpy) is the sum of product entropies (enthalpies) minus the sum of reactant entropies (enthalpies).

$$\Delta S_j = \sum_{i=1}^{N} \nu_{ij} S_i \quad \text{and} \quad \Delta H_j = \sum_{i=1}^{N} \nu_{ij} H_i, , \qquad j = 1, \ldots, M \tag{8}$$

Entropy and enthalpy are thermodynamic quantities that are defined as follows, where C_p is the heat capacity at constant pressure.

$$H_i = \int_{T_0}^{T} C_{p,i}\,(T)\ \mathrm{d}T, \qquad i = 1, \ldots, N \tag{9}$$

$$S_i = \int_{T_0}^{T} \frac{C_{p,i}\,(T)}{T}\ \mathrm{d}T, \qquad i = 1, \ldots, N \tag{10}$$

Instead of determining the heat capacity of each chemical species from first principles, we approximate the heat capacity as well as integrated values of enthalpy and entropy using the 7th order NASA polynomials, which are are given by

$$\frac{C_{p,i}}{R} = a_{i1} + a_{i2}T + a_{i3}T^2 + a_{i4}T^3 + a_{i5}T^4$$

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

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

for species $i = 1, \ldots, N$.

Summary of notation

ν'_{ij} : Stoichiometric coefficients of reactants,

ν''_{ij} : Stoichiometric coefficients of products,

ω_j : Progress rate of reaction j,

x_i : Concentration of specie i,

$k_j^{(f)}$: forward reaction rate coefficient for reaction j,

$k_j^{(b)}$: backward reaction rate coefficient for reaction j,

K_j: *equilibrium coefficient* for reaction j,

p_0: pressure of the reactor (usually 10^5 Pa),

ΔS_j: the entropy change of reaction j,

ΔH_j: the enthalpy change of reaction j,

$C_{p,i}$: specific heat at constant pressure (given by the NASA polynomial)

Quantitative modeling of chemical reactions

The clients could call the chemkin package and obtained the right-hand-side of an ODE. They can then use it as the righ-hand-side of the ODE, or in a neural net code to learn new reaction pathways.

We offered two options for the user after obtaining the right-hand-side of an ODE. The user could choose to solve the ODE with the deterministic simulator or simulate the abundances of all species using the stochastic simulator. The simulation result will be presented by plots showing the trajectories of species abundances over time.

For deterministic simulator, we implemented three ODE solvers, backward euler method, backward differentiation formula and Runge-Kutta-Fehlberg method. To solve ODE problems in chemical kinetics, however, Runge-Kutta-Fehlberg method (rk45) is NOT recommended as the functions we are dealing with are usually stiff. It is recommended that the user sets the step size to be small (eg. 0.01) to achieve higher simulation accuracy when using the backward euler method and the backward differentiation. If the abundances of species become negative in the simulation process, the simulation will raise a `ValueError` and stop as abundances should never be negative.

For stochastic simulator, we implemented the Gillespie algorithm. The principle behind the algorithm is that waiting times between reaction events are exponentially distributed; thus the time until the next reaction is drawn from an exponential distribution (the simulation is advanced in time by this amount). The particular reaction event that occurs is randomly selected from among the possible reactions in proportion to their probabilities.

The following is a summary of the simulation process: 1. Generate two random numbers r_1, r_2 uniformly distribution in (0,1). 2. Compute the propensity function of each reaction and compute

$$\alpha_0 = \sum_{i=1}^{q} \alpha_i(t)$$

3. Compute the time interval until the next chemical reaction via

$$\tau = \frac{1}{\alpha_0} ln[1/r_1]$$

4. Compute which reaction occurs. Find j such that

$$r_2 \geq \frac{1}{\alpha_0} \sum_{i=1}^{j-1} \alpha_i$$

$$r_2 < \frac{1}{\alpha_0} \sum_{i=1}^{j} \alpha_i$$

5. The jth reaction takes place. Update the numbers of chemical species accordingly. 6. Continue simulation by returning to step 1.

To elaborate, the propensity function for a chemical reaction describes the probability of a particular reaction happening. For instance, consider the following bimolecular reaction:

$$A + A \rightarrow B; \quad rate\ k \qquad (1)$$

The quantity of species `A` present at time `t` is given by `A(t)`. The propensity function for the above reaction is $A(t)(A(t) - 1)k$. In a more complex reaction system, each elementary reaction will have its own propensity, and it is necessary to calculate the propensity of all elementary reactions in order to find the total "propensity" for a reaction to occur, in order to draw time τ until the next reaction from an exponential distribution.

Installation

A latest version of `chemical-kinetics` can downloaded from Github here.

Installation instructions

Obtain the latest version from github, change to the directory, and install using pip.

```
git clone https://github.com/cs207-2017-group13/cs207-FinalProject.git
cd cs207-FinalProject
pip3 install ./
```

Testing

Users can run the test suite by calling pytest from the main directory, i.e.

```
pytest --cov=src
```

Dependencies

Our package depends on `scipy`, `numpy`, `xml`, and `matplotlib` packages.

Contributing to the development version

(Fork and pull request)

Basic Usage and Examples

`XMLReader` class: Read and parse XML input file

The user should have an XML input file containing all the chemical reactions. `XMLReader` will read in the file and parse the file with the `xml.etree` library. It will output a list of dictionaries containing all the elements needed to calculate reaction rate coefficients, progress rates and reaction rates. It will create `ElementaryReaction` objects inside the `get_reaction_systems` function, and then put all `ElementaryReaction` objects in a reaction system into a list. It will then create `ReactionSystem` objects.

This class contains one method: `get_reaction_systems()`.

Example:

```
reader = XMLReader("tests/rxns.xml")
reaction_systems = reader.get_reaction_systems()
```

`reaction_systems` is a list containing multiple `ReactionSystem` instances. The length of `reaction_systems` is the number of reaction systems, and the length of each list element is the number of reactions in a reaction system.

`ElementaryReaction` class: Class for each elementary reaction

Takes a dictionary of properties from the XMLReader class for each elementary reaction. Calculates the rate coefficient for each elementary reaction and passes it to the ReactionSystem class. It also returns a dictionary of recatants and products to the ReactionSystem class.

This class has thee public methods, two private methods a special method: - `__repr__()`: Returns a string containing basic information about the elementary reaction - `get_reactants()`: Returns a dictionary with reactants as key and the stoichiometric coeff of reactants as value for each elementary reaction. - `get_products()`: Returns a dictionary with products as key and the stoichiometric coeff of products as value for each elementary reaction. - `calculate_rate_coefficient(T)`: Calculates and returns the rate coeffiecient of the reaction based on the type of rate coefficient required. - `_constant_rate(k=1.0)`: Returns a constant reaction rate coefficient with default value as 1.0 - `_k_arrhenius(A, E, T, b=0.0, R=8.314)`: Returns a reaction rate coefficient according to the Arrhenius equation. The Arrhenius equation relates the rate constant, k, of a chemical reaction to parameters A (pre-exponential factor), E (activation energy), T (absolute temperature), and b (exponential indicating temperature-dependence of pre-exponential factor)::

A nonzero value for b gives the modified Arrhenius equation.

$$k_{\text{mod arr}} = AT^b \exp\left(-\frac{E}{RT}\right) \qquad \text{(Modified Arrhenius)}$$

When b = 0, the above formula corresponds to the Arrhenius equation.

$$k_{\text{arr}} = A \exp\left(-\frac{E}{RT}\right) \qquad \text{(Arrhenius)}$$

Example:

```
python elementary_reaction = ElementaryReaction(reaction_properties) reactants =
elementary_reaction.get_reactants() products = elementary_reaction.get_products()
rate_coeff = elementary_reaction.calculate_rate_coefficient(1000) repr(elementary_reaction)
```

`ReactionSystem` class: Class for a system of reactions

Takes a list of ElementaryReaction instances and a list of species. Builds stoichiometric coefficient matrices for the reactants and products and calculates the corresponding progress rates and reaction rates.

This class has five methods, and two special methods: - `__repr__`: Returns a string containing basic information for the reaction system - `__len__`: Returns the number of species in the reaction system - `calculate_progress_rate(concs, temperature)`: Returns the progress rate of a system of elementary reactions - `calculate_reaction_rate(concs, temperature)`: Returns the reaction rate of a system of elementary reactions - `get_rate_coefficients()`: Calculate reaction rate coefficients - `build_reactant_coefficient_matrix()`: Build a reactant coefficients matrix for the reaction system - `build_product_coefficient_matrix()`: Build a product coefficients matrix for the reaction system - `check_reversible`: Check if each elementary reaction is reversible

Example:

```
len(reaction_system[0])
concs = [1., 2., 1., 3., 1.]
reaction_system[0].build_reactant_coefficient_matrix()
reaction_system[0].build_product_coefficient_matrix()
print(reaction_system[0])
reaction_system[0].calculate_progress_rate(concs, 300)
reaction_system[0].calculate_reaction_rate(concs, 300)
```

`Thermochem` class: Class for calculating the backward reaction rate

Construct with Rxnset class object, the default pressure of the reactor and the default ideal gas constant. The values of pressure of the reactor and ideal gas constant can be changed. It calculates backward reaction rate using the temperature passed into the functions and the corresponding NASA polynomial coefficients.

This class has four methods: - `Cp_over_R(T)`: Returns specific heat of each species given by the NASA polynomials - `H_over_RT(T)`: Returns the enthalpy of each species given by the NASA polynomials - `S_over_R(T)`: Returns the entropy of each species given by the NASA polynomials - `backward_coeffs(kf, T)`: Returns the backward reaction rate coefficient for reach reaction

Example:

```
reader = chemkin.XMLReader("tests/rxns_reversible.xml")
reaction_system = reader.get_reaction_systems()[0]
nu = reaction_system.product_coefficients - reaction_system.reactant_coefficients
rxnset = thermodynamics.Rxnset(reaction_system.species, nu)
thermo =  thermodynamics.Thermochem(rxnset)
thermo.Cp_over_R(800)
```

```
thermo.H_over_RT(800)
thermo.S_over_R(800)
kf = reaction_system.get_rate_coefficients(800)
thermo.backward_coeffs(kf, 800)
```

`Rxnset` class: Read and store NASA polynomial coefficients

This class reads the NASA polynomial coefficients for all species in the reaction system from the SQL database which contains coefficients for all species. It stores the coefficients and temperature ranges in a dictionary of dictionaries where the name of species is the key. For each reaction system, the class just need to read from the database once, and check the range the given temperature is in every time the temperature of the reaction system changes afterwards. If the `get_nasa_coefficients` function is called twice for the same reaction system and the same temperature, the cached value is returned.

This class has two methods: - `get_nasa_coefficients(T)`: Returns the corresponding NASA polynomial coefficients for all species at the given temperature - `read_nasa_coefficients()`: Return NASA polynomial coefficients for all species involved in the reaction system

Example:

```
reader = chemkin.XMLReader("tests/rxns_reversible.xml")
reaction_system = reader.get_reaction_systems()[0]
nu = reaction_system.product_coefficients - reaction_system.reactant_coefficients
rxnset = thermodynamics.Rxnset(reaction_system.species, nu)
rxnset.get_nasa_coefficients(800)
```

`memoized` class: Caches a function's return value each time it is called

Decorator class. Caches a function's return value each time it is called. If called later with the same arguments, the cached value is returned (not reevaluated).

This class has three special methods: - `__call__(*args)`: Cached a function's return value - `__repr__`: Return the function's docstring - `__get__(obj, objtype)`: Support instance methods

`ODE_solver` class: Solve ordinary differential equation with three methods

Implemented three ode solvers: backward euler method, Runge-Kutta-Fehlberg, and backward differentiation formula. Backward euler method and backward differentiation formula can be used to solve stiff ODE problems, while Runge-Kutta-Fehlberg moethod is more accurate for non-stiff problems. For the purposed of solving ODE problems in chemical kinetics, our default method is backward differentiation formula.

This class has five methods: - `backward_euler()`: Solve the ODE using backward euler method. - `backward_euler_step()`: Solve the ODE one step/time forward using backward euler method. We use fixed point iterations to find the optimal root. - `rk45()`: Solve the ODE using Runge-Kutta-Fehlberg method. - `rk45_step()`: Solve the ODE one step/time forward using Runge-Kutta-Fehlberg method. - `BDF()`: Solve the ODE using backward differentiation formula.

Example:

```
func = lambda t, y: 2*t
obj = ODE_solver(func, 0.8, [1, 2], 0.1)
obj.backward_euler()
obj.backward_euler_step()
```

```
obj.rk45()
obj.rk45_step()
obj.BDF()
```

`DeterministicSimulator` class: Class for deterministic simulation

This class is inherited from the `ReactionSimulator` class. It simulate species abundances deterministically.

This class has two methods: - simulate(method='bdf', epsilon = 1e-06): We implemented three methods to solve the ordinary differential equation. Backward differentiation formula and backward eulerare good for stiff functions, and rk45 is accurate for non-stiff functions. BDF is the most suitable for solving ODE problems in chemical kinetics, so our default method is set as BDF. - diff_func(t, y): In order for the ode solver to work, we need a function with t (time) and y as parameters. However, the original calculate reaction rate function is a function of y and temperature. We need to transform the original function.

Example:

```
import chemkin.chemkin as chemkin
import chemkin.simulator as simulator
concs = np.array([1., 2., 1., 3., 1.])*1e-05
reader = chemkin.XMLReader("tests/rxns.xml")
reaction_system = reader.get_reaction_systems()[0]
det_sim = simulator.DeterministicSimulator(reaction_system, concs, 800, [0, 1], dt=0.01)
det_sim.simulate()
```