
Stochastic Simulation Algorithms in Python Documentation

Release 0.9.1

Dileep Kishore, Srikanth Chandrasekaran

Jul 18, 2020

Contents

1	Introduction	3
2	Install	5
3	Documentation	7
4	Usage	9
5	License	13
6	Credits	15
	Index	35

Fig. 1: Logo for cayenne

CHAPTER 1

Introduction

`cayenne` is a Python package for stochastic simulations. It offers a simple API to define models, perform stochastic simulations with them and visualize the results in a convenient manner.

Currently under active development in the `develop` branch.

CHAPTER 2

Install

Install with pip:

```
$ pip install cayenne
```


CHAPTER 3

Documentation

- General: <https://cayenne.readthedocs.io>.
- Benchmark repository, comparing `cayenne` with other stochastic simulation packages: <https://github.com/Heuro-labs/cayenne-benchmarks>

CHAPTER 4

Usage

A short summary follows, but a more detailed tutorial can be found [here](#). You can define a model as a Python string (or a text file, see [docs](#)). The format of this string is loosely based on the excellent [antimony](#) library, which is used behind the scenes by `cayenne`.

```
from cayenne.simulation import Simulation
model_str = """
    const compartment compl;
    compl = 1.0; # volume of compartment

    r1: A => B; k1;
    r2: B => C; k2;

    k1 = 0.11;
    k2 = 0.1;
    chem_flag = false;

    A = 100;
    B = 0;
    C = 0;
    """
sim = Simulation.load_model(model_str, "ModelString")
# Run the simulation
sim.simulate(max_t=40, max_iter=1000, n_rep=10)
sim.plot()
```

4.1 Change simulation algorithm

You can change the algorithm used to perform the simulation by changing the `algorithm` parameter (one of "direct", "tau_leaping" or "tau_adaptive")

```
sim.simulate(max_t=150, max_iter=1000, n_rep=10, algorithm="tau_leaping")
```

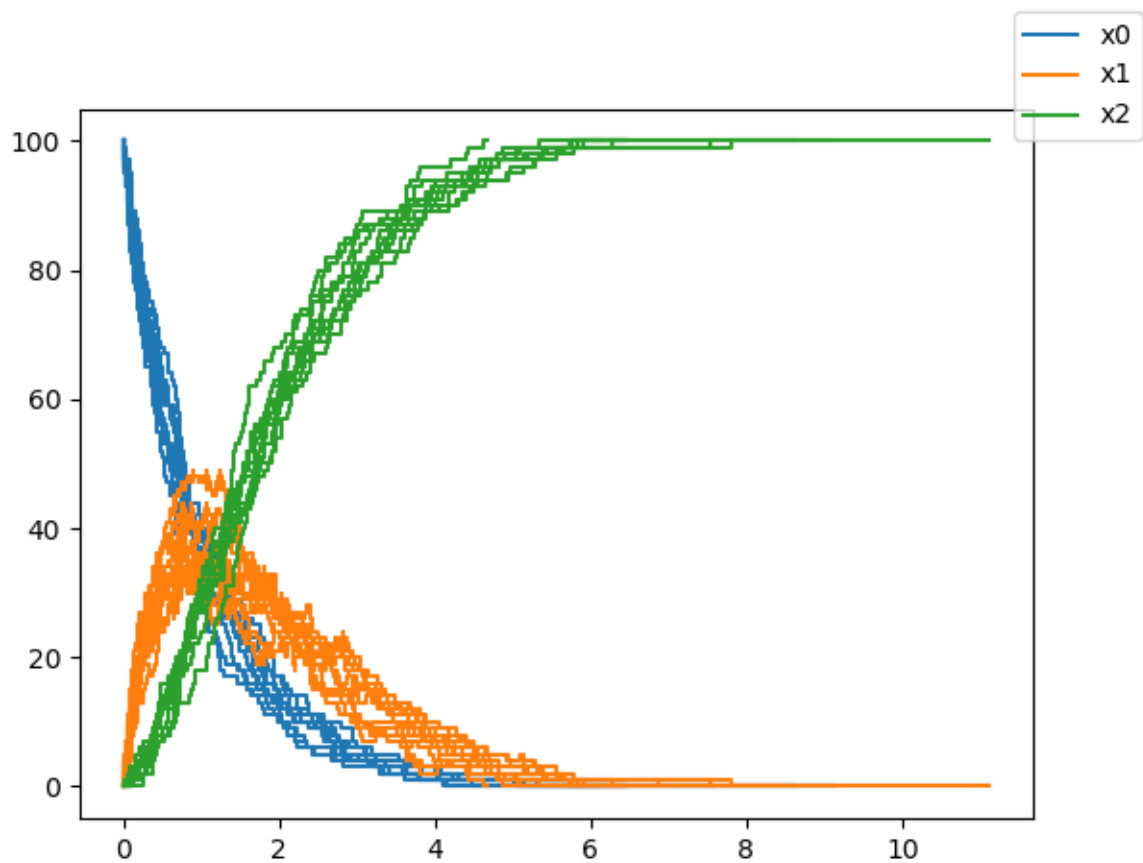


Fig. 1: Plot of species A, B and C

Our `benchmarks` are summarized *below*, and show `direct` to be a good starting point. `tau_leaping` offers greater speed but needs specification and tuning of the `tau` hyperparameter. The `tau_adaptive` is less accurate and a work in progress.

4.2 Run simulations in parallel

You can run the simulations on multiple cores by specifying the `n_procs` parameter

```
sim.simulate(max_t=150, max_iter=1000, n_rep=10, n_procs=4)
```

4.3 Accessing simulation results

You can access all the results or the results for a specific list of species

```
# Get all the results
results = sim.results
# Get results only for one or more species
results.get_species(["A", "C"])
```

You can also access the final states of all the simulation runs by

```
# Get results at the simulation endpoints
final_times, final_states = results.final
```

Additionally, you can access the state at a particular time point of interest `t`. `cayenne` will interpolate the value from nearby time points to give an accurate estimate.

```
# Get results at timepoint "t"
t = 10.0
states = results.get_state(t) # returns a list of numpy arrays
```

Benchmarks

	<code>direct</code>	<code>tau_leaping</code>	<code>tau_adaptive</code>
<code>cayenne</code>	:heavy_check_mark: Most accurate yet	:heavy_check_mark: Very fast but may need manual tuning	Less accurate than GillespieSSA's version
<code>Tellurium</code>	exclamation Inaccurate for 2nd order	N/A	N/A
<code>GillespieSSA</code>	Very slow	exclamation Inaccurate for initial zero counts	exclamation Inaccurate for initial zero counts
<code>BioSimulator.jl</code>	exclamation Inaccurate interpolation	exclamation Inaccurate for initial zero counts	exclamation Inaccurate for initial zero counts

CHAPTER 5

License

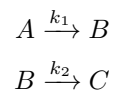
Copyright (c) 2018-2020, Dileep Kishore, Srikanth Chandrasekaran. Released under: Apache Software License 2.0

- Cython
- antimony
- pytest
- Cookiecutter
- audreyr/cookiecutter-pypackage
- black

6.1 Tutorial

6.1.1 Model Building

Consider a simple system of chemical reactions given by:



Suppose $k_{0.11} = 1$, $k_{0.1} = 1$ and there are initially 100 units of A. Then we have the following model string

```
>>> model_str = """
    const compartment compl;
    compl = 1.0; # volume of compartment

    r1: A => B; k1; # differs from antimony
    r2: B => C; k2; # differs from antimony

    k1 = 0.11;
    k2 = 0.1;
    chem_flag = false;
```

(continues on next page)

(continued from previous page)

```
A = 100;
B = 0;
C = 0;

"""
```

The format of the model string is based on the [antimony modeling language](#), but with one key difference. Antimony allows the user to specify custom rate *equations* for each reaction. *cayenne* automatically generates the rate equations behind the scenes, and user need only supply the rate *constants*. The format is discussed below:

Model format

```
const compartment compl;
```

This defines the compartment in which the reactions happen.:

```
compl = 1.0;
```

This defines the volume of the compartment in which reactions happen. For zero and first order reactions, this number does not matter. For second and third order reactions, varying compartment volume will affect the kinetic outcomes even when the rest of the model is not changed. A blank line after this separates these definitions from the reactions.:

```
r1: A => B; k1; # differs from antimony
r2: B => C; k2; # differs from antimony
```

Here *r1* and *r2* refer to the names of the reactions. This is followed by a colon and the reactants in that reaction. In *r1* there is only one reactant, A. Additional reactants or stoichiometries can be written like $A + 2B$. This is followed by \Rightarrow which separates reactants and products. Products are written in a fashion similar to the reactants. A semi-colon indicates the end of the products. This is followed by a symbol depicting the rate constant e.g. *k1*, and the reaction ends with a second semi-colon. A blank line after this separates these reactions from rate-constant assignments.:

```
k1 = 0.11;
k2 = 0.1;
```

The rate constants are assigned one per line, with each line ending in a semi-colon. Every rate constant defined in the reactions must be assigned a numerical value at this stage, or *cayenne* will throw a *cayenne.model_io.RateConstantError*.:

```
chem_flag = false;
```

An additional element that is included at this stage is the *chem_flag* boolean variable. This is discussed more in detail in the documentation of *cayenne.Simulation* class under the notes section. Briefly, if

1. the system under consideration is a chemical system and the supplied rate constants are in units of molarity or M or mol/L, *chem_flag* should be set to *true*
2. the system under consideration is a biological system and the supplied rate constants are in units of copies/L or CFU/L, *chem_flag* should be set to *false*

A blank line after this separates rate constants from initial values for the species.:

```
A = 100;
B = 0;
C = 0;
```

The initial values for species are assigned one per line, with each line ending in a semi-colon. Every species defined in the reactions must be assigned an integer initial value at this stage, or `cayenne` will throw a `cayenne.model_io.InitialStateError`.

Note: `cayenne` only accepts zero, first, second and third order reactions. We decided to not allow custom rate equations for stochastic simulations for two reasons:

1. A custom rate equation, such as the Monod equation (see [here](#) for background) equation below, may violate the [assumptions](#) of stochastic simulations. These assumptions include a well stirred chamber with molecules in Brownian motion, among others.

$$\mu = \frac{\mu_{max}S}{K_S + S}$$

2. An equation resembling the Monod equation, the [Michaelis-Menten](#) equation, is grounded chemical kinetic theory. Yet the rate expression (see below) does not fall under 0-3 order reactions supported by `cayenne`. However, the *elementary* reactions that make up the Michaelis-Menten kinetics are first and second order in nature. These *elementary* reactions can easily be modeled with `cayenne`, but with the specification of additional constants (see [examples](#)). A study shows that using the rate expression of Michaelis-Menten kinetics is valid under [some conditions](#).

$$\frac{dP}{dt} = \frac{\mu_{max}S}{K_S + S}$$

Note: The `chem_flag` is set to `True` since we are dealing with a chemical system. For definition of `chem_flag`, see the notes under the definition of the `Simulation` class.

These variables are passed to the `Simulation` class to create an object that represents the current system

```
>>> from cayenne import Simulation
>>> sim = Simulation.load_model(model_str, "ModelString")
```

```
class cayenne.simulation.Simulation(species_names: List[str], rxn_names: List[str], react_stoic: numpy.ndarray, prod_stoic: numpy.ndarray, init_state: numpy.ndarray, k_det: numpy.ndarray, chem_flag: bool = False, volume: float = 1.0)
```

A main class for running simulations.

Parameters

- **species_names** (`List[str]`) – List of species names
- **rxn_names** (`List[str]`) – List of reaction names
- **react_stoic** (`(ns, nr) ndarray`) – A 2D array of the stoichiometric coefficients of the reactants. Reactions are columns and species are rows.
- **prod_stoic** (`(ns, nr) ndarray`) – A 2D array of the stoichiometric coefficients of the products. Reactions are columns and species are rows.
- **init_state** (`(ns,) ndarray`) – A 1D array representing the initial state of the system.
- **k_det** (`(nr,) ndarray`) – A 1D array representing the deterministic rate constants of the system.
- **volume** (`float, optional`) – The volume of the reactor vessel which is important for second and higher order reactions. Defaults to 1 arbitrary units.

- **chem_flag** (*bool, optional*) – If True, divide by Na (Avogadro’s constant) while calculating stochastic rate constants. Defaults to False.

results

The results instance

Type *Results*

Raises `ValueError` – If supplied with order > 3.

Examples

```
>>> V_r = np.array([[1,0],[0,1],[0,0]])
>>> V_p = np.array([[0,0],[1,0],[0,1]])
>>> X0 = np.array([10,0,0])
>>> k = np.array([1,1])
>>> sim = Simulation(V_r, V_p, X0, k)
```

Notes

Stochastic reaction rates depend on the size of the system for second and third order reactions. By this, we mean the volume of the system in which the reactants are contained. Intuitively, this makes sense considering that collisions between two or more molecules becomes rarer as the size of the system increases. A detailed mathematical treatment of this idea can be found in³.

In practice, this means that `volume` and `chem_flag` need to be supplied for second and third order reactions. `volume` represents the size of the system containing the reactants.

In chemical systems `chem_flag` should generally be set to True as `k_det` is specified in units of molarity or M or mol/L. For example, a second order rate constant could be = 0.15 mol / (L s). Then Avogadro’s constant (N_a) is used for normalization while computing `k_stoc` (c_μ in³) from `k_det`.

In biological systems, `chem_flag` should be generally be set to False as `k_det` is specified in units of copies/L or CFU/L. For example, a second order rate constant could be = 0.15 CFU / (L s).

References**6.1.2 Running Simulations**

Suppose we want to run 10 repetitions of the system for at most 1000 steps / 40 time units each, we can use the `simulate` method to do this.

```
>>> sim.simulate(max_t=40, max_iter=1000, n_rep=10)
```

`Simulation.simulate` (*max_t: float = 10.0, max_iter: int = 1000, seed: int = 0, n_rep: int = 1, n_procs: Optional[int] = 1, algorithm: str = 'direct', debug: bool = False, **kwargs*) → None

Run the simulation

Parameters

- **max_t** (*float, optional*) – The end time of the simulation. The default is 10.0.

³ Gillespie, D.T., 1976. A general method for numerically simulating the stochastic time evolution of coupled chemical reactions. J. Comput. Phys. 22, 403–434. doi:10.1016/0021-9991(76)90041-3.

- **max_iter** (*int*, *optional*) – The maximum number of iterations of the simulation loop. The default is 1000 iterations.
- **seed** (*int*, *optional*) – The seed used to generate simulation seeds. The default value is 0.
- **n_rep** (*int*, *optional*) – The number of repetitions of the simulation required. The default value is 1.
- **n_procs** (*int*, *optional*) – The number of cpu cores to use for the simulation. Use `None` to automatically detect number of cpu cores. The default value is 1.
- **algorithm** (*str*, *optional*) – The algorithm to be used to run the simulation. The default value is "direct".

Notes

The status indicates the status of the simulation at exit. Each repetition will have a status associated with it, and these are accessible through the `Simulation.results.status_list`.

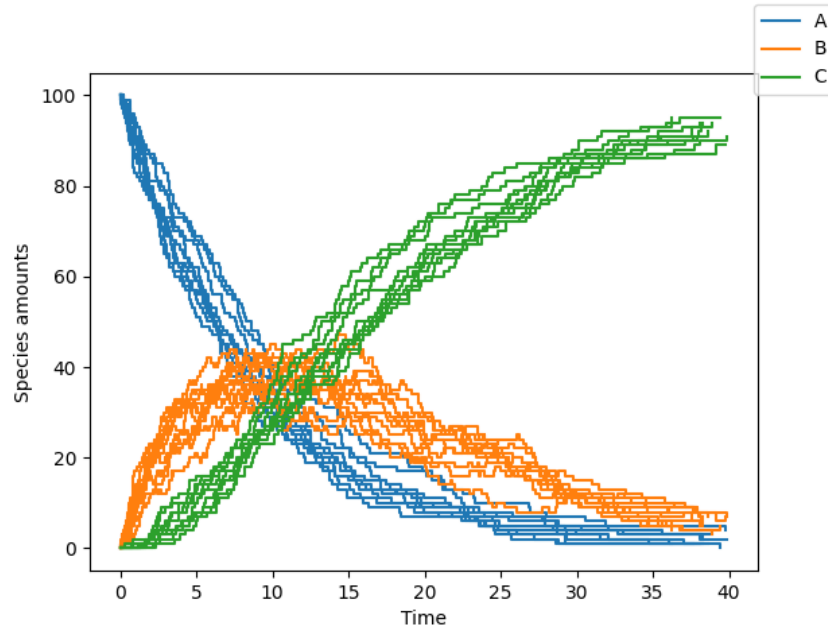
status: int Indicates the status of the simulation at exit.

- 1: Successful completion, terminated when `max_iter` iterations reached.
- 2: Successful completion, terminated when `max_t` crossed.
- 3: Successful completion, terminated when all species went extinct.
- 1: Failure, order greater than 3 detected.
- 2: Failure, propensity zero without extinction.

6.1.3 Plotting

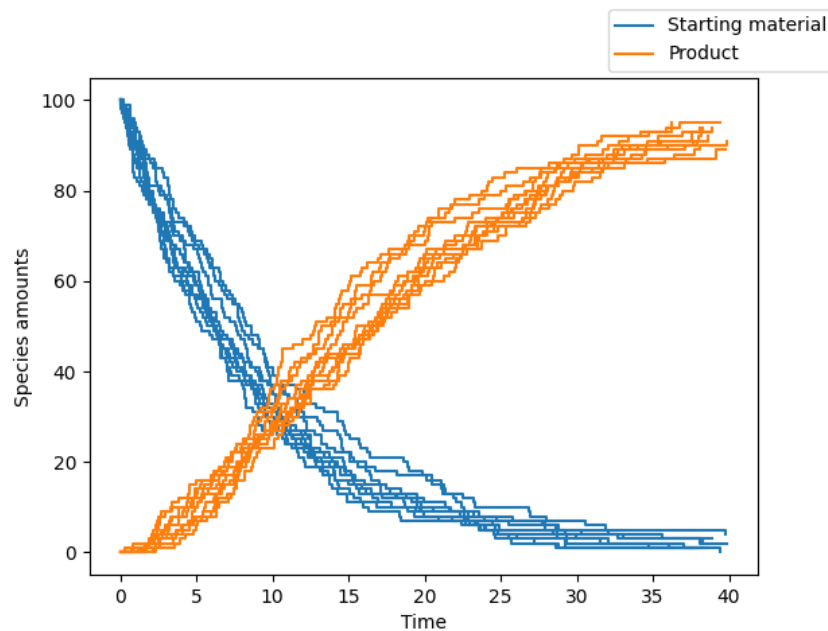
To plot the results on the screen, we can simply plot all species concentrations at all the time-points using:

```
>>> sim.plot()
```



A subset of the species can be plotted along with custom display names by supplying additional arguments to `Simulation.plot` as follows:

```
>>> sim.plot(species_names = ["A", "C"], new_names = ["Starting material", "Product"])
```



By default, calling the plot object returns the `matplotlib` figure and axis objects. To display the plot, we just do:

```
>>> sim.plot()
>>> import matplotlib.pyplot as plt
>>> plt.show()
```

Instead to save the figure directly to a file, we do:


```
>>> sim.plot()
>>> import matplotlib.pyplot as plt
>>> plt.savefig("plot.png")
```

`Simulation.plot` (*species_names: list = None, new_names: list = None*)

Plot the simulation

Parameters

- **species_names** (*list, optional*) – The names of the species to be plotted (*list* of *str*). The default is *None* and plots all species.
- **new_names** (*list, optional*) – The names of the species to be plotted. The default is "xi" for species *i*.

Returns

- **fig** (*class 'matplotlib.figure.Figure'*) – Figure object of the generated plot.
- **ax** (*class 'matplotlib.axes._subplots.AxesSubplot'*) – Axis object of the generated plot.

Note:

1. The `sim.plot` method needs to be run after running `sim.simulate`
2. More detailed plots can be created manually by accessing the `sim.results` object

6.1.4 Accessing the results

The results of the simulation can be retrieved by accessing the `Results` object as

```
>>> results = sim.results
>>> results
<Results species=('A', 'B', 'C') n_rep=10algorithm=direct sim_seeds=[8325804,
↳1484405 2215104 5157699 8222403 7644169 5853461 6739698 374564
2832983]>
```

The `Results` object provides abstractions for easy retrieval and iteration over the simulation results. For example you can iterate over every run of the simulation using

```
>>> for x, t, status in results:
...     pass
```

You can access the results of the *n* th run by

```
>>> nth_result = results[n]
```

You can also access the final states of all the simulation runs by

```
>>> final_times, final_states = results.final

# final times of each repetition
>>> final_times
array([6.23502469, 7.67449057, 6.15181435, 8.95810706, 7.12055223,
       7.06535004, 6.07045973, 7.67547689, 9.4218006 , 9.00615099])

# final states of each repetition
```

(continues on next page)

(continued from previous page)

```
>>> final_states
array([[ 0,  0, 100],
       [ 0,  0, 100],
       [ 0,  0, 100],
       [ 0,  0, 100],
       [ 0,  0, 100],
       [ 0,  0, 100],
       [ 0,  0, 100],
       [ 0,  0, 100],
       [ 0,  0, 100],
       [ 0,  0, 100]])
```

You can obtain the state of the system at a particular time using the `get_state` method. For example to get the state of the system at time `t=5.0` for each repetition:

```
>>> results.get_state(5.0)
[array([ 1,  4, 95]),
 array([ 1,  2, 97]),
 array([ 0,  2, 98]),
 array([ 3,  4, 93]),
 array([ 0,  3, 97]),
 array([ 0,  2, 98]),
 array([ 1,  1, 98]),
 array([ 0,  4, 96]),
 array([ 1,  6, 93]),
 array([ 1,  3, 96])]
```

Additionally, you can also access a particular species' trajectory through time across all simulations with the `get_species` function as follows:

```
>>> results.get_species(["A", "C"])
```

This will return a list with a numpy array for each repetition. We use a list here instead of higher dimensional ndarray for the following reason: any two repetitions of a stochastic simulation need not return the same number of time steps.

```
class cayenne.results.Results(species_names: List[str], rxn_names: List[str], t_list:
                             List[numpy.ndarray], x_list: List[numpy.ndarray], status_list:
                             List[int], algorithm: str, sim_seeds: List[int])
```

A class that stores simulation results and provides methods to access them

Parameters

- **species_names** (*List[str]*) – List of species names
- **rxn_names** (*List[str]*) – List of reaction names
- **t_list** (*List[float]*) – List of time points for each repetition
- **x_list** (*List[np.ndarray]*) – List of system states for each repetition
- **status_list** (*List[int]*) – List of return status for each repetition
- **algorithm** (*str*) – Algorithm used to run the simulation
- **sim_seeds** (*List[int]*) – List of seeds used for the simulation

Notes

The status indicates the status of the simulation at exit. Each repetition will have a status associated with it, and these are accessible through the `status_list`.

- 1: Successful completion, terminated when `max_iter` iterations reached.
- 2: Successful completion, terminated when `max_t` crossed.
- 3: Successful completion, terminated when all species went extinct.
- 1: Failure, order greater than 3 detected.
- 2: Failure, propensity zero without extinction.

<code>Results.__iter__()</code>	Iterate over each repetition
<code>Results.__len__()</code>	Return number of repetitions in simulation
<code>Results.__contains__(ind)</code>	Returns True if <code>ind</code> is one of the repetition numbers
<code>Results.__getitem__(ind)</code>	Return <code>sim</code> .
<code>Results.final</code>	Returns the final times and states of the system in the simulations
<code>Results.get_state(t)</code>	Returns the states of the system at time point <code>t</code> .

6.1.5 Algorithms

The `Simulation` class currently supports the following algorithms (see *Algorithms*):

- 1. *Gillespie's direct method*
- 2. *Tau leaping method method*
- 3. *Adaptive tau leaping method (experimental)*

You can change the algorithm used to perform a simulation using the `algorithm` argument

```
>>> sim.simulate(max_t=150, max_iter=1000, n_rep=10, algorithm="tau_leaping")
```

6.2 Algorithms

`cayenne` currently has 3 algorithms:

- 1. *Gillespie's direct method (direct)* (accurate, may be slow)
- 2. *Tau leaping method (tau_leaping)* (approximate, faster, needs to be tuned)
- 3. *Adaptive tau leaping method (experimental, tau_adaptive)* (approximate, faster, largely self-tuning)

Methods are described more in depth below.

6.2.1 Gillespie's direct method (`direct`)

Implementation of Gillespie's Direct method. This is an exact simulation algorithm that simulates each reaction step. This makes it slower than other methods, but it's a good place to start.

class `cayenne.algorithms.direct`
Runs the Direct Stochastic Simulation Algorithm¹

Parameters

- **react_stoic** (*(ns, nr) ndarray*) – A 2D array of the stoichiometric coefficients of the reactants. Reactions are columns and species are rows.
- **prod_stoic** (*(ns, nr) ndarray*) – A 2D array of the stoichiometric coefficients of the products. Reactions are columns and species are rows.
- **init_state** (*(ns,) ndarray*) – A 1D array representing the initial state of the system.
- **k_det** (*(nr,) ndarray*) – A 1D array representing the deterministic rate constants of the system.
- **max_t** (*float*) – The maximum simulation time to run the simulation for.
- **max_iter** (*int*) – The maximum number of iterations to run the simulation for.
- **volume** (*float*) – The volume of the reactor vessel which is important for second and higher order reactions. Defaults to 1 arbitrary units.
- **seed** (*int*) – The seed for the numpy random generator used for the current run of the algorithm.
- **chem_flag** (*bool*) – If True, divide by Na (Avogadro’s constant) while calculating stochastic rate constants. Defaults to False.

Returns

- **t** (*ndarray*) – Numpy array of the times.
- **x** (*ndarray*) – Numpy array of the states of the system at times in **t**.
- **status** (*int*) – Indicates the status of the simulation at exit.
 - 1 - Successful completion, terminated when **max_iter** iterations reached.
 - 2 - Successful completion, terminated when **max_t** crossed.
 - 3 - Successful completion, terminated when all species went extinct.
 - 1 - Failure, order greater than 3 detected.
 - 2 - Failure, propensity zero without extinction.

References

6.2.2 Tau leaping method (`tau_leaping`)

Implementation of the [tau leaping algorithm](#). This is an approximate method that needs to be tuned to the system at hand (by modifying the time step or the `tau` parameter). A default `tau=0.1` is assumed by `cayenne`. This algorithm is approximate and faster than the Direct algorithm, but it must be used with caution. Smaller time steps make the simulation more accurate, but increase the code run time. Larger time steps makes the simulations less accurate and speeds up code run time.

class `cayenne.algorithms.tau_leaping`
Runs the Tau Leaping Simulation Algorithm. Exits if negative population encountered.

¹ Gillespie, D.T., 1976. A general method for numerically simulating the stochastic time evolution of coupled chemical reactions. J. Comput. Phys. 22, 403–434. doi:10.1016/0021-9991(76)90041-3.

Parameters

- **react_stoic** ((*ns*, *nr*) *ndarray*) – A 2D array of the stoichiometric coefficients of the reactants. Reactions are columns and species are rows.
- **prod_stoic** ((*ns*, *nr*) *ndarray*) – A 2D array of the stoichiometric coefficients of the products. Reactions are columns and species are rows.
- **init_state** ((*ns*,) *ndarray*) – A 1D array representing the initial state of the system.
- **k_det** ((*nr*,) *ndarray*) – A 1D array representing the deterministic rate constants of the system.
- **tau** (*float*) – The constant time step used to tau leaping.
- **max_t** (*float*) – The maximum simulation time to run the simulation for.
- **volume** (*float*) – The volume of the reactor vessel which is important for second and higher order reactions. Defaults to 1 arbitrary units.
- **seed** (*int*) – The seed for the numpy random generator used for the current run of the algorithm.
- **chem_flag** (*bool*) – If True, divide by Na (Avogadro’s constant) while calculating stochastic rate constants. Defaults to False.

Returns

- **t** (*ndarray*) – Numpy array of the times.
- **x** (*ndarray*) – Numpy array of the states of the system at times in *t*.
- **status** (*int*) – Indicates the status of the simulation at exit.
 - 1: Successful completion, terminated when *max_iter* iterations reached.
 - 2: Successful completion, terminated when *max_t* crossed.
 - 3: Successful completion, terminated when all species went extinct.
 - 1: Failure, order greater than 3 detected.
 - 2: Failure, propensity zero without extinction.
 - 3: Negative species count encountered.

6.2.3 Adaptive tau leaping method (experimental, tau_adaptive)

Experimental implementation of the [tau adaptive algorithm](#). This is an approximate method that builds off the tau_leaping method. It self-adapts the value of `tau` over the course of the simulation. For systems with a small number of molecules, it will be similar in speed to the direct method. For systems with a large number of molecules, it will be much faster than the direct method.

class `cayenne.algorithms.tau_adaptive`
 Runs the adaptive tau leaping simulation algorithm²

Parameters

- **react_stoic** ((*ns*, *nr*) *ndarray*) – A 2D array of the stoichiometric coefficients of the reactants. Reactions are columns and species are rows.

² Cao, Y., Gillespie, D.T., Petzold, L.R., 2006. Efficient step size selection for the tau-leaping simulation method. J. Chem. Phys. 124, 044109. doi:10.1063/1.2159468

- **prod_stoic** (*ns, nr ndarray*) – A 2D array of the stoichiometric coefficients of the products. Reactions are columns and species are rows.
- **init_state** (*ns, ndarray*) – A 1D array representing the initial state of the system.
- **k_det** (*nr, ndarray*) – A 1D array representing the deterministic rate constants of the system.
- **hor** – A 1D array of the highest order reaction in which each species appears.
- **nc** (*int*) – The criticality threshold. Reactions with that cannot fire more than *nc* times are deemed critical.
- **epsilon** (*float*) – The epsilon used in tau-leaping, measure of the bound on relative change in propensity.
- **max_t** (*float*) – The maximum simulation time to run the simulation for.
- **max_iter** (*int*) – The maximum number of iterations to run the simulation for.
- **volume** (*float*) – The volume of the reactor vessel which is important for second and higher order reactions. Defaults to 1 arbitrary units.
- **seed** (*int*) – The seed for the numpy random generator used for the current run of the algorithm.
- **chem_flag** (*bool*) – If True, divide by Na (Avogadro’s constant) while calculating stochastic rate constants. Defaults to False.

Returns

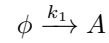
- **t** (*ndarray*) – Numpy array of the times.
- **x** (*ndarray*) – Numpy array of the states of the system at times in *t*.
- **status** (*int*) – Indicates the status of the simulation at exit.
 - 1 - Successful completion, terminated when *max_iter* iterations reached.
 - 2 - Successful completion, terminated when *max_t* crossed.
 - 3 - Successful completion, terminated when all species went extinct.
 - 1 - Failure, order greater than 3 detected.
 - 2 - Failure, propensity zero without extinction.
 - 3 - Negative species count encountered

References

6.3 Examples

Here we discuss some example systems and how to code them up using `cayenne`.

6.3.1 Zero order system



$$k_1 = 1.1$$

$$A(t=0) = 100$$

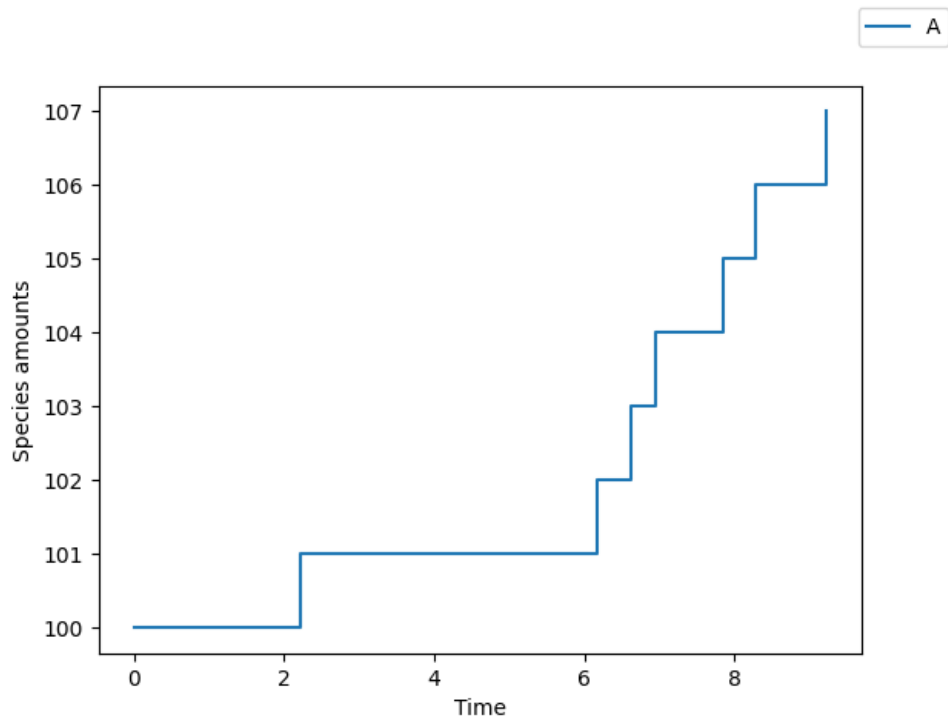
This can be coded up with:

```
>>> from cayenne import Simulation
>>> model_str = """
    const compartment compl;
    compl = 1.0; # volume of compartment

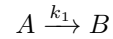
    r1: => A; k1;

    k1 = 1.1;
    chem_flag = false;

    A = 100;
    """
>>> sim = Simulation.load_model(model_str, "ModelString")
>>> sim.simulate()
>>> sim.plot()
```



6.3.2 First order system



$$k_1 = 1.1$$

$$A(t=0) = 100$$

$$B(t=0) = 20$$

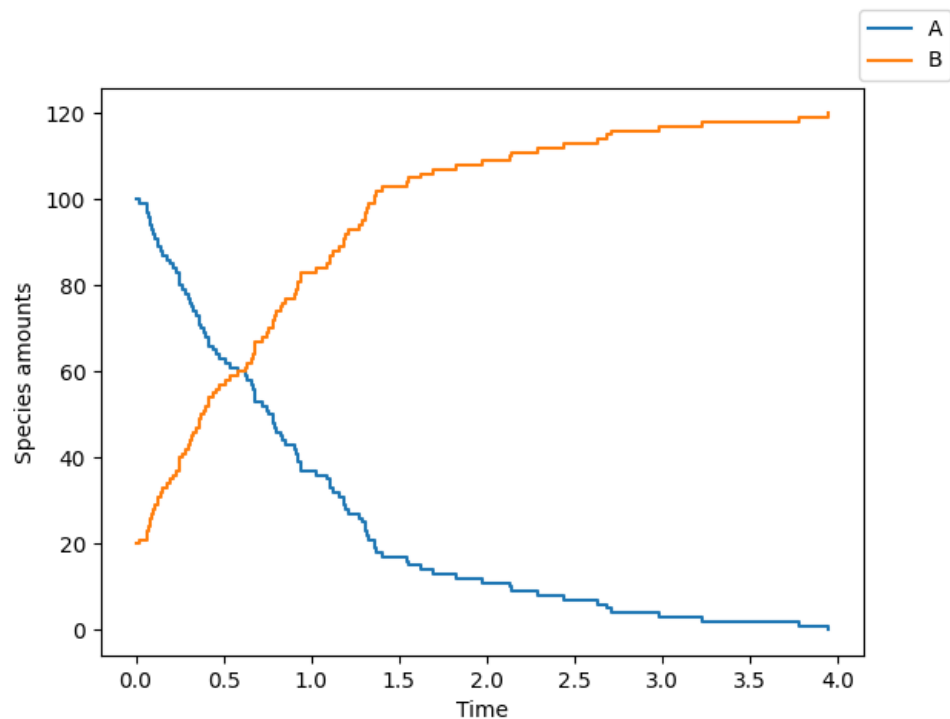
This can be coded up with:

```
>>> from cayenne import Simulation
>>> model_str = """
    const compartment comp1;
    comp1 = 1.0; # volume of compartment

    r1: A => B; k1;

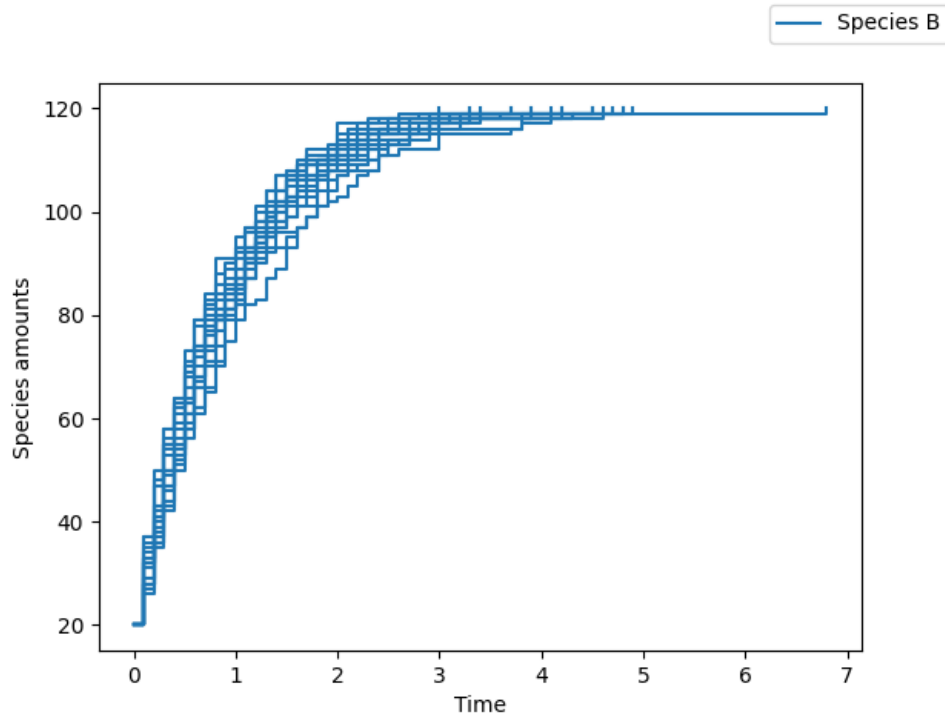
    k1 = 1.1;
    chem_flag = false;

    A = 100;
    B = 20;
    """
>>> sim = Simulation.load_model(model_str, "ModelString")
>>> sim.simulate()
>>> sim.plot()
```

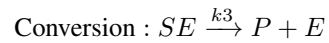
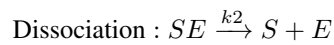
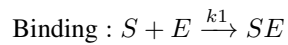


Suppose you want to use the `tau_leaping` algorithm, run 20 repetitions and plot only species `B`. Then do:

```
>>> sim.simulate(algorithm="tau_leaping", n_rep=20)
>>> sim.plot(species_names=["B"], new_names=["Species B"])
```

6.3.3 Enzyme kinetics (second order system with multiple reactions)



$$k_1 = 0.006$$

$$k_2 = 0.005$$

$$k_3 = 0.1$$

$$S(t=0) = 200$$

$$E(t=0) = 50$$

$$SE(t=0) = 0$$

$$P(t=0) = 0$$

This can be coded up with:

```
>>> from cayenne import Simulation
>>> model_str = """
    const compartment compl;
    compl = 1.0; # volume of compartment

    binding: S + E => SE; k1;
    dissociation: SE => S + E; k2;
    conversion: SE => P + E; k3;

    k1 = 0.006;
```

(continues on next page)

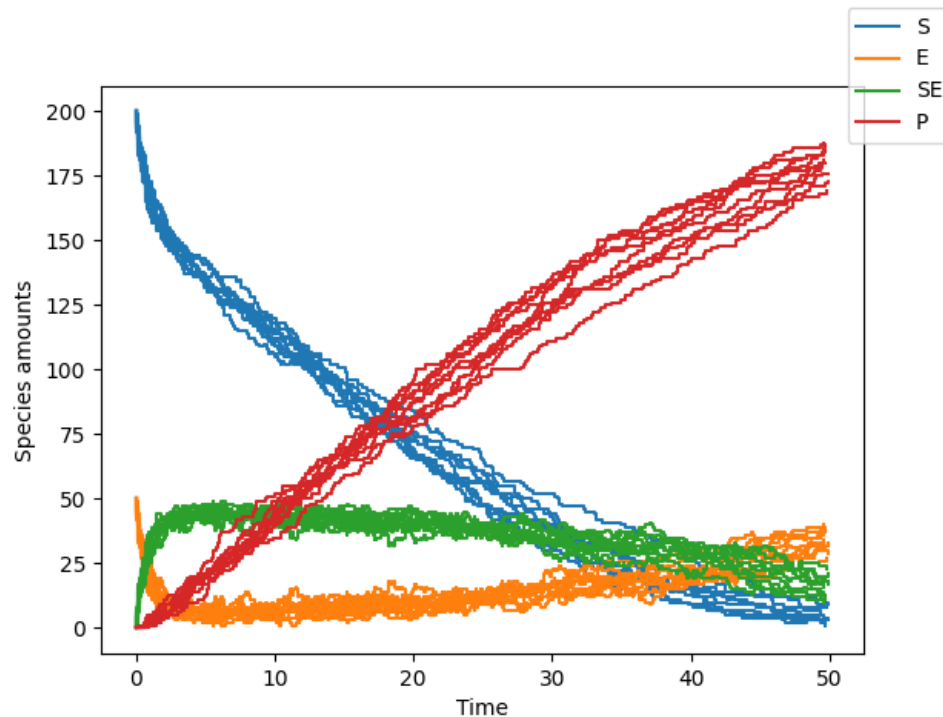
(continued from previous page)

```

k2 = 0.005;
k3 = 0.1;
chem_flag = false;

S = 200;
E = 50;
SE = 0;
P = 0;
"""
>>> sim = Simulation.load_model(model_str, "ModelString")
>>> sim.simulate(max_t=50, n_rep=10)
>>> sim.plot()

```



Since this is a second order system, the size of the system affects the reaction rates. What happens in a larger system?

```

>>> from cayenne import Simulation
>>> model_str = """
const compartment compl;
compl = 5.0; # volume of compartment

binding: S + E => SE; k1;
dissociation: SE => S + E; k2;
conversion: SE => P + E; k3;

k1 = 0.006;
k2 = 0.005;
k3 = 0.1;
chem_flag = false;

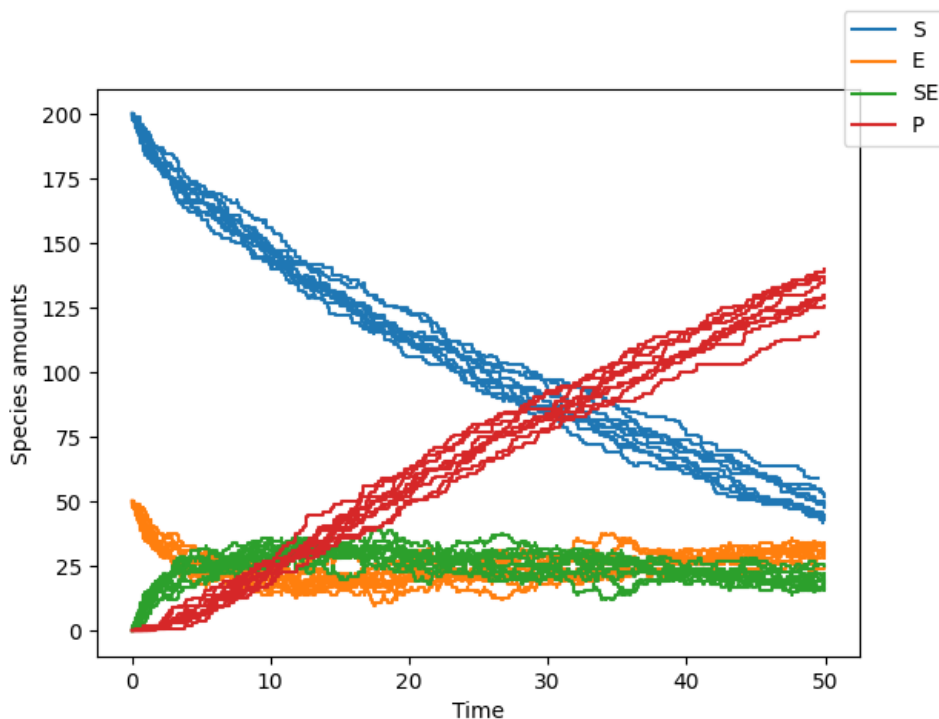
S = 200;
E = 50;

```

(continues on next page)

(continued from previous page)

```
SE = 0;
P = 0;
"""
>>> sim = Simulation.load_model(model_str, "ModelString")
>>> sim.simulate(max_t=50, n_rep=10)
>>> sim.plot()
```



Here we see that the reaction proceeds slower. Less of the product is formed by $t=50$ compared to the previous case.

6.4 Contributing

Contributions are welcome, and they are greatly appreciated! Every little bit helps, and credit will always be given.

You can contribute in many ways:

6.4.1 Types of Contributions

Report Bugs

Report bugs at <https://github.com/dileep-kishore/cayenne/issues>.

If you are reporting a bug, please include:

- Your operating system name and version.
- Any details about your local setup that might be helpful in troubleshooting.
- Detailed steps to reproduce the bug.

Fix Bugs

Look through the GitHub issues for bugs. Anything tagged with “bug” and “help wanted” is open to whoever wants to implement it.

Implement Features

Look through the GitHub issues for features. Anything tagged with “enhancement” and “help wanted” is open to whoever wants to implement it.

Write Documentation

Stochastic Simulation Algorithms in Python could always use more documentation, whether as part of the official Stochastic Simulation Algorithms in Python docs, in docstrings, or even on the web in blog posts, articles, and such.

Submit Feedback

The best way to send feedback is to file an issue at <https://github.com/dileep-kishore/cayenne/issues>.

If you are proposing a feature:

- Explain in detail how it would work.
- Keep the scope as narrow as possible, to make it easier to implement.
- Remember that this is a volunteer-driven project, and that contributions are welcome :)

6.4.2 Get Started!

Ready to contribute? Here’s how to set up *cayenne* for local development.

1. Fork the *cayenne* repo on GitHub.
2. Clone your fork locally:

```
$ git clone git@github.com:your_name_here/cayenne.git
```

3. Install your local copy into a virtualenv. Assuming you have virtualenvwrapper installed, this is how you set up your fork for local development:

```
$ mkvirtualenv cayenne
$ cd cayenne/
$ python setup.py develop
```

4. Create a branch for local development:

```
$ git checkout -b name-of-your-bugfix-or-feature
```

Now you can make your changes locally.

5. When you’re done making changes, check that your changes pass flake8 and the tests, including testing other Python versions with tox:

```
$ flake8 cayenne tests
$ python setup.py test or py.test
$ tox
```

To get flake8 and tox, just pip install them into your virtualenv.

6. Commit your changes and push your branch to GitHub:

```
$ git add .  
$ git commit -m "Your detailed description of your changes."  
$ git push origin name-of-your-bugfix-or-feature
```

7. Submit a pull request through the GitHub website.

6.4.3 Pull Request Guidelines

Before you submit a pull request, check that it meets these guidelines:

1. The pull request should include tests.
2. If the pull request adds functionality, the docs should be updated. Put your new functionality into a function with a docstring, and add the feature to the list in README.rst.
3. The pull request should work for Python 2.7, 3.4, 3.5 and 3.6, and for PyPy. Check https://travis-ci.org/dileep-kishore/cayenne/pull_requests and make sure that the tests pass for all supported Python versions.

6.4.4 Tips

To run a subset of tests:

```
$ py.test tests.test_cayenne
```

6.4.5 Deploying

A reminder for the maintainers on how to deploy. Make sure all your changes are committed (including an entry in HISTORY.rst). Then run:

```
$ bumpversion patch # possible: major / minor / patch  
$ git push  
$ git push --tags
```

Travis will then deploy to PyPI if tests pass.

6.5 Credits

6.5.1 Development Lead

- Dileep Kishore <k.dileep1994@gmail.com>
- Srikirán Chandrasekaran <srikiranc@gmail.com>

6.5.2 Contributors

None yet. Why not be the first?

C

`cayenne.algorithms.direct` (*module*), [23](#)
`cayenne.algorithms.tau_adaptive` (*module*),
[25](#)
`cayenne.algorithms.tau_leaping` (*module*),
[24](#)

D

`direct` (*class in cayenne.algorithms*), [23](#)

P

`plot()` (*cayenne.simulation.Simulation method*), [21](#)

R

`results` (*cayenne.simulation.Simulation attribute*), [18](#)
`Results` (*class in cayenne.results*), [22](#)

S

`simulate()` (*cayenne.simulation.Simulation method*),
[18](#)
`Simulation` (*class in cayenne.simulation*), [17](#)

T

`tau_adaptive` (*class in cayenne.algorithms*), [25](#)
`tau_leaping` (*class in cayenne.algorithms*), [24](#)