ppsim

Release 0.1.6

Eric Severson and David Doty

Jul 19, 2021

CONTENTS:

`ppsim` is a package for the simulation of population protocols.

For an introduction, see https://github.com/UC-Davis-molecular-computing/ppsim#readme

Table of Contents

CHAPTER

ONE

PPSIM.SIMULATION MODULE

A module for simulating population protocols.

The main class *Simulation* is created with a description of the protocol and the initial condition, and is responsible for running the simulation.

The general syntax is

```
a, b, u = 'A', 'B', 'U'
approx_majority = {
    (a,b): (u,u),
    (a,u): (a,a),
    (b,u): (b,b),
}
n = 10 ** 5
init_config = {a: 0.51 * n, b: 0.49 * n}
sim = Simulation(init_config=init_config, rule=approx_majority)
sim.run()
sim.history.plot()
```

More examples given in https://github.com/UC-Davis-molecular-computing/ppsim/tree/main/examples

time_trials() is a convenience function used for gathering data about the convergence time of a protocol.

ppsim.simulation.**Rule**
TODO

Type Type representing transition rule for protocol. Is one of three types

alias of Union[Callable[[Hashable, Hashable], Union[Tuple[Hashable, Hashable], Dict[Tuple[Hashable, Hashable], float]]], Dict[Tuple[Hashable, Hashable], Union[Tuple[Hashable, Hashable], Dict[Tuple[Hashable, Hashable], float]]], Iterable[*ppsim.crn.Reaction*]]

ppsim.simulation.**state_enumeration**(*init_dist*, *rule*)
Finds all reachable states by breadth-first search.

Parameters

- **init_dist** (*Dict[Hashable, int]*) – dictionary mapping states to counts (states are any hashable type, commonly NamedTuple or String)
- **rule** (*Callable[[Hashable, Hashable], Union[Tuple[Hashable, Hashable], Dict[Tuple[Hashable, Hashable], float]]]*) – function mapping a pair of states to either a pair of states or to a dictionary mapping pairs of states to probabilities

Returns a set of all reachable states

Return type Set[Hashable]

class ppsim.simulation.**Simulation**(*init_config*, *rule*, *simulator_method='MultiBatch'*, *transition_order='symmetric'*, *seed=None*, *volume=None*, *continuous_time=False*, *time_units=None*, ***kwargs*)

Class to simulate a population protocol.

Initialize a Simulation.

Parameters

- **init_config** (*Dict[Hashable, int]*) – The starting configuration, as a dictionary mapping states to counts.
- **rule** (*Rule*) – A representation of the transition rule. The first two options are a dictionary, whose keys are tuples of 2 states and values are their outputs, or a function which takes pairs of states as input. For a deterministic transition function, the output is a tuple of 2 states. For a probabilistic transition function, the output is a dictionary mapping tuples of states to probabilities. Inputs that are not present in the dictionary, or return None from the function, are interpreted as null transitions that return the same pair of states as the output. The third option is a list of *Reaction* objects describing a CRN, which will be parsed into an equivalent population protocol.
- **simulator_method** (*str*) – Which Simulator method to use, either `'MultiBatch'` or `'Sequential'`.

  'MultiBatch': *SimulatorMultiBatch* does O(sqrt(n)) interaction steps in parallel using batching, and is much faster for large population sizes and relatively small state sets.

  'Sequential': *SimulatorSequentialArray* represents the population as an array of agents, and simulates each interaction step by choosing a pair of agents to update. Defaults to 'MultiBatch'.
- **transition_order** (*str*) – Should the rule be interpreted as being symmetric, either `'asymmetric'`, `'symmetric'`, or `'symmetric_enforced'`. Defaults to 'symmetric'.

  'asymmetric': Ordering of the inputs matters, and all inputs not explicitly given as assumed to be null interactions.

  'symmetric': The input pairs are interpreted as unordered. If rule(a, b) returns None, while rule(b, a) has a non-null output, then the output of rule(a, b) is assumed to be the same as rule(b, a). If rule(a, b) and rule(b, a) are each given, there is no effect. Asymmetric interactions can be explicitly included this way.

  'symmetric_enforced': The same as symmetric, except that if rule(a, b) and rule(b, a) are non-null and do not give the same set of outputs, a ValueError is raised.
- **seed** (*Optional[int]*) – An optional integer used as the seed for all pseudorandom number generation. Defaults to None.
- **volume** (*Optional[float]*) – If a list of *Reaction* objects is given for a CRN, then the parameter volume can be passed in here. Defaults to None. If None, the volume will be assumed to be the population size n.
- **continuous_time** (*bool*) – Whether continuous time is used. Defaults to False. If a CRN as a list of reactions is passed in, this will be set to True.
- **time_units** (*Optional[str]*) – An optional string given the units that time is in. Defaults to None. This must be a valid string to pass as unit to pandas.to_timedelta.
- ****kwargs** – If *rule* is a function, any extra function parameters are passed in here, beyond the first two arguments representing the two agents. For example, if *rule* is defined:

```
def rule(sender: int, receiver: int, threshold: int) -> Tuple[int,␣
↪int]:
    if sender + receiver > threshold:
        return 0, 0
    else:
        return sender, receiver+1
```

To use a threshold of 20 in each interaction, in the *Simulation* constructor, use

```
sim = Simulation(init_config, rule, threshold=20)
```

simulator: *ppsim.simulator.Simulator*
An internal *Simulator* object, whose methods actually perform the steps of the simulation.

seed: **Optional[int]**
The optional integer seed used for rng and inside cython code.

rng: **numpy.random._generator.Generator**
A numpy random generator used to sample random variables outside the cython code.

steps_per_time_unit: **float**
Number of simulated interactions per time unit.

time_units: **Optional[str]**
The units that time is in.

continuous_time: **bool**
Whether continuous time is used. The regular discrete time model considers *steps_per_time_unit* steps to be 1 unit of time. The continuous time model is a poisson process, with expected *steps_per_time_unit* steps per 1 unit of time.

state_list: **List[Hashable]**
A sorted list of all reachable states.

state_dict: **Dict[Hashable, int]**
Maps states to their integer index to be used in array representations.

column_names: **Union[pandas.core.indexes.multi.MultiIndex, List[str]]**
Columns representing all states for pandas dataframe. If the State is a tuple, NamedTuple, or dataclass, this will be a pandas MultiIndex based on the various fields. Otherwise it is list of str(State) for each State.

configs: **List[numpy.ndarray]**
A list of all configurations that have been recorded during the simulation, as integer arrays.

times: **List[Union[float, datetime.timedelta]]**
A list of all the corresponding times for configs.

time: **float**
The current time.

snapshots: **List[*ppsim.snapshot.Snapshot*]**
A list of *Snapshot* objects, which get periodically called during the running of the simulation to give live updates.

rule(*a*, *b*)
The rule, as a function of two input states.

initialize_simulator(*config*)
Build the data structures necessary to instantiate the *Simulator* class.

Parameters **config** – The config array to instantiate *Simulator*.

array_from_dict(*d*)
Convert a configuration dictionary to an array.

Parameters **d** (*Dict*) – A dictionary mapping states to counts.

Returns An array giving counts of all states, in the order of self.state_list.

Return type numpy.ndarray

run(*run_until=None*, *history_interval=1.0*, *stopping_interval=1.0*, *timer=True*)
Runs the simulation.

Can give a fixed amount of time to run the simulation, or a function that checks the configuration for convergence.

Parameters

- **run_until** (*Optional[Union[float, Callable[[Dict[Hashable, int]], bool]]]*) – The stop condition. To run for a fixed amount of time, give a numerical value. To run until a convergence criterion, give a function mapping a configuration (as a dictionary mapping states to counts) to a boolean. The run will stop when the function returns True. Defaults to None. If None, the simulation will run until the configuration is silent (all transitions are null). This only works with the multibatch simulator method, if another simulator method is given, then using None will raise a ValueError.
- **history_interval** (*Union[float, Callable[[float], float]]*) – The length to run the simulator before recording data, in current time units. Defaults to 1. This can either be a float, or a function that takes the current time and and returns a float.
- **stopping_interval** (*float*) – The length to run the simulator before checking for the stop condition.
- **timer** (*bool*) – If True, and there are no other snapshot objects, a default *TimeUpdate* snapshot will be created to print updates with the current time. Defaults to True.

Return type None

property reactions: str
A string showing all non-null transitions in reaction notation.

Each reaction is separated by newlines, so that `print(self.reactions)` will display all reactions. Only works with simulator method multibatch, otherwise will raise a ValueError.

property enabled_reactions: str
A string showing all non-null transitions that are currently enabled.

This can only check the current configuration in self.simulator. Each reaction is separated by newlines, so that `print(self.enabled_reactions)` will display all enabled reactions.

reset(*init_config=None*)
Reset the simulation.

Parameters **init_config** (*Optional[Dict[Hashable, int]]*) – The configuration to reset to. Defaults to None. If None, will use the old initial configuration.

Return type None

set_config(*config*)
Change the current configuration.

Parameters **config** (*Union[Dict[Hashable, int], numpy.ndarray]*) – The configuration to change to. This can be a dictionary, mapping states to counts, or an array giving counts in the order of *state_list*.

Return type None

time_to_steps(*time*)
Convert simulated time into number of simulated interaction steps.

Parameters **time** (*float*) – The amount of time to convert.

Return type int

property config_dict: Dict[Hashable, int]
The current configuration, as a dictionary mapping states to counts.

property config_array: numpy.ndarray
The current configuration in the simulator, as an array of counts.

The array is given in the same order as self.state_list. The index of state s is self.state_dict[s].

property history: pandas.core.frame.DataFrame
A pandas dataframe containing the history of all recorded configurations.

property null_probability: float
The probability the next interaction is null.

times_in_units(*times*)
If *time_units* is defined, convert time list to appropriate units.

Parameters **times** (*Iterable[float]*) –

Return type Iterable[Any]

add_config()
Appends the current simulator configuration and time.

Return type None

set_snapshot_time(*time*)
Updates all snapshots to the nearest recorded configuration to a specified time.

Parameters **time** (*float*) – The parallel time to update the snapshots to.

Return type None

set_snapshot_index(*index*)
Updates all snapshots to the configuration *configs* [index].

Parameters **index** (*int*) – The index of the configuration.

Return type None

add_snapshot(*snap*)
Add a new *Snapshot* to *snapshots*.

Parameters **snap** (ppsim.snapshot.Snapshot) – The *Snapshot* object to be added.

Return type None

snapshot_slider(*var='index'*)
Returns a slider that updates all *Snapshot* objects.

Returns a slider from the ipywidgets library.

Parameters **var** (*str*) – What variable the slider uses, either 'index' or 'time'.

Return type widgets.interactive

sample_silence_time()
Starts a new trial from the initial distribution and return time until silence.

Return type float

sample_future_configuration(*time*, *num_samples=100*)

Repeatedly samples the configuration at a fixed future time.

Parameters

- **time** (*float*) – The amount of time ahead to sample the configuration.
- **num_samples** (*int*) – The number of samples to get.

Returns A dataframe whose rows are the sampled configuration.

Return type pandas.core.frame.DataFrame

ppsim.simulation.**time_trials**(*rule*, *ns*, *initial_conditions*, *convergence_condition=None*, *convergence_check_interval=0.1*, *num_trials=100*, *max_wallclock_time=86400*, ***kwargs*)

Gathers data about the convergence time of a rule.

Parameters

- **rule** (*Union[Callable[[Hashable, Hashable], Union[Tuple[Hashable, Hashable], Dict[Tuple[Hashable, Hashable], float]]], Dict[Tuple[Hashable, Hashable], Union[Tuple[Hashable, Hashable], Dict[Tuple[Hashable, Hashable], float]]], Iterable[ppsim.crn.Reaction]]*) – The rule that is used to generate the *Simulation* object.
- **ns** (*List[int]*) – A list of population sizes n to sample from. This should be in increasing order.
- **initial_conditions** (*Union[Callable, List]*) – An initial condition is a dict mapping states to counts. This can either be a list of initial conditions, or a function mapping population size n to an initial condition of n agents.
- **convergence_condition** (*Optional[Callable]*) – A boolean function that takes a configuration dict as input and returns True if that configuration has converged. Defaults to None. If None, the simulation will run until silent (all transitions are null), and the data will be for silence time.
- **convergence_check_interval** (*float*) – How often (in parallel time) the simulation will run between convergence checks. Defaults to 0.1. Smaller values give better resolution, but spend more time checking for convergence.
- **num_trials** (*int*) – The maximum number of trials that will be done for each population size n, if there is sufficient time. Defaults to 100. If you want to ensure that you get the full num_trials samples for each value n, use a large value for time_bound.
- **max_wallclock_time** (*float*) – A bound (in seconds) for how long this function will run. Each value n is given a time budget based on the time remaining, and will stop before doing num_trials runs when this time budget runs out. Defaults to 60 * 60 * 24 (one day).
- ****kwargs** – Other keyword arguments to pass into *Simulation*.

Returns A pandas dataframe giving the data from each trial, with two columns `'n'` and `'time'`. A good way to visualize this dataframe is using the seaborn library, calling `sns.lineplot(x='n', y='time', data=df)`.

Return type pandas.core.frame.DataFrame

CHAPTER

TWO

PPSIM.SNAPSHOT MODULE

A module for *Snapshot* objects used to visualize the protocol during or after the simulation has run.

Snapshot is a base class for snapshot objects that get are updated by *Simulation*.

Plotter is a subclass of *Snapshot* that creates a matplotlib figure and axis. It also gives the option for a state_map function which maps states to the categories which will show up in the plot.

StatePlotter is a subclass of *Plotter* that creates a barplot of the counts in categories.

HistoryPlotter is a subclass of *Plotter* that creates a lineplot of the counts in categories over time.

class ppsim.snapshot.**Snapshot**

Base class for snapshot objects.

Return type None

simulation

The *Simulation* object that initialized and will update the *Snapshot*. This attribute gets set when the *Simulation* object calls *add_snapshot*.

update_time

How many seconds will elapse between calls to update while in the *Simulation.run* method of *simulation*.

time

The time at the current snapshot. Changes when *Snapshot.update* is called.

config

The configuration array at the current snapshot. Changes when *Snapshot.update* is called.

Init constructor for the base class.

Parameters can be passed in here, and any attributes that can be defined without the parent *Simulation* object can be instantiated here, such as *update_time*.

initialize()

Method which is called once during *add_snapshot*.

Any initialization that requires accessing the data in *simulation* should go here.

Return type None

update(*index=None*)

Method which is called while *Snapshot.simulation* is running.

Parameters **index** (*Optional[int]*) – An optional integer index. If present, the snapshot will use the data from configuration *configs* [index] and time *times* [index]. Otherwise, the snapshot will use the current configuration *config_array* and current time.

Return type None

class ppsim.snapshot.**TimeUpdate**(*time_bound=None*, *update_time=0.2*)
Simple *Snapshot* that prints the current time in the *Simulation*.

When calling *Simulation.run*, if *snapshots* is empty, then this object will get added to provide a basic progress update.

Init constructor for the base class.

Parameters can be passed in here, and any attributes that can be defined without the parent *Simulation* object can be instantiated here, such as *update_time*.

Parameters

- **time_bound** (*Optional[float]*) –
- **update_time** (*float*) –

Return type None

initialize()
Method which is called once during *add_snapshot*.

Any initialization that requires accessing the data in *simulation* should go here.

Return type None

update(*index=None*)
Method which is called while *Snapshot.simulation* is running.

Parameters **index** (*Optional[int]*) – An optional integer index. If present, the snapshot will use the data from configuration *configs* [index] and time *times* [index]. Otherwise, the snapshot will use the current configuration *config_array* and current time.

Return type None

class ppsim.snapshot.**Plotter**(*state_map=None*, *update_time=0.5*, *yscale='linear'*, *sort_by='categories'*)
Base class for a *Snapshot* which will make a plot.

Gives the option to map states to categories, for an easy way to visualize relevant subsets of the states rather than the whole state set. These require an interactive matplotlib backend to work.

fig
The matplotlib figure that is created.

ax
The matplotlib axis object that is created. Modifying properties of this object is the most direct way to modify the plot.

yscale
The scale used for the yaxis, passed into ax.set_yscale.

state_map
A function mapping states to categories, which acts as a filter to view a subset of the states or just one field of the states.

categories
A list which holds the set {state_map(state)} for all states in *state_list*.

sort_by

_matrix
A (# states)x(# categories) matrix such that for the configuration array (indexed by states), matrix * config gives an array of counts of categories. Used internally to get counts of categories.

Initializes the *Plotter*.

Parameters

- **state_map** (*Optional[Callable[[Hashable], Any]]*) – An optional function mapping states to categories.
- **yscale** – The scale used for the yaxis, passed into ax.set_yscale. Defaults to 'linear'.
- **sort_by** (*str*) –

Return type None

initialize()
Initializes the plotter by creating a fig and ax.

Return type None

class ppsim.snapshot.**StatePlotter**(*state_map=None*, *update_time=0.5*, *yscale='linear'*, *sort_by='categories'*)
Plotter which produces a barplot of counts.

Initializes the *Plotter*.

Parameters

- **state_map** (*Optional[Callable[[Hashable], Any]]*) – An optional function mapping states to categories.
- **yscale** – The scale used for the yaxis, passed into ax.set_yscale. Defaults to 'linear'.
- **sort_by** (*str*) –

Return type None

initialize()
Initializes the barplot.

If *state_map* gets changed, call *initialize* to update the barplot to show the new set *categories*.

Return type None

update(*index=None*)
Update the heights of all bars in the plot.

Parameters **index** (*Optional[int]*) –

Return type None

class ppsim.snapshot.**HistoryPlotter**(*state_map=None*, *update_time=0.5*, *yscale='linear'*, *sort_by='categories'*)
Plotter which produces a lineplot of counts over time.

Initializes the *Plotter*.

Parameters

- **state_map** (*Optional[Callable[[Hashable], Any]]*) – An optional function mapping states to categories.
- **yscale** – The scale used for the yaxis, passed into ax.set_yscale. Defaults to 'linear'.
- **sort_by** (*str*) –

Return type None

update(*index=None*)
Make a new history plot.

Parameters **index** (*Optional[int]*) –

Return type None

CHAPTER

THREE

PPSIM.SIMULATOR MODULE

The cython module which contains the internal simulator algorithms.

This is not intended to be interacted with directly. It is intended for the user to only interact with the class *Simulation*.

class ppsim.simulator.**Simulator**

Base class for the algorithm that runs the simulation.

The configuration is stored as an array of size q, so the states are the indices 0, . . . , q-1.

config

The integer array of counts representing the current configuration.

n

The population size (sum of config).

t

The current number of elapsed interaction steps.

q

The total number of states (length of config).

is_random

A boolean that is true if there are any random transitions.

random_depth

The largest number of random outputs from any random transition.

gen

A BitGenerator object which is the pseudorandom number generator.

bitgen

A pointer to the BitGenerator, needed to for the numpy random C-API.

Initializes the main data structures for *Simulator*.

Parameters

- **init_array** – An integer array of counts representing the initial configuration.
- **delta** – A q x q x 2 array representing the transition function. Delta[i, j] gives contains the two output states.
- **null_transitions** – A q x q boolean array where entry [i, j] says if these states have a null interaction.
- **random_transitions** – A q x q x 2 array. Entry [i, j, 0] is the number of possible outputs if transition [i, j] is random, otherwise it is 0. Entry [i, j, 1] gives the starting index to find the outputs in the array random_outputs if it is random.

- **random_outputs** – An array containing all outputs of random transitions, whose indexing information is contained in random_transitions.
- **transition_probabilities** – An array containing all random transition probabilities, whose indexing matches random_outputs.
- **seed** (*optional*) – An integer seed for the pseudorandom number generator.

reset()
Base function which will be called to reset the simulation.

Parameters

- **config** – The configuration array to reset to.
- **t** – The new value of self.t. Defaults to 0.

run()
Base function which will be called to run the simulation for a fixed number of steps.

Parameters

- **num_steps** – The number of steps to run the simulation.
- **max_wallclock_time** – A bound in seconds on how long the simulator will run for.

class ppsim.simulator.**SimulatorMultiBatch**
Uses the MultiBatch algorithm to simulate O(sqrt(n)) interactions in parallel.

The MultiBatch algorithm comes from the paper *Simulating Population Protocols in Subconstant Time per Interaction* (https://arxiv.org/abs/2005.03584). Beyond the methods described in the paper, this class also dynamically switches to Gillespie's algorithm when the number of null interactions is high.

urn
An *Urn* object which stores the configuration and has methods for sampling.

updated_counts
An additional *Urn* where agents are stored that have been updated during a batch.

logn
Precomputed log(n).

batch_threshold
Minimum number of interactions that must be simulated in each batch. Collisions will be repeatedly sampled up until batch_threshold interaction steps, then all non-colliding pairs of 'delayed agents' are processed in parallel.

row_sums
Array which stores sampled counts of initiator agents (row sums of the 'D' matrix from the paper).

row
Array which stores the counts of responder agents for each type of initiator agent (one row of the 'D' matrix from the paper).

m
Array which holds the outputs of samples from a multinomial distribution for batch random transitions.

do_gillespie
A boolean determining if the we are currently doing Gillespie steps.

silent
A boolean determining if the configuration is silent (all interactions are null).

reactions
A (num_reactions) x 4 array giving a list of reactions, as [input input output output]

enabled_reactions
An array holding indices of all currently applicable reactions.

num_enabled_reactions
The number of meaningful indices in enabled_reactions.

propensities
A num_reactions x 1 array holding the propensities of each reaction. The propensity of a reaction is the probability of that reaction * (n choose 2).

reaction_probabilities
A num_reactions x 1 array giving the probability of each reaction, given that those two agents interact.

gillespie_threshold
The probability of a non-null interaction must be below this threshold to keep doing Gillespie steps.

coll_table
Precomputed values to speed up the function sample_coll(r, u).

coll_table_r_values
Values of r, giving one axis of coll_table.

coll_table_u_values
Values of u, giving the other axis of coll_table.

num_r_values
len(coll_table_r_values), first axis of coll_table.

num_u_values
len(coll_table_u_values), second axis of coll_table.

r_constant
Used in definition of coll_table_r_values.

Initializes all additional data structures needed for MultiBatch Simulator.

get_enabled_reactions()
Updates *enabled_reactions* and *num_enabled_reactions*.

get_total_propensity()
Calculates the probability the next interaction is non-null.

gillespie_step()
Samples the time until the next non-null interaction and updates.

Parameters **t_max** – Defaults to 0. If positive, the maximum value of *t* that will be reached. If the sampled time is greater than t_max, then it will instead be set to t_max and no reaction will be performed. (Because of the memoryless property of the geometric, this gives a faithful simulation up to step t_max).

multibatch_step()
Sample collisions to build a batch, then update the entire batch in parallel.

See the paper for a more detailed explanation of the algorithm.

reset()
Reset to a given configuration.

Sets all parameters necessary to change the configuration.

Parameters

- **config** – The configuration array to reset to.
- **t** – The new value of *t*. Defaults to 0.

run()
Run the simulation for a fixed number of steps.

Parameters

- **end_step** – Will run until self.t = end_step.
- **max_wallclock_time** – A bound in seconds this will run for.

run_until_silent()
Run the simulation until silent.

set_n_parameters()
Initialize all parameters that depend on the population size n.

class ppsim.simulator.**SimulatorSequentialArray**
A Simulator that sequentially chooses random agents from an array.

population
A length-n array with entries in 0, …, q-1 giving the states of each agent.

Initializes Simulator, then creates the population array.

make_population()
Creates the array self.population.

This is an array of agent states, where the count of each state comes from *Simulator.config*.

reset()
Reset to a given configuration.

Sets all parameters necessary to change the configuration.

Parameters

- **config** – The configuration array to reset to.
- **t** – The new value of *t*. Defaults to 0.

run()
Samples random pairs of agents and updates them until reaching end_step.

class ppsim.simulator.**Urn**
Data structure for a multiset that supports fast random sampling.

config
The integer array giving the counts of the multiset.

bitgen
Pointer to a BitGenerator, needed for numpy random C API.

order
An integer array giving the ranking of the counts from largest to smallest.

size
sum(config).

length
len(config).

CHAPTER

FOUR

PPSIM.CRN MODULE

Module for expression population protocols using CRN notation. Ideas and much code taken from https://github.com/enricozb/python-crn.

The general syntax is

```
a, b, u = species('A B U')
approx_majority = [
    a + b >> 2 * u,
    a + u >> 2 * a,
    b + u >> 2 * b,
]
n = 10 ** 5
init_config = {a: 0.51 * n, b: 0.49 * n}
sim = Simulation(init_config=init_config, rule=approx_majority)
```

In other words, a list of reactions is treated by the ppsim library just like the other ways of specifying population protocol transitions (the *rule* parameter in the constructor for *Simulation*, which also accepts a dict or a Python function).

More examples given in https://github.com/UC-Davis-molecular-computing/ppsim/tree/main/examples

ppsim.crn.**species**(*sp*)

Create a list of *Specie* (Single species *Expression*'s), or a single one.

Parameters **sp** (*Union[str, Iterable[str]]*) – An string or Iterable of strings representing the names of the species being created. If a single string, species names are interpreted as space-separated.

Return type Tuple[*ppsim.crn.Specie*, ...]

Examples:

```
w, x, y, z = species('W X Y Z')
rxn = x + y >> z + w
```

```
w, x, y, z = species(['W', 'X', 'Y', 'Z'])
rxn = x + y >> z + w
```

ppsim.crn.**replace_reversible_rxns**(*rxns*)

Parameters **rxns** (*Iterable[ppsim.crn.Reaction]*) – list of *Reaction*'s

Return type List[*ppsim.crn.Reaction*]

Returns: list of *Reaction*'s, where every reversible reaction in *rxns* has been replaced by two irreversible reactions, and all others have been left as they are

ppsim.crn.**reactions_to_dict**(*reactions*, *n*, *volume*)

Returns dict representation of *reactions*, transforming unimolecular reactions to bimolecular, and converting rates to probabilities, also returning the max rate so the *Simulation* knows how to scale time.

Parameters

- **reactions** (*Iterable[ppsim.crn.Reaction]*) – list of *Reaction*'s
- **n** (*int*) – the population size, necessary for rate conversion
- **volume** (*float*) – parameter as defined in Gillespie algorithm

Returns (transitions_dict, max_rate), where *transitions_dict* is the dict representation of the transitions, and *max_rate* is the maximum rate for any pair of reactants, i.e., if we have reactions (a + b >> c + d).k(2) and (a + b >> x + y).k(3), then the ordered pair (a,b) has rate 2+3 = 5

Return type Tuple[Dict[Tuple[*Specie*, *Specie*], Union[Tuple[*Specie*, *Specie*], Dict[Tuple[*Specie*, *Specie*], float]]], float]

ppsim.crn.**convert_unimolecular_to_bimolecular_and_flip_reactant_order**(*reactions*, *n*, *volume*)

Process all reactions before being added to the dictionary.

bimolecular reactions have their rates multiplied by the corrective factor (n-1) / (2 * volume). Bimolecular reactions with two different reactants are added twice, with their reactants in both orders.

Parameters

- **reactions** (*Iterable[ppsim.crn.Reaction]*) –
- **n** (*int*) –
- **volume** (*float*) –

Return type List[*ppsim.crn.Reaction*]

class ppsim.crn.**Specie**(*name: 'str'*)

Parameters **name** (*str*) –

Return type None

class ppsim.crn.**Expression**(*species*)

Class used for very basic symbolic manipulation of left/right hand side of stoichiometric equations. Not very user friendly; users should just use the *species* functions and manipulate those to get their reactions.

Parameters **species** (*List[ppsim.crn.Specie]*) –

Return type None

species: List[*ppsim.crn.Specie*]

ordered list of species in expression, e.g, A+A+B is [A,A,B]

get_species()

Returns the set of species in this expression, not their coefficients.

Return type Set[*ppsim.crn.Specie*]

ppsim.crn.**concentration_to_count**(*concentration*, *volume*)

Parameters

- **concentration** (*float*) – units of M (molar) = moles / liter
- **volume** (*float*) – units of liter

Returns count of molecule with *concentration* in *volume*

Return type int

class ppsim.crn.**RateConstantUnits**(*value*)
An enumeration.

stochastic = 'stochastic'
Units of L/s. Multiple by Avogadro's number to convert to mass-action units.

mass_action = 'mass_action'
Units of /M/s. Divide by Avogadro's number to convert to stochastic units.

class ppsim.crn.**Reaction**(*reactants*, *products*, *k=1*, *r=1*, *rate_constant_units=RateConstantUnits.stochastic*, *rate_constant_reverse_units=RateConstantUnits.stochastic*, *reversible=False*)
Representation of a stoichiometric reaction using a pair of Expressions, one for the reactants and one for the products.

Parameters

- **reactants** (ppsim.crn.Expression) – left side of species in the reaction
- **products** (ppsim.crn.Expression) – right side of species in the reaction
- **k** (*float*) – Rate constant of forward reaction
- **r** (*float*) – Rate constant of reverse reaction (only used if *Reaction.reversible* is true
- **rate_constant_units** (ppsim.crn.RateConstantUnits) – Units of forward rate constant
- **rate_constant_reverse_units** (ppsim.crn.RateConstantUnits) – Units of reverse rate constant
- **reversible** (*bool*) – Whether reaction is reversible

Return type None

reactants: *ppsim.crn.Expression*
The left side of species in the reaction.

products: *ppsim.crn.Expression*
The right side of species in the reaction.

rate_constant: float = 1
Rate constant of forward reaction.

rate_constant_reverse: float = 1
Rate constant of reverse reaction (only used if *Reaction.reversible* is true).

rate_constant_units: *ppsim.crn.RateConstantUnits* **= 'stochastic'**
Units of forward rate constant.

rate_constant_reverse_units: *ppsim.crn.RateConstantUnits* **= 'stochastic'**
Units of reverse rate constant.

reversible: bool = False
Whether reaction is reversible, i.e. products → reactants is a reaction also.

is_unimolecular()
Returns: true if there is one reactant

Return type bool

`is_bimolecular()`
Returns: true if there are two reactants

Return type bool

`symmetric()`
Returns: true if there are two reactants that are the same species

Return type bool

`symmetric_products()`
Returns: true if there are two products that are the same species

Return type bool

`num_reactants()`
Returns: number of reactants

Return type int

`num_products()`
Returns: number of products

Return type int

`is_conservative()`
Returns: true if number of reactants equals number of products

Return type bool

`reactant_if_unimolecular()`
Returns: unique reactant if there is only one Raises: ValueError if there are multiple reactants

Return type *ppsim.crn.Specie*

`product_if_unique()`
Returns: unique product if there is only one Raises: ValueError if there are multiple products

Return type *ppsim.crn.Specie*

`reactants_if_bimolecular()`
Returns: pair of reactants if there are exactly two Raises: ValueError if there are not exactly two reactants

Return type Tuple[*ppsim.crn.Specie*, *ppsim.crn.Specie*]

`reactant_names_if_bimolecular()`
Returns: pair of reactant names if there are exactly two Raises: ValueError if there are not exactly two reactants

Return type Tuple[str, str]

`products_if_exactly_two()`
Returns: pair of products if there are exactly two Raises: ValueError if there are not exactly two products

Return type Tuple[*ppsim.crn.Specie*, *ppsim.crn.Specie*]

`product_names_if_exactly_two()`
Returns: pair of product names if there are exactly two Raises: ValueError if there are not exactly two products

Return type Tuple[str, str]

`property rate_constant_stochastic: float`
forward rate constant in stochastic units (converts from mass-action if necessary)

Type Returns

property rate_constant_reverse_stochastic: float
reverse rate constant in stochastic units (converts from mass-action if necessary)

Type Returns

k(*coeff*, *units=RateConstantUnits.stochastic*)
Changes the reaction coefficient to *coeff* and returns *self*.

This is useful for including the rate constant during the construction of a reaction. For example

```
x, y, z = species("X Y Z")
rxns = [
    (x + y >> z).k(2.5),
    (z >> x).k(1.5),
    (z >> y).k(0.5)),
]
```

Parameters

- **coeff** (*float*) – float The new reaction coefficient
- **units** (ppsim.crn.RateConstantUnits) – float units of rate constant (default stochastic)

Return type *ppsim.crn.Reaction*

r(*coeff*, *units=RateConstantUnits.stochastic*)
Changes the reverse reactionn reaction rate constant to *coeff* and returns *self*.

This is useful for including the rate constant during the construction of a reaction. For example

```
x, y, z = species("X Y Z")
rxns = [
    (x + y >> z).k(2.5),
    (z >> x).k(1.5),
    (z >> y).k(0.5)),
]
```

Parameters

- **coeff** (*float*) – float The new reverse reaction rate constant
- **units** (ppsim.crn.RateConstantUnits) – float units of rate constant (default stochastic)

Return type *ppsim.crn.Reaction*

get_species()
Return: the set of species present in the products and reactants.

Return type Set[*ppsim.crn.Specie*]

ppsim.crn.**species_in_rxns**(*rxns*)

Parameters **rxns** (*Iterable[ppsim.crn.Reaction]*) – iterable of *Reaction*'s

Return type List[*ppsim.crn.Specie*]

Returns: list of species (without repetitions) in *Reaction*'s in *rxns*

ppsim.crn.**gillespy2_format**(*init_config*, *rxns*, *volume=1.0*)
Create a gillespy2 Model object from a CRN description.

Parameters

- **init_config** (*Dict[ppsim.crn.Specie, int]*) – dict mapping each *Specie* to its initial count
- **rxns** (*Iterable[ppsim.crn.Reaction]*) – reactions to translate to StochKit format
- **volume** (*float*) – volume in liters
- **name** – name of the CRN

Returns An equivalent gillespy2 Model object

Return type Any

ppsim.crn.**stochkit_format**(*init_config*, *rxns*, *volume=1.0*, *name='CRN'*)

Parameters

- **rxns** (*Iterable[ppsim.crn.Reaction]*) – reactions to translate to StochKit format
- **init_config** (*Dict[ppsim.crn.Specie, int]*) – dict mapping each *Specie* to its initial count
- **volume** (*float*) – volume in liters
- **name** (*str*) – name of the CRN

Returns string describing CRN in StochKit XML format

Return type str

ppsim.crn.**write_stochkit_file**(*filename*, *rxns*, *init_config*, *volume=1.0*, *name='CRN'*)
Write stochkit file :param filename: name of file to write :param rxns: reactions to translate to StochKit format :param init_config: dict mapping each *Specie* to its initial count :param volume: volume in liters :param name: name of the CRN

Parameters

- **filename** (*str*) –
- **rxns** (*Iterable[ppsim.crn.Reaction]*) –
- **init_config** (*Dict[ppsim.crn.Specie, int]*) –
- **volume** (*float*) –
- **name** (*str*) –

Return type None

CHAPTER

FIVE

INDICES AND TABLES

- genindex
- modindex
- search

PYTHON MODULE INDEX

p

INDEX

Symbols

A

B

C

D

E

F

G

H

I

S

T

U

W

Y